CEUR-WS.org/Vol-3428/short4.pdf

Explainable Answer Set Programming for Legal Decision Support

Daniele Theseider Dupré*

DISIT, Università del Piemonte Orientale, Viale Michel 11, 15125 Alessandria, Italy

Abstract

Computational models of arguments have been shown to be useful for decision support in legal cases; in particular, the ANGELIC methodology was successfully used to develop a Prolog program which is able to predict decisions of the European Court of Human Rights with high accuracy — significantly better than the one achieved by machine learning approaches. In addition, the program provides as explanation a trace in natural language of the reasoning steps, quite valuable in order to make decision support trustworthy. The result is based on the backtracking execution model of Prolog programs including cuts and the built-in predicate for writing. This paper introduces an alternative approach, based on Answer Set Programming and the *xclingo* system for solving and, at the same time, generating explanations for solutions. As usual for ASP vs Prolog, in this proposal representation and inference are less tightly coupled, so that the same representation can be used for different inference tasks. Similarly, explanation in *xclingo* comes from annotations which are not directly part of the representation, being, for example, ignored by a "pure" ASP solver.

Introduction

The use of Computational Logic for legal reasoning dates back to the formalization of the British Nationality Act as Prolog rules [1]. In the last decade, Al-Abdulkarim et al. [2] defined a methodology for designing Prolog programs for deciding legal cases, based on computational models of arguments, and, in particular, Abstract Dialectical Frameworks (ADFs) [3] to represent the *factors* to be taken into account and their relations.

The approach has been applied by Collenette et al. [4, 5] for reproducing the decisions of the European Court of Human Rights (ECtHR) related to a specific article (Art.6) of the European Convention on Human Rights (ECHR). Some effort was required in developing the representation, but the resulting system achieved a high accuracy (97%) in predicting the outcomes, compared with a 70-85% accuracy of machine learning approaches.

An even more important advantage lies in the fact that the structured representation in the ADF is used as a basis for *explanation* of the results of reasoning, which is of utmost importance for making an automated system acceptable and trustworthy for users in this domain — as well as in other ones. A system based on logical rules is, in fact, much more a *white box* with respect to a predictor or classifier resulting from machine learning, especially in case modelers introduce intermediate concepts that are important for humans in relating inputs to a conclusion

CILC'23: 38th Italian Conference on Computational Logic, June 21–23, 2023, Udine, Italy
dtd@uniupo.it (D. Theseider Dupré)
0000-0001-6798-4380 (D. Theseider Dupré)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

and are therefore used in the logical rules. An explanation could then relate the inputs to the conclusion in terms of such concepts.

In the Prolog programs, both inference and explanation are achieved relying on the backtracking execution model, using cuts and the built-in predicate for writing. Therefore, the program is not exactly a model of the domain, while it works well as a model of reasoning for the purpose of estabilishing the conclusion in a case and for providing a trace of reasoning in natural language. A different implementation was also provided [5], where an ad hoc program for interpreting the specific form of ADF adopted there is used as a back end for a web-based interface for explanation.

In this paper we propose a different approach where:

- Answer Set Programming is used for modelling the domain, so that different forms of reasoning can be provided on the same model as Answer Set Solving: reasoning that is not just predicting the outcome from a complete set of inputs (called *base level factors*, as we shall see), but may include, e.g., how to complete the input, or to modify the input, in order to achieve a given outcome;
- information for explanation is associated with the representation, but not part of it; explanation may be provided by suitable extensions of solvers and/or visualization tools (e.g., *xASP* [6], *xASP2* [7], *clingraph* [8]), and in this paper we consider the *xclingo* [9] extension of the *clingo* solver for Explainable Answer Set Programming, where annotations of rules and facts can used to provide a tree-like explanation.

In the following we provide more details on the work in [2, 4, 5] and then sketch how some of the goals of such previous work can be dealt with in Explainable Answer Set Programming, pointing out some advantages.

