Automating Defeasible Reasoning in Law with Answer Set Programming

Lim How Khang, Avishkar Mahajan, Martin Strecker and Meng Weng Wong

Singapore Management University

Abstract

The paper studies defeasible reasoning in rule-based systems, in particular about legal norms and contracts. We identify rule modifiers that specify how rules interact and how they can be overridden. We then define rule transformations that eliminate these modifiers, leading in the end to a translation of rules to formulas. For reasoning with and about rules, we contrast two approaches, one in a classical logic with SMT solvers, which is only briefly sketched, and one using non-monotonic logic with Answer Set Programming solvers, described in more detail.

Keywords

Knowledge representation and reasoning, Argumentation and law, Computational Law, Defeasible reasoning

1. Introduction

Computer-supported reasoning about law is a longstanding effort of researchers from different disciplines such as jurisprudence, artificial intelligence, logic and philosophy. What originally may have appeared as an academic playground is now evolving into a realistic scenario, for various reasons.

On the *demand* side, there is a growing number of human-machine or machine-machine interactions where compliance with legal norms or with a contract is essential, such as in sales, insurance, banking and finance or digital rights management, to name but a few. Innumerable "smart contract" languages attest to the interest to automate these processes, even though many of them are dedicated programming languages rather than formalisms intended to express and reason about regulations.

On the *supply* side, decisive advances have been made in fields such as automated reasoning and language technologies, both for computerised domain specific languages (DSLs) and natural languages. Even though a completely automated processing of traditional law texts capturing the subtleties of natural language is currently out of scope, one can expect to code a law text in a DSL that is amenable to further processing.

This "rules as code" approach is the working hypothesis of our CCLAW project[1]: law texts are formalised in a DSL called L4 that is sufficiently precise to avoid ambiguities of natural languages

2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22), August 1, 2022

0000-0002-9333-1364 (L. H. Khang); 0000-0002-9925-1533 (A. Mahajan); 0000-0001-9953-9871 (M. Strecker); 0000-0003-0419-9443 (M. W. Wong)

CEUR Workshop Proceedings (CEUR-WS.org)

[1] https://cclaw.smu.edu.sg/

and at the same time sufficiently close to a traditional law text with its characteristic elements such as cross references, prioritisation of rules and defeasible reasoning. Indeed, presenting these features is one of the main topics of this paper. Once a law has been coded in L4, it can be further processed: it can be converted to natural language [1] to be as human-readable as a traditional law text, and efficient executable code can be extracted, for example to perform tax calculations (all this is not the topic of the present paper). It can also be analysed, to find faults in the law text on the meta level (such as consistency and completeness of a rule set), but also on the object level, to decide individual cases.

Overview of the paper The main emphasis of this paper is on the L4 DSL that is currently under definition, which in particular features a formalism for transcribing rules and reasoning support for verifying their properties. The rule language will be dissected in Section 2. We will in particular describe mechanisms for prioritisation and defeasibility of rules that are encoded via specific keywords in law texts. We then define a precise semantics of these mechanisms, by a translation to logic. Classical, monotonic logic has received surprisingly little attention in this area, even though proof support in the form of SAT/SMT solvers has made astounding progress in recent years. It is not developed in detail here, but see [2]. An alternative approach, based on Answer Set Programming, is described in Section 3. In Section 3 we show how to handle defeasibility operators via encodings in Answer Set Programming (ASP) which has only *negation as failure*, interpreted under the *stable model semantics* as the core non-monotonic operator. We conclude in Section 4.

Related work There is a huge body of work both on computer-assisted legal reasoning and (not necessarily related) defeasible reasoning. In a seminal work, Sergot and Kowalski [3, 4] code the British Nationality Act in Prolog, exploiting Prolog's negation as failure for default reasoning.

The Catala language [5], extensively used for coding tax law and resembling more a high-level programming language than a reasoning formalism, includes default rules, which are however not entirely disambiguated during compile time so that run time exceptions can be raised.

