



Compliance checking on first-order knowledge with conflicting and compensatory norms: a comparison among currently available technologies

Livio Robaldo¹ · Sotiris Batsakis^{2,3} · Roberta Calegari⁴ ·
Francesco Calimeri⁵ · Megumi Fujita⁸ · Guido Governatori⁷ ·
Maria Concetta Morelli⁵ · Francesco Pacenza⁵ · Giuseppe Pisano⁴ ·
Ken Satoh⁶ · Ilias Tachmazidis³ · Jessica Zangari⁵

Accepted: 26 March 2023
© Crown 2023

Abstract

This paper analyses and compares some of the automated reasoners that have been used in recent research for compliance checking. Although the list of the considered reasoners is not exhaustive, we believe that our analysis is representative enough to take stock of the current state of the art in the topic. We are interested here in formalizations at the *first-order* level. Past literature on normative reasoning mostly focuses on the *propositional* level. However, the propositional level is of little usefulness for concrete LegalTech applications, in which compliance checking must be enforced on (large) sets of individuals. Furthermore, we are interested in technologies that are *freely available* and that can be further investigated and compared by the scientific community. In other words, this paper does not consider technologies only employed in industry and/or whose source code is non-accessible. This paper formalizes a selected use case in the considered reasoners and compares the implementations, also in terms of simulations with respect to shared synthetic datasets. The comparison will highlight that lot of further research still needs to be done to integrate the benefits featured by the different reasoners into a single standardized first-order framework, suitable for LegalTech applications. All source codes are freely available at <https://github.com/liviorobaldo/compliancecheckers>, together with instructions to locally reproduce the simulations.

Keywords Compliance checking · First-order knowledge · LegalTech

The main author of this paper was Livio Robaldo, who coordinated and organized the overall work. All other authors contributed equally to the research presented here and, therefore, they have been listed in alphabetical order on their surnames.

✉ Livio Robaldo
livio.robaldo@swansea.ac.uk

Extended author information available on the last page of the article

1 Introduction

Formalizing the meaning of norms has been the objective of deontic logic over the last fifty years. Formalization of norms requires deontic operators to represent the involved modalities (obligatory, permitted, and prohibited), as well as non-monotonic operators fit to handle the central role of defeasibility in normative reasoning (Gabbay et al. 2013).

Formalizations found in past relevant literature are typically propositional, i.e. their basic components are propositions, which generally denote whole sentences. The most popular, and freely available,¹ normative reasoner is perhaps SPINdle (Lam and Governatori 2009), which only accepts propositional formulae as input.

However, propositions are of little usefulness for real-world LegalTech applications dealing with big data [see (Antoniou et al. 2021)], due to their very limited expressivity. It is then necessary to enhance the expressivity of the underlying logical formats to the first-order level, fit to distinguish individuals from predicates and allow the evaluation of deontic formulae to iterate over (large) sets of individuals.

Alternatively, it is possible to execute first-order rules within SPINdle (or within any other propositional reasoner) by *grounding* the rules, i.e., by transforming them into sets of propositional rules. This is usually done by instantiating the first-order rules on all possible input facts (e.g., from $\forall_x [A(x) \rightarrow B(x)]$ to $A(a) \rightarrow B(a)$, $A(b) \rightarrow B(b)$, $A(c) \rightarrow B(c)$, assuming that $\{a, b, c\}$ is the set of individuals in the reference state of affairs).

Grounding is a widely used technique in modern reasoners (Kaufmann et al. 2016). It allows the use of propositional reasoners on finite domains, e.g., on existing sets of tuples from real-world databases. However, it is computationally expensive, so solutions have been investigated to identify ground sets of rules smaller in size than the full instantiation but with the same semantics (Gebser et al. 2011; Faber et al. 2012).

1.1 Compliance checking

This paper focuses on compliance checking with conflicting and compensatory norms. Compliance checking aims at assessing whether a certain state of affairs complies or not with a set of norms. We are interested here in sets of norms where some norms conflict with others, for which it is necessary to establish preference criteria among norms and to introduce defeasible operators to implement the overriding. On the other hand, compensatory norms denote obligations associated with violations of other norms; once these obligations are fulfilled, they “compensate” the associated violations so that the state of affairs is again compliant with the norms.

Compensatory norms have been scarcely investigated in the literature. To the best of our knowledge, only (Governatori and Rotolo 2006, 2019) offer formalizations of these type of norms. The formalization in (Governatori and Rotolo 2006) has been implemented within the RuleRS system (Islam and Governatori 2018) and the Regorous system (Governatori 2015), which are both based on the aforementioned SPINdle

¹ <https://sourceforge.net/projects/spindlereasoner>.

reasoner. However, the implementations of the two systems are not publicly available as they are both protected by Data61 copyright.² Further details about the (propositional) formalization in (Governatori and Rotolo 2006) can be found below in Sect. 3; on the other hand, this paper offers further (first-order) implementations of compensatory norms in the legal reasoners evaluated and compared below in Sect. 4.

On the other hand, examples of formalizations of conflicting norms may be found in Palmirani and Governatori (2018), among others. The proposal in Palmirani and Governatori (2018) distinguishes between *monotonic* knowledge, encoded within an OWL ontology for the GDPR called PrOnto (Palmirani et al. 2018), and *non-monotonic* knowledge, i.e., deontic and defeasible legal rules that implement some selected GDPR norms. Legal rules are encoded within a separate knowledge base in LegalRuleML (Athanasopoulos et al. 2015), which is a recent OASIS XML standard³ for legal reasoning.

Since a reasoner that directly acts on LegalRuleML representations is not currently available, the LegalRuleML rules in Palmirani and Governatori (2018) are converted into the input format of the SPINdle reasoner.

Following Palmirani et al. (2018), in this paper we will formalize the *monotonic* knowledge of our selected use case in OWL and the *non-monotonic* legal rules in the formats that we will compare. The comparison will include experiments with large synthetic datasets whose sizes are comparable to the ones of databases used in industry.

The next section illustrates the use case. This is taken from Batsakis et al. (2018), who present an analysis similar to the one of this paper but, again, at the propositional level only. Propositional formalizations, as mentioned above and as further discussed in Sect. 3 below, represent the current state of the art in legal reasoning.

Section 4 contains the core of the paper. It proposes possible formalizations of the use case in some of the main contemporary automated reasoners used for compliance checking. After introducing in Sect. 4.1 the reference OWL ontology that encodes the monotonic knowledge shared by all reasoners, the section will formalize the use case in SHACL (Sect. 4.2), ASP (Sect. 4.3), DLV (Sect. 4.4), PROLEG (Sect. 4.5), Arg2P (Sect. 4.6), and SPINdle (Sect. 4.7). The latter is the single reasoner among those considered in this paper that has a propositional input format. On the contrary, the input formats of all the other reasoners are first-order. Still, we decided to include SPINdle in our analysis because, as pointed out above, it is widely acknowledged in the literature in legal reasoning.

A computational comparison of the legal reasoners is shown in Sect. 5. This is given in terms of evaluations with respect to automatically generated synthetic datasets of increasing size. Finally, Sect. 6 discusses the computational results as well as the pros and cons of each reasoner. The paper concludes by pointing out directions of future research fit to combine the benefits of each reasoner into a single integrated framework.

2 The use case

In this paper, we use the following use case:

² <https://research.csiro.au/data61/regorous>; <https://research.csiro.au/bpli/tools/rulers>.

³ <https://www.oasis-open.org/committees/legalruleml>.

- (1)
- *Article 1.* The Licensor grants the Licensee a licence to evaluate the Product.
 - *Article 2.* The Licensee must not publish the results of the evaluation of the Product without the approval of the Licensor. If the Licensee publishes results of the evaluation of the Product without approval from the Licensor, the material must be removed.
 - *Article 3.* The Licensee must not publish comments about the evaluation of the Product, unless the Licensee is permitted to publish the results of the evaluation.
 - *Article 4.* If the Licensee is commissioned to perform an independent evaluation of the Product, then the Licensee has the obligation to publish the evaluation results.

The use case in (1) is a simplification of use case 2 from Batsakis et al. (2018). This is in turn taken from Governatori et al. (2018). We simplified the use case by removing all temporal information. For instance, in the original version of Article 2 the Licensee is obliged to remove the material *within 24h* after its publication. Although time may be modelled in OWL (Batsakis et al. 2017) and other first-order formats such as ASP (Giordano et al. 2013), we believe that adding time management will not constitute a relevant additional element of comparison with respect to the goal of modelling conflicting and compensatory norms. Instead, it demands more fine-grained formalizations to distinguish between achievement and maintenance obligations (Governatori and Rotolo 2019), which we consider as part of our future work and not in the scope of this study.

Since the representations investigated here do not model time, we interpret all norms with respect to the state of affairs held at the time “now”. Thus, if “now” the Licensee has published the material without the Licensor’s approval *and* they have “now” removed it, then the Licensee is “now” complying with Article 2; otherwise, they are not.

2.1 Identifying norms from text

Formalizing the articles in (1) requires identifying the several norms they denote and encoding these norms in some machine-readable format. According to standard legal theory (Sartor 2009), norms are formalized as if-then rules having a deontic statement (i.e., an obligation, a permission, or a prohibition) in the consequent and, in the antecedent, the conditions for this statement to hold true in the context.

Norms and corresponding if-then rules may be defeasible, in the sense that some of them may *override* others. Therefore, in order to properly formalize the articles in (1), we must also identify and formalize which norms override which other ones.

Finally, as stated above, some of the rules may *compensate* violation of others. These rules specify obligations that, when fulfilled, repair the non-compliance of other rules. For instance, Article 2 of the use case specifies that if the Licensee publishes the results of the evaluation without the Licensor’s approval, a new obligation occurs: the Licensee is obliged to remove the results. In case this obligation is fulfilled, the violation had still taken place, but it was repaired/compensated.

In this paper, the articles in (1) are formalized as in (2). Each item in (2) either specifies an if-then rule or which if-then rule overrides or compensates which other one.

Article 1(a) is modeled as an if-then rule having the antecedent as true (symbol “ \top ”), i.e., an if-then rule in the form “ $\top \rightarrow P$ ”, where P is the prohibition to evaluate the product. The if-then rule states that P *always* holds. Thus, Article 1(a) conflicts with Article 1(b) in the specific scenario in which the Licensor grants the Licensee a licence to evaluate the Product. Article 1(a) is then seen as a general rule of which Article 1(b) represents a context-specific exception. In order to solve the conflict, Article 1(c) specifies that when the exception holds, the general rule is overridden.

Similar considerations hold between Article 2(a) and Article 2(b). Whenever they both hold, the latter overrides the former, as specified in Article 2(d).

Article 2(c) represents an obligation that holds in case Article 2(a) has been violated. Article 2(c) entails an obligation for the Licensee to *compensate* for their violation.

Article 3 and Article 4 provide further examples of conflicting rules, which are again solved by specifying which rules override which other ones. These conflicting rules do not feature any relevant difference with respect to the previous ones. The only observation worth noticing is that Article 4(b) cannot indeed be deduced from any explicit linguistic marker in (1). In other words, (1) does not actually specify that, when an independent evaluation of the Product is performed *and* the Licensee does not have the Licensor’s approval, the obligation of publishing the results is “stronger” than the prohibition of not publishing them. Therefore, it might be contested that the norms in (1) leave a blank, as they do not actually specify what the Licensee should do in such a context. In this paper, Article 4(b) is assumed to solve the conflict between Article 2(a) and Article 4(a), but this is actually an addition of the paper’s authors.

More generally, sometimes it could be difficult to formalize norms from legislation. Since natural language may be ambiguous, certain norms could be formalized in several ways, depending on how we interpret the original intentions of the legislator.

As this paper is not concerned with the *extraction* of (formal) if-then rules representing natural language norms, and all related problems arising from ambiguous interpretations of the text, we will simply assume here that the articles in (1) have the meaning in (2).

(2) • **Article 1.**

- (a) The Licensee is prohibited to evaluate the Product.
- (b) If the Licensor grants the Licensee a licence to evaluate the Product, then the Licensee is permitted to evaluate the Product.
- (c) Article 1b overrides Article 1a.

• **Article 2.**

- (a) If the Product has been evaluated, then the Licensee is prohibited to publish the results of the evaluation of the Product.
- (b) If the Licensor approves the publishing of the results of the evaluation of the Product, then the Licensee is permitted to publish the results of the evaluation of the Product.

- (c) If the Licensee publishes the results of the evaluation of the Product without approval of the Licensor, then the Licensee is obliged to remove the results of the evaluation of the Product.
- (d) Article 2b overrides Article 2a.
- (e) Article 2c compensates Article 2a.
- **Article 3.**
 - (a) If the Product has been evaluated, then the Licensee is prohibited to publish comments about the evaluation of the Product.
 - (b) If the Licensee is permitted to publish the results of the evaluation of the Product, then the Licensee is permitted to publish comments about the evaluation of the Product.
 - (c) Article 3b overrides Article 3a.
- **Article 4.**
 - (a) If the Licensee is commissioned to perform an independent evaluation of the Product, then the Licensee is obliged to publish the results of the evaluation of the Product.
 - (b) Article 4a overrides Article 2a.

To summarize, the use case in (2) contains eight norms in total, constituting three prohibitions, three permissions, and two obligations. Four of these norms, i.e., Article 1b, Article 2b, Article 3b, and Article 4a, override other norms. One of the two obligations, i.e., Article 2c, compensates the violation of one of the four prohibitions, i.e., Article 2a. Finally, Article 3b is an example of “nested permission”, as defined in Robaldo et al. (2020 §4.1); its condition is another permission, i.e., the one triggered by Article 2b.

We deem the variety of these norms to be representative enough both with respect to real-world legislation and with respect to the sample use cases investigated so far in the literature in legal reasoning.

Concerning real-world legislation, we direct the reader to the aforementioned work in Robaldo et al. (2020), which translates the norms of the European General Data Protection Regulation (GDPR)⁴ into first-order if-then rules encoded in LegalRuleML. The knowledge base, called “DAPRECO knowledge base” (“D-KB” for short), includes about 1000 rules and it is, to date, the biggest repository in the XML standard.

All types of if-then rules within the D-KB are exemplified in our use case. In addition, our use case includes a type of norm that does not occur in the D-KB, i.e., compensatory norms. As already pointed out above, compensations have been mainly neglected in past literature, the only exceptions being (Governatori and Rotolo 2006, 2019, and some other related works by the same authors.

It is also worth noticing that the LegalRuleML standard does include a special XML tag “<SuborderList>”⁵ to specify the sequence of compensatory norms associated with a violation. Still, to the best of our knowledge, LegalRuleML corpora tagging existing legislation and using this tag are not available in the literature. Besides the

⁴ <https://eur-lex.europa.eu/eli/reg/2016/679/oj>.

⁵ See §4.2.3.2 of the official specification at <https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/os/legalruleml-core-spec-v1.0-os.pdf>.

aforementioned D-KB, another work using LegalRuleML to tag existing legislation, specifically the Smoking Prohibition (Scotland) Act 2016,⁶ is Nazarenko et al. (2018).

Robaldo et al. (2020), Nazarenko et al. (2018) developed and used suitable Web editors to create their corpora. The lack of available annotated corpora of norms from existing legislation is perhaps due to the difficulties of creating such editors. These difficulties have been recently addressed in Libal (2022) and further investigated in ongoing research activities by the same author.

The *sample* use cases investigated in past literature in legal reasoning do not seem to substantially differ from the use case in (2) either, at least from the point of view of the variety of norms included therein. As an example, we address the reader to the recent contribution in Francesconi and Governatori (2022), which formalizes compliance checking in OWL⁷ while exemplifying the framework on a use case including three prohibitions, two permissions, and one obligation, with four of these norms overriding one of the others. Therefore, this use case does not include compensatory norms or nested permissions/obligations, which are instead considered here. Indeed, it is not clear how to model these types of norms in OWL. For future developments of their framework, Francesconi and Governatori (2022) suggest using SHACL in place of OWL. SHACL is the first language in which we will formalize our use case, below in Sect. 4.2.

3 Formalizing norms at the propositional level

This section discusses how to formalize norms in propositional logic to enable compliance checking. The rest of the paper will evolve these formalizations to the first-order level.

Formalization of norms requires deontic and defeasible operators. Concerning deontic operators, the standard solution is to introduce special *modal* operators to model obligations, permissions, or prohibitions. This solution is also adopted in SPINdle, in which Article 1(a-b) from (2) may be formalized as follows:

```
Art1a[O] : => -Evaluate
Art1b[P] : hasLicence => Evaluate
```

Evaluate and hasLicence are two propositional symbols respectively referring to the whole sentences “The Licensee evaluates the Product” and “The Licensee has the Licence”. The symbols “-” and “=>” implement standard negation and entailment. Finally “[O]” and “[P]” are SPINdle’s deontic modal operators for obligation and permission respectively. Whenever the antecedent of the if-then rules is satisfied, SPINdle applies the deontic modality to the consequent and stores the result in its working memory.

Note that SPINdle represents prohibitions as obligations, in virtue of the equivalence that something is prohibited if and only if its contrary is obligatory. Thus, the two formulae above read that it is always obligatory to *not* evaluate the product

⁶ <https://www.legislation.gov.uk/asp/2016/3>.

⁷ <https://www.w3.org/OWL>.

(“[O]-Evaluate” always holds) and that if the Licensee has the licence, they are permitted to evaluate the product (if “hasLicence” holds, then “[P]Evaluate” is added to the working memory).