1. The ANGELIC Methodology and ADFs

The ANGELIC methodology described in [2] is based on Abstract Dialectical Frameworks (ADFs) which are defined as follows by Brewka et al. [3]:

Definition 1. *An ADF is a tuple $\langle S, L, C\rangle$ where:*

- *S is a set of statements (nodes);*
- *L is a subset of $S \times S$, a set of links; for $s \in S$, $par(s) = \{s' \in S, (s', s) \in L\}$ is the set of parents of s;*
- *$C = \{C_{s\in S}\}$ is a set of total functions $C_s : 2^{par(s)} \rightarrow \{t, f\}$, one for each statement s; C_s is called the acceptance condition of s.*

Prioritized ADFs are also introduced in [3], where links are partitioned in *supporting* and *attacking* links, and a partial order on S is given, representing *global* preferences among nodes; the semantics of such ADFs is given via a mapping to basic ADFs, providing a general acceptance condition for a node based on the supporting/attacking links and the global preference.

For the ANGELIC methodology, as described in [2, 4], a variant of such ADFs is used, which is essentially an instance of (basic) ADFs where:

- S and L form an acyclic graph and, in particular, one or more trees; therefore, different semantics for ADFs coincide;
- there are supporting and attacking links for nodes (similarly to prioritized ADFs);
- explicit acceptance conditions (like in plain ADFs) are provided, for each internal node, as a series of rules whose order, local to a node s, provides a form of priority.

More in detail, a tree root corresponds to a main decision, other nodes to *factors* relevant for analysing a case and establishing the decision; tree leaves are *base level factors* whose truth has to be established in ways that are not provided in this model. Given the tree form of ADFs, the term *children* is used for the nodes on which the acceptance of a node depends (they would be *parents* in the ADF terminology). Even considering the polarity of links, the tree does not contain all information for determining the acceptance of nodes, unlike in prioritized ADF; rather, as in plain ADFs, an explicit acceptance condition is given for each internal `Node`, as a *sequence* of rules of the form:

```
Accept/Reject Node if Condition1
...
Accept/Reject Node if ConditionN
```

where the conditions are conjunctions of literals built from *children* of `Node` in the tree, and the order of rules means that one can be applied if the previous ones cannot. In some cases, accepting and rejecting rules are alternated, and the last rule is used, with an empty precondition, to provide a default value for `Node`, e.g. :

```
Accept Node if Condition1
Reject Node if Condition2
Accept Node
```

The priority based on the order means that the preconditon of a rule implicitly contains the negation of the preconditions for previous rules. In the associated Prolog program, this is of course realized using cuts; in [2], negation as failure is stated to be used for the negative literals in the conditions, while in [4] a `valid/invalid` argument is used in atoms of the form `factor_i(Case,valid/invalid)`, meaning that `factor_i` holds / does not hold for the case described by `Case`, and rules are provided for both the `valid` and `invalid` case, at least to provide part of the explanation; negation as failure is only used to check whether a base level factor is not member of the list of base level factors that have been established to hold.

Based on an example in [4], the corresponding of the abstract rules above would be:

```
 node_i(Case,valid):- condition1(..), write("...")), nl,!.
 node_i(Case,invalid):- condition2(..), write("...")), nl,!.
 node_i(Case,valid):- write("...")), nl.
```

writing suitable strings, stating that the factor corresponding to the node is found to be present or not. In [2], cuts are not used. The output resulting from the single solution (or the first solution, with no cuts) provides a *textual explanation* including, for example, for a case in the domain of Art.6 of the ECHR, "The case is well founded...", "The case is therefore admissible".