An entirely different approach to tool support is taken with the LogiKEy [6] workbench that codes legal reasoning in the Isabelle interactive proof assistant, paving the way for a very expressive formalism. In contrast, we have opted for a DSL with fully automated proofs which are provided by SMT respectively ASP solvers. These do not permit for human intervention in the proof process, which would not be adequate for the user group we target. Symboleo [7] and the NAI Suite [8] emphasise deontic logic rather than defeasible reasoning (the former is so far not considered in our L4 version).

As a result of a long series of logics, [9] and colleagues have developed the Turnip system that is based on a combination of defeasible and deontic logic and which is applied, among others, to modelling traffic rules [10].

It seems vain to attempt an exhaustive review of defeasible reasoning. Before the backdrop of foundational law theory [11], there are sometimes diverging proposals for integrating defeasibility, sometimes opting for non-monotonic logics [12], sometimes taking a more classical stance [13].

On a more practical side, Answer Set Programming (ASP) [14, 15] goes beyond logic programming and increasingly integrates techniques from constraint solving, such as in the sCASP system [16]. In spite of a convergence of SMT and CASP technologies, there are few attempts to use SMT for ASP, see [17]. For the technologies used in our own implementation, please see Section 4.

This paper is an excerpt of a longer publication [2] that contrasts SMT and CASP technologies in more detail and provides full proofs – we here concentrate on the ASP aspect.

2. Reasoning with and about Rules

We can only give a brief outline of the L4 rule format here and defer a more thorough discussion of the L4 language to the full paper [2]. We will illustrate the main concepts with an example, a (fictitious) regulation of speed limits for different types of vehicles, subdivided into class `Car` and its subclass `SportsCar`, furthermore classes `Day` and `Road`. We will in particular be interested in specifying the maximal speed `maxSp` of a vehicle on a particular day and type of road, and this will be the purpose of the rules.

In its most complete form, a *rule* is composed of a list of variable declarations introduced by the keyword `for`, a precondition introduced by `if` and a post-condition introduced by `then`. Figure 1 gives an example of rules of our speed limit scenario, stating, respectively, that the maximal speed of cars is 90 km/h on a workday, and that they may drive at 130 km/h if the road is a highway. Note that in general, both pre- and post-conditions are Boolean formulas that can be arbitrarily complex, thus are not limited to conjunctions of literals in the preconditions or atomic formulas in the post-conditions. Rules whose precondition is `true` can be written as `fact`.

```
rule <maxSpCarWorkday>
  for v: Vehicle, d: Day, r: Road if isCar v && isWorkday d then maxSp v d r
      90
rule <maxSpCarHighway>
  for v: Vehicle, d: Day, r: Road if isCar v && isHighway r then maxSp v d r
      130
```

Figure 1: Rules of speed limit example

```
assert <maxSpFunctional> {SMT: {valid}}
   maxSp instCar instDay instRoad instSpeed1 &&
   maxSp instCar instDay instRoad instSpeed2
   --> instSpeed1 == instSpeed2
```

Figure 2: Assertions of speedlimit example

The purpose of our formalization efforts is to be able to make assertions and prove them, such as the statement in Figure 2 which claims that the predicate `maxSp` behaves like a function, *i.e.* given the same car, day and road, the speed will be the same. Instead of a universal quantification, we here use variables `inst...` that have been declared globally, because they

produce more readable (counter-)models.

Given a plethora of different notions of *defeasibility*, we had to make a choice as to which notions to support, and which semantics to give to them. We will here concentrate on two concepts, which we call *rule modifiers*, that limit the applicability of rules and make them "defeasible". They will be presented informally in the following. A semantics based on Answer Set Programming will be provided in Section 3.

We will concentrate on two rule modifiers that restrict the applicability of rules and that frequently occur in law texts: *subject to* and *despite*, further illustrated by our running example.