The two assertions “[O]-Evaluate” and “[P]Evaluate” cannot of course exist together in the working memory, otherwise, SPINdle would detect an inconsistency. In order to avoid that, a preference between the two rules must be asserted. In our use case, whenever the antecedent of the rule with the label “Art1b” is satisfied, “[P]Evaluate” overrides “[O]-Evaluate”. In SPINdle, this preference is enforced by adding the following superiority relation, stating that whenever the rule with label “Art1b” is triggered, the consequent of the rule with label “Art1a” is overridden:

$$\text{Art1b} > \text{Art1a}$$

Superiority relations are the defeasible operators used in SPINdle to solve conflicts between rules. Superiority relations are also used in other popular legal reasoners such as PROLEG (Sato et al. 2011) (see Sect. 4.5 below).

Article 2(a–b), Article 2(d), Article 3(a–c), and Article 4(a–b) in (2) are similarly modelled on SPINdle’s input format (see Sect. 4.7 below).

However, SPINdle version 2.2.4, i.e., the version available online and distributed under GNU LGPL, does not provide explicit operators to implement compensations of rules, e.g., Article 2(e). These must be externally implemented, as is done in the aforementioned systems RuleRS and Regorous. Both RuleRS and Regorous use SPINdle to compute legal rules, but they add the operator “ \otimes ” from Governatori and Rotolo (2006) to specify which norms compensate which other norms, in cases where the latter are violated. Article 2(a) may be then formalized as follows:

$$\text{Art2a}[O]: \text{Evaluate} \Rightarrow \text{-Publish} \otimes \text{Remove}$$

meaning that if the evaluation took place, it is obligatory to not publish the results. The proposition `-Publish` is, however, related to the proposition `Remove` via the \otimes operator, meaning that removing what has been published will indeed compensate for the violation. @@ As pointed out above, the source codes of RuleRS and Regorous are not available. Therefore, it is actually unknown how they implement the “ \otimes ” operator. Governatori (2015) specify that Regorous uses some internal sets to maintain the if-then rules that have been violated as well as those that have been compensated for. Indeed, the latter might be simply computed by removing from the former all obligations/prohibitions associated via the “ \otimes ” operator with an obligation that has been fulfilled. In other words, even if the technical details of how RuleRS and Regorous implement and enforce compensations are unknown, the task appears to be rather intuitive.

Below in Sect. 4.7, compensatory norms will be implemented by introducing specific additional SPINdle rules and facts. Similar rules will be asserted to handle compensations also in the other reasoners considered below.

4 Formalizing norms at the first-order level

This section proposes possible implementations of the if-then rules in (2) at the first-order level. In line with Palmirani et al. (2018), we will collect and store all the monotonic knowledge of the use case within a shared OWL ontology. On the other hand, the if-then rules representing the norms will be formalized and stored within separate knowledge bases in the formats that we will compare. The if-then rules, in any of the considered formats, will all refer to the content of the OWL ontology. Specifically, they will involve predicates corresponding 1:1 to the OWL classes or properties of the ontology.

4.1 The reference ontology

The reference ontology used in this paper⁸ defines the minimal set of concepts needed to model the norms in the considered use case. The ontology includes OWL resources for modelling the actions together with their thematic roles, as well as OWL resources to model the modalities together with their violations, exceptions, and compensations.

Figure 1 shows the screenshot in the interface of the Protégè editor for the OWL resources that model actions and thematic roles. These resources include three classes: *Actor*, *Action*, and *Object*. Individuals of *Action* are instances of the possible actions that may be carried out by the *Licensee*(s) or the *Licensor*(s), which are *Actor*(s). These actions are connected to the individuals in *Actor* and *Object* via some standard thematic roles (*has-agent*, *has-theme*, etc.), implemented as OWL object properties.

Some simple intuitive OWL constraints and restrictions are defined for these classes and properties, but they will not be described in detail here. For instance, as shown in Fig. 1, the class *Approve* includes the OWL restrictions *has-agent* exactly 1 *Licensor* and *has-theme* exactly 1 *Publish*, stating that each approval is performed by exactly one licensor and concerns exactly one action of publishing.

To model our use case, we will only consider *Action*(s) that exhaustively specify all (and only) their thematic roles. In other words, although RDFs/OWL makes the open-world assumption, and so allowing the addition of instances of *Action* that do *not* specify the values of the properties *has-agent*, *has-theme*, etc., in our ABox we do not allow so: the synthetic datasets that we generate to run our simulations (see Sect. 5.1 below) fully specify the thematic roles for every instance of *Action*.

Figure 2 shows a screenshot of the interface of the Protégè editor for the OWL resources that model modalities together with their violations, exceptions, and compensations.

Most of the reasoners illustrated below do not use modal operators to model deontic modalities nor do they use superiority relations to model defeasibility. One of the reasons to avoid modal operators and superiority relations is that they are second-order operators whose management introduces extra complexities that could seriously hinder performance [cf. Sun and Robaldo (2017)].

⁸ The ontology is available at <https://github.com/liviorobaldo/compliancecheckers/blob/main/SHACL/licenceusecaseTBox.owl>.

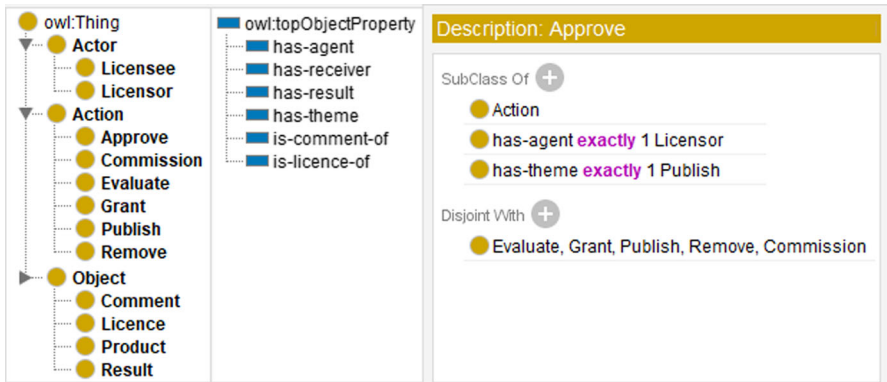
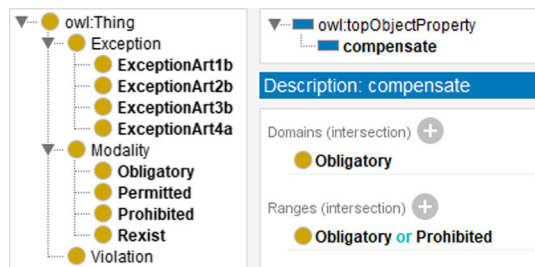


Fig. 1 OWL resources to model actions together with their thematic roles. In the figure, the class `Approve` is shown in full detail

Fig. 2 OWL resources to model modalities together with their violations, exceptions, and compensations



Therefore, following Batsakis et al. (2018), deontic modalities will be represented, in the input format of these reasoners, via additional first-order predicates corresponding each to an ontological class: `Obligatory`, `Permitted`, and `Prohibited`. These classes are all subclasses of another class `Modality`.

Furthermore, following Robaldo and Sun (2017), we introduce another class `Rexist` to mark actions that really take place in the state of affairs. `Rexist` follows the assumption those actions are not “true” or “false”; actions can really exist or not, thus we need an explicit modality to encode their real existence in the state of affairs.

In place of the superiority relations, the ontology includes OWL classes to assert whether certain actions represent *exceptions* of other if-then rules. In Fig. 2, exceptions are declared as subclasses of the class `Exception`. Therefore, the general rules are triggered only in cases where the actions do *not* belong to these classes, i.e., only in cases where these exceptions do not hold. These classes are usually termed in the literature as “undercutting defeaters”; they were originally introduced in Nute (1994).

Finally, the ontology includes a further object property `compensate` that relates individuals of the class `Obligation` with individuals of the (union) class `Obligatory` \cup `Prohibited` as well as a further class `Violation` whose individuals will refer to the violated (and not compensated) obligations or prohibitions.

4.2 Implementing the use case in SHACL

In this subsection, we formalize the if-then rules in (2) as SHACL rules. This is the first implementation we present because SHACL rules can be directly executed on ABox(es) of RDF triples populating the (TBox) ontology described in the previous subsection. The use of SHACL for legal reasoning has been recently investigated in Robaldo (2021).

The Shapes Constraint Language (SHACL)⁹ is a W3C recommendation to validate and reason with RDF triplestores. SHACL was originally proposed to define special conditions, called “SHACL shapes”, against which RDF graphs can be validated. However, a current W3C Working Group Note proposes to enrich SHACL shapes with “SHACL rules”¹⁰ to derive inferred triples from asserted ones, thus transforming SHACL into a (logical) programming language that may be used to further purposes besides validation.

SHACL shapes and rules are themselves written in RDF. In other words, the W3C recommendation defines a set of RDF classes and properties, identified by the namespace <http://www.w3.org/ns/shacl#> and usually associated with the prefix “sh”, to assert shapes and rules. These are then executed against another RDF graph.

4.2.1 Representing the norms as SHACL rules

We have now all the ingredients to formalize the if-then rules in (2) as SHACL rules. The one corresponding to Article 1(a) in (2), stating that the Licensee is prohibited to evaluate the Product, is shown in (3).

```
(3) :evaluatingProductsIsProhibitedUnlessLicence rdf:type sh:NodeShape;
    sh:rule [rdf:type sh:SPARQLRule; sh:order 1;
            sh:prefixes[sh:declare
                        [sh:prefix "rdf"; sh:namespace "http://..."],
                        [sh:prefix "TBox"; sh:namespace "http://..."]];
            sh:construct """
                CONSTRUCT { $this rdf:type TBox:Prohibited. }
                WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
                        $this TBox:has-theme ?p. ?p rdf:type TBox:Product.
                        NOT EXISTS{$this rdf:type TBox:ExceptionArt1b}. }""";
            sh:targetClass TBox:Evaluate.
```

The rule in (3) is a special type of SHACL rule called “SPARQLRule”¹¹ and identified by the class `sh:SPARQLRule`. These rules are particularly flexible and expressive in that they allow to embed standard SPARQL CONSTRUCT-WHERE assertions within the rule. Simply put, SHACL rules of type `sh:SPARQLRule` are just standard SPARQL rules in the form CONSTRUCT-WHERE enriched with priorities.

In (3), TBox is the prefix of the ontology in Figs. 1 and 2. The rule in (3) is triggered on every individual of `sh:targetClass`, i.e., on every evaluating action. This is referred within the SHACL rule via the special term `$this`. For every evaluating

⁹ <https://www.w3.org/TR/shacl>.

¹⁰ See <https://www.w3.org/TR/shacl-af>.

¹¹ <https://www.w3.org/TR/shacl-af/#SPARQLRule>.

action for which the `WHERE` clause is true, the `CONSTRUCT` clause creates new triples in the RDF graph.

`NOT-EXISTS` is the SPARQL clause to implement negation-as-failure. The clause is true if the triple as its argument does not occur in the RDF graph.

Therefore, the rule in (3) asserts as type `Prohibited` every evaluating action that is not asserted as type `ExceptionArt1b`. In other words, the rule states that evaluating actions are prohibited unless they are also classified as exceptions under Article 1b.

The rule corresponding to Article 1b is shown in (4). This is executed *before* the rule in (3) because its `sh:order` is lower. `sh:order` is the SHACL operator to specify priorities on the rules; these are executed from the lowest value in `sh:order` to the highest.

Rule (4) states that if the theme of the evaluating action `$this` is a product `?p` with licence `?l`, and this licence is, in turn, the theme of a granting action `?eg` that really exists such that the receiver of `?eg` is the same agent of `$this`, then `$this` is classified as `Permitted` as well as `ExceptionArt1b`. Consequently, the rule (3) will no longer classify it as prohibited in that the negation-as-failure on `ExceptionArt1b` will turn out false.

```
(4)  sh:rule [rdf:type sh:SPARQLRule; sh:order 0;
        sh:prefixes[sh:declare
            [sh:prefix "rdf"; sh:namespace "http://..."],
            [sh:prefix "TBox"; sh:namespace "http://..."]];
        sh:construct ""
        CONSTRUCT { $this rdf:type TBox:Permitted.
                    $this rdf:type TBox:ExceptionArt1b. }
        WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
                $this TBox:has-theme ?p. ?p rdf:type TBox:Product.
                ?l TBox:is-licence-of ?p. ?l rdf:type TBox:Licence.
                ?eg TBox:has-agent ?y. ?y rdf:type TBox:Licensor.
                ?eg TBox:has-theme ?l. ?eg rdf:type TBox:Grant.
                ?eg rdf:type TBox:Rexist. ?eg TBox:has-receiver ?x. }""];
```

Article 2a is represented with the rule in (5). The rule is triggered on every action of type `Publish` whose theme is the result `?r` of an evaluation that really exists and for which neither of the exceptions in Article 2b and Article 4a holds in the state of affairs. Each publishing action as such is classified as `Prohibited`.

```
(5)  sh:rule [rdf:type sh:SPARQLRule; sh:order 2;
        sh:prefixes[sh:declare
            [sh:prefix "rdf"; sh:namespace "http://..."],
            [sh:prefix "TBox"; sh:namespace "http://..."]];
        sh:construct ""
        CONSTRUCT { $this rdf:type TBox:Prohibited. }
        WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
                $this TBox:has-theme ?r. ?r rdf:type TBox:Result.
                ?ev TBox:has-result ?r. ?ev rdf:type TBox:Evaluate.
                ?ev rdf:type TBox:Rexist.
                NOT EXISTS{$this rdf:type TBox:ExceptionArt2b}.
                NOT EXISTS{$this rdf:type TBox:ExceptionArt4a}. }""];
```

On the other hand, Article 2c is represented as in (6). Note that the `WHERE` clause of the rule in (6) is the same as the rule in (5) plus an additional triple requiring the

referred publishing action to also really exist in the state of affairs. In other words, the rule in (6) triggers for prohibitions that have been *violated*.

For each of these publishing actions, the CONSTRUCT clause asserts a new anonymous individual within the classes `Obligatory` and `Remove` whose theme is the result `?r`.

Anonymous individuals are represented in RDF by enclosing them among “[...]”. When inserted within a CONSTRUCT clause, they correspond to existential quantifications. Thus, rule (6) creates a *new* obligation: the removal of the result is obligatory whenever the Licensee published it, even though they were prohibited to do so. The new obligation is related to the prohibited publishing action via the object property `compensate`.

```
(6) sh:rule [rdf:type sh:SPARQLRule; sh:order 3;
  sh:prefixes[sh:declare
    [sh:prefix "rdf"; sh:namespace "http://..."],
    [sh:prefix "TBox"; sh:namespace "http://..."]];
  sh:construct """
    CONSTRUCT { [rdf:type TBox:Obligatory; rdf:type TBox:Remove;
      TBox:has-agent ?x; TBox:has-theme ?r;
      TBox:compensate $this]. }
    WHERE { $this rdf:type TBox:Reexist. $this TBox:has-agent ?x.
      ?x rdf:type TBox:Licensee. $this TBox:has-theme ?r.
      ?r rdf:type TBox:Result. ?ev TBox:has-result ?r.
      ?ev rdf:type TBox:Evaluate. ?ev rdf:type TBox:Reexist.
      NOT EXISTS{$this rdf:type TBox:ExceptionArt2b}.
      NOT EXISTS{$this rdf:type TBox:ExceptionArt4a. }""";
```

The rule that asserts publishing actions as individuals of the class `ExceptionArt2b` is very similar to (4) so we have omitted it in this paper.

The rule that asserts publishing actions as individuals of the class `ExceptionArt4a` is shown in (7). If there is a commission action to evaluate the product that really exists, then the publishing of the evaluation’s result is obligatory and it counts as `ExceptionArt4a`, so that rules (5) and (6) are blocked.

```
(7) sh:rule [rdf:type sh:SPARQLRule; sh:order 0;
  sh:prefixes[sh:declare
    [sh:prefix "rdf"; sh:namespace "http://..."],
    [sh:prefix "TBox"; sh:namespace "http://..."]];
  sh:construct """
    CONSTRUCT { $this rdf:type TBox:Obligatory.
      $this rdf:type TBox:ExceptionArt4a. }
    WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
      $this TBox:has-theme ?r. ?r rdf:type TBox:Result.
      ?ev TBox:has-result ?r. ?ev rdf:type TBox:Evaluate.
      ?ev rdf:type TBox:Reexist. ?ec TBox:has-theme ?ev.
      ?ec rdf:type TBox:Commission. ?ec rdf:type TBox:Reexist. }""";
```

Finally, the SHACL rule that implements Article 3a is shown in (8). The rule states that publishing comments of evaluations is prohibited unless `ExceptionArt3b` holds.

```
(8) sh:rule [rdf:type sh:SPARQLRule; sh:order 2;
sh:prefixes[sh:declare
  [sh:prefix "rdf"; sh:namespace "http://..."],
  [sh:prefix "TBox"; sh:namespace "http://..."]];
sh:construct ""
CONSTRUCT { $this rdf:type TBox:Prohibited. }
WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
  $this TBox:has-theme ?c. ?c rdf:type TBox:Comment.
  ?c TBox:is-comment-of ?ev. ?ev rdf:type TBox:Evaluate.
  ?ev rdf:type TBox:Rexist.
  NOT EXISTS{$this rdf:type TBox:ExceptionArt3b. }""};
```

ExceptionArt3b is entailed by the rule (9), which implements Article 3b. The rule states that publishing comments of evaluation is permitted if there is another publishing action ?epr that is permitted and whose theme is the result of the same evaluation.

```
(9) sh:rule [rdf:type sh:SPARQLRule; sh:order 1;
sh:prefixes[sh:declare
  [sh:prefix "rdf"; sh:namespace "http://..."],
  [sh:prefix "TBox"; sh:namespace "http://..."]];
sh:construct ""
CONSTRUCT { $this rdf:type TBox:Permitted.
  $this rdf:type TBox:ExceptionArt3b. }
WHERE { $this TBox:has-agent ?x. ?x rdf:type TBox:Licensee.
  $this TBox:has-theme ?c. ?c rdf:type TBox:Comment.
  ?c TBox:is-comment-of ?ev. ?ev rdf:type TBox:Evaluate.
  ?ev rdf:type TBox:Rexist. ?ev TBox:has-result ?r.
  ?epr TBox:has-agent ?x. ?epr TBox:has-theme ?r.
  ?epr rdf:type TBox:Publish. ?epr rdf:type TBox:Permitted.}""};
```

4.2.2 Compliance checking via SHACL rules

Once the SHACL rules representing the regulative norms of our use case have been executed, some actions might have been inferred as either *Obligatory* or *Prohibited*.