Even though "input cases determine the labelling of base level factors (the leaf nodes)" [2] "each case is represented by a list of base level factors" [4], as a consequence of the priority based on rule order, there are cases where the value of some base level factor is irrelevant (at

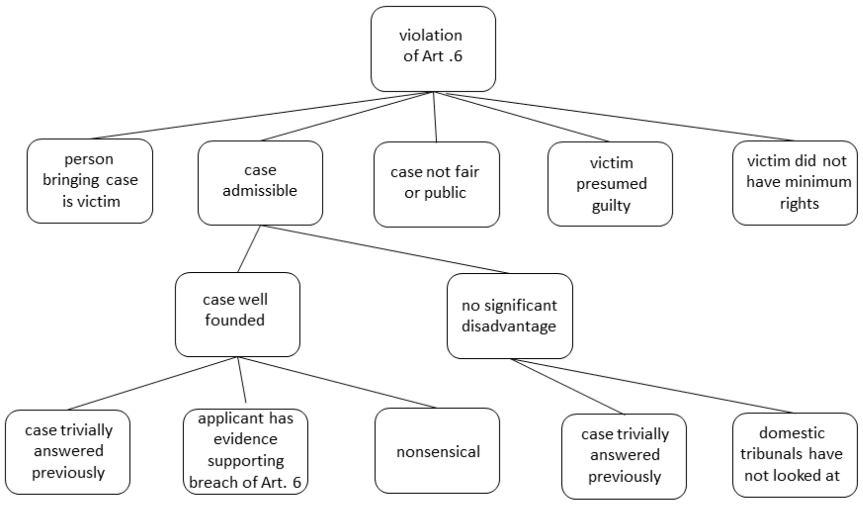
violation of Art .6
person bringing case is victim
case admissible
case not fair or public
victim presumed guilty
victim did not have minimum rights
case well founded
no significant disadvantage
case trivially answered previously
applicant has evidence supporting breach of Art. 6
nonsensical
case trivially answered previously
domestic tribunals have not looked at

Figure 1: Tree structure of the ADF fragment

least in case only the first solution is searched, if cuts are not used): in the example above, the ones for establishing `Condition2` in case `Condition1` holds.

The approach in [5] slightly departs from the earlier work: the acceptance conditions for a node are given as boolean formulae, with no priorities.

The work on the ECHR is based on "law and guidance documents that the ECtHR has provided, rather than analysing individual cases" [5]; the latter is the way to go in *case law* or *common law* used in several countries including UK and USA (considered in [2]), as opposed to *civil law* based on codified principles. In the context of civil law, deriving a model might be easier; in this short paper we do not address the differences between the two contexts, assuming that knowledge is available in one of the ADF variants in the papers cited above.

As pointed out in [5], devising an ADF makes it easier to ensure that all relevant factors are taken into account, and that no conflicts occur: the factors on which a node depends are made explicit in the tree, and potential conflicts are resolved locally to such nodes (possibly using priority of rules). The tree structure is also a guidance for changing the knowledge base.

2. Modeling, Reasoning and Explaining in the Legal Domain with Explainable ASP

We now sketch how some of the goals of the work described in the previous section can be achieved in Explainable Answer Set Programming. We consider the fragment of ADF in [4], table 1 (with some correction based on [5]), with the following textual representation and the tree structure in figure 1:

Reject `Violation of Art 6` **if** `person bringing case is not a victim`
Reject `Violation of Art 6` **if** `not case admissible`
Accept `Violation of Art 6` **if** `case was not fair or public`

Accept `Violation of Art 6` **if** `victim was presumed guilty`
Accept `Violation of Art 6` **if** `victim did not have the minimum rights`
Reject `Violation of Art 6` **otherwise**

Reject `case admissible` **if** `not case well founded`
Reject `case admissible` **if** `(there was) no significant disadvantage`
Accept `case admissible` **otherwise**

Accept `case well founded` **if** `not case was trivially answered previously`
Accept `case well founded` **if** `applicant has evidence supporting breach of art. 6`
Accept `case well founded` **if** `not nonsensical`
Reject `case well founded` **otherwise**

Accept `no significant disadvantage` **if** `no reason to look at the case`
Accept `no significant disadvantage` **if** `there are domestic tribunals that have not looked at the case`
Reject `no significant disadvantage` **otherwise**

