asplain: An ASP System for Contrastive Explanations

Susana Hahn[1], Brais Muñiz[2], Orkunt Sabuncu[1], Torsten Schaub[1], Tran Cao Son[1] and Hannes Weichelt[1]

[1]Potassco Solutions GmbH, Germany
[1]University of Coruña, Spain

Abstract

This paper focuses on answering "why-not" questions in Answer Set Programming, with an emphasis on user-oriented explanations. Such questions ask why a user's expectations are not met within an answer set or why a program is unsatisfiable. We address these questions using contrastive explanations, which explain an outcome by contrasting a reference program and, when available, one of its models, with an alternative program–model pair that satisfies the query. The alternative program is obtained through a controlled process of rule addition and removal. Our approach emphasizes flexibility, adapting to different domain settings and supporting preference-based selection among possible explanations. To improve interpretability, we propose a graph-based representation that highlights contrasts while leveraging domain knowledge captured by integrity constraints. These ideas are realized in *asplain*, a system that computes contrastive explanation graphs via meta-programming in *clingo*, allows users to configure explanation generation through a tagging mechanism, and supports interactive exploration of the explanation space through a user-friendly interface. Finally, we present a preliminary experiment using large language models to generate natural language explanations from contrastive explanation graphs.

1. Introduction

Humans often seek explanations when observed outcomes differ from their expectations, prompting them to ask *"why not"* questions. In the context of Answer Set Programming (ASP), such questions arise, for example, when a user expects an atom to appear (or not appear) in an answer set, or when a solution is expected but the program is unsatisfiable. This differs from the more commonly studied questions in the context of ASP, which focus on understanding why certain atoms are present in an answer set [1, 2]. The survey on Explainable AI (XAI) in ASP [3] provides a comprehensive overview of methods, systems, and perspectives in this area.

As a running example, consider rules (1)–(6) shown below, depicting an attempt to poison James Bond [1].

$$\{d\} \quad (1) \qquad \leftarrow p, c \quad (2) \qquad p \leftarrow t \quad (3) \qquad p \leftarrow d, \neg a \quad (4) \qquad a \quad (5) \qquad c \quad (6)$$

Rule (1) allows Bond to drink d a martini, while rule (2) is an integrity constraint stating that he cannot be poisoned p if he is careful c. Rules (3) and (4) encode two possible causes of poisoning: by contact with a toxin t, or by drinking without taking an antidote a. Finally, rules (5) and (6) state that Bond takes the antidote and is careful. This program has two stable models, $M_1 = \{a, c\}$ and $M_2 = \{a, c, d\}$, in both of which Bond is not poisoned; they differ only in whether he drinks.

Consider a situation where the user is shown model M_1, but expects James to be poisoned, and wants to know why it is not the case. Social sciences accounts that humans answer such questions by identifying hypothetical situations (called foils) that match these unmet expectations [4]. These foils

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.

susana.hahn@potassco.com (S. Hahn); brais.mcastro@udc.es (B. Muñiz); orkunt.sabuncu@potassco.com (O. Sabuncu); torsten.schaub@potassco.com (T. Schaub); tran.cao.son@potassco.com (T. C. Son); hannes.weichelt@potassco.com (H. Weichelt)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]More examples can be found in https://potassco.org/asplain/docs/examples

CEUR-WS.org/Vol-4311/ASPOCP_regular_02.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

present slight changes over what was actually observed, that would have lead to what was expected. These changes are most often perceived by humans as good explanations. In our example, it's easy to imagine that, by removing rules (5) and (6), thus not taking the antidote and not being careful, this could be a situation where James is actually poisoned. Alternatively, we could still remove (6) and add rule (7) which leads to poisoning by the toxin.

$$t \tag{7}$$

We propose an approach that mimics this process: intuitively, we start from a reference program and a query that encodes the user's expectations, we modify the reference program in a controlled way by adding and removing rules to obtain a foil program, with a foil model such that the foil model satisfies the query. Although this process can be considered a form of abduction for computing correction sets, no publicly available system uses these techniques for explanation in ASP. Furthermore, we select the best foils according to different criteria via a cost function. This provides flexibility for the modeler to customize the preferred explanations, such as those closer to the original program and/or model, or any ad hoc criteria for particular problems. Finally, using a graph based representation, we contrast and visualize the differences between the reference and foil worlds.