Compliance checking is enforced by additional SHACL rules. This checks that *all* instances of *Obligatory* also belong to *Rexist* and that *no* instance of *Prohibited* belongs to *Rexist*. In other words, a violation may be inferred either when an action is obligatory but it does *not* take place in the state of affair or when an action is prohibited but it *does* take place in the state of affair.

However, not all these instances truly correspond to violations: some of them could have been compensated. Therefore, the rule for compliance checking, shown in (10), will also check that the obligatory or prohibited action is *not* related through the *compensate* object property to another action belonging to the class *Rexist*. In order to infer whether the compensatory obligation really exists or not, we will execute a third rule, shown below in (11). The latter must be of course executed before the ones checking the obligations and prohibitions, thus it will have higher priority.

```
(10)  sh:rule [rdf:type sh:SPARQLRule; sh:order 1;
sh:prefixes[sh:declare
  [sh:prefix "rdf"; sh:namespace "http://..."],
  [sh:prefix "TBox"; sh:namespace "http://..."]];
sh:construct """
CONSTRUCT { [rdf:type TBox:Violation; TBox:refer-to $this]. }
WHERE { { $this rdf:type TBox:Obligatory.
        NOT EXISTS{$this rdf:type TBox:Rexist}}
        UNION
        { $this rdf:type TBox:Prohibited.
          $this rdf:type TBox:Rexist. }
        NOT EXISTS{?c TBox:compensate $this.
                  ?c rdf:type TBox:Rexist} } """;
```

In other words, the rule in (11) “solves” the existential quantification represented by the anonymous individual by searching for an action with the same type and the same thematic roles that really exists in the model. If this action is found, the anonymous individual is also asserted as really existing.¹²

```
(11)  sh:rule[rdf:type sh:SPARQLRule; sh:order 0;
sh:prefixes[sh:declare
  [sh:prefix "rdf"; sh:namespace "http://..."],
  [sh:prefix "rdfs"; sh:namespace "http://..."],
  [sh:prefix "TBox"; sh:namespace "http://..."]];
sh:construct """
CONSTRUCT { $this rdf:type TBox:Rexist. }
WHERE { $this TBox:compensate ?a. $this rdf:type ?class.
        ?class rdfs:subClassOf TBox:Action. ?er rdf:type ?class.
        $this TBox:has-agent ?x. $this TBox:has-theme ?r.
        ?er TBox:has-agent ?x. ?er TBox:has-theme ?r.
        ?er rdf:type TBox:Rexist. } """;
```

4.3 Implementing the use case in ASP

In this subsection, we formalize the if-then rules in (2) as Answer Set Programming (ASP) rules. Contrary to SHACL rules, ASP rules cannot be directly executed on RDF triples. These triples will need to be converted into input ASP facts (see Sect. 5 below).

ASP is a widely used logical language for knowledge representation and reasoning. The ASP programming methodology was originally introduced in Gelfond and Lifschitz (1991). Since then, the scientific community have been carrying out a significant amount of work that made ASP one of the most popular formalisms for AI, even at the industrial level (Leone and Ricca 2015; Calimeri et al. 2019; Reale et al. 2022).

ASP is a purely declarative formalism based on (if-then) rules. A set of ASP rules defines an ASP program; typically, a given computational problem is solved via ASP by building a declarative logic program whose intended models, called *answer sets*, correspond one-to-one to the solution of the problem at hand.

¹² In RDF, it is possible to have two individuals referring to the same action because RDF does not observe the Unique Name Assumption. Thus, the anonymous individual representing the compensatory norm is matched to the one quantified by `?er` in (11): they refer to the same action. If the ABox would be encoded in OWL, the rule in (11) should also assert that the two individuals are connected through the `owl:sameAs` relation (see <https://www.w3.org/TR/owl-ref/#sameAs-def>).

The research in ASP has led over the decades to the definition of a variety of ASP “dialects”, supported by corresponding ASP reasoners. The scientific community recently agreed on the definition of a standard input language for ASP systems: ASP-Core-2 (Calimeri et al. 2020b). ASP-Core-2 is also the official language of the ASP Competition series (Calimeri et al. 2016). This paper will also use ASP-Core-2.

Since ASP is purely *declarative* in nature, the order of the rules in a program is irrelevant; also, the basic language does not provide an explicit means for expressing priority among rules, even though this can be accomplished by means of other language features. Knowledge is just additive, and the ASP reasoner solves a program by searching for answer sets that satisfy all rules at once. This is different from the SHACL rules seen above, in which we had to establish priorities among the rules because once new triples are asserted in the ABox, these cannot be deleted by other SHACL rules.

4.3.1 Representing the norms as ASP rules

The ASP rules corresponding to the use case in (2) involve unary and binary first-order predicates corresponding 1:1 to the classes and object properties of the ontology.

The ASP rule encoding Article 1(a) in (2), which states that the Licensee is prohibited to evaluate the Product unless `exceptionArt1b` holds, is shown in (12).

```
(12) prohibited(Ev) :- evaluate(Ev), hasAgent(Ev,X), licensee(X),
                        hasTheme(Ev,P), product(P), not exceptionArt1b(Ev).
```

(12) parallels the SHACL rule in (3); “not” is the ASP operator for negation-as-failure.

`exceptionArt1b(Ev)` holds if the agent of `Ev` is granted a licence to evaluate the product. In such a case, the evaluation is permitted. In SHACL this was modelled with a rule, shown in (4) above, that entails both `exceptionArt1b(Ev)` and `permitted(Ev)`. The basic ASP language does not support conjunctions of literals in rule heads; hence, in order to model such situations the typical approach consists of introducing a specific rule to define the condition, and then using it as antecedent of more than one rule. Article 1(b) in (2) is then modelled as in (13).

```
(13) condition1(Ev) :- evaluate(Ev), hasAgent(Ev,X), licensee(X),
                        hasTheme(Ev,P), product(P), isLicenceOf(L,P),
                        licence(L), grant(Eg), rexist(Eg), hasTheme(Eg,L),
                        hasAgent(Eg,Y), licensor(Y), hasReceiver(Eg,X).

exceptionArt1b(Ev) :- condition1(Ev).

permitted(Ev) :- condition1(Ev).
```

Since the predication `condition1(Ev)` is only used in these three if-then rules, the first if-then rule in (13) is logically equivalent to a bi-implication, i.e., a definition.

Article 2 of the use case specifies both a prohibition and a compensatory obligation. Licensees are prohibited to publish the result of an evaluation unless this was approved by the licensor (first exception) or unless they were commissioned to perform an independent evaluation (second exception). Licensees who violate this prohibition are obliged to remove the published material. The rules in (14) define the prohibition described in Article 2a. They parallel the SHACL rule in (5). The two mentioned exceptions are represented by the predicates `exceptionArt2b` and `exceptionArt4a`

respectively. We omit the ASP rules that entail `exceptionArt2b` because they are very similar to (13). On the other hand, the ones that entail `exceptionArt4a` will be shown below in (19).

```
(14) condition2(Ep, X, R) :- publish(Ep), hasAgent(Ep, X), licensee(X),
                               hasTheme(Ep, R), result(R), hasResult(Ev, R),
                               evaluate(Ev), reXist(Ev),
                               not exceptionArt2b(Ep), not exceptionArt4a(Ep).

    prohibited(Ep) :- condition2(Ep, X, R).
```

In order to model the compensatory obligation in Article 2c, we must introduce a set of ASP rules that allow us to derive the same knowledge expressed by the following first-order logic well-formed formula:

```
(15)  $\forall Ep \forall X \forall R [ (\text{reXist}(Ep) \wedge \text{condition2}(Ep, X, R)) \rightarrow$ 
       $\exists Y [ \text{obligatory}(Y) \wedge \text{remove}(Y) \wedge \text{hasAgent}(Y, X) \wedge$ 
       $\text{hasTheme}(Y, R) \wedge \text{compensate}(Y, Ep) ] ]$ 
```

In SHACL, we chose to implement the existential quantification by introducing an anonymous individual, which is then possibly matched with a `remove` action having `X` as an agent and `R` as a theme during the compliance checking phase (cf. Sect. 4.2.2 above).

This solution cannot be implemented in ASP because the language does not allow the introduction of anonymous individuals. In ASP, we alternatively replace the existential quantifier “ $\exists Y$ ” with a function symbol “`ca`” (as for “compensatory action”) over the universally quantified variables `Ep`, `X`, and `R`. In other words, we *Skolemize* the formula in (15) into the following one:

```
(16)  $\forall Ep \forall X \forall R [ (\text{reXist}(Ep) \wedge \text{condition2}(Ep, X, R)) \rightarrow$ 
       $(\text{obligatory}(\text{ca}(Ep, X, R)) \wedge \text{remove}(\text{ca}(Ep, X, R))) \wedge$ 
       $\text{hasAgent}(\text{ca}(Ep, X, R), X) \wedge \text{hasTheme}(\text{ca}(Ep, X, R), R) \wedge$ 
       $\text{compensate}(\text{ca}(Ep, X, R), Ep) ]$ 
```

The first-order logic formula in (16) can be directly implemented in ASP, as follows:

```
(17) obligatory(ca(Ep, X, R)) :- reXist(Ep), condition2(Ep, X, R).
    remove(ca(Ep, X, R)) :- reXist(Ep), condition2(Ep, X, R).
    hasAgent(ca(Ep, X, R), X) :- reXist(Ep), condition2(Ep, X, R).
    hasTheme(ca(Ep, X, R), R) :- reXist(Ep), condition2(Ep, X, R).
    compensate(ca(Ep, X, R), Ep) :- reXist(Ep), condition2(Ep, X, R).
```

ASP enriched with functional symbols is Turing-complete in the general case and so undecidable (Calimeri et al. 2009), even if there are some subclasses of programs whose answer sets are finite and computable (Calautti et al. 2017). In particular, ASP programs featuring functions in recursion can have a non-finite relevant ground program; intuitively, this can cause the so-called *value invention* when both domains and co-domains are asserted on the same predicates, e.g., “ $\text{p}(\text{f}(x)) : \neg \text{p}(x)$ ”. However, in our modeling such recursive patterns cannot occur. We use functional symbols only to represent sequences of compensatory norms. In existing legislation, these sequences are rather limited in size and, most important of all, do not contain cycles; a cycle would

in fact correspond to a compensatory norm that compensates the violation of itself, which would not make much sense in real-world scenarios. Therefore, in our ASP formalization the aforementioned recursive patterns cannot occur.¹³

Article 3 of the use case in (2) defines the prohibition to publish comments on the evaluation of the product unless the licensee is allowed to publish the results of the evaluation of the product. The rules encoding the prohibition described in Article 3a in cases where the exception described in Article 3b does not occur are below. The rules in (18) parallel the SHACL rules shown in (8) and (9) above.

```
(18) prohibited(Ep) :- publish(Ep), hasAgent(Ep, X), licensee(X),
                        hasTheme(Ep, C), comment(C), isCommentOf(C, Ev),
                        evaluate(Ev), reXist(Ev), not exceptionArt3b(Ep).

condition4(Ep) :- publish(Ep), hasAgent(Ep, X), licensee(X),
                  hasTheme(Ep, C), comment(C), isCommentOf(C, Ev),
                  evaluate(Ev), reXist(Ev), hasResult(Ev, R),
                  hasTheme(Epr, R), hasAgent(Epr, X), publish(Epr),
                  permitted(Epr).

exceptionArt3b(Ep) :- condition4(Ep).

permitted(Ep) :- condition4(Ep).
```

Finally, Article 4 establishes the obligation to publish the results of the evaluation of a product in cases where the evaluation was commissioned, and thus an exception to Article 2a. The ASP rules in (19) parallels the SHACL rule in (7).

```
(19) condition5(Ep) :- publish(Ep), hasAgent(Ep, X), licensee(X),
                        hasTheme(Ep, R), result(R), hasResult(Ev, R),
                        evaluate(Ev), reXist(Ev), hasTheme(Ec, Ev),
                        commission(Ec), reXist(Ec).

exceptionArt4a(Ep) :- condition5(Ep).

obligatory(Ep) :- condition5(Ep).
```

4.3.2 Compliance checking via ASP rules

The ASP rules shown in the previous subsection infer which actions are either prohibited or obligatory. Further ASP rules, which parallel the SHACL rules shown above in Sect. 4.2.2, are then needed to infer the violations occurring in the state of affairs.

We remind the reader that a violation occurs either in cases where an action is performed even if prohibited or in cases where an action is not performed even if obligatory. However, in both cases, if the action is associated with a compensatory obligation and the latter was performed, the former does not indeed trigger any violation.

¹³ These ASP patterns will specifically parallel the items in the LegalRuleML tag <SuborderList> mentioned above in Sect. 2.1, which in turn parallel the operator “ \otimes ” from Governatori and Rotolo (2006) mentioned above in Sect. 3. <SuborderList> and “ \otimes ” represent ordered finite lists with no duplicated elements.

The ASP rules in (20), which parallel the SHACL rule shown above in (10), are able to carry out the desired inferences.

```
(20) compensated(X) :- compensate(Y, X), reXist(Y).
    violation(viol(X)) :- obligatory(X), not reXist(X), not compensated(X).
    violation(viol(X)) :- prohibited(X), reXist(X), not compensated(X).
    referTo(viol(X), X) :- violation(viol(X)).
```

Finally, we add the ASP rule in (21) in order to intercept the occurrence in the state of affairs of a removal action that has the properties required by the removal action denoted by $ca(Ep, X, R)$ in (17). The rule in (21) parallels the SHACL rule shown above in (11) and is needed to solve the existential quantification, represented as a Skolemized functional symbol in ASP and as an anonymous individual in RDF/SHACL.

```
(21) reXist(ca(Ep, X, R)) :- remove(ca(Ep, X, R)), hasTheme(ca(Ep, X, R), R),
    hasAgent(ca(Ep, X, R), X), reXist(Er), remove(Er),
    hasTheme(Er, R), hasAgent(Er, X).
```

4.4 Implementing the use case in DLV

Buccafurri et al. (2002) proposed an extension of standard Answer Set Programming that allows rules to be organized into *inheritance networks*. The extension is implemented by the DLV reasoner (Leone et al. 2006).

Inheritance networks resemble SPINdle's superiority rules, discussed above in Sect. 3. The input format of the DLV reasoner allows ASP rules to be clustered within curly brackets "{...}". Each cluster is labelled either with a single label "1" or with a double label "1₁:1₂", meaning that in cases where a rule in the cluster of "1₁" conflicts with a rule in the cluster of "1₂", the former prevails and the latter is overridden. Apart from the use of inheritance networks in place of the special predicates `exceptionArt1b`, `exceptionArt4a`, etc., the formalization is the same as the one presented in the previous subsection.

4.4.1 Representing the norms as DLV rules

The DLV rules encoding Article 1(a) and Article 1(b) from (2) are shown in (22).

```
(22) art1a{ prohibited(Ev) :- evaluate(Ev), hasAgent(Ev, X), licensee(X),
    hasTheme(Ev, P), product(P). }

art1b:art1a{ condition1(Ev) :- evaluate(Ev), hasAgent(Ev, X),
    licensee(X), hasTheme(Ev, P), product(P),
    isLicenceOf(L, P), licence(L), grant(Eg), reXist(Eg),
    hasTheme(Eg, L), hasAgent(Eg, Y), licensor(Y),
    hasReceiver(Eg, X).

    -prohibited(Ev) :- condition1(Ev).

    permitted(Ev) :- condition1(Ev). }
```

The ASP rule in (12), but the literal `not exceptionArt1b(Ev)`, has been inserted within a cluster with the label "art1a" On the other hand, the ASP rules

in (13) has been inserted within a second cluster with the label “art1b:art1a”; the consequent of the second rule in (13) has been modified: rather than entailing `exceptionArt1b(Ev)`, the rule entails `-prohibited(Ev)`, i.e., the negation of the consequent of the (single) rule in the cluster with the label “art1a”. Finally, the label “art1b:art1a” specifies that the rules in the second cluster are stronger than the rules in the first one. Thus, in cases where all rules in (22) are triggered, `prohibited(Ev)` is overridden by `-prohibited(Ev)`.

Article 2a and Article 2c from (2) are implemented via the cluster with label “art2a” shown in (23). This corresponds to the ASP rules shown in (14) and (17) above.

```
(23) art2a{ condition2(Ep, X, R) :- publish(Ep), hasAgent(Ep, X),
                                licensee(X), hasTheme(Ep, R), result(R),
                                hasResult(Ev, R), evaluate(Ev), reexist(Ev).

    prohibited(Ep) :- condition2(Ep, X, R).

    obligatory(ca(Ep,X,R)) :- reexist(Ep), condition2(Ep,X,R).

    remove(ca(Ep,X,R)) :- reexist(Ep), condition2(Ep,X,R).

    hasAgent(ca(Ep,X,R),X) :- reexist(Ep), condition2(Ep,X,R).

    hasTheme(ca(Ep,X,R),R) :- reexist(Ep), condition2(Ep,X,R).

    compensate(ca(Ep,X,R),Ep) :- reexist(Ep), condition2(Ep,X,R). }
```

The rules in (23) state that publishing the results of existing evaluations is prohibited; if these results are published despite the prohibition, it is obligatory to remove them.

