4. E. De Angelis, M. Proietti, F. Fioravanti & A. Pettorossi (2022): *Verifying Catamorphism-Based Contracts using Constrained Horn Clauses*. Theory and Practice of Logic Programming 22(4), pp. 555–572, doi:10.1017/S1471068422000175.
5. H. Hojjat & Ph. Rümmer (2018): *The ELDARICA Horn Solver*. In: Formal Methods in Computer Aided Design, FMCAD '18, IEEE, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.
6. A. Komuravelli, A. Gurfinkel & S. Chaki (2016): *SMT-Based Model Checking for Recursive Programs*. Formal Methods in System Design 48(3), pp. 175–205, doi:10.1007/s10703-016-0249-4.

# Towards a Representation of Decision Theory Problems with Probabilistic Answer Set Programs

**Damiano Azzolini** (*University of Ferrara*)
**Elena Bellodi** (*University of Ferrara*)
**Fabrizio Riguzzi** (*University of Ferrara*)

## Abstract

Probabilistic Answer Set Programming under the Credal Semantics is a recently introduced formalism to express uncertainty with answer set programming where the probability of a query is described by a range. We propose to extend it to encode decision theory problems.

## Encoding

Probabilistic Answer Set Programming under the Credal Semantics [5] extends answer set programming [3] with ProbLog [6] probabilistic facts of the form $\Pi :: f$, where $\Pi$ is a floating point value (between 0 and 1) representing the probability of $f$ of being true. A selection of a truth value for every probabilistic fact identifies a world whose probability is the product of the probabilities of the probabilistic facts true times the product of one minus the probability of every probability fact false. In formula:

$$P(w) = \prod_{f_i \text{ true} \in w} \Pi_i \cdot \prod_{f_i \text{ false} \in w} (1 - \Pi_i).$$

In Probabilistic Logic Programming [6] (PLP), a world is a Prolog program, so it has exactly one model. In Probabilistic Answer Set Programming under the credal semantics (PASP), every world is an answer set program [8], so it has zero or more (stable) models. Every world is required to have at least one stable model. In PASP, the probability of a query (i.e., a conjunction of ground atoms) is represented with a probability range, composed of a lower and upper bound. A world contributes to both the lower and upper bounds if the query is true in every answer set while it contributes only to the upper bound if it is true in some of the answer sets. That is, $P(q) = [\underline{P}(q), \overline{P}(q)]$ where

$$\underline{P}(q) = \sum_{w_i \mid \forall m \in AS(w_i),\, m \models q} P(w_i),\ \overline{P}(q) = \sum_{w_i \mid \exists m \in AS(w_i),\, m \models q} P(w_i).$$

The probability bounds can be computed with, for example, the tool described in [1]. The DTProbLog framework [4] allows to express decision theory problems with a ProbLog [6] program. It introduces a set $U$ of utility atoms of the form $utility(a, r)$ where $r$ is the reward obtained to satisfy $a$ (i.e., when $a$ is true) and a set of decision atoms $D$. Every subset of decision atoms defines a strategy $\sigma$ (a ProbLog program) and its utility is computed as

$$Util(\sigma) = \sum_{(a,r) \in U} r \cdot P_\sigma(a)$$

where $P_\sigma(a)$ is the probability of $a$ computed in the ProbLog program identified by fixing the decision atoms in $\sigma$ to true. The goal is to find the strategy $\sigma^*$ that maximizes the utility, i.e., $\sigma^* = argmax_\sigma Util(\sigma)$.