The paper is structured as follows. Section 2 provides a brief introduction to ASP. Section 3 presents our approach to constructing and selecting foils, followed by Section 4, which describes our contrastive graph-based explanation that compares the reference and foil worlds. Section 5 describes the tool *asplain*, which implements the proposed approach. Finally, Sections 6 and 7 discuss limitations, future directions, and related work.

2. Answer Set Programming

A *logic program* P consists of rules of the following forms:

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \tag{8}$$

$$\{a_1;\ \ldots;\ a_m\} \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \tag{9}$$

where $1 \leq m \leq n \leq o$. We call each a_i, for $i \leq o$, an *atom*. A *literal* is either an atom (*positive literal*) or a default-negated atom (*negative literal*). For a rule r, we denote its head by $H(r) = \{a_1, \ldots, a_m\}$, its positive body by $B^+(r) = \{a_{m+1}, \ldots, a_n\}$, and its negative body by $B^-(r) = \{a_{n+1}, \ldots, a_o\}$, with $B(r) = B^+(r) \cup B^-(r)$.

Rules of the form (8) are *disjunctive rules*, while rules of the form (9) are called *choice rules*. A disjunctive rule where $m = 0$ is called an *integrity constraint*, and when $m = n = o = 1$ it is called a *fact*. Given a logic program P, we denote by $\mathcal{R}_P$ its set of rules and by $AS(P)$ its set of answer sets. The satisfaction relation between a model M and an atom a is defined as $M \models a$ iff $a \in M$, $M \models \neg a$ iff $a \notin M$, and for a set of literals L, $M \models L$ iff $M \models l$ for each $l \in L$. We refer to [5] for a detailed introduction to ASP.

We consider ground logic programs, where all elements in $H(r) \cup B(r)$ are propositional atoms. Non-ground input programs are handled via *clingo*'s grounder. Some features of the full *clingo* input language are not supported, including optimization statements, aggregates, and explicit bounds in choice rules, as grounding introduces auxiliary atoms. Meta-programming plays a central role in *asplain*, which builds on *clingo*'s reified representation of logic programs [6]. In this representation, rules and atoms are encoded as facts, allowing a logic program to reason about another program's structure. *asplain* exploits this to analyze, edit, and compare logic programs, as described in Section 5.

3. Foil Finding

We start the explanation process from a reference program P_r and a query Q that encodes the user's expectations as a set of literals. We modify P_r in a controlled way by adding and removing rules to

obtain a foil program P_f, with a foil model M_f such that $M_f \models Q$. We will refer to this process as *foil-finding*. The rules that can be added and removed, R_{add} and R_{rem}, respectively, are also referred to as *optional rules*. To allow for controlled modifications, we limit these sets so that only rules in P_r can be removed, that is $R_{rem} \subseteq P_r$. With this, we define a *foil-finding setup* $\mathcal{S}$ as the tuple $\langle P_r, R_{add}, R_{rem}, Q \rangle$.

Definition 1 (Foil). *Given a foil-finding setup $\mathcal{S} = \langle P_r, R_{add}, R_{rem}, Q \rangle$, we say the pair (P_f, M_f) is a* foil *with respect to $\mathcal{S}$ if the following conditions hold:*

(i) $P_r \setminus R_{rem} \subseteq P_f \subseteq (P_r \cup R_{add})$ *(ii)* $M_f \in AS(P_f) \ \wedge \ M_f \models Q$

We denote as $Foils_{\mathcal{S}}$ the set of all foils w.r.t. $\mathcal{S}$. The set of potential rules and atoms from $\mathcal{S}$ are denoted by $\mathcal{R}_{\mathcal{S}} = P_r \cup R_{add}$ and $\mathcal{A}_{\mathcal{S}} = \mathcal{A}_{P_r \cup R_{add}}$, respectively.