Two articles from (2) override the prohibition of publishing the results and, consequently, the obligation to remove them: Article 2b, stating that the publishing is permitted if approved by the licensor, and Article 4a, stating that the publishing is obligatory if the evaluation has been commissioned. These are represented as in (24) and (25).

```
(24) art2b:art2a{ condition3(Ep,X,R) :- publish(Ep), hasAgent(Ep,X),
                                licensee(X), hasTheme(Ep,R), result(R), hasResult(Ev,R),
                                evaluate(Ev), reexist(Ev), hasTheme(Ea,Ep), approve(Ea),
                                reexist(Ea), hasAgent(Ea,Y), licensor(Y).

    -condition2(Ep,X,R) :- condition3(Ep,X,R).

    permitted(Ep) :- condition3(Ep,X,R). }
```

```
(25) art4a:art2a{ condition5(Ep,X,R) :- publish(Ep), hasAgent(Ep,X),
                                licensee(X), hasTheme(Ep,R), result(R), hasResult(Ev,R),
                                evaluate(Ev), reexist(Ev), hasTheme(Ec,Ev),
                                commission(Ec), reexist(Ec).

    -condition2(Ep,X,R) :- condition5(Ep,X,R).

    obligatory(Ep) :- condition5(Ep,X,R). }
```

In case the second rule in (24) or the second rule in (25) is triggered, `condition2(Ep,X,R)` is overridden by its negation and so all rules in (23) are blocked.

The rules corresponding to Article 3 are shown in (26); these parallel the ASP rules shown above in (18). In cases where the publishing of the results is permitted, the

publishing of the comments is not prohibited, i.e., `-prohibited(Ep)` is asserted. This overrides the opposite literal from the rule in the cluster with the label “art3a”.

Finally, the rules for compliance checking are the same as those shown above in Sect. 4.3.2, therefore we will not repeat them here.

```
(26) art3a{ prohibited(Ep):- publish(Ep), hasAgent(Ep,X), licensee(X),
                           hasTheme(Ep,C), comment(C), evaluate(Ev),
                           reexist(Ev), isCommentOf(C,Ev). }

    art3b:art3a{ condition4(Ep):- publish(Ep), licensee(X), comment(C),
                                hasAgent(Ep,X), hasTheme(Ep,C), isCommentOf(C,Ev),
                                evaluate(Ev), reexist(Ev), hasResult(Ev,R),
                                hasTheme(Epr,R), hasAgent(Epr,X), publish(Epr),
                                permitted(Epr).

    -prohibited(Ep):- condition4(Ep).

    permitted(Ep):- condition4(Ep). }
```

4.5 Implementing the use case in PROLEG

PROLEG (PROlog based LEGal reasoning support system) (Satoh et al. 2011) is a legal reasoning system based on standard Prolog.

Many other Prolog-based systems for defeasible reasoning have been investigated for decades, e.g., d-Prolog (Nute 1988). However, PROLEG is, to our knowledge, the only one that specifically aims at modelling lawyers’ and judges’ reasoning; see, e.g., Fungwacharakorn et al. (2020), Satoh et al. (2021).

While compliance checking with conflicting norms in PROLEG has been widely studied in past literature, this paper represents the first attempt to also incorporate compensatory norms therein, thus it contributes to the literature of the reasoner in this respect.

PROLEG is able to provide human-understandable explanations of its derivations. These are usually provided as a narrative that simulates a hypothetical dispute between a “plaintiff” and a “defendant” (Satoh et al. 2009). The narrative starts with the plaintiff who tries to build a burden of proof for a given allegation against the defendant by using the rules from legislation and known facts. The defendant tries to provide defenses against the plaintiff’s burden of proof, each of which is in turn associated with another burden of proof. If proved, these defenses constitute *exceptions* to the conclusions achieved by the plaintiff. These are modelled in terms of an operator “exception” that parallels SPINdle’s superiority relations and DLV’s inheritance networks.

Another crucial feature of PROLEG, which is not shared by any of the other reasoners considered here, is the neat distinction between known (input) facts and inferred (output) facts; these correspond to two different sets of predicates (Satoh et al. 2011).

4.5.1 Representing the norms as PROLEG rules

PROLEG is available as a Prolog library that may be imported within a standard Prolog program. The library defines additional Prolog operators and meta-predicates, e.g., the operator “`<=`”, used to model defeasible inferences, and the aforementioned

operator “exception”, used to state when a conclusion overrides another one. As mentioned above, PROLEG distinguishes between input facts and inferred facts; in the formalization below, the former is characterized by a suffix “_f” at the end of the predicate name.

The PROLEG formalization of Article 1 is shown in (27). (27) correspond to the SHACL rules in (3) and (4), the ASP rules in (12) and (13), and the DLV rules in (22).

```
(27) prohibited(Ev) <= evaluate_f(Ev), hasAgent_f(Ev,X), licensee_f(X),
      hasTheme_f(Ev,P), product_f(P).

permitted(Ev) <= evaluate_f(Ev), hasAgent_f(Ev,X), licensee_f(X),
      hasTheme_f(Ev,P), product_f(P), isLicenceOf_f(L,P),
      licence_f(L), grant_f(Eg), reexist_f(Eg),
      hasTheme_f(Eg,L), hasAgent_f(Eg,Y), licensor_f(Y),
      hasReceiver_f(Eg,X).

exception(prohibited(Ev), permitted(Ev)).
```

All facts in the bodies of the rules in (27) are input facts. When these are included in the description of the state of affairs, we infer that the action *Ev* is prohibited and/or permitted. However, in cases where both are derived, the prohibition is overridden by the permission, due to the exception stated in the last line of (27).

Contrary to SHACL, ASP, and DLV, PROLEG/Prolog is query-based, thus the reasoner additionally takes an input query from the user. PROLEG returns an explanation in natural language of the derivations triggered by the query. Since these explanations are quite verbose, we will omit them here; the reader is invited to directly use the reasoner after downloading the source files from the GitHub repository.

Article 2a-b is represented via the rules in (28); these rules include the intermediate predicate *condition2*(*Ep*, *X*, *R*), used to avoid repeating the antecedents.

```
(28) prohibited(Ep) <= condition2(Ep,X,R).

condition2(Ep,X,R) <= publish_f(Ep), hasAgent_f(Ep,X),
      licensee_f(X), hasTheme_f(Ep,R), result_f(R),
      hasResult_f(Ev,R), evaluate_f(Ev), reexist_f(Ev).

permitted(Ep) <= publish_f(Ep), hasAgent_f(Ep,X), licensee_f(X),
      hasTheme_f(Ep,R), result_f(R), hasResult_f(Ev,R),
      evaluate_f(Ev), reexist_f(Ev), approve_f(Ea),
      reexist_f(Ea), hasTheme_f(Ea,Ep), hasAgent_f(Ea,Y),
      licensor_f(Y).

exception(condition2(Ep,X,R),permitted(Ep)).
```

In cases where *condition2*(*Ep*, *X*, *R*) holds and the results have been really published, then the licensee is obliged to remove them. This is formalized as in (29). As in ASP and DLV, the removal action corresponds to a Skolemized functional symbol that replaces the existential quantification.

```
(29) obligatory(ca(Ep,X,R)) <= reexist_f(Ep), condition2(Ep,X,R).

remove(ca(Ep,X,R)) <= reexist_f(Ep), condition2(Ep,X,R).

hasTheme(ca(Ep,X,R),R) <= reexist_f(Ep), condition2(Ep,X,R).

hasAgent(ca(Ep,X,R),X) <= reexist_f(Ep), condition2(Ep,X,R).

compensate(ca(Ep,X,R),Ep) <= reexist_f(Ep), condition2(Ep,X,R).
```

Article 4a, stating that when an evaluation has been commissioned the publication of its results is obligatory, corresponds to a formalization very similar to (28), thus we omit it.

On the other hand, Article 3 is represented as in (27). Note that the body of the second rule in (27) includes the derived predicate `permitted(Epr)` besides the fact predicates describing the publishing of the comments. In (27), in cases where the publishing of the results is permitted by the third rule in (28) above, so is the publishing of the comments.

```
(30) prohibited(Ep) <= publish_f(Ep), hasAgent_f(Ep,X), licensee_f(X),
      hasTheme_f(Ep,C), comment_f(C), evaluate_f(Ev),
      reexist_f(Ev), isCommentOf_f(C,Ev).

      permitted(Ep) <= publish_f(Ep), hasAgent_f(Ep,X), licensee_f(X),
      hasTheme_f(Ep,C), comment_f(C), isCommentOf_f(C,Ev),
      evaluate_f(Ev), reexist_f(Ev), hasResult_f(Ev,R),
      publish_f(Epr), hasAgent_f(Epr,X),
      hasTheme_f(Epr,R), permitted(Epr).

      exception(prohibited(Ep),permitted(Ep)).
```

4.5.2 Compliance checking via PROLEG rules

Since PROLEG uses exceptions in place of negation-as-failure to model defeasibility, the rules to infer whether the obligations and prohibitions holding in the state of affairs have been violated or not are different from the ones in SHACL and ASP seen above.

Given an obligatory action, we derive that it has been violated by checking that it does *not* really exist, i.e., that the `reexist` predicate has not been asserted on that action. In PROLEG, this is achieved by stating that an obligatory action entails a violation *by default*; however, if the action really exists that default violation is overridden. The formulae in (31) represent the desired inferences.

```
(31) violation(viol(X)) <= obligatory(X).
      -violation(viol(X)) <= obligatory(X), reexist(X).
      exception(violation(X),-violation(X)).
```

On the other hand, we derive that a prohibited action has been violated when it *does* really exist. Therefore, for prohibitions we do not need to use the operator “exception”. We do instead need it to model compensations, which constitute exceptions of violations. The entailments in (32) are added in order to handle prohibitions and compensations.

```
(32) violation(viol(X)) <= prohibited(X), reexist(X).
      compensated(X) <= compensate(Y,X), reexist(Y).
      exception(violation(viol(X)),compensated(X)).
```

Finally, we add an additional PROLEG rule that parallels the SHACL rule in (11) and the ASP rule in (21). This rule, shown in (33), derives that the individual denoted by

the first-order term $ca(Ep, X, R)$ really exists when the removal of the results really took place in the state of affairs.

```
(33)  reexist(ca(Ep,X,R)) <= remove(ca(Ep,X,R)), hasAgent(ca(Ep,X,R),X),
      hasTheme(ca(Ep,X,R),R), hasTheme_f(Er,R),
      hasAgent_f(Er,X), remove_f(Er), reexist_f(Er).
```

The rules in (31), (32), and (33) are used to detect when violations occur. In order to do so, we query the reasoner by asking whether the predicate `violation(viol(a))` holds true, where “a” is a possible prohibited or obligatory action. If so, the explanation returned by PROLEG details the reasons why the action has been violated.

4.6 Implementing the use case in Arg2P

Several modern approaches to legal reasoning are based on structured argumentation [see Prakken and Sartor (2015)]. The idea is to reflect that defeasibility in the law results from having arguments both for and against the application of the norms.

These approaches provide an extra layer to the representation of the inferences by including therein the graph of the arguments that either support or reject the conclusions.

However, as it has been argued in Billi et al. (2021), such an explicit representation of the arguments might not be so relevant when modeling *static* norms from legislation, as in the use case considered here. Legislation already states which norms override which other ones, i.e., this knowledge is already known a priori and so there is no need to infer it again from the argument graph. On the other hand, argumentation is suitable to model reasoning in judicial proceedings, in which the parties involved can have different arguments in favour of or against the interpretation of the rules. These competing arguments are approved or rejected over time by different courts and authorities, which have different weights. Therefore, in judicial reasoning we need to *dynamically* infer which arguments override which other ones and, consequently, the priorities among the rules.

Although argumentation offers more functionalities than what we need to model our use case, we still decided to consider it in our analysis given the prominent role that it is increasingly assuming in LegalTech.

Three well-known argumentation frameworks used nowadays are DeLP (García and Simari 2004), ABA⁺ (Cyras and Francesca 2016), and ASPIC⁺ (Modgil and Prakken 2014). These frameworks have been implemented in several reasoners available online. In this paper we consider Arg2P (Calegari et al. 2022), a recent implementation of ASPIC⁺.

Arg2P is a lightweight Prolog-based implementation for structured argumentation in compliance with the micro-intelligence definition (Calegari et al. 2022). The research in Arg2P is oriented to identify different features and functionalities offered by available defeasible reasoners and to allow Arg2P’s users to configure the reasoner on the ones they need for the purposes of their projects. See Billi et al. (2021) for a discussion as well as for a comparison between Arg2P and other contemporary defeasible reasoners.

4.6.1 Representing the norms as Arg2P rules

The input format of Arg2P allows the encoding of labelled defeasible inference rules each from a conjunction of premises to a conclusion. Overriding among rules is achieved via superiority relations as in SPINdle or Proleg. On the other hand, as in SPINdle but contrary to Proleg, the input format of Arg2P includes explicit modal operators,¹⁴ “o” and “p”, respectively stating whether an action is obligatory or permitted.

Article 1(a-c) in (2) is formalized via the following formulae, corresponding to the SHACL formulae in (3) and (4), the ASP formulae in (12) and (13), the DLV rules in (22), and the PROLEG rules in (27). The sup predicate implement superiority relations.

```
(34) art1a: evaluate(Ev), hasAgent(Ev,X), licensee(X), hasTheme(Ev,P),
      product(P) => o(-evaluate(Ev)).

      art1b: evaluate(Ev), hasAgent(Ev,X), licensee(X), hasTheme(Ev,P),
      product(P), isLicenceOf(L,P), licence(L), hasTheme(Eg,L),
      hasAgent(Eg,Y), licensor(Y), grant(Eg), rexist(Eg),
      hasReceiver(Eg,X) => p(evaluate(Ev)).

      sup(art1b, art1a).
```

If the licensor grants the licensee a licence, the two rules in (34) derive that the evaluation of the product is both prohibited ($o(-evaluate(Ev))$) and permitted ($p(evaluate(Ev))$); however, as the superiority relation $sup(art1b, art1a)$ states that the rule with label art1b is stronger than the rule with label art1a, only $p(evaluate(Ev))$ is inferred.

The if-then rules encoding the prohibition described in Article 2a are shown in (35):

```
(35) art2aPart1: evaluate(Ev), rexist(Ev), hasResult(Ev,R), result(R),
      publish(Ep), hasAgent(Ep,X), licensee(X), hasTheme(Ep,R)
      => condition2(Ep,X,R).

      art2aPart2: condition2(Ep,X,R) => o(-publish(Ep)).
```

The Arg2P rule implementing the obligations from Article 4a is shown in (36):

```
(36) art4a: publish(Ep), hasAgent(Ep,X), licensee(X), hasTheme(Ep,R),
      result(R), hasResult(Ev,R), evaluate(Ev), rexist(Ev),
      hasTheme(Ec,Ev), commission(Ec), rexist(Ec) => o(publish(Ep)).

      sup(art4a, art2aPart2).
```

The superiority relation in (36) blocks the inference of the first rule in (35) in cases where the evaluation of the product has been commissioned and so the publishing of the evaluation’s results is obligatory. A similar rule corresponding to Article 2b, which we omit in this paper, blocks the inference of the first rule in (35) in cases where the licensor approved the publishing of the evaluation’s results and so this is actually permitted.

In order to represent Article 2c and Article 2e of the use case in (2), we introduce a set of rules analogues to the ones defined for the ASP case in (17) above: when multiple

¹⁴ See <https://pika-lab.gitlab.io/argumentation/arg2p-kt/wiki/syntax>.

literals are inferred by a specific piece of knowledge, multiple rules are added, having the same antecedent which defines that knowledge.

However, note that there is a crucial difference between the ASP rules in (17) and the Arg2P rules in (37). The shared antecedent of the latter includes the three literals $\text{condition2}(\text{Ep}, X, R)$, $\text{o}(\text{-publish}(\text{Ep}))$, and $\text{rexist}(\text{Ep})$, while the shared antecedent of the former only the two literals $\text{condition2}(\text{Ep}, X, R)$ and $\text{rexist}(\text{Ep})$.

The reason is that both formalizations need $\text{condition2}(\text{Ep}, X, R)$ in the shared antecedents because the corresponding consequents are asserted on the variables Ep, X, and R. However, in the case of ASP, this literal is enough also to implement the overriding of the rules in (17) because the predicates $\text{exceptionArt2b}(\text{Ep})$ and $\text{exceptionArt4a}(\text{Ep})$ block this literal: $\text{condition2}(\text{Ep}, X, R)$ is true only if the two exceptions are negated by failure. On the contrary, in Arg2P the superiority relations that replace the predicates $\text{exceptionArt2b}(\text{Ep})$ and $\text{exceptionArt4a}(\text{Ep})$, e.g., $\text{sup}(\text{art4a}, \text{art2aPart2})$ in (36), block the literal $\text{o}(\text{-publish}(\text{Ep}))$. Thus, in order to achieve the same truth conditions, we must add this literal to the antecedent of the Arg2P rules in (37).

```
(37) art2cPart1: condition2(Ep, X, R), o(-publish(Ep)), rexi
    => o(remove(ca(Ep, X, R))).
    art2cPart2: condition2(Ep, X, R), o(-publish(Ep)), rexi
    => remove(ca(Ep, X, R)).
    art2cPart3: condition2(Ep, X, R), o(-publish(Ep)), rexi
    => hasTheme(ca(Ep, X, R), R).
    art2cPart4: condition2(Ep, X, R), o(-publish(Ep)), rexi
    => hasAgent(ca(Ep, X, R), X).
    art2e: condition2(Ep, X, R), o(-publish(Ep)), rexi
    => compensate(ca(Ep, X, R), Ep).
```

Article 3 is formalized as in (38). These formulae correspond to the SHACL rules in (8) and (9), the ASP rules in (18), the DLV rules in (26), and the PROLEG rules in (27). The publishing of the comments is permitted in case the publishing of the results is. To achieve this, $\text{p}(\text{publish}(\text{Epr}))$ is added to the antecedent of the rule with label `art3b`.

```
(38) art3a: publish(Ep), hasAgent(Ep, X), licensee(X), hasTheme(Ep, C),
    comment(C), isCommentOf(C, Ev), evaluate(Ev), rexi
    => o(-publish(Ep)).
    art3b: publish(Ep), hasAgent(Ep, X), licensee(X), hasTheme(Ep, C),
    comment(C), isCommentOf(C, Ev), evaluate(Ev), rexi
    hasResult(Ev, R), hasTheme(Epr, R), hasAgent(Epr, X),
    publish(Epr), p(publish(Epr)) => p(publish(Ep)).
    sup(art3b, art3a).
```

4.6.2 Compliance checking via Arg2P rules

Arg2P represents obligatory, prohibited, and permitted actions via two modal operators “o” and “p”. Contrary to SHACL, ASP, and Proleg, which use explicit predicates obligatory, prohibited and permitted, modal operators such as “o” and

“p” prevents us from defining *general* Arg2P rules to detect violations of obligations and prohibitions.