We apply this framework to PASP, where every world has one or more stable models. Thus, the utility is described by a lower and an upper bound:

$$Util(\sigma) = \left[\underline{Util}(\sigma), \overline{Util}(\sigma)\right] = \left[\sum_{(a,r) \in U} r \cdot \underline{P}_\sigma(a), \sum_{(a,r) \in U} r \cdot \overline{P}_\sigma(a)\right]$$

We have now to select whether to optimize the lower or upper utility, and so there are two possible goals:

$$\sigma^* = argmax_\sigma(\underline{Util}(\sigma)).$$

or

$$\sigma^* = argmax_\sigma(\overline{Util}(\sigma)).$$

depending on the considered target probability. Note that with this representation the lower utility can be greater than the upper utility, when, for example, there are both positive and negative rewards. We consider here only the strategies where the lower utility is less or equal than the upper utility. To clarify, consider this simple program:
*0.7::pa. 0.2::pb.*
*decision da. decision db.*
*utility(r0,3). utility(r1,-11).*
*r0:- pa, da.*
*r0;r1:- pb, db.*
$pa$ and $pb$ are two probabilistic facts and $da$ and $db$ two decision atoms. The rewards we get to satisfy $r0$ and $r1$ are respectively 3 and -11. There are 4 possible strategies (no decision atoms selected, only one decision atom selected, and both decision atoms selected). Each of these have 4 worlds (no probabilistic facts true, only one probabilistic fact true, and both probabilistic facts true). If we consider the strategy $\sigma_{da} = \{da\}$, we have $P_{\sigma_{da}}(r0) = [0.7, 0.7]$ and $P_{\sigma_{da}}(r1) = [0, 0]$ (only the first rule is considered), and $Util(\sigma_{da}) = [3 \cdot 0.7 + (-11) \cdot 0, 3 \cdot 0.7 + (-11) \cdot 0] = [2.1, 2.1]$. If we repeat this process for all the strategies and we consider the upper utility as target, we get that $\sigma_{da} = \{da\}$ is the one that maximizes the upper utility.

A naive algorithm consists in enumerating all the possible strategies and then computing the lower and upper probability for every decision atom [1]. However, this clearly cannot scale, since the number of possible strategies is $2^{|D|}$. Moreover, the algorithm of [1] adopts enumeration of stable models and projection on probabilistic facts [7], that also requires the generation of an exponential number of answer sets. Thus, a practical solution should adopt both approximate methods for the generation of the possible strategies and for inference. For the former, greedy algorithms such as genetic algorithms [9] can be a possible solution. For inference, approximate methods based on sampling, such as the one proposed in [2] can be of interest.

**Acknowledgements**

This research was partly supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No. 952215.

### References

1. Damiano Azzolini, Elena Bellodi, and Fabrizio Riguzzi (2022): Statistical Statements in Probabilistic Logic Programming. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors: Logic Programming and Non-monotonic Reasoning, Springer International Publishing, Cham, pp. 43-55, doi:10.1007/978-3-031-15707-3_4.
2. Damiano Azzolini, Elena Bellodi, and Fabrizio Riguzzi (2023): Approximate Inference in Probabilistic Answer Set Programming for Statistical Probabilities. In Agostino Dovier, Angelo Montanari, and Andrea Orlandini, editors: AIxIA 2022 - Advances in Artificial Intelligence, Springer International Publishing, Cham, pp. 33-46, doi:10.1007/978-3-031-27181-6_3.
3. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski (2011): Answer Set Programming at a Glance. Communications of the ACM 54(12), pp. 92-103, doi:10.1145/2043174.2043195.
4. Guy Van den Broeck, Ingo Thon, Martijn van Otterlo, and Luc De Raedt (2010): DTProbLog: A Decision-Theoretic Probabilistic Prolog. In Maria Fox and David Poole, editors: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI Press, pp. 1217-1222.
5. Fabio Gagliardi Cozman and Denis Deratani Mauá (2020): The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference. International Journal of Approximate Reasoning 125, pp. 218-239, doi:10.1016/j.ijar.2020.07.004.
6. Luc De Raedt, Angelika Kimmig, and Hannu Toivonen (2007): ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In Manuela M. Veloso, editor: 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), 7, AAAI Press, pp. 2462-2467.
7. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub (2009): Solution Enumeration for Projected Boolean Search Problems. In Willem-Jan van Hoeve and John Hooker, editors: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Springer Berlin Heidelberg, pp. 71-86, doi:10.1007/978-3-642-01929-6_7.
8. Michael Gelfond and Vladimir Lifschitz (1988): The stable model semantics for logic programming. In: 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988), 88, MIT Press, pp. 1070-1080.
9. Xin-She Yang (2021): Chapter 6 - Genetic Algorithms. In Xin-She Yang, editor: Nature-Inspired Optimization Algorithms (Second Edition), second edition edition, Academic Press, pp. 91-100, doi:10.1016/B978-0-12-821986-7.00013-5.