Example 1. *The rules* `maxSpCarHighway` *and* `maxSpCarWorkday` *are not mutually exclusive and contradict another because they postulate different maximal speeds. For disambiguation, we would like to say:* `maxSpCarHighway` *holds* despite *rule* `maxSpCarWorkday`*. In L4, rule modifiers are introduced with the aid of* rule annotations*, with a list of rule names following the keywords* `subjectTo` *and* `despite`*. Thus, we modify rule* `maxSpCarHighway` *of Figure 1 with*

```
rule <maxSpCarHighway> {restrict: {despite: maxSpCarWorkday}}
# rest of rule unchanged
```

Furthermore, to the delight of the public of the country with the highest density of sports cars, we also introduce a new rule `maxSpSportsCar` *that holds* subject to `maxSpCarWorkday` *and* despite `maxSpCarHighway`*:*

```
rule <maxSpSportsCar>
  {restrict: {subjectTo: maxSpCarWorkday, despite: maxSpCarHighway}}
  for v: Vehicle, d: Day, r: Road
  if isSportsCar v && isHighway r then maxSp v d r 320
```

We will now give an informal characterization of these modifiers:

- r_1 *subject to* r_2 and r_1 *despite* r_2 are complementary ways of expressing that one rule may override the other rule. They have in common that r_1 and r_2 have contradicting conclusions. The conjunction of the conclusions can either be directly unsatisfiable (such as: "may hold" vs. "must not hold") or unsatisfiable *w.r.t.* an intended background theory (obtaining different maximal speeds is inconsistent when expecting `maxSp` to be functional in its fourth argument).
- Both modifiers differ in that *subject to* modifies the rule to which it is attached, whereas *despite* has a remote effect on the rule given as argument.
- They permit to structure a legal text, favouring conciseness and modularity: In the case of *despite*, the overridden, typically more general rule need not be aware of the overriding, typically subordinate rules.
- Even though these modifiers appear to be mechanisms on the meta-level in that they reasoning about rules, they can directly be reflected on the object-level.

3. Defeasible Reasoning with Answer Set Programming

3.1. Introduction

The purpose of this section is to give an account of the work we have been doing using Answer Set Programming (ASP) to formalize and reason about legal rules. This approach is complementary to the one described before using SMT solvers. Our intention is to present how some core legal reasoning tasks can be implemented in ASP while keeping the ASP representation readable and intuitive and respecting the idea of having an 'isomorphism' between the rules and the encoding. Please see the appendix for a brief overview of ASP and references for further reading.

Our work in this section is inspired by [18] and we borrow some of their notation/terminology. Readers will note that there are similarities between the use of predicates such as $according\_to$, $defeated$, $opposes$ in our ASP encoding, to reason about rules interacting with each other, and similar predicates that the authors of [18] use in their work. However our ASP implementation is much more specific to legal reasoning whereas they seek to implement very general logic based reasoning mechanisms. We independently developed our 'meta theory' for how rule modifiers interact with the rules and with each other and there are further original contributions like a proposed axiom system for what we call 'legal models'. An interesting avenue of future work could be to compare our approaches within the framework of legal reasoning.

The work in this section builds on the work in [19] and hence uses many of the same predicates/notation and terminology.

3.2. Formal Setup

Let the tuple $Config = (R, F, M, I)$ denote a $configuration$ of legal rules. The set R denotes a set of rules of the form $pre\_con(r) \rightarrow concl(r)$. These are 'naive' rules with no information pertaining to any of the other rules in R. F is a set of positive atoms that describe facts of the legal scenario we wish to consider. M is a set of the binary predicates $despite$, $subject\_to$ and $strong\_subject\_to$. I is a collection of minimal inconsistent sets of positive atoms. Henceforth for a rule r, we may write C_r for its conclusion $Concl(r)$.

Note that, throughout this section, given any rule r, C_r is assumed to be a single positive atom. That is, there are no disjunctions or conjunctions in rule conclusions. Also any rule pre-condition ($pre\_con(r)$) is assumed to be a conjunction of positive and negated atoms. Here negation denotes 'negation as failure'.