To see why, consider the ASP rule from (20) above to detect violations of prohibitions. The rule is copied again in (39) for reader’s convenience. The rule infers a violation `viol(X)` for *any* prohibited action `X` that really exists and has not been compensated.

```
(39) violation(viol(X)) :- prohibited(X), reXist(X), not compensated(X).
```

In Arg2P, the literal `prohibited` is replaced by the modal operator “o” applied to the negation of an action. Since Arg2P’s input format does not allow to quantify over the predicates outscoped by “o”, we must assert a different rule for *each* action that may be prohibited. In our use case, two actions may be prohibited: the evaluation of the product and the publishing of either its results or comments about it. Each of these two actions is associated with a different Arg2P rule that detects the violation of its prohibition. The two rules are shown in (40). “~” is the Arg2P operator for negation-as-failure; it parallels the ASP operator “not” in (39).

```
(40) ccRuleEv: o(-evaluate(Ev)), reXist(Ev), ~(compensated(Ev))
      => violation(viol(Ev)).

      ccRuleEp1: o(-publish(Ep)), reXist(Ep), ~(compensated(Ep))
      => violation(viol(Ep)).
```

On the other hand, in our use case there are two actions that may be obligatory: the publishing of the results, which is obligatory in cases where the evaluation has been commissioned (see rule (36) above), and the removal of the results, which is obligatory in cases where the licensee publishes the results although they were not allowed to do so (see rule (37) above). Again, we need to define a *specific* rule for each of these two actions, as shown in (41). By contrast, in SHACL, ASP, and Proleg, a single rule is needed.

```
(41) ccRuleEp2: o(publish(Ep)), ~(reXist(Ep)), ~(compensated(Ep))
      => violation(viol(Ep)).

      ccRuleEr: o(remove(Er)), ~(reXist(Er)), ~(compensated(Er))
      => violation(viol(Er)).
```

Finally, we need rules to infer when `ca(Ep, X, R)` really exists and, consequently, when the prohibited publishing has been compensated. These rules, which parallel the SHACL rule in (11), the ASP rules in (21), and the PROLEG rules in (32) and (33), are:

```
(42) ccRuleComp1: remove(ca(Ep, X, R)), hasTheme(ca(Ep, X, R), R),
                  hasAgent(ca(Ep, X, R), X), reXist(Er), remove(Er),
                  hasTheme(Er, R), hasAgent(Er, X) => reXist(ca(Ep, X, R)).

      ccRuleComp2: compensate(Y, X), reXist(Y) => compensated(X).
```

4.7 Implementing the use case in SPINdle, after grounding

As stated earlier, although SPINdle is propositional, we decided to include it in our investigation because it is widely acknowledged in the literature on legal reasoning.

In order to use SPINdle with first-order rules, these must be first *grounded*, i.e., transformed into the set of propositional counterparts obtained by replacing the free variables with all possible combinations of individuals in the state of affairs.

We note that grounding is also implicitly implemented within traditional ASP reasoners (Faber et al. 2012; Kaufmann et al. 2016) and lot of research has been done in the last decades in order to improve the performances of the “ground&solve” approach used within those reasoners, a recent contribution being (Cuteri et al. 2020). Still, the input language of modern ASP reasoners is *first-order*: grounding is simply part of the *internal* implementation to reason with first-order formulae.

This is different from SPINdle that, on the contrary, only accepts *propositional* formulae in input. Therefore, first-order formulae such as the one considered in the previous sections must be first *externally* grounded into the SPINdle propositional input format. These grounded propositional formulae might be then processed via the reasoner.

4.7.1 Representing the norms as SPINdle rules

This subsection will present a (propositional) formalization in SPINdle of the if-then rules in (2). The dataset generator discussed below in Sect. 5.1 will create further versions of these rules by indexing them with respect to the individuals in the synthetic states of affairs. In other words, the grounding of the SPINdle input formulae will be (externally) carried out by the dataset generator, before invoking the reasoner.

We formalize Article 1(a-c) in (2) via the following SPINdle’s input formulae, which correspond to the SHACL formulae in (3) and (4), the ASP formulae in (12) and (13), the DLV rules in (22), the PROLEG rules in (27), and the Arg2P formulae in (34).

```
(43) Art1a[O]: => -Evaluate_ev
      Art1b[P]: Grant_eg, Rexist_eg => Evaluate_ev
      Art1b > Art1a
```

The first line in (43) is an if-then rule with empty antecedent, meaning that the antecedent is always true. The if-then rule asserts the literal [O]-Evaluate_ev, where “[O]” is the deontic operator applied by the rule to the consequent; in SPINdle input format, the deontic operator is specified before the “:” and after the label of the rule, which is Art1a in this case. On the other hand, “-” is the symbol for the negation while Evaluate_ev is a *propositional* symbol, denoting the whole proposition. This includes both the main action and its thematic roles. We chose to use a more compact propositional symbol Evaluate_ev rather than a more verbose one that encompasses all thematic roles, e.g., the propositional symbol Evaluate_ev_hasAgent_x_licensee_x_hasTheme_p_product_p.

The second rule in (43) asserts the literal [P]Evaluate_ev in cases where the rule’s antecedent is true. In this rule, the antecedent is not empty but it corresponds to a conjunction of two propositional symbols that must both hold true in the state of affairs. Grant_eg refers to the whole proposition “The licensor grants the licensee a licence to evaluate the product” while Rexist_eg states that this granting action really exists. In our logical account, Rexist_eg represents a modality alternative

to the three deontic ones, represented in SPINdle as “[O]” (obligation), “[O]–” (prohibition) and “[P]” (permission), so it must be asserted separately from the whole proposition it applies to.

Finally, the last line in (43) is a superiority relation to solve the conflict between the two previous if-then rules, since the two literals [O]-Evaluate_ev and [P]Evaluate_ev cannot be asserted together as they express opposite deontic modalities. The superiority relation uses the labels of the two rules to state that in cases where both rules are triggered, only the literal [P]Evaluate_ev must be asserted.

On the other hand, Article 2(a-e) in (2) are formalized as follows:

```
(44) Art2a[O]: Evaluate_ev, Rexist_ev => -Publish_epr
      Art2b[P]: Approve_ea, Rexist_ea => Publish_epr
      Art2b > Art2a
      Art2c[O]: [O]-Publish_epr, Rexist_epr => Remove_er
      Art2e: [O]-Publish_epr, Rexist_epr => compensate_er_epr
```

The first rule in (44) states that if the evaluation really exists, then the publishing of its result is prohibited, i.e., that [O]-Publish_epr holds true in the context. This entailment is, however, blocked by the superiority relation `Art2b > Art2a` in cases where the approval is given by the licensor, i.e., in cases where both the propositions `Approve_ea` and `Rexist_ea` hold in the context.

Finally, the fourth and fifth rules in (44) state that in cases where the publishing event `epr` is prohibited *and* it really exists then `Remove_er` is obligatory, i.e., it is obligatory to remove the published result, and this action will compensate the prohibited publishing, i.e., that `compensate_er_epr` also holds true in the context.

Article 3(a-c) in (2) is represented in SPINdle as in (45). (45) state that the publication of comments about the evaluation is prohibited, i.e., [O]-Publish_epc holds in the state of affairs, unless the publication of the results is permitted, i.e., unless [P]Publish_epr holds in the state of affairs.

```
(45) Art3a[O]: Evaluate_ev, Rexist_ev => -Publish_epc
      Art3b[P]: [P]Publish_epr => Publish_epc
      Art3b > Art3a
```

Finally, Article 4(a-b) in (2) is represented in SPINdle as in (46). (2) states that in cases where the evaluation has been commissioned and it really took place, then the publishing of its results is obligatory. This obligation can possibly override the prohibition from (44), as specified by the superiority relation `Art4a > Art2a`.

```
(46) Art4a[O]: Commission_ec, Rexist_ec, Evaluate_ev, Rexist_ev => Publish_epr
      Art4a > Art2a
```

4.7.2 Compliance checking via SPINdle rules

Given the propositional nature of SPINdle’s input language, and the fact that it does not support negation-as-failure, the rules to infer violations in SPINdle are higher

in number and more complex than their counterparts in the other logical languages considered above.

Since in propositional logic we cannot quantify over the individuals in the domain, we need to assert specific rules for every propositional symbol. For instance, the rule to infer that the prohibition of an evaluation has been violated is the following:

```
(47) ccRuleEv: [O]-Evaluate_ev, Rexist_ev => violated_ev
```

In cases where the first rule in (43) is triggered, `[O]-Evaluate_ev` is asserted. If it also holds that the evaluation really exists, (47) infers that the action `ev` has been violated.

The rule to infer that the prohibited publication of the results has been violated (literal `[O]-Publish_epr`) is similar to (43); however, since this violation could be compensated, we also require the literal `compensated_er_epr` to be false:

```
(48) ccRuleEpr1: [O]-Publish_epr, Rexist_epr, -compensated_er_epr
      => violated_epr
```

We remind the reader that the operator “-” of SPINdle’s input format denotes standard negation, not negation-as-failure. Thus, as we did for the formalization for Arg2P discussed in the previous subsection, we need to use a superiority relation to achieve the same semantics. In other words, we infer *by default* that `compensated_er_epr` is false. In cases where the remove action really exists (i.e., in case `Rexist_er` holds in the state of affairs) `-compensated_er_epr` is overridden. This is handled by the following rules:

```
(49) ccRuleComp1: compensate_er_epr => -compensated_er_epr
      ccRuleComp2: compensate_er_epr, Rexist_er => compensated_er_epr
      ccRuleComp2 > ccRuleComp1
```

The remove action `er` is asserted as obligatory whenever the publishing of the results is asserted as prohibited (see (44) above). Compliance with this obligation must be checked as well: whenever `[O]Remove_er` is asserted (through the fourth rule in (44)), a violation must be inferred in case `Rexist_er` does *not* exist. To achieve the desired inferences we must again use standard negation together with a superiority relation:

```
(50) ccRuleEr1: [O]Remove_er => violated_er
      ccRuleEr2: [O]Remove_er, Rexist_er => -violated_er
      ccRuleEr2 > ccRuleEr1
```

Also the publishing of the results can be inferred as obligatory. This happens whenever the evaluation has been commissioned, i.e., whenever the first rule in (46) above is triggered. The rules in (51) are therefore added to the input of the SPINdle reasoner:

```
(51) ccRuleEpr2: [O]Publish_epr => violated_epr
      ccRuleEpr3: [O]Publish_epr, Rexist_epr => -violated_epr
      ccRuleEpr3 > ccRuleEpr2
```

Finally, the publishing of the comments may be prohibited. If this is the case, we derive a violation in cases where `Rexist_epc` also holds:

```
(52) ccRuleEpc: [O]-Publish_epc, Rexist_epc => violated_epc
```

5 Evaluating and comparing the formalizations

The objective of this paper is to take stock of the current state of the art in legal reasoning. We aim in particular at identifying which legal reasoners might be employed within concrete and marketable LegalTech applications *dealing with big data*.

In light of this, it is of course necessary to evaluate the performance of the reasoners, specifically their scalability with respect to large datasets of rules and input facts.

Nevertheless, computational performance cannot be the sole criterion for comparison. Before processing norms from legislation, these norms must be formalized in machine-readable formats and the formalizations must be checked/debugged. This task should be done by lawyers and other domain experts, who are likely unfamiliar with technical details. Therefore, other criteria such as human-readability, ease of editing and debugging, explainability, etc. of the format chosen to encode the if-then rules are likewise pivotal.

While this section only focuses on the computational performance of the legal reasoners presented above, Sect. 6 below will discuss the reasoners from a broader perspective, which also includes non-computational elements of comparison.

In order to evaluate the computational performance of the legal reasoners, we developed an automated generator that creates synthetic datasets of increasing size in the input format of every considered reasoner. All reasoners are then executed on their corresponding datasets. Testing AI systems through artificially created datasets is a widely used and consolidated strategy [see Maher et al. (2001), among others].

The generator is able to create datasets of increasing size along two dimensions: the number of input facts, i.e., the number of states of affairs that must comply with the in-force norms, and the number of if-then rules, i.e., the formalizations of these norms.

As discussed in Antoniou et al. (2021), which we consider as a seminal work in large-scale legal reasoning, in real-world LegalTech applications dealing with big data the number of input facts is usually much larger than the number of rules with respect to which the compliance of the former is checked. For example, Antoniou et al. (2021) discusses FinTech applications, i.e., LegalTech applications operating in the financial domain. Millions of financial transactions take place every day and each of them must be checked against in-force norms about anti-fraud, anti-money laundering, taxation, etc., which are much smaller in number and not so frequently updated.

In light of this, the next subsection focuses on the first dimension, i.e., input facts, which we consider more important from a “big data” perspective. We first discuss how the generator creates synthetic datasets of input facts with increasing size, then we present the time performance of the reasoners on these datasets.

Section 5.2, on the other hand, focuses on the second dimension, i.e., the rules. The generator will be extended in order to duplicate the rules of the use case.

The reader can download the dataset generator from GitHub and generate further synthetic datasets by changing the parameters associated with the two dimensions.

5.1 Generating and evaluating synthetic datasets of input facts

As mentioned earlier, the generator is able to create synthetic datasets of input facts in the considered formats. Each dataset represents the same knowledge thus making it possible to double-check the compliance checking results across the reasoners. The generator produces datasets of different sizes, depending on some input configurable parameters.

In each dataset, the actions that really took place in the state of affairs are randomly generated. As explained in the previous sections, these actions are those asserted on the `Rexist` predicate; this predicate denotes the “really exist” modality. Therefore, in order to randomly generate actions that really took place in the state of affairs, we simply randomly select which actions are asserted on the `Rexist` predicate.

To do so, the software generates a random number from 0 to 100 for each action of our use case (`Publish`, `Approve`, `Remove`, etc., see Fig. 1). If the number is lower than the input probability, it asserts `Rexist` on the corresponding action.

In order to create datasets of input facts with different sizes, an incremental index is assigned to each tuple of actions. As an example consider the facts in (53), which are some of the input facts automatically generated for the ASP implementation.

```
(53) licensee(x_0). licensor(y_0). result(r_0). evaluate(ev_0). rexist(ev_0).
    hasAgent(ev_0,x_0). hasTheme(ev_0,p_0). product(p_0). hasResult(ev_0,r_0).

    licence(l_0). isLicenceOf(l_0,p_0). grant(eg_0). hasAgent(eg_0,y_0).
    hasReceiver(eg_0,x_0). hasTheme(eg_0,l_0).

    licensee(x_1). licensor(y_1). result(r_1). evaluate(ev_1). rexist(ev_1).
    hasAgent(ev_1,x_1). hasTheme(ev_1,p_1). product(p_1). hasResult(ev_1,r_1).

    licence(l_1). isLicenceOf(l_1,p_1). grant(eg_1). hasAgent(eg_1,y_1).
    hasReceiver(eg_1,x_1). hasTheme(eg_1,l_1). rexist(eg_1).
```

From the facts in (53), the ASP rules in (12) and (13) above derive `prohibited(ev_0)` and `permitted(ev_1)`. In fact, both facts `rexist(ev_0)` and `rexist(ev_1)` hold in the state of affairs. However, only `rexist(eg_1)` holds, while `rexist(eg_0)` does not.

The dataset generator creates equivalent input facts for SHACL, DLV, PROLEG, and Arg2P; for PROLEG, the generator also creates in the dataset of the input facts a query for each possible individual action that might be violated. On the other hand, for SPINdle the generator must create indexed (propositional) rules besides indexed input facts. We remind the reader that SPINdle’s input format is propositional; therefore, the rules must be *externally* grounded before invoking the reasoner. For the example in (53), the generator creates the following rules and facts for SPINdle:

```
(54) Art1a_0[O]: => -Evaluate_ev_0
    Art1b_0[P]: Grant_eg_0, Rexist_eg_0 => Evaluate_ev_0
    Art1b_0 > Art1a_0
    Art1a_1[O]: => -Evaluate_ev_1
    Art1b_1[P]: Grant_eg_1, Rexist_eg_1 => Evaluate_ev_1
    Art1b_1 > Art1a_1
    » Evaluate_ev_0
    » Evaluate_ev_1
    » Grant_eg_1
```


This paper omits further technical details about the dataset generator; the interested reader is directed to the source code available on the GitHub repository,¹⁵ from which the dataset generator may be downloaded.