Intuitively, a foil can also be obtained without performing any modifications to the reference program P_r, provided there is a model of P_r that satisfies the query, that is $P_r = P_f$ and $M_f \in AS(P_r)$. Furthermore, notice that $Foils_{\mathcal{S}}$ might be empty, which means that there is no way to modify P_r considering the optional rules to obtain a program P_f that has a model satisfying Q, so there is no explanation.

Following our example, let $P_1 = \{(1) \dots (6)\}$ be the reference program, and consider the foil-finding setup $\mathcal{S}_1 = \langle P_1, \{(7)\}, \{(5), (6)\}, \{p\} \rangle$, posing the question *"Why-not p?"*. In the set $Foils_{\mathcal{S}_1}$ we will find a total of five foils satisfying p, namely, $F_1 = (P_1 \setminus \{(6), (5)\}, \{d, p\})$, $F_2 = (\{(7)\} \cup P_1 \setminus \{(6)\}, \{a, p, t\})$, $F_3 = (\{(7)\} \cup P_1 \setminus \{(6)\}, \{a, d, p, t\})$, $F_4 = (\{(7)\} \cup P_1 \setminus \{(6), (5)\}, \{p, t\})$ and $F_5 = (\{(7)\} \cup P_1 \setminus \{(6), (5)\}, \{d, p, t\})$. All foils remove rule (6) since this is a necessary change to allow p given constraint (2). The first foil F_1 removes (5) eliminating the antidote, and requires that Bond chooses to drink. The foils F_2 and F_3 add (7) instead, thus obtaining p from the toxin via rule (3), differing only in the drinking d which becomes irrelevant. Finally, F_4 and F_5 both remove (5) and add (7).

Notice that Definition 1 does not have a requirement on the satisfiability of the reference program. If P_r is unsatisfiable, it becomes necessary to modify it in order to get a satisfiable P_f. In fact, if what we want is to only recover satisfiability of P_r, and not forcing the truth value of any atom, the query can be just set to $\emptyset$.

Consider $P_2 = P_1 \cup (7)$ which is unsatisfiable as there is no answer set that can satisfy rules (7), (6) and (2) at the same time. To recover satisfiability, we can use the foil-finding setup $\mathcal{S}2 = \langle P_2, \emptyset, \{(7), (6), (2)\}, \emptyset \rangle$ to obtain foils that make the program satisfiable by removing any (non-empty) combination of the conflicting rules. Since $Q = \emptyset$, any model M_f of a satisfiable program P_f will form a foil (P_f, M_f).

In order to provide a flexible way to select among all foils according to different criteria, we introduce an abstract *cost function* that assigns each foil a numerical value, thus establishing an order relation over foils. Each particular implementation of the abstract function represents a criterion that can be used to select among foils by minimization. The identification, definition, and implementation of such criteria deserve dedicated effort, as they directly affect the relevance and usefulness of the explanations from a user's perspective.

Unlike the foil-finding process, which is agnostic to the existence of $M_r \in AS(P_r)$, comparing M_f to M_r becomes essential for foil selection. To avoid defining special cases for unsatisfiable reference programs and to encode both reference and foil program–model pairs uniformly, we introduce program and model membership functions.

Intuitively, given a foil-finding setup $\mathcal{S}$, $\Theta_{\mathcal{S}}$ denotes the class of program membership functions $\mathcal{R}_{\mathcal{S}} \to 2^{\{ref, foil\}}$, which assign to each potential rule the programs in which it occurs. Similarly, $\Sigma_{\mathcal{S}}$ denotes the class of model membership functions $\mathcal{A}_{\mathcal{S}} \to 2^{\{ref, foil\}}$, which specify for each atom the models in which it holds.

Definition 2 (Program membership function). *Given a foil-finding setup $\mathcal{S}$ and programs P_r and P_f, the* program membership function $\theta_{\mathcal{S}}^{P_r, P_f} \in \Theta_{\mathcal{S}}$ *is defined such that, for every rule $r \in \mathcal{R}_{\mathcal{S}}$:*

(i) $ref \in \theta_{\mathcal{S}}^{P_r, P_f}(r)$ iff $r \in P_r$, and (ii) $foil \in \theta_{\mathcal{S}}^{P_r, P_f}(r)$ iff $r \in P_f$.