Throughout this document, whenever we use an uppercase or lowercase letter (like r, r_1, R etc.) to denote a rule that is an argument, in a binary predicate, we mean the unique integer rule ID associated with that rule. The binary predicate $legally\_valid(r, c)$ intuitively means that the rule r is 'in force' and it has conclusion c. Here r typically is an integer referring to the rule ID and c is the atomic conclusion of the rule. The unary predicate $is\_legal(c)$ intuitively means that the atom c legally holds/has legal status. The predicates $despite$, $subject\_to$ and $strong\_subject\_to$ all cause some rules to override others. Their precise properties will be given next.

3.3. Semantics

A set S of is_legal and $legally\_valid$ predicates is called a *legal model* of $Config = (R, F, M, I)$, if and only if

(A1) $\forall f \in F\ is\_legal(f) \in S$.

(A2) $\forall r \in R$, if $legally\_valid(r, C_r) \in S$. then $S \models is\_legal(pre\_con(r))$ and $S \models is\_legal(C_r)$ [2]

(A3) $\forall c$, if $is\_legal(c) \in S$, then either $c \in F$ or there exists $r \in R$ such that $legally\_valid(r, C_r) \in S$ and $c = C_r$.

(A4) $\forall r_i, r_j \in R$, if $despite(r_i, r_j) \in M$ and $S \models is\_legal(pre\_con(r_j))$, then $legally\_valid(r_i, C_{r_i}) \notin S$

(A5) $\forall r_i, r_j \in R$, if $strong\_subject\_to(r_i, r_j) \in M$ and $legally\_valid(r_i, C_{r_i}) \in S$, then $legally\_valid(r_j, C_{r_j}) \notin S$

(A6) $\forall r_i, r_j \in R$ if $subject\_to(r_i, r_j) \in M$, and $legally\_valid(r_i, C_{r_i}) \in S$ and there exists a minimal conflicting set $k \in I$ such that $C_{r_i} \in k$ and $C_{r_j} \in k$ and $is\_legal(k \setminus \{C_{r_j}\}) \subseteq S$, then $legally\_valid(r_j, C_{r_j}) \notin S$. Note than in our system, any minimal inconsistent set must contain at least 2 atoms. [3]

(A7) $\forall r \in R$, if $S \models pre\_con(r)$, but $legally\_valid(r, C_r) \notin S$, then it must be the case that at least one of A4 or A5 or A6 has caused the exclusion of $legally\_valid(r, C_r)$. That is if $S \models pre\_con(r)$, then unless this would violate one of A5, A6 or A7, it must be the case that $legally\_valid(r, C_r) \in S$.

3.4. Some remarks on axioms A1–A7

We now give some informal intuition behind some of the axioms and their intended effects.

A1 says that all facts in F automatically gain legal status, that is, they legally hold. The set F represents indisputable facts about the legal scenario we are considering.

A2 says that if a rule is 'in force' then it must be the case that both the pre-condition and conclusion of the rule have legal status. Note that it is not enough if simply require that the conclusion has legal status as more than one rule may enforce the same conclusion or the conclusion may be a fact, so we want to know exactly which rules are in force as well as their conclusions.

A3 says that anything that has legal status must either be a fact or be a conclusion of some rule that is in force.

A4–A6 describe the semantics of the three modifiers. The intuition for the three modifiers will be discussed next. Firstly, it may help the reader to read the modifiers in certain ways. $despite(r_i, r_j)$ should be read as 'despite r_i, r_j'. Thus r_i here is the 'subordinate rule' and

[2] By $S \models is\_legal(pre\_con(r))$ we mean that for each positive atom b_i in the conjunction, $is\_legal(b_i) \in S$ and for each negation-as-failure body atom $not\ b_j$ in the conjunction $is\_legal(b_j) \notin S$