5.1.1 Evaluating the reasoners on the datasets of input facts

The SHACL, ASP, DLV, PROLEG, Arg2P, and SPINdle formalizations illustrated in the previous sections have been executed on synthetic datasets of input facts with different sizes. For SHACL, we used TopBraid SHACL v.1.3.2.¹⁶ For ASP we used both the Clingo reasoner v5.5.1¹⁷ and DLV2 (Calimeri et al. 2020a) while for ASP enriched with inheritance networks we used the original DLV (Leone et al. 2006); both DLV and DLV2 are freely distributed for research purposes by DLVSystem,¹⁸ a spin-off company of the University of Calabria. For PROLEG, we used SWI Prolog.¹⁹ Finally, for Arg2P and SPINdle we used the libraries freely distributed on the homepages of the two reasoners.²⁰

Table 1 shows the time performance on three datasets respectively including 10, 30, and 50 states of affairs. All experiments reported in this paper were run on a PC with Intel(R) Core(TM) at 1.8 GHz, 16 GB RAM, and Windows 10.

Table 1 Time performance of the compliance checkers

Size	SHACL (s)	ASP (clingo) (s)	ASP (DLV2) (s)	DLV (s)	PROLEG (s)	Arg2P (s)	SPINdle (s)
10	0.091	0.019	0.0552	0.0347	0.398	398.338	0.063
30	0.122	0.025	0.0337	0.0505	0.631	1039.668	0.099
50	0.148	0.051	0.0553	0.097	1.374	1927.389	0.187

From the results reported in Table 1, it is evident that PROLEG and, in particular, Arg2P are much slower than the other reasoners. As will be discussed in the next section, PROLEG and Arg2P are the only reasoners, among the ones considered in this paper, that currently provide explanations of their inferences, specifically the trace of the applied rules and the graphs of the arguments. Explainability is of course a highly desirable property for modern LegalTech applications, but Table 1 undoubtedly shows that it comes with a (huge) price to pay.

We were surprised in particular by Arg2P's computational performance. We knew that building and processing the graph of the arguments would require extra computations and so we were indeed expecting lower time performances; still, we were not

¹⁵ <https://github.com/liviorobaldo/compliancecheckers>.

¹⁶ <https://repo1.maven.org/maven2/org/topbraid/shacl/1.3.2>.

¹⁷ <https://github.com/potassco/clingo/releases>.

¹⁸ <https://www.dlvsystem.it>.

¹⁹ <https://www.swi-prolog.org>.

²⁰ These are respectively available at <https://apice.unibo.it/xwiki/bin/view/Arg2p/WebHome> and <https://sourceforge.net/projects/spindlereasoner>.

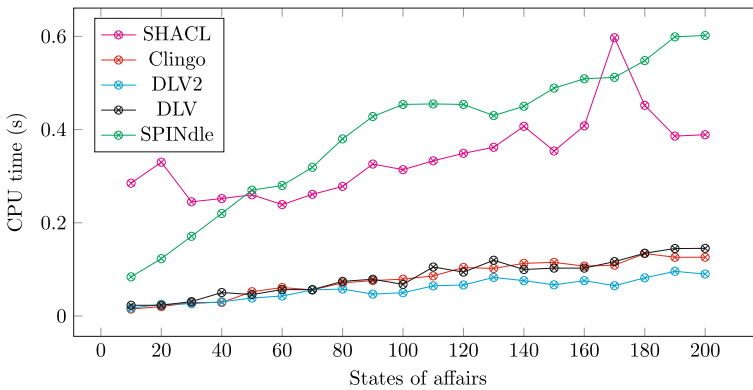


Fig. 3 Execution of SHACL, Clingo, DLV2, DLV, and SPINdle on datasets with increasing number of states of affairs (the probability for an action to `ReXist` is set to 50%)

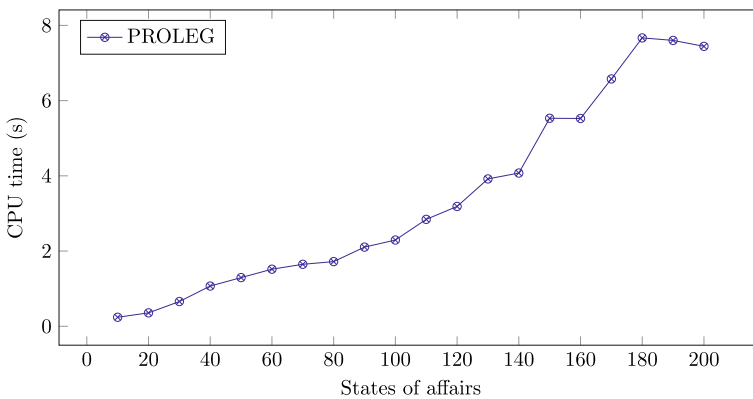


Fig. 4 Execution of PROLEG on the same datasets used in Fig. 3

expecting them to be *so much* lower. For this reason, we will not further report the time performance of Arg2P for the datasets considered below, which are even larger in size.

Further computational results are reported in Figs. 3 and 4. Both figures consider the same datasets including from 10 to 200 states of affairs. The chart in Fig. 3 shows the time performances of all reasoners apart from Arg2P and PROLEG. The time performance of the latter is shown in Fig. 4; we decided to show it in a separate figure because it is far superior to the ones of the other reasoners.

From Figs. 3 and 4 it is evident that the three ASP-based reasoners outperform the other reasoners, with DLV2 being the fastest reasoner among them.

5.2 Generating and evaluating synthetic datasets of rules

After showing how the reasoners scale on datasets with input facts of increasing size (while keeping constant the set of rules, i.e., the ones of our use case), we present a

new set of experiments that show how the reasoners scale on datasets with sets of rules of increasing size (while keeping constant the set of input facts).

The aim is to test the reasoners on large sets of (machine-readable translations of) norms, which, however, we do not have a disposal. To create these sets we should develop a user friendly editor and annotate existing legislation, as was done in Robaldo et al. (2020), Nazarenko et al. (2018). This is much beyond the scope of this paper, although it is definitely part of our future works (see Sect. 6.2.5 below).

Therefore, we decide instead to generate more synthetic rules by *duplicating* the rules in the use case for increasing numbers of times. In particular, we created new rules by indexing the predicates, thus actually creating *new* predicates. For example, from the ASP if-then rule corresponding to Article 1a and shown again in (55):

```
(55) prohibited(Ev) :- evaluate(Ev), hasAgent(Ev,X), licensee(X),
                        hasTheme(Ev,P), product(P), not exceptionArt1b(Ev).
```

We created the following indexed rules:

```
(56) prohibited_1(Ev) :- evaluate_1(Ev), hasAgent_1(Ev,X), licensee_1(X),
                        hasTheme_1(Ev,P), product_1(P), not exceptionArt1b_1(Ev).
```

```
prohibited_2(Ev) :- evaluate_2(Ev), hasAgent_2(Ev,X), licensee_2(X),
                        hasTheme_2(Ev,P), product_2(P), not exceptionArt1b_2(Ev).
```

Etc.

In other words, by indexing the predicates in the norms we can double, triple, etc. the dataset of norms processed by the automated reasoners, i.e., we can extend the dataset by adding therein an equal number of *symmetric* norms.

The additional norms, being symmetric to the ones of the use case, do not of course add any further variety to the dataset. Still, they allow the testing of the scalability of the reasoners on sets of rules of increasing size, which is the objective of the present subsection. On the other hand, we already discussed in Sect. 2.1 above that, in our view, the variety of our norms is acceptable, at least compared to the ones of contemporary approaches, e.g., Francesconi and Governatori (2022). The evaluation of the reasoners on larger sets of rules, both in size and in variety, is left for future investigations, once the LegalRuleML editor advocated below in Sect. 6.2.5 will become available.

Finally, concerning the input states of affairs in the ABox, the new version of the dataset generator must create synthetic individuals for each predicate, i.e., it must also index these individuals on the associated predicate. In other words, while in the previous section, for instance, from Ev we were generating the individuals ev_1, ev_2, etc., in this section, from Ev we generate the individuals ev_1_1, ev_1_2, etc., ev_2_1, ev_2_2, etc., in which the first index corresponds to the index of the predicates on which the individuals are asserted as true; therefore, for instance, in ASP the dataset generator adds to the ABox the predications evaluate_1(ev_1_1), evaluate_1(ev_1_2), etc., evaluate_2(ev_2_1), evaluate_2(ev_2_2), etc.

In the new set of experiments, we generated synthetic datasets in which the norms in the use cases have been duplicated 10, 20, ..., 200 times. For each of these datasets, the number of states of affairs remains constant to 10. Results are shown in Fig. 5.

In Fig. 5, PROLEG and SPINdle have the worst computational times: each of the two reasoners takes more than 15s to process the dataset with the rules of the use

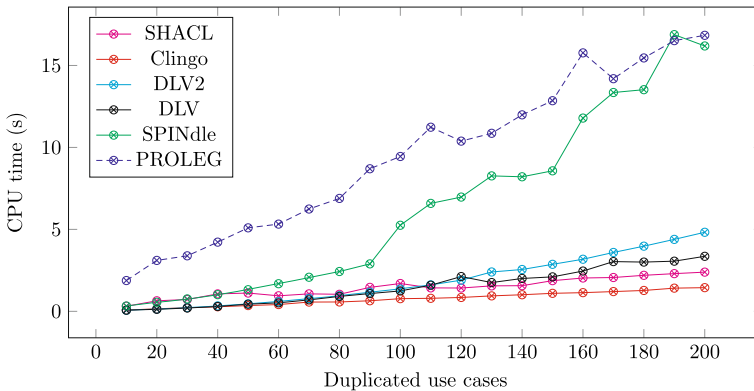


Fig. 5 Execution of SHACL, Clingo, DLV2, DLV, SPINdle and PROLEG on datasets with increasing number of duplicated use cases (the number of facts is set to 10 for every set of rules while the probability for an action to `ReXist` is set to 50%)

case duplicated 200 times. For this reason, we will no longer consider PROLEG and SPINdle in the next and final experiment (Fig. 6 below). We observe that PROLEG was already rather slow for processing the datasets considered in the previous subsection while SPINdle was not (see Figs. 3, 4 above). The results in Fig. 5 show that SPINdle becomes as slow as PROLEG when the number of rules increases. In particular, in Fig. 5 it is possible to notice that SPINdle's performance substantially gets worse when the rules in the use cases are duplicated 100, 160, and 190 times.

Figure 5 also highlights an interesting difference between the ASP-based reasoners: DLV2, i.e., the reasoner with the best assessed performance in the previous experiment, takes more time to process the datasets considered in Fig. 5 than Clingo, DLV, and even SHACL, which is not even ASP-based. In other words, the scalability of DLV2 is worse than Clingo's, DLV's, and SHACL's when the number of rules increases.

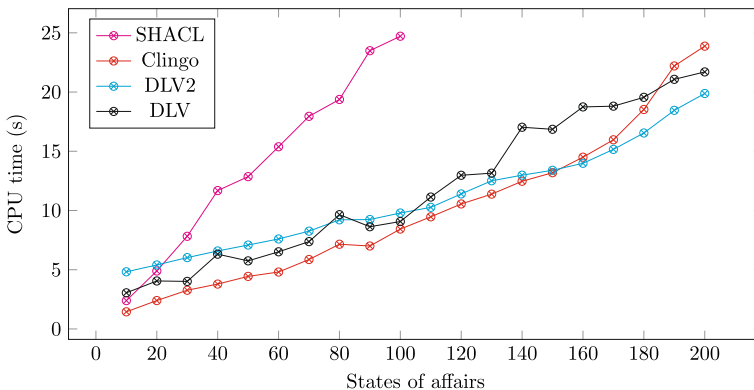


Fig. 6 Execution of SHACL, Clingo, DLV2, DLV, and SPINdle on datasets with increasing number of states of affairs (the number of duplicated use cases is set to 200 while the probability for an action to `ReXist` is set to 50%)

In light of these results, we decided to run a final experiment: we executed the reasoners on the dataset with 200 duplicated use cases, i.e., the larger one among those considered in Fig. 5, while varying the size of the input states of affairs from 10 to 200. Results are shown in Fig. 6. The time performance of SHACL is reported only until 100 states of affairs, in which the reasoner's time already exceeds 24 s.

Figure 6 shows that, in case the set of rules remains constant, regardless of its size, DLV2 outperforms again Clingo and DLV when processing large datasets of input facts.

6 Discussion and future works

This section compares and discusses the formalizations presented above under several perspectives. The discussion will set the ground for further directions of research.

6.1 Computational performance

The experiments reported in the previous section highlight that, in almost all cases, the ASP-based reasoners outperform all other reasoners. This is not surprising, as ASP has been primarily designed to cope with NP-hard search problems (Calimeri et al. 2016).

DLV2 shows a better scalability when the number of input facts increases; however, its performance gets worse when the number of input facts remains constant and the number of rules increases, in which case the fastest reasoner is Clingo.

Indeed, in this case, dealing with an increased number of rules requires more effort in the solving phase, while having a greater number of input facts impacts mainly on the instantiation/grounding phase. Thus, we observe that DLV2 better scales during the grounding phase, while Clingo achieves better scalability in the solving phase; related discussion is available in Calimeri et al. (2020a).

Therefore, in light of the discussion at the beginning of Sect. 5 above, in turn inspired by the discussions in Antoniou et al. (2021), it may be concluded that DLV2 should be preferred within LegalTech applications that check compliance on big data, in which the size of the ABox is deemed to be more crucial than the size of the set of if-then rules. By contrast, for applications or use cases in which the number of rules is much superior to the number of facts, and for which computational performance is a relevant requirement, Clingo appears to be a better choice.

Arg2P, PROLEG, and SPINdle are much slower than the other reasoners. In Arg2P in particular, the construction of the argument graph greatly slows down the process of compliance checking. For instance, as shown in Table 1, Arg2P takes more than 30 min to process the norms of the use case with respect to the synthetic dataset with 50 states of affairs for which DLV2 takes less than 0.06 s.

Since Arg2P is one of the most modern implemented reasoners for structured argumentation, the assessed slow performance definitely demands much further research on the topic. Structured argumentation has been mainly studied so far from a theoretical point of view but it is time now to research ways of making the theoretical findings

usable in practice. This could be perhaps achieved by modeling problems in argumentation precisely as problems in Answer Set Programming, in order to make the most of the format's efficiency, a solution already advocated in Brewka et al. (2011).

On the other hand, as explained at the beginning of Sect. 4.6 above, one of the objectives of Arg2P's implementation is to allow the platform's users to configure the reasoner only on the functionalities they need in their legal domain and for the purposes of their projects. Since the argument graph is not always needed, future versions of Arg2P will possibly allow for this option to be disabled, for instance when performing reasoning on "static" norms from legislation such as those in the use case considered here.

Similar considerations hold for PROLEG. However, contrary to Arg2P, PROLEG is not a stand-alone legal reasoner. It is a library that must be loaded within other Prolog reasoners, e.g., SWI Prolog.²¹ Thus, carrying out further research to improve PROLEG efficiency most likely amounts to carrying out further research to improve the efficiency of reasoners for standard Prolog.

An alternative solution could be to define translation algorithms from PROLEG's input format to ASP. Satoh et al. (2012) showed that ASP and PROLEG are indeed mathematically equivalent. Thus, domain experts could still take advantage of PROLEG's explainability (see Sect. 6.2.4 below) to build the initial formulae; these could be then converted into ASP and used to efficiently check compliance on big data.

Concerning SPINdle, we remind the reader that we included it in our comparison, although its input format is propositional, only because it is widely acknowledged in past literature on legal reasoning. In order to carry out the inferences in SPINdle, the dataset generator had to ground the rules before executing them on the (propositional) input facts. The time that the dataset generator took to ground the rules should be then added to the ones shown in Table 1. In other words, it could be argued that the reported execution time for SPINdle does not fully reflect the computational performances of the reasoner.

Following the same line of reasoning, we observe that, actually, Table 1 reports the times that each reasoner took to process the datasets *encoded in its input format*.

SHACL is the single reasoner considered above that directly processes RDF data and, indeed, this should be considered as one of its advantages given the increasing availability of (big) data in RDF, the standard language for Semantic Web. In fact, in order to process RDF triples in Clingo or DLV2 these must be first converted in ASP and the computational time needed for the conversion should be therefore added to the computational time needed to perform the inferences.

These considerations are very well-known in the ASP scientific community. The increasing use of ASP in real-world scenarios over the last years has been continuously fostering definitions and extensions of methods and tools for successfully dealing with practical applications, up to the industrial level. For instance, there is a common agreement on the need for efficiently handling multiple queries and reasoning tasks over large-sized knowledge bases in RDF. In light of this, for instance, the DLV2 reasoner has been extended in order to connect with both relational and graph databases

²¹ <https://www.swi-prolog.org>.

via explicit directives for importing/exporting data. For further details, we refer the reader to Calimeri et al. (2017), Leone et al. (2019).

6.2 Ease of representation and expressivity

Although computational performance is of paramount importance in the big data era, it is not the only metric through which available compliance checkers should be compared. As pointed out above, other crucial elements of comparison are the ease of representation and the expressivity of the formal languages used by these compliance checkers. This subsection compares the formal languages considered above from these points of view.

In relation to this, we remind the reader that one of our objectives was to use the same predicates/granularity across the formal languages considered. In all formalizations above we kept the correspondence between the predicates used therein and the concepts in the shared ontology from Fig. 1. This was done because we are interested in formulae that may be easily shared, i.e., that facilitate *interoperability* between different reasoners.

In other words, with the exception of the (propositional) SPINdle reasoner, we did not want to consider here non-interoperable, ad-hoc, and abstract formalizations just to fit with the limits of the reasoners. We believe that only by using shared concepts/predicates will the comparisons be truly meaningful. We used OWL as the intermediate language to represent the shared predicates given its wide adoption in the Semantic Web.

6.2.1 Modal operators

While formalizing the use case, we found that modal operators, such as the operators “o” and “p” of Arg2P, are rather difficult to use in conjunction with first-order formulae. On the contrary, unary first-order predicates applied to terms that directly refer to the actions appear to be easier for editing, reading, and debugging the formulae.