Definition 3 (Model membership function). *Given a foil-finding setup $\mathcal{S}$ and models M_r and M_f, the* model membership function $\sigma_{\mathcal{S}}^{M_r,M_f} \in \Sigma_{\mathcal{S}}$ *is defined such that, for every atom $a \in \mathcal{A}_{\mathcal{S}}$:*
(i) ref $\in \sigma_{\mathcal{S}}^{M_r,M_f}(a)$ iff $a \in M_r$, and (ii) foil $\in \sigma_{\mathcal{S}}^{M_r,M_f}(a)$ iff $a \in M_f$.

To support different foil selection criteria for a given foil-finding setup $\mathcal{S}$, we define a *foil cost function* as $\delta_{\mathcal{S}} : \Theta_{\mathcal{S}} \times \Sigma_{\mathcal{S}} \to \mathbb{N}$. Intuitively, for $\mathcal{S} = \langle P_r, R_{add}, R_{rem}, Q \rangle$, the cost function $\delta_{\mathcal{S}}(\theta_{\mathcal{S}}^{P_r,P_f}, \sigma_{\mathcal{S}}^{M_r,M_f})$ assigns a numerical value to the foil $(P_f, M_f) \in \mathit{Foils}_{\mathcal{S}}$, measuring its cost relative to the reference model M_r (or $\emptyset$ if P_r is unsatisfiable).

The following concrete foil cost functions are of particular interest. The *program-difference cost function* δ^{pdiff} counts the number of rules that differ between the reference and foil programs, i.e., rules that are added or removed, thus favoring foils whose program P_f is syntactically closest to P_r. For a given foil-finding setup $\mathcal{S}$,

$$\delta_{\mathcal{S}}^{pdiff}(\theta, \sigma) = |\{\, r \in \mathcal{R}_{\mathcal{S}} \mid |\theta(r)| = 1 \,\}|\,.$$

Analogously, one can define a *model-difference cost function* δ^{mdiff} that counts the number of atoms that differ between the reference and the foil model. Those two functions encode the often preferred criteria of minimizing changes with respect to a reference state (i.e. diagnosis, belief revision or theory repair). Besides, the program membership function alone can easily be used for debugging faulty programs. Further cost functions can encode domain-specific preferences, or assign higher costs to certain rules and/or atoms, e.g., to prioritize changes over the input, or the removal of integrity constraints over facts.

Provided a foil-finding setup $\mathcal{S}$, and a vector of cost functions $\mathit{costs} = (\delta_{\mathcal{S}}^1, \ldots, \delta_{\mathcal{S}}^n)$, The set of best foils is selected by the lexicographic ordering defined as

$$\mathit{BestFoils}_{\mathcal{S}}^{\mathit{costs}}(M_r) = \operatorname*{arg\,lex\,min}_{(P_f,M_f)\in \mathit{Foils}_{\mathcal{S}}} \big(\delta_{\mathcal{S}}^1(\theta_{\mathcal{S}}^{P_r,P_f}, \sigma_{\mathcal{S}}^{M_r,M_f}), \ldots, \delta_{\mathcal{S}}^n(\theta_{\mathcal{S}}^{P_r,P_f}, \sigma_{\mathcal{S}}^{M_r,M_f})\big)$$

Let us revisit our running example and discuss how some foils are more relevant than others. Provided the two alternative changes needed for having p under the foil-finding setup $\mathcal{S}_1$ i.e. not taking the antidote a (5) and touching the toxin t (7), foils F_4 and F_5 are clearly less concise than the rest since they require both changes. Among the other three foils, and considering $M_r = M_1 = \{a, c\}$ as the reference model, we can easily check how F_2 is the one that differs the least from M_1 as it does not require James to drink. This choice aligns with the foil selected by $\mathit{BestFoils}_{\mathcal{S}_1}^{(\delta^{pdiff},\delta^{mdiff})}(M_1)$, which minimizes the distance vector $\big((2,4),(2,3),(2,4),(3,4),(3,5)\big)$ over foils F_1 to F_5.