[3] For a set of atoms A, by $is\_legal(A)$, we mean the set $\{is\_legal(a) \mid a \in A\}$

r_j is the 'dominating' rule. The idea here is that once the precondition of the dominating rule r_j is true, it invalidates the subordinate rule r_i regardless of whether the dominating rule itself is then invalidated by some other rule. For *strong subject to*, the intended reading for $strong\_subject\_to(r_i, r_j)$ is something like '(strong) subject to r_i, r_j'. Here r_i can be considered the dominating rule and r_j the subordinate. Once the dominating rule is in force, then it invalidates the subordinate rule. The intended reading for $subject\_to(r_i, r_j)$ is 'subject to r_i, r_j'. For the subordinate rule r_j to be invalidated, it has to be the case that the dominating rule r_i is in force and there is a minimal inconsistent set k in I that contains the two atoms in the conclusions of the two rules and, $is\_legal(k \setminus \{C_{r_j}\}) \subseteq S$. These minimal inconsistent sets along with the *subject to* modifier give us a way to incorporate a classical-negation-like effect into our system. We are able to say which things contradict each other. Note that in our system, if say $\{a, b\}$ is a minimal inconsistent set, then it is possible for both $is\_legal(a)$ and $is\_legal(b)$ to be in a single legal model, if they are both facts or they are conclusions of rules that have no modifiers linking them. These minimal inconsistent sets only play a role where a $subject\_to$ modifier is involved. The reason for doing this is that this offers greater flexibility rather than treating a and b as pure logical negatives of each other that cannot be simultaneously true in a legal model. We will give examples later on to illustrate these modifiers.

A7 says essentially that A4–A6 represent the only ways in which a rule whose pre-condition is true may nevertheless be invalidated, and any rule whose precondition is satisfied and is not invalidated directly by some instance of A4–A6, must be in force.

Note that there maybe legal rule configurations for which no legal models exist. See the appendix for a discussion of some 'pathological' rule configurations.

3.5. ASP encoding

Here is an ASP encoding scheme for a configuration $Config = (R, F, M, I)$ of legal rules.

```
% For any f in F, we have:
is_legal(f).
% All the modifiers get added as facts like for example:
despite(1,2).
% Any rule r in R is encoded using the general schema:
according_to(r,C_r):-is_legal(pre_con(r)).
% Given a minimal inconsistent set {a_1,a_2,...,a_n}, this corresponds to a
    set of rules:
opposes(a_1,a_2):-is_legal(a_2),is_legal(a_3),...,is_legal(a_n).
opposes(a_1,a_3):-is_legal(a_2),is_legal(a_4)...,is_legal(a_n). % etc ...
opposes(a_n-1,a_n):-is_legal(a_1),...,is_legal(a_n-2).
% Opposes is a symmetric relation
opposes(X,Y):-opposes(Y,X).
% Encoding for 'despite'
defeated(R,C,R1) :-
   according_to(R,C), according_to(R1,C1), despite(R,R1).
%Encoding for 'subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   opposes(C,C1), subject_to(R1,R).
% Encoding for 'strong_subject_to'
```

```
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   strong_subject_to(R1,R).

not_legally_valid(R) :- defeated(R,C,R1).
legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).
is_legal(C):-legally_valid(R,C).
```

3.6. Proposition

Proposition 1. *For a configuration $Config = (R, F, M, I)$, let the above encoding be the program ASP_{Config}. Then given an answer set A_{Config} of ASP_{Config} let $S_{A_{Config}}$ be the set of is_legal and $legally\_valid$ predicates in A_{Config}. Then $S_{A_{Config}}$ is a legal model of $Config$.*