# The ICARUS-System for Interactive Coherence Establishment in Logic Programs

**Andre Thevapalan** (*TU Dortmund, Germany*)
**Jesse Heyninck** (*Open Universiteit, the Netherlands*)

When modelling expert knowledge as logic programs, default negation is very useful, but might lead to there being no stable models. Detecting the exact causes of the incoherence in the logic program manually can become quite cumbersome, especially in larger programs. Moreover, establishing coherence requires expertise regarding the modelled knowledge as well as technical knowledge about the program and its rules. In this demo, we present the implementation of a workflow that enables knowledge experts to obtain a coherent logic program by modifying it in interaction with a system that explains the causes of the incoherence and provides possible solutions to choose from.

### Introduction

Due to its declarative nature, and with the availability of both strong and default negation, answer set programming offers valuable features for the usage in real-world applications. But often knowledge bases are subject to change. Adding rules to a logic program is not as straightforward as it might seem. Due to its declarative nature, adding new information can easily lead to contradictory knowledge. Such contradictions can result from rules whose conclusions are contradictory while both their premises can potentially hold simultaneously. However, by using default negation, another form of inconsistency can appear which is called *incoherence*. Incoherent answer set programs do not possess answer sets. In many cases, though, the cause of the incoherence generally comprises only one or a couple of rules. Several approaches show how these causes can be found and which part of the programs are affected and thus have to be revised in order to restore the coherence of a program ([4][5][10]). These approaches do not describe, however, how to establish coherence. In recent work ([11]), we have defined a method that offers strategies for computing suitable solutions without the need for additional information from the user and which in turn can be used to enable implementations where coherence can be interactively established in collaboration with an expert. This method is based on results on coherence in argumentation graphs ([10]) for normal logic programs which in turn enables us to locate those parts of the program that are responsible for the incoherence.

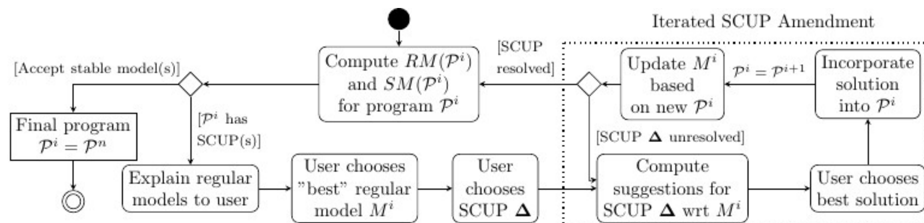

Figure 1: Amendment Workflow

### The ICARUS-System

We implemented the framework from ([11]) in a prototype system[1]. In this section, we describe and illustrate this implementation.

Figure 1 shows the general workflow that is implemented in this prototype for establishing the coherence in a given incoherent program $\mathcal{P}$.

In the following, we will describe the general workflow for this application, and how it helps the knowledge expert to gradually restore coherence. For this consider the following program $\mathcal{P}^1$ inspired by the medical domain will be used as a running example[2]:

$r_1 : fever \leftarrow highTemp, {\sim}sunstrk.$
$r_2 : fatigue \leftarrow lowEnergy, {\sim}stress, {\sim}sports.$
$r_3 : highTemp \leftarrow temp > 37.$
$r_4 : sunstrk \leftarrow highTemp, {\sim}fever.$
$r_5 : allrg \leftarrow pollenSeason, fatigue, {\sim}flu, {\sim}cold.$
$r_6 : cold \leftarrow fatigue, soreThroat, {\sim}migrn.$
$r_7 : flu \leftarrow fatigue, {\sim}migrn.$
$r_8 : migrn \leftarrow headache, {\sim}allrg.$
$r_9 : soreThroat.$
$r_{10} : headache.$
$r_{11} : pollenSeason.$