4. Graph-Based Contrastive Explanations

Once suitable foils have been computed and the most appropriate one has been selected, we can compare it with the reference world to construct a meaningful explanation. For this, humans typically focus on the differences between a reference and a foil world. In this sense, explanations are inherently contrastive. Representing such differences across programs and models calls for an appropriate representation. As noted by Miller, explanations are also causal, which makes directed graph representations an appropriate formalism, as edges can express cause–effect relationships. Moreover, logic programs themselves admit a natural graph-based representation.

In this section, we first introduce the notion of a program graph, which represents a ground logic program via a directed graph whose vertices correspond to atoms and rules, together with typing functions for vertices and edges, distinguishing rule types (choice vs. disjunctive) and positive and negative dependencies. We then build on this representation with the membership functions to define contrastive explanation graphs, supporting both representation and visualization of differences between reference and foil worlds.

Let us assume a bijective labeling function Lb that associates each rule r with a unique label $Lb(r)$, with Lb^{-1} as the inverse. By a slight abuse of notation, we extend Lb to sets of rules by defining $Lb(\mathcal{R}) = \{Lb(r) \mid r \in \mathcal{R}\}$.

Definition 4 (Program Graph). *Given a logic program P, its corresponding* program graph PG_P *is defined as the tuple $\langle G, \nu, \xi \rangle$, where $G = (V, E)$ is a directed graph with $V = Lb(P) \cup \mathcal{A}_P$ and $E = \{(a, Lb(r)) \mid a \in B(r),\ r \in P\} \cup \{(Lb(r), a) \mid a \in H(r),\ r \in P\}$.*

The vertex typing function $\nu : V \to \{choice, disj, atom\}$ and the edge typing function $\xi : E \to \{0, 1\}$ are defined as follows. For every rule $r \in P$, $\nu(Lb(r)) = choice$ iff r is a choice rule, and $\nu(Lb(r)) = disj$ iff r is a disjunctive rule, while $\nu(a) = atom$ for every $a \in \mathcal{A}_P$. For an edge $e = (a, Lb(r))$, we have $\xi(e) = 1$ iff $a \in B^+(r)$ and $\xi(e) = 0$ iff $a \in B^-(r)$, while for an edge $e = (Lb(r), a)$, $\xi(e) = 1$.

Figure 1 visualizes $PG_{\mathcal{R}_{\mathcal{S}_1}}$. Circle nodes represent atoms in $\mathcal{R}_{\mathcal{S}_1}$, i.e., nodes $v \in V$ with $\nu(v) = atom$. Rectangle nodes represent choice and disjunctive rules, i.e., nodes $v \in V$ with $\nu(v) \in \{choice, disj\}$. Dashed edges indicate negative literals, that is, edges $(a, \ell) \in E$ with $\xi((a, \ell)) = 0$. All remaining edges are solid, representing either positive literals or rule–head relationships. As expected, facts appear as square nodes without incoming edges, while integrity constraints appear as square nodes without outgoing edges.

Since a program graph fully encodes a ground logic program, we can work entirely at the graph level without referring back to the original program. This representation serves both as the conceptual basis of our explanations and as the core structure used throughout the implementation. Based on this, we now introduce *contrastive explanation graphs*, which make the differences between two programs and their models explicit.

Definition 5 (Contrastive Explanation Graph). *Given a foil-finding setup $\mathcal{S}$, along with program and model membership functions $\theta_{\mathcal{S}}$ and $\sigma_{\mathcal{S}}$, a* contrastive explanation graph *is defined as the tuple $\langle PG_{\mathcal{R}_{\mathcal{S}}}, (\boldsymbol{\theta}_{\mathcal{S}}, \sigma_{\mathcal{S}}) \rangle$. Where $\boldsymbol{\theta}_{\mathcal{S}} = \theta_{\mathcal{S}} \circ Lb^{-1}$ is the composition of $\theta_{\mathcal{S}}$ with the inverse of a particular labeling function Lb^{-1}, assigning program membership to the rule labels in the graph.*