Proof See Appendix of full paper. □

3.7. Example

Let us see how the running example would work in the ASP setting. We have 3 rules encoded as below, there are no minimal inconsistent sets. There are 3 modifiers: $despite(2, 3)$, $strong\_subject\_to(1, 3)$, $strong\_subject\_to(1, 2)$

```
despite(2,3).
strong_subject_to(1,3).
strong_subject_to(1,2).
according_to(1,max_spd(v,d,r,90)):-is_legal(is_workday(d)),
is_legal(is_car(v)).
according_to(2,max_spd(v,d,r,130)):-is_legal(is_highway(r)),
  is_legal(is_car(v)).
according_to(3,max_spd(v,d,r,320)):-is_legal(is_highway(r)),
  is_legal(is_sports_car(v)).

% Encoding for 'despite'
defeated(R,C,R1) :-
   according_to(R,C), according_to(R1,C1), despite(R,R1).
%Encoding for 'subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   opposes(C,C1), subject_to(R1,R).
% Encoding for 'strong_subject_to'
defeated(R,C,R1) :-
   according_to(R,C), legally_valid(R1,C1),
   strong_subject_to(R1,R).

not_legally_valid(R) :- defeated(R,C,R1).
legally_valid(R,C):-according_to(R,C),not not_legally_valid(R).
is_legal(C):-legally_valid(R,C).
```

When the initial set of facts F is the set

```
is_legal(is_workday(d)).
is_legal(is_car(v)).
is_legal(is_highway(r)).
is_legal(is_sports_car(v)).
```

we get exactly one legal max speed given by

```
is_legal(max_spd(v,d,r,90)).
```

One can check this by adding the rule

```
legal_max_spd(X):- is_legal(max_spd(v,d,r,X)).
```

and running the s(CASP) query $?-legal\_max\_spd(X).$, which returns the binding $X = 90$. This is the unique legal maximum speed which can be seen via use of the rule

```
legal_max_spd(X):- X > 90, is_legal(max_spd(v,d,r,X)).
```

Now running the query as above we see that there is no solution. When removing $is\_legal(is\_workday(d))$ from F, we get exactly one legal max speed of 320, and when $is\_legal(is\_sports\_car(v))$, and $is\_legal(is\_workday(d))$ are both removed from F we get exactly one exactly legal max speed of 130.

4. Conclusions

This paper has discussed different approaches for representing defeasibility as used in law texts, by annotating rules with modifiers that explicate their relation to other rules. We have notably presented encodings based on Answer Set Programming, in Section 3. All the encodings are motivated by the need to explore the implementation of various forms of defeasibility in logics for which well developed and powerful solvers such as SMT solvers and ASP solvers already exist. The ASP based and SMT based approaches are complimentary to each other. ASP's closed-world-assumption is useful when a legal verdict must be reached with potentially incomplete information, when the truth value of every atom is not explicitly known. For example in the speed limit example, unless it is known explicitly that the car in question was a sports car, the ASP solver would always return a speed limit lower than 320. The SMT solver would generate two models, one in which the car is a sports car and one in which it is not. This could lead to two different speed limits, hence requiring human intervention to determine the true speed limit in the scenario being considered. On the other hand, when one does model checking or wants to reason about meta-properties of the rules such as soundness of the rule-set, the SMT solver approach is more suited than the ASP approach. Intuitively, here we want to reason over all possible scenarios rather than restricting the reasoning to a particular scenario being considered.

However, we are still at the beginning of the journey. To allow the two approaches to work together coherently and seamlessly, a theoretical comparison of the classical and ASP semantics presented here still has to be carried out, and it has to be propped up by an empirical evaluation. For this purpose, we are currently in the process of coding some real-life law texts in L4, such

as Singapore's Personal Data Protection Act[4]. An implementation of the L4 ecosystem is under way[5], providing a transpilation of L4 rules to both the SMT and the ASP world. The interaction with SMT solvers is done through an SMT-LIB [20] interface. Advanced solvers, such as Z3 [21], provide good support for quantification.

Acknowledgments

The contributions of the members of the L4 team to this common effort are thankfully acknowledged, in particular of Jason Morris who contributed his experience with Answer Set Programming; of Jacob Tan and Ruslan Khafizov who have participated in discussions about its contents and commented on the paper; and of Liyana Muthalib who has proof-read a previous version.

This research is supported by the National Research Foundation (NRF), Singapore, under its Industry Alignment Fund — Pre-Positioning Programme, as the Research Programme in Computational Law. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

References

[1] I. Listenmaa, M. Hanafiah, R. Cheong, A. Källberg, Towards CNL-based verbalization of computational contracts, in: Proceedings CNL, 2021.