The factors for which no rule is provided in this fragment (tree leaves) are considered base level ones. An ASP model corresponding to the rules above can be provided as follows, using explicit negation (to correspond to the valid/invalid factors in [4]), default negation, and, where useful, additional atoms to represent that a previous set of rules does not apply.

```
-violationart6 :- c11.

c11 :- -person_bringing_case_is_victim.
c11 :- -case_admissible.

-c11 :- person_bringing_case_is_victim, case_admissible.

violationart6 :- c12.

c12 :- -c11, case_not_fair_or_public.
c12 :- -c11, victim_is_presumed_guilty.
c12 :- -c11, victim_did_not_have_minimum_rights.

-c12 :- -case_not_fair_or_public,-victim_is_presusmall_guilty,
        -victim_did_not_have_minimum_rights.

-violationart6 :- -c11, -c12.

-case_admissible :- -case_well_founded.
-case_admissible :- no_significant_disadvantage.
case_admissible :- not -case_admissible.

case_well_founded :- -case_trivially_answered_previously.
case_well_founded :- applicant_has_evidence_supporting_breach_of_art6.
case_well_founded :- -nonsensical.
-case_well_founded :- not case_well_founded.

no_significant_disadvantage :- no_reason_to_look_at.
no_significant_disadvantage :- domestic_tribunals_have_not_looked_at.
-no_significant_disadvantage :- not no_significant_disadvantage.
```

The model can be used to predict the decision for a *case description* providing exactly one of the facts `f` , `-f` for each base level factor `f`. Adding, instead, choice rules `1 {f;-f} 1` leads to considering all possible combinations of base level factors, i.e., all possible case descriptions. In this case, given that the model is derived from an ADF, checking that for all case descriptions exactly one of the decisions is derived may not be essential (except for ensuring absence of errors in modeling).

The model with choice rules could however be used for analysing the domain (or, more precisely, the domain model). For example, we might want to find for teaching purposes case descriptions that satisfy some given condition, e.g., a case that is admissible but for which there is no violation of Art. 6, which can be done adding:

```
:- not -violationart6.
:- not case_admissible.
```

Another example is the one of a lawyer analyizing a case where base level factors are not all clearly established, in order to find which ones should be supported with evidence in order to predict a "violation" outcome; in that case, of course, the constraint `:- not violationart6.` should be imposed, and the known base level factors asserted.

The use of explicit and default negation could also allow, as usual, distinguishing whether a factor has been established to be false or it has not been established to hold. This may not be essential if the two cases lead to the same decision: e.g., when the burden of proof for a factor is on one side, if that side does not provide a proof, decision should proceed as if the factor is false. In the ASP model above, for internal nodes, default rules `f :- not -f` (or viceversa) are used, and, in general, to be closer to [4] the rules are intended to derive a conclusion for a complete case description. But for base level factors, there could be a mix of known ones and unknown ones; using `1 {f;-f} 1` for unknown ones, cautious reasoning could conclude a decision even in case some base factor is unknown.

2.1. Adding Explanation

The *xclingo* system[1] [9] provides ways for producing explanations for the solutions of an ASP solver, in the form of trees built from annotations of some of the rules in the ASP model and/or some of the atoms. Annotations are on lines starting with `%!` and are therefore considered comments by a non-explaining ASP solver, while *xclingo* is based on a translation and the use of the *clingo* Python API[2]. In this paper we rely on atom annotations, which can be kept separate from the model itself (e.g. in a separate file). The annotations:

```
%!show_trace violationart6.
%!show_trace -violationart6.
```

are used to mean that, in case the corresponding atoms are in an answer set, we are interested in explaining how they are derived (multiple explanations could be given for the same atom). Annotations such as:

```
%!trace "Violation of art. 6" violationart6.
%!trace "No violation of art. 6" -violationart6.
```

[1]https://github.com/bramucas/xclingo
[2]https://potassco.org/clingo/python-api/current/clingo

are used for the atoms corresponding to the nodes in the ADF, to mean that if they are used in a proof, the associated string should be included in the tree-like explanation. The auxiliary atoms associated with groups of rules are not annotated.