Once a contrastive explanation graph $\mathbb{G} = \langle \langle (V, E), \nu, \xi \rangle, (\boldsymbol{\theta}, \sigma) \rangle$ is constructed, the foil-finding setup $\mathcal{S}$ does not need to be referenced explicitly. From this structure, we can infer additional information that is useful for interpreting the graph. In particular, the function $\sigma^{\mathbb{G}} : V \to 2^{\{ref, foil\}}$ extends model membership from atoms to rule nodes, indicating whether the body of a rule is satisfied by the reference and foil models. For each $w \in \{ref, foil\}$ and each $v \in V$ such that $\nu(v) \in \{choice, disj\}$, we have $w \in \sigma^{\mathbb{G}}(v)$ if and only if the following conditions hold: (i) $w \in \sigma(a)$ for all $(a, v) \in E$ with $\xi(a, v) = 1$, and (ii) $w \notin \sigma(a)$ for all $(a, v) \in E$ with $\xi(a, v) = 0$. For readability, we write σ instead of $\sigma^{\mathbb{G}}$ whenever the context makes the distinction clear.

Notice that for any integrity constraint r, we have that $\sigma^{\mathbb{G}}(Lb(r)) = \emptyset$, since its body must not be satisfied. Nevertheless, integrity constraints often encode relevant domain knowledge, as in the case of rule (2), which models how being careful c prevents the poisoning. To capture their impact within our explanations, we identify those constraints that would be violated when the reference and foil models are considered simultaneously. Or, informally speaking, those constraints that would have been fired under specific changes to the reference program.

Let $C_{\mathbb{G}} = \{v \in V \mid \nu(v) = disj \wedge \forall a \in V, (v, a) \notin E\}$ be the set of all rule nodes in the graph corresponding to integrity constraints. We then define the set of *fired constraints* as

$$FC_{\mathbb{G}} = \{c \in C_{\mathbb{G}} \mid \forall (a, c) \in E : \xi(a, c) = 1 \Rightarrow |\sigma(a)| \geq 1, \xi(a, c) = 0 \Rightarrow |\sigma(a)| \leq 1\}$$

Intuitively, we check if the literals within the body of each constraint are individually satisfied in at least one of the models. That is, $\sigma(a) \neq \emptyset$ for positive literals (i.e. ξ(a,c)=1) and $\sigma(a) \neq \{ref, foil\}$ for negative literals (i.e. ξ(a,c)=0).

With our example foil-finding setup $\mathcal{S}_1$, after observing model $M_1 = \{a, c\}$, and selecting the best foil $F_2 = (\{(7)\} \cup P_1 \setminus \{(6)\}, \{a, p, t\})$, we can construct the contrastive explanation graph as $\mathbb{G}_1 =$

$\langle PG_{\mathcal{R}_{\mathcal{S}_1}}, \theta_1, \sigma_1 \rangle$. Where $\theta_1(r) = \{ref, foil\}$ for $r \in \{(1)\ldots(5)\}$, $\theta_1((7)) = \{foil\}$, $\theta_1((6)) = \{ref\}$, $\sigma_1(a) = \{ref, foil\}$, $\sigma_1(t) = \sigma_1(p) = \{foil\}$, and $\sigma_1(c) = \{ref\}$. We visualize the graph in Figure 2, featuring both the explanation of the query as well as the foil-finding process. The model and the program membership functions σ_1 and θ_1 are used to provide meaning to the elements in $PG_{\mathcal{R}_{\mathcal{S}_1}}$ via color and style highlighting. When satisfied by the reference model, nodes are coloured in purple, whereas for the foil model we use green and higher opacity. The constraints in $FC_{\mathbb{G}_1} = \{(2)\}$ are colored in red. We show added rules with blue borders, while removed rules are shown with red borders. The graph can be read as *James would be poisoned if he hadn't been careful and had been in contact with a toxin.*

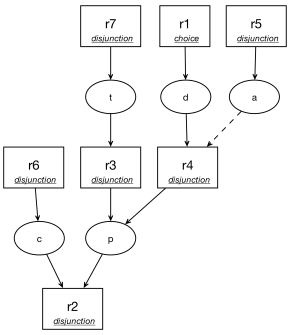


Figure 1: Program graph $PG_{\mathcal{R}_{\mathcal{S}_1}}$.