[2] H. K. Lim, A. Mahajan, M. Strecker, M. W. Wong, Automating defeasible reasoning in law, 2022. URL: https://arxiv.org/abs/2205.07335.

[3] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, H. T. Cory, The british nationality act as a logic program, CACM 29 (1986) 370–386.

[4] R. A. Kowalski, Legislation as logic programs, in: Informatics and the Foundations of Legal Reasoning, Springer, 1995, pp. 325–356.

[5] D. Merigoux, N. Chataing, J. Protzenko, Catala: a programming language for the law, Proc. ACM Program. Lang. 5 (2021) 1–29.

[6] C. Benzmüller, A. Farjami, D. Fuenmayor, P. Meder, X. Parent, A. Steen, L. van der Torre, V. Zahoransky, LogiKEy workbench: Deontic logics, logic combinations and expressive ethical and legal reasoning, Data in Brief 33 (2020) 106409.

[7] S. Sharifi, A. Parvizimosaed, D. Amyot, L. Logrippo, J. Mylopoulos, Symboleo: Towards a specification language for legal contracts, in: RE, IEEE, 2020, p. 364–369. doi:10.1109/RE48521.2020.00049.

[8] T. Libal, A. Steen, The NAI suite - drafting and reasoning over legal texts, in: M. Araszkiewicz, V. Rodríguez-Doncel (Eds.), Proceedings JURIX, volume 322 of *Frontiers in AI and Applications*, IOS Press, 2019, pp. 243–246.

[9] G. Governatori, F. Olivieri, Unravel legal references in defeasible deontic logic, in: Proc. International Conference on Artificial Intelligence and Law (ICAIL), 2021.

[4] https://sso.agc.gov.sg/Act/PDPA2012

[5] https://github.com/smucclaw/baby-l4

[10] H. Bhuiyan, G. Governatori, A. Bond, S. Demmel, M. B. Islam, A. Rakotonirainy, Traffic rules encoding using defeasible deontic logic, in: V. Serena, J. Harasta, P. Kremen (Eds.), Proceedings JURIX, IOS Press, 2020, p. 3–12.
[11] H. L. A. Hart, The concept of law, Clarendon Press, 1997.
[12] J. Hage, Law and defeasibility, Artificial Intelligence and Law 11 (2003) 221–243. doi:10.1023/B:ARTI.0000046011.13621.08.
[13] C. E. Alchourrón, D. Makinson, Hierarchies of Regulations and their Logic, Springer, Dordrecht, 1981, p. 125–148.
[14] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012.
[15] J. Arias, M. Carro, Z. Chen, G. Gupta, Constraint answer set programming without grounding and its applications, in: M. Alviano, A. Pieris (Eds.), Datalog 2.0, volume 2368 of *CEUR Workshop Proceedings*, CEUR-WS.org, Philadelphia, PA (USA), 2019, pp. 22–26. URL: http://ceur-ws.org/Vol-2368/paper2.pdf.
[16] J. Arias, Advanced Evaluation Techniques for (Non)-Monotonic Reasoning Using Rules with Constraints, Ph.D. thesis, Universidad Politécnica de Madrid, 2019.
[17] D. Shen, Y. Lierler, SMT-based constraint answer set solver EZSMT+ for non-tight programs, in: Proceedings KR, AAAI Press, 2018, p. 67–71.
[18] H. Wan, B. N. Grosof, M. Kifer, P. Fodor, S. Liang, Logic programming with defaults and argumentation theories, in: Proc. ICLP, Springer, 2009, pp. 432–448.
[19] J. P. Morris, Constraint answer set programming as a tool to improve legislative drafting: A rules as code experiment, in: Proceedings ICAIL, 2021.
[20] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), http://smtlib.cs.uiowa.edu, 2016.
[21] L. De Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Proceedings TACAS, Springer, 2008, p. 337–340.