Arg2P’s modal operators can outscope a *single* predicate. On the other hand, for representing the actions together with their thematic roles we need a *conjunction* of predicates, e.g., `publish(Ep)`, `hasAgent(Ep, X)`, `licensee(X)`, etc. in (36) above.

In our view, the only way to achieve the desired truth conditions in the current version of Arg2P is to assert the conjunction of predicates in the antecedent of the rule and the modal operator applied to the “main” predicate in the consequent. This was done, for instance, for rule “art4a” in (36), copied again in (57) for reader’s convenience.

```
(57) art4a: publish(Ep), hasAgent(Ep, X), licensee(X), hasTheme(Ep, R),
      result(R), hasResult(Ev, R), evaluate(Ev), reXist(Ev),
      hasTheme(Ec, Ev), commission(Ec), reXist(Ec) => o(publish(Ep)).
```

However, we observe that allowing the Arg2P operator “o” to also accept a conjunction of predicates as argument does not appear to solve the problem so simply. In Standard

Deontic Logic (SDL),²² which inspired the definition of these modal operators, the axiom $\circ(P_1, P_2) \rightarrow (\circ(P_1), \circ(P_2))$ holds. Thus, whenever the modal operator would outscope a conjunction, it would distribute over the conjuncts, e.g.:

(58) $\circ(\text{publish}(x, r), \text{licensee}(x), \text{result}(r))$
 $\Rightarrow \circ(\text{publish}(x, r)), \circ(\text{licensee}(x)), \circ(\text{result}(r))$

Nevertheless, (58) means that it is obligatory for the individual x to be a licensee and for the individual r to be a result, which sounds weird and counter-intuitive.

The modal operators of the LegalRuleML standard²³ also suffer from the same problem. Possible solutions within future versions of the standard are still under discussion within the LegalRuleML technical committee.²⁴

We observe that these counter-intuitive derivations are not found with propositional symbols, e.g., in SPINdle: it is a problem related to the use of modal deontic operators in conjunction with *first-order* formulae, for which one should perhaps define an alternative semantics for the modal operators that does not encompass the SDL axiom exemplified in (58). However, this solution requires much further research to properly investigate whether it could lead or not to other counter-intuitive derivations.

Another solution could be to avoid conjunctions of predicates by replacing them with single “huge” predicates that encompass all thematic roles. By adopting this strategy, in place of the antecedent in (58), we would assert a single literal such as:

(59) $\circ(\text{publishResultsOfEvaluationByLicensee}(x, r))$.

However, this is precisely an example of “non-interoperable, ad-hoc, and abstract formalization just to fit with the limits of the reasoners” that we mentioned above. In this paper, we do not want to consider solutions such as this; we want instead to maintain the 1:1 correspondence with the concepts in the shared ontology shown in Sect. 4.1.

After some further attempts we concluded that the formalization in Sect. 4.6 is the only available one to achieve the desired truth conditions in the current version of Arg2P.

6.2.2 Compliance checking rules

Another drawback related to the use of modal operators is that they require the introduction of multiple specific rules to infer violations, as we already noticed above in Sect. 4.6.2 (Arg2P) and Sect. 4.7.2 (SPINdle).

For instance, in Arg2P we need to add the ad-hoc rules in (60) to detect the four possible violations that could arise in our use case: “ccRuleEv” detects when a prohibited evaluation took place in the state of affair, “ccRuleEp1” detects when a prohibited publishing took place in the state of affair, “ccRuleEp2” detects when an obligatory publishing did not take place in the state of affair, and “ccRuleEr” detects when an obligatory removal did not take place in the state of affair.

²² <https://plato.stanford.edu/entries/logic-deontic/#StanDeonLogi>.

²³ https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/os/legalruleml-core-spec-v1.0-os.html#_Toc38017882.

²⁴ Personal communications within the LegalRuleML technical committee, during its recent activities to support the evolution of the standard.


```
(60) ccRuleEv: o(-evaluate(Ev)), reXist(Ev), ~(compensated(Ev))
      => violation(viol(Ev)).

ccRuleEp1: o(-publish(Ep)), reXist(Ep), ~(compensated(Ep))
      => violation(viol(Ep)).

ccRuleEp2: o(publish(Ep)), ~reXist(Ep), ~(compensated(Ep))
      => violation(viol(Ep)).

ccRuleEr: o(remove(Er)), ~reXist(Er), ~(compensated(Er))
      => violation(viol(Er)).
```

By contrast, in the other logical formalisms we need a single rule to detect violated prohibitions and another one to detect violated obligations. The two rules detecting violations in ASP are shown in (61).

```
(61) violation(viol(X)) :- obligatory(X), not reXist(X), not compensated(X).
     violation(viol(X)) :- prohibited(X), reXist(X), not compensated(X).
```

Therefore, the formalizations in Arg2P and SPINdle do not appear to be practical for handling bigger use cases. These formalizations require the introduction of ad-hoc rules such as the ones in (60) for *every* action that may be prohibited or obliged.

The reason why Arg2P requires the introduction of multiple rules for detecting violations is because the input format does not allow the predicates denoting the actions to be quantified. A solution, that we will possibly investigate in our future work, could be then to enrich Arg2P's input format with second-order quantifiers to this end. Thus, the four if-then rules in (60) could be replaced by two if-then rules such as:

```
(62) ccRuleProhibition: ∃P[o(-P(X)), reXist(X), ~(compensated(X))]
      => violation(viol(X)).

ccRuleObligation: ∃P[o(P(X)), ~reXist(X), ~(compensated(X))]
      => violation(viol(X)).
```

In (60), “ \exists_P ” is a second-order quantifier that requires the existence of a predicate P for which the antecedent is true; violations are asserted on the argument X of this predicate.

Finally, since in this paper we also considered compensatory norms, the compliance checking rules had to include additional rules to infer whether violated obligations or violated prohibitions have been compensated or not.

We believe that the proper modelling of compensatory norms is to introduce functional terms in case violations occur. In SHACL, due to the nature of the RDF format, these functional terms correspond to anonymous individuals that are created “on the fly”.

This choice has been made to follow the intuition that compensatory norms must be added in the state of affairs only when violations occur, i.e., they *functionally depend* on the obligations/prohibitions that have been violated.

Nevertheless, the disadvantage of this choice is that we then need additional ad-hoc rules to check whether the state of affairs includes an individual that satisfies the same predicates of the functional term, i.e., an individual to which the functional term could be *matched*. In other words, contrary to the rules to infer violations, in which we need such specific ad-rule rules only in Arg2P and SPINdle, to infer compensations we

need such specific ad-rule rules in *every* language we have considered in this paper. For instance, in ASP we need the rule shown above in (21) and copied again in (63) for reader convenience.

```
(63)    $\text{reXist}(\text{ca}(\text{Ep}, \text{X}, \text{R})) :- \text{remove}(\text{ca}(\text{Ep}, \text{X}, \text{R})), \text{hasTheme}(\text{ca}(\text{Ep}, \text{X}, \text{R}), \text{R}),$   

        $\text{hasAgent}(\text{ca}(\text{Ep}, \text{X}, \text{R}), \text{X}), \text{reXist}(\text{Er}), \text{remove}(\text{Er}),$   

        $\text{hasTheme}(\text{Er}, \text{R}), \text{hasAgent}(\text{Er}, \text{X}).$ 
```

Rule (63) looks for another remove action Er in the state of affairs and checks whether the thematic roles hasAgent and hasTheme are the same of the functional term $(\text{ca}(\text{Ep}, \text{X}, \text{R}))$, in (63)). If this is true and if Er really exists, then also $\text{ca}(\text{Ep}, \text{X}, \text{R})$ really exists. Alternatively, the rule could assert that the two terms are actually the same term, i.e., that $\text{Er} = \text{ca}(\text{Ep}, \text{X}, \text{R})$ holds; in SHACL this could be implemented by using the OWL tag `owl:sameAs` to state that Er and the anonymous individual corresponding to the functional term are the same individual.²⁵

The removal of the prohibited publishing is the single compensatory norm of our use case, thus in every format we considered in the previous section we only needed a single extra rule that parallels (63). However, use cases that include multiple compensatory actions would require an additional rule for each compensatory action because each of these actions would feature a different set of thematic roles to check.

It is then evident that a proper implementation of compensatory norms in contemporary legal reasoners deserves much further research. Indeed, as we already observed above in Sect. 3, although compensatory norms are rather pervasive in legal texts, their implementation for legal reasoning has been scarcely investigated so far in the literature. To our knowledge, only (Governatori and Rotolo 2006) offers a serious analysis of compensatory norms. These have been implemented in the RuleRS and the Regorous systems [see Islam and Governatori (2018), Governatori (2015) respectively], but their source code is not publicly available.

Of course, since the rules for compliance checking do not vary/depend on the formalized norms, perhaps the best solution would be to just hard-code and optimize the whole compliance checking procedure within the reasoners, as is done in RuleRS and Regorous. Thus, the users would only need to encode the norms and then query the reasoner to automatically compute violations with respect to a given state of affairs.

6.2.3 Negation-as-failure versus superiority relations

Two alternative constructs have been used in the formalizations above for representing defeasible rules: negation-as-failure and superiority relations. The former are used in SHACL and ASP while the latter are used in DLV, PROLEG, Arg2P, and SPINdle. It is worth noticing also that LegalRuleML, designed to be a rule-based interchange XML standard language for the legal domain, implements defeasibility via a special tag “OverrideStatement” that resembles superiority relations.²⁶

Most of the considered reasoners and the LegalRuleML standard use superiority relations because they are more intuitive and easier to manipulate than negation-as-failure. This is particularly true for lawyers, as extensively argued in Satoh et al. (2011).

²⁵ See <https://www.w3.org/TR/owl-ref/#sameAs-def>.

²⁶ See https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/os/legalruleml-core-spec-v1.0-os.html#_Toc38017880.

The original version of PROLEG indeed employed negation-as-failure. However, in light of the feedback from lawyers and law school students working with the reasoner, its developers decided to replace it with the “exception” predicate seen in Sect. 4.5 above.

Our experience in formalizing the use case in the several formats further supports the claims from Satoh et al. (2011). We indeed found it simpler to *explicitly* indicate which rules override which other ones via superiority relations than by doing so *implicitly* by introducing further special predicates on which applying negation-as-failure.

As shown in Fig. 2, we had to insert four additional special predicates/classes in the ontology to handle defeasibility via negation-as-failure. More complex use cases would require an even bigger set of these special predicates, one for each possible overriding of the rules. Therefore, it could definitely be hard for lawyers to keep track of which special predicates are used within which if-then rules, while editing or revising the formulae.

A single meta-predicate is instead required to formalize superiority relations. This single operator allows the intuitive definition of the directed acyclic graph representing which rules override which other ones.

In light of this, it is evident that further research is needed to implement superiority relations within SHACL and ASP, in order to reconcile ease of editing/debugging with the peculiarities of the two formalisms.

This could be implemented again in terms of translation algorithms from superiority relations into SHACL or ASP constructs for negation-as-failure.

The alternative solution is to add further constructs to the SHACL or ASP vocabularies, as it has been done in DLV. As shown in Sect. 4.4 above, DLV introduced inheritance networks in the ASP vocabulary in order to reconcile easy and intuitive editing with ASP computational performance. The new DLV2 system, which aims at reimplementing DLV from scratch by using modern software engineering methodologies and technologies, does not currently encompass inheritance networks; however, this is part of the ongoing/future work of the reasoner.

6.2.4 Explainability

Explainability is deemed by many as one of the most desirable features for next-generation AI systems, both in academia and in industry. In the last decade, research in AI has mainly focused on Machine Learning, which is known to be a “black box” unable to provide explanations of its derivations/decisions. In order to advance academic research as well as to create new competitive advantage in industry, novel methods to develop explainable AI solutions have been greatly advocated and are currently under study.

The results of our experiment undoubtedly show that a lot of further research still needs to be done to implement explainability in legal reasoning.

Arg2P and PROLEG cannot be used to process big data, given their very slow computational performances. Future research will address scalability within the two reasoners.

However, for Arg2P the solution does not appear to be so simple. Arg2P has been designed to represent *more knowledge* than PROLEG and the other reasoners con-

sidered above. Besides the derivations, Arg2P represents the graph of the arguments that either support or reject certain facts or derivative rules, together with all meta-information associated with these arguments, e.g., which legal authorities endorse which arguments, which other arguments these authorities have the power to attack or defend, etc.

Arg2P could perhaps be extended to allow disabling the construction of the argument graph for tasks that do not need it, e.g., the use case from (2). On the other hand, for many other use cases and applications the graph would be indeed useful, in that it allows the building of fine-grained explanations, thus leading to novel cutting-edge legal services.

Norms are highly subject to legal interpretations. Rules and even single words occurring in legislation can be interpreted in different ways depending on the context. These interpretations are often stretched to the maximum as they are used in disputes that represent different interests, so that lawyers spend a considerable amount of their working time investigating legal interpretations of the norms that mostly favour their clients, as well as ways to rebut the arguments of their counter-parties.

The analysis of all possible contextual meanings that norms from legislation may denote is of course a huge and time-consuming work. Therefore, a legal reasoner able to model inferences on the arguments that were or might be used in trials can be of great help for lawyers, worth the investments to enhance the computational performance.

In our future work, we will investigate in particular how this may be achieved in Arg2P. This is one of the main research outputs of the ongoing ERC project *CompuLaw*²⁷ and it aims at becoming a reference framework for structured argumentation.

In Sect. 6.1 above it has been pointed out that the performances of PROLEG and Arg2P could be improved via a translation from their input formats into a faster reasoning language, e.g., ASP. However, it must be also pointed out that the translation in ASP, although it could possibly improve the performance, may hinder explainability.

Achieving explainability in ASP could be difficult because, as explained in Sect. 4.3 above, ASP is a declarative language in which the reasoner tries to satisfy all rules *at once*. The returned answer set does not specify which rules have been applied to obtain the facts within the answer set. This knowledge must be inferred through an additional “reverse engineering” process, from the returned answer set to the asserted rules. The process is particularly complex in cases where we are trying to explain why certain facts were *not* inferred, i.e., why they do *not* belong to the output answer set.

Achieving explainability in ASP is a matter of ongoing research (Dauphin and Satoh 2019). Several techniques and methodologies to debug answer-set programs have been proposed, such as those by Gebser et al. (2008), Oetsch et al. (2018), Cuteri et al. (2019). The common insight of these solutions is to add an extra-layer that relates the facts in the returned answer set with the rules that derive them, thus allowing the inferential process to be traced.

Implementing corresponding debugging procedures in SHACL appears to be simpler because SHACL rules are associated with priorities while ASP’s are not. Priorities of course facilitate explainability as they define a partial workflow of the rules’ application.

²⁷ <https://site.unibo.it/compulaw/en>.

Achieving efficient explainability in legal reasoning will be one of the main research objectives of our future works.

6.2.5 Standardization

Building symbolic knowledge is highly time-consuming, especially in LegalTech where the knowledge originates from norms written in natural language. Moreover, norms from real legislation are highly dependent on the legal domain they regulate (finance, health, etc.); thus, their proper formalization must necessarily involve lawyers or other domain experts, many of whom are unfamiliar with logic and technical details.

In our view, the involvement of domain experts towards the creation of large knowledge bases of machine-readable formulae associated with existing legislation might be achieved only by defining a *standardized methodology*, from the norms in natural language to the executable formalizations in some implemented legal reasoner.

In our future work, we intend to define such a methodology around LegalRuleML, which became an OASIS standard very recently, specifically on August 30th, 2021.²⁸

LegalRuleML is an XML-based semi-formal language that aims to enhance the interplay between experts in law and experts in logic. By “semi-formal” we mean that no formal model-theoretic semantics is associated with LegalRuleML. Well-formed LegalRuleML representations need to be translated into another language having such a semantics, e.g., the input formats of the legal reasoners considered above, similarly to what is done in Reaction RuleML 1.0 via the so-called “semantics profiles” (Paschke 2014).

Still, LegalRuleML defines a specification, in terms of an XML vocabulary and composition rules, that is able to represent the particularities of the legal normative rules with a rich, articulated, and meaningful markup language.

LegalRuleML aims at enabling domain experts to identify the if-then rules denoted by the norms, which if-then rules override or compensate which other ones, etc. For instance, in Sect. 2 above, we could have used LegalRuleML to encode the knowledge in (2), denoted by the natural language norms in (1).

The advocated annotations in LegalRuleML may be facilitated via a special editor that allows the composition of the if-then rules and stores them in the XML standard, similarly to what has been recently done in Robaldo et al. (2020). The LegalRuleML annotations so produced can be then automatically translated in a computational language to check the compliance of the denoted norms with respect to a given state of affairs.

In our future works, we will design and implement advanced editors for LegalRuleML, as well as translation algorithms from LegalRuleML to executable formats. The advocated editors could likewise integrate NLP procedures to assist the construction of the LegalRuleML annotations from legal texts (Robaldo et al. 2019).

²⁸ See <https://www.oasis-open.org/2021/09/08/legalruleml-core-specification-v1-0-oasis-standard-published>.

7 Conclusions

In this paper, we investigated some of current technologies for compliance checking at the *first-order* level, with conflicting and compensatory norms.

Our investigation has been motivated by the fact that most implemented legal reasoners, first of all SPINdle, are propositional. Nevertheless, propositional logic is too limited for existing LegalTech applications. As shown in Sect. 4.7 above, in order to use SPINdle or any other propositional legal reasoner the rules must be externally grounded on all possible input facts, which is unfeasible in particular when dealing with big data.

We investigated and compared some of main current reasoning languages with respect to a shared use case in the legal domain and a shared vocabulary of atomic predicates.