The following are the explanations obtained for three case descriptions; in each of them a different group of reject/accept clauses is applied for the root node of the ADF.

```
*
|__No violation of art. 6
|  |__the case is not admissible
|  |  |__there was no significant disadvantage
|  |  |  |__there are domestic tribunals that have not looked at the case

*
|__Violation of art. 6
|  |__the case is admissible
|  |__the person bringing the case is the victim
|  |__the case was not fair and public

*
|__No violation of art. 6
|  |__the case is admissible
|  |__the person bringing the case is the victim
|  |__the case was fair and public
|  |__the victim was not presumed guilty
|  |__the victim had the minimum rights
```

In the first one, the trace why the case is not admissible is provided, via a further internal node of the ADF (there was no significant disadvantage). The tree-like explanations provided by *xclingo* — even though in textual form, in the current version — are indeed particularly suited for tree-like ADFs. In the second one, the reasons why the first two "reject" rules did not apply are given, together with the reason why an "accept" rule does. The last one is a case where the default "reject" rule applies.

3. Conclusions

We have sketched how legal reasoning with tree-form textual explanation can be achieved in Explainable Answer Set Programming. This could be a step in order to achieve results similar to the ones in [4, 5] with some advantages. As usual, the availability of a model of a domain allows for reasoning that is not just predicting an output for a complete set of inputs. Other ASP tools for explanation and visualization can be used to provide further ways for presenting the results to non-technical users. Moreover, using a representation and reasoning framework like ASP could lead to integrating knowledge corresponding to ADFs with other forms of knowledge, and to rely on extensions of the representation and reasoning framework, without implementing ad-hoc reasoners.

References

[1] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, H. T. Cory, The British Nationality Act as a logic program, Commun. ACM 29 (1986) 370–386. URL: https://doi.org/10.1145/5689.5920. doi:10.1145/5689.5920.

[2] L. Al-Abdulkarim, K. Atkinson, T. J. M. Bench-Capon, A methodology for designing systems to reason with legal cases using Abstract Dialectical Frameworks, Artif. Intell. Law 24 (2016) 1–49. URL: https://doi.org/10.1007/s10506-016-9178-1. doi:10.1007/s10506-016-9178-1.

[3] G. Brewka, H. Strass, S. Ellmauthaler, J. P. Wallner, S. Woltran, Abstract Dialectical Frameworks Revisited, in: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI/AAAI, 2013, pp. 803–809.

[4] J. Collenette, K. Atkinson, T. J. M. Bench-Capon, An explainable approach to deducing outcomes in European Court of Human Rights cases using ADFs, in: Computational Models of Argument, COMMA 2020, volume 326 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2020, pp. 21–32. URL: https://doi.org/10.3233/FAIA200488. doi:10.3233/FAIA200488.

[5] J. Collenette, K. Atkinson, T. J. M. Bench-Capon, Explainable AI tools for legal reasoning about cases: A study on the European Court of Human Rights, Artif. Intell. 317 (2023) 103861. URL: https://doi.org/10.1016/j.artint.2023.103861. doi:10.1016/j.artint.2023.103861.

[6] L. L. T. Trieu, T. C. Son, M. Balduccini, xASP: An explanation generation system for answer set programming, in: Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 363–369.

[7] M. Alviano, L. L. Trieu, T. C. Son, M. Balduccini, Advancements in xASP, an XAI System for Answer Set Programming , in: Proceedings of the 38th Italian Conference on Computational Logic, CEUR Workshop Proceedings, 2023.

[8] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, Clingraph: Asp-based visualization, in: Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 401–414.

[9] P. Cabalar, J. Fandinno, B. Muñiz, A system for Explainable Answer Set Programming, in: ICLP Technical Communications 2020, volume 325 of *EPTCS*, 2020, pp. 124–136. URL: https://doi.org/10.4204/EPTCS.325.19. doi:10.4204/EPTCS.325.19.