So far these reasoning languages were mostly studied in isolation. Investigating them together with respect to a shared use case and a shared vocabulary of predicates allowed their respective peculiarities to be highlighted.

To the best of our knowledge, this paper represents the first work in which some of the main automated reasoners used in recent research for compliance checking have been compared and evaluated on the same use case. Comparisons are crucial to research optimal solutions. We therefore hope that this paper will pioneer and kick off further research in legal reasoning based on (empirical) comparisons among the alternative solutions. This will be possible only by publicly releasing source codes and datasets on GitHub or similar platforms, in line with the FAIR principles.²⁹ All our source codes and datasets are freely available at <https://github.com/liviorobaldo/compliancecheckers>, together with instructions to locally reproduce the simulations.

Arg2P and PROLEG are inefficient, in particular Arg2P. However, these two reasoners are currently the only ones able to explain their derivations. Conversely, ASP is very efficient but lacks explainability because the declarative nature of the language itself makes it difficult to debug the inferences.

Furthermore, ASP uses negation-as-failure in place of the superiority relations used in Arg2P and PROLEG. The latter have been proved to be more readable and intuitive than the former: while superiority relations straightforwardly allow the encoding of the directed acyclic graph representing which rules override which other ones, negation-as-failure requires the introduction of additional predicates that refer to the exceptions. As these additional predicates increase in number along with the number of exceptions, it might be harder for a human to keep track and organize them when translating large sets of norms. DLV provides an intermediate solution in this respect: it is based on ASP but it models overriding via special superiority relations called “inheritance networks”.

SHACL rules can be directly applied to the RDF standard for the Semantic Web. We deem this as an advantage of SHACL in that more and more RDF triplestores are becoming available nowadays. In order to process them via the other reasoners, it is first necessary to translate the RDF triples into their input format, thus slowing down the whole process. Nevertheless, SHACL features some disadvantages with respect to the other reasoners: it is not as efficient as ASP and it also implements overriding via

²⁹ <https://www.go-fair.org/fair-principles>.

negation-as-failure. It is also worth noticing that SHACL rules have explicit priorities that, on the one hand, may facilitate explainability but, on the other hand, represent a further burden for human editors, as they are also in charge of deciding the execution order of the rules. The rules' order is instead immaterial in ASP, as the reasoner searches for answer sets that satisfy all rules at once.

All these observations raise interesting research questions that we will address in our future works: how and to what extent is it possible to encompass the benefits featured by the several implementations within a single integrated framework for legal reasoning while maximizing human interaction and efficient explainability? Perhaps the advocated framework could be defined in terms of one-to-one translation procedures from/to the different formats, centered on the LegalRuleML standard.

Acknowledgements Livio Robaldo has been supported by the Legal Innovation Lab Wales operation within Swansea University's Hillary Rodham Clinton School of Law. The operation has been part-funded by the European Regional Development Fund through the Welsh Government. Francesco Calimeri, Maria Concetta Morelli, Francesco Pacenza, and Jessica Zangari acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), Spoke 9 - Green-aware AI, under the NRRP MUR program funded by the NextGenerationEU, and the support of the project PRIN PE6, Title: "Declarative Reasoning over Streams", funded by the Italian Ministero dell'Università, dell'Istruzione e della Ricerca (MIUR), CUP:H24I17000080001. The research of Roberta Calegari and Giuseppe Pisano has been partially supported by the "CompuLaw" project, funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement No. 833647).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Antoniou G, Atkinson K, Baryannis G, Batsakis S, Di Caro L, Governatori G, Robaldo L, Siragusa G, Tachmazidis I (2021) Large-scale legal reasoning with rules and databases. *J Appl Logics IfCoLog J* 8(4):911
- Athan T, Governatori G, Palmirani M, Paschke A, Wyner A (2015) *LegalRuleML: design principles and foundations*. Springer
- Batsakis S, Petrakis EGM, Tachmazidis I, Grigoris A (2017) Temporal representation and reasoning in OWL 2. *Semant Web* 8(6):1–20
- Batsakis S, Baryannis G, Governatori G, Tachmazidis I, Antoniou G (2018) Legal representation and reasoning in practice: a critical comparison. In: Palmirani M (ed) *Legal knowledge and information systems, JURIX 2018*, volume 313 of *Frontiers in Artificial Intelligence and Applications*. IOS Press
- Billi M, Calegari R, Contissa G, Laggioia F, Pisano G, Sartor G, Sartor G (2021) Argumentation and defeasible reasoning in the law. *J* 4(4):897 (**special issue The Impact of Artificial Intelligence on Law**)
- Brewka G, Eiter T, Truszczynski M (2011) Answer set programming at a glance. *Commun ACM* 54(12):92–103
- Buccafurri F, Faber W, Leone N (2002) Disjunctive logic programs with inheritance. *Theory Pract Logic Program* 2:293
- Calautti M, Greco S, Trubitsyna I (2017) Detecting decidable classes of finitely ground logic programs with function symbols. *ACM Trans Comput Logic* 18(4):1–42


- Calegari R, Omicini A, Pisano G, Sartor G (2022) Arg2P: an argumentation framework for explainable intelligent systems. *J Logic Comput* 32(2):369
- Calimeri F, Cozza S, Ianni G, Leone N (2009) An ASP system with functions, lists, and sets. In: Erdem E, Lin F, Schaub T (eds) *Logic programming and nonmonotonic reasoning*. Springer, Berlin
- Calimeri F, Gebser M, Maratea M, Ricca F (2016) Design and results of the fifth answer set programming competition. *Artif Intell* 231:151–181
- Calimeri F, Fuscà D, Perri S, Zangari J (2017) External computations and interoperability in the new DLV grounder. In: Floriana E, Roberto B, Stefano F, Francesca AL (eds.) *Proceedings of 16th international conference of the Italian association for artificial intelligence (AI*IA 2017)*, volume 10640 of *Lecture Notes in Computer Science*. Springer
- Calimeri F, Fuscà D, Germano S, Perri S, Zangari J (2019) Fostering the use of declarative formalisms for real-world applications: the embasp framework. *New Gener Comput* 37(1):29–65
- Calimeri F, Dodaro C, Fuscà D, Perri S, Zangari J (2020a) Efficiently coupling the I-DLV grounder with ASP solvers. *Theory Pract Logic Program* 20(2):205
- Calimeri F, Faber W, Gebser M, Ianni G, Kaminski R, Krennwallner T, Leone N, Ricca F, Schaub T (2020b) Asp-core-2 input language format. *Theory Pract Logic Program* 20(2):294–309
- Cuteri, B, Dodaro C, Ricca F, Schüller P (2020) Overcoming the grounding bottleneck due to constraints in ASP solving: constraints become propagators. In: Christian B (ed.) *Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI 2020*
- Cuteri B, Dodaro C, Ricca F (2019) Debugging of answer set programs using paracoherent reasoning. In: Alberto C, Eugenio GO (eds.) *Proceedings of the 34th Italian conference on computational logic (CILC 2019)*, volume 2396 of *CEUR Workshop Proceedings*. CEUR-WS.org
- Cyras K, Francesca T (2016) ABA+: assumption-based argumentation with preferences. In: Baral C, Delgrande JP, Wolter F (eds) *Principles of knowledge representation and reasoning: Proceedings of the fifteenth international conference, KR 2016, Cape Town, AAAI Press*
- Dauphin J, Satoh K (2019) Explainable ASP. In: Matteo B, Mehdi D, Beishui L, Yuko S, Rym Z-W (eds) *Proceedings of 22nd international conference on principles and practice of multi-agent systems (PRIMA 2019)*, volume 11873 of *Lecture Notes in Computer Science*. Springer
- Faber W, Leone N, Perri S (2012) The intelligent grounder of DLV. In: Erdem E, Lee J, Lierler Y, Pearce D (eds) *Correct reasoning—essays on logic-based AI in Honour of Vladimir Lifschitz, vol 7265*. *Lecture Notes in Computer Science*. Springer, pp 247–264
- Francesconi E, Governatori G (2022) Patterns for legal compliance checking in a decidable framework of linked open data. *Artif Intell Law* (to appear)
- Fungwacharakorn W, Tsumishima K, Satoh K (2020) On the legal revision in PROLEG program. In: Yada K, Katagami D, Takama Y, Ito T, Abe A, Sato-Shimokawara E, Mori J, Matsumura N, Kashima H (eds) *Advances in artificial intelligence—selected papers from the annual conference of japanese society of artificial intelligence, JSAI 2020, Kumamoto-ken, Japan, 9–12 June 2020*, volume 1357 of *Advances in Intelligent Systems and Computing*. Springer
- Gabbay D, Horty J, Parent X, van der Meyden R, van der Torre L (2013) *Handbook of deontic logic and normative systems*. College Publications
- García AJ, Simari GR (2004) Defeasible logic programming: an argumentative approach. *Theory Pract Logic Program* 4(1–2):95
- Gebser M, Pührer J, Schaub T, Tompits H (2008) A meta-programming technique for debugging answer-set programs. In: *Proceedings of the 23rd national conference on artificial intelligence - Volume 1*. AAAI Press
- Gebser M, Kaminski R, König A, Schaub T (2011) *Advances in gringo series 3*. In: James PD, Wolfgang F (eds) *Logic programming and nonmonotonic reasoning—11th international conference, LPNMR 2011, Vancouver, Canada, May 16–19, 2011*. *Proceedings*, volume 6645 of *Lecture Notes in Computer Science*. Springer
- Gelfond M, Lifschitz V (1991) Classical negation in logic programs and disjunctive databases. *New Gener Comput* 9:365–385
- Giordano L, Martelli A, Dupré D (2013) Temporal deontic action logic for the verification of compliance to Norms in ASP. In: *Proceedings of the 145th international conference on artificial intelligence and law (ICAIL 2013)*. Association for Computing Machinery
- Governatori G (2015) The rigorous approach to process compliance. In: Kolb J, Weber B, Hallé S, Mayer W, Ghose AK, Grossmann G (eds) *19th IEEE international enterprise distributed object computing workshop*. IEEE Computer Society

- Governatori G, Rotolo A (2006) Logic of violations: a Gentzen system for reasoning with contrary-to-duty obligations. *Australas J Logic* 4:193
- Governatori G, Rotolo A (2019) Time and compensation mechanisms in checking legal compliance. *J Appl Logics IfCoLog J* 6(5):815–846
- Governatori G, Idelberger F, Milosevic Z, Riveret R, Sartor G, Xiwei X (2018) On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artif Intell Law* 26(4):377–409
- Islam MB, Governatori G (2018) Rulers: a rule-based architecture for decision support systems. *Artif Intell Law* 26(4):315–344
- Kaufmann B, Leone N, Perri S, Schaub T (2016) Grounding and solving in answer set programming. *AI Mag* 37(3):25
- Lam H-P, Governatori G (2009) The making of SPINdle. In: Adrian P, Guido G, John H (eds) Proceedings of international symposium on rule interchange and applications (RuleML 2009), <http://spindle.data61.csiro.au/spindle>. Springer
- Leone N, Pfeifer G, Faber W, Eiter T, Gottlob G, Perri S, Scarcello R (2006) The DLV system for knowledge representation and reasoning. *ACM Trans Comput Logic* 7(3):499
- Leone N, Allocca C, Alviano M, Calimeri F, Civili C, Costabile R, Fiorentino A, Fuscà D, Germano S, Labocetta G, Cuteri B, Manna M, Perri S, Reale K, Ricca F, Veltri P, Zangari J (2019) Enhancing DLV for large-scale reasoning. In: Marcello B, Yuliya L, Stefan W (eds) Proceedings of 15th international conference in logic programming and nonmonotonic reasoning LPNMR, volume 11481 of Lecture Notes in Computer Science. Springer
- Leone N, Ricca F (2015) Answer set programming: a tour from the basics to advanced development tools and industrial applications. In: Wolfgang F, Adrian P (eds) Reasoning Web. Web Logic Rules—11th international summer school 2015, Berlin, Germany, July 31–August 4, 2015, Tutorial Lectures, volume 9203 of Lecture Notes in Computer Science. Springer, pp 308–326
- Libal T (2022) The legal editor: a tool for the construction of legal knowledge bases. In: JURIX, volume 362 of Frontiers in artificial intelligence and applications. IOS Press, pp 286–289
- Maher MJ, Rock A, Antoniou G, Billington D, Miller T (2001) Efficient defeasible reasoning systems. *Int J Artif Intell Tools* 10(4):483
- Modgil S, Prakken H (2014) The *ASPIC*⁺ framework for structured argumentation: a tutorial. *Argum Comput* 5(1):31–62
- Nazarenko A, Lévy L, Wyner A (2018) An annotation language for semantic search of legal sources. In: Proceedings of international conference on language resources and evaluation
- Nute D (1988) Defeasible reasoning: a philosophical analysis in prolog. Springer, Netherlands
- Nute D (1994) Defeasible logic. In: Gabbay (ed) Handbook of logic in artificial intelligence and logic programming. Oxford University Press
- Oetsch J, Pührer J, Tompits H (2018) Stepwise debugging of answer-set programs. *Theory Pract Logic Program* 18(1):30
- Palmirani M, Governatori G (2018) Modelling legal knowledge for GDPR compliance checking. In: Legal knowledge and information systems JURIX, volume 313 of Frontiers in Artificial Intelligence and Applications. IOS Press
- Palmirani M, Martoni M, Rossi A, Bartolini C, Robaldo L (2018) Pronto: privacy ontology for legal compliance. In: Proceedings of the 18th European conference on digital government (ECDG)
- Paschke A (2014) Reaction ruleml 1.0 for rules, events and actions in semantic complex event processing. In: Bikakis A, Fodor P, Roman D (eds) Rules on the Web. From Theory to Applications. Springer
- Prakken H, Sartor G (2015) Law and logic: a review from an argumentation perspective. *Artif Intell Law* 227:214–245
- Reale K, Calimeri F, Leone N, Ricca F (2022) Smart devices and large scale reasoning via ASP: tools and applications. In: Cheney J, Perri S (eds) Practical aspects of declarative languages—24th international symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings, vol 13165. Lecture Notes in Computer Science. Springer, pp 154–161
- Robaldo L (2021) Towards compliance checking in reified I/O logic via SHACL. In: Maranhão J, Wyner A (eds) Proceedings of 18th international conference for artificial intelligence and law (ICAAIL 2021). ACM
- Robaldo L, Sun X (2017) Reified input/output logic: combining input/output logic and reification to represent norms coming from existing legislation. *J Logic Comput* 7:2471
- Robaldo L, Villata S, Wyner A, Grabmair M (2019) Introduction for artificial intelligence and law: special issue “natural language processing for legal texts”. *Artif Intell Law* 27(2):113

- Robaldo L, Bartolini C, Palmirani M, Rossi A, Martoni M, Lenzini G (2020) Formalizing GDPR provisions in reified I/O logic: the DAPRECO knowledge base. *J Logic Lang Inf* 29:401
- Sartor G (2009) Legal concepts as inferential nodes and ontological categories. *Artif Intell Law* 17(3):217–251
- Satoh K, Asai K, Kogawa T, Kubota M, Nakamura M, Nishigai Y, Shirakawa K, Takano C (2011) PROLEG: an implementation of the presupposed ultimate fact theory of Japanese Civil Code by PROLOG technology. In: Onada T, Bekki D, McCreedy E (eds) *New frontiers in artificial intelligence*. Springer
- Satoh K, Giordano L, Baldoni M (2021) Implementation of choice of jurisdiction and law in private international law by PROLEG meta-interpreter. In: Baroni P, Benzmüller C, Wáng Y (eds) *Proceedings of 4th international conference in logic and argumentation*, volume 13040 of *Lecture Notes in Computer Science*. Springer
- Satoh K, Kogawa T, Okada N, Omori K, Omura S, Tsuchiya K (2012) On generality of proleg knowledge representation. In: *Proceedings of the 6th international workshop on juris-informatics (JURISIN 2012)*, Miyazaki, Japan
- Satoh K, Kubota M, Nishigai Y, Takano C (2009) Translating the Japanese presupposed ultimate fact theory into logic programming. In: *Proceedings of the 22nd conference on Legal Knowledge and Information Systems JURIX 2009*. IOS Press
- Sun X, Robaldo L (2017) On the complexity of input/output logic. *J Appl Logic* 25:69–88

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Livio Robaldo¹  · **Sotiris Batsakis**^{2,3} · **Roberta Calegari**⁴ · **Francesco Calimeri**⁵ · **Megumi Fujita**⁸ · **Guido Governatori**⁷ · **Maria Concetta Morelli**⁵ · **Francesco Pacenza**⁵ · **Giuseppe Pisano**⁴ · **Ken Satoh**⁶ · **Ilias Tachmazidis**³ · **Jessica Zangari**⁵

Sotiris Batsakis
sbatsakis@isc.tuc.gr; s.batsakis@hud.ac.uk

Roberta Calegari
roberta.calegari@unibo.it

Francesco Calimeri
francesco.calimeri@unical.it

Megumi Fujita
watanahakase@gmail.com

Guido Governatori
guido@governatori.net

Maria Concetta Morelli
maria.morelli@unical.it

Francesco Pacenza
francesco.pacenza@unical.it

Giuseppe Pisano
g.pisano@unibo.it

Ken Satoh
ksatoh@nii.ac.jp

Ilias Tachmazidis
i.tachmazidis@hud.ac.uk

Jessica Zangari
jessica.zangari@unical.it

- 1 Legal Innovation Lab Wales, Swansea University, Swansea, UK
- 2 Technical University of Crete, Chania, Greece
- 3 Huddersfield University, Huddersfield, UK
- 4 University of Bologna, Bologna, Italy
- 5 University of Calabria, Arcavacata, Italy
- 6 National Institute of Informatics of Japan, Tokyo, Japan
- 7 Brisbane, Australia
- 8 Tokyo, Japan