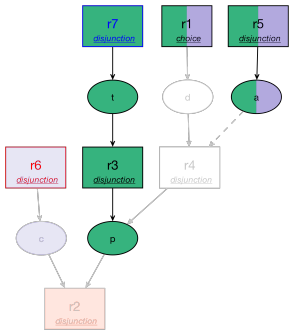


Figure 2: Contrastive explanation graph $\mathbb{G}_1$ corresponding to foil F_2.

Finally, notice that after the foil-finding process, it can be the case that $(P_f, M_f) = (P_r, M_r)$, and $M_r \models Q$, or in other words, that no changes are needed to answer the query. Since there is no difference between the reference and the foil in such a case, the explanation is not contrastive, but rather just justifies the truth value of the literals in Q in the model M_r.

5. Implementation

We implemented our approach in a system called *asplain*, which builds on the *clingo* system, leveraging its API and meta-programming features [6] to construct explanations. The system consists of a collection of *clingo* encodings, together with Python scripts that orchestrate the generation of contrastive explanation graphs. In addition, *asplain* provides a user interface for exploring and visualizing explanations, paired with natural language explanations generated using large language models (LLMs).

In this section, we describe how the facts representing a contrastive explanation graph $\mathbb{G} = \langle PG_{\mathcal{R}_{\mathcal{S}}}, (\boldsymbol{\theta}_{\mathcal{S}}, \sigma_{\mathcal{S}}) \rangle$ are constructed in ASP. The foil-finding setup $\mathcal{S} = \langle P_r, R_{add}, R_{rem}, Q \rangle$ is given as input by a logic program P_r augmented with special comments for tags, together with a query Q represented as facts. The explanation generation proceeds in two main steps. First, the tagged program is reified to extract the structural information of P_r, as well as the optional rules from the tags. Second, using a collection of meta-encodings, we construct the program graph $PG_{\mathcal{R}_{\mathcal{S}}} = \langle (V, E), \nu, \xi \rangle$ and perform the foil-finding process, resulting in the program and model membership functions $\boldsymbol{\theta}_{\mathcal{S}}$ and $\sigma_{\mathcal{S}}$. Throughout this process, the reference and foil programs P_r and P_f are not represented explicitly, but instead implicitly captured by the program membership function $\boldsymbol{\theta}_{\mathcal{S}}$.

Following our running example, Listing 1 shows the resulting facts obtained by the system which represent the contrastive explanation graph $\mathbb{G}_1$ (Figure 2). The query $Q = \{p\}$ is represented in Line 1 by the predicate `query/2`, where the second argument denotes positive and negative literals with `1` and `0`, respectively. The sets of optional rules R_{rem} and R_{add} are encoded using the predicate `optional/2` in Line 2. The vertex set V is realized in Lines 4–6, where the node typing function ν is captured by the second argument of predicate `node/2`. Edges are represented accordingly, with the first argument encoding the directed edge and the second argument representing the value of ξ, where `1` and `0` correspond to positive and negative edges. Lines 8–11 encode the edges associated with

```
query(p,1).
optional(add,7).   optional(remove,6).    optional(remove,5).

node(1,           rule(choice)).
node(2..7,        rule(disjunction)).
node((a;p;c;t;d), atom).

edge((1,d),1).
edge((p,2),1).  edge((c,2),1).
edge((3,p),1).  edge((t,3),1).
edge((4,p),1).  edge((d,4),1).  edge((a,4),0).
edge((5,a),1).  edge((6,c),1).  edge((7,t),1).

program(1..5, ref).     program(6, ref).
program(1..5, foil).    program(7, foil).

model((c;a),    ref).   model((1;6;5),   ref).
model((t;c;p),  foil).  model((1;3;6;7), foil).

fired(2).
```

Listing 1: Facts representing $\mathbb{G}_1$

rules (1)–(4), respectively, while Line 12 corresponds to the facts (5)–(7). Note that the addable rule (7) is included since the graph represents $PG_{\mathcal{R}_{\mathcal{S}_1}}$. The program membership function $\boldsymbol{\theta}_1$ is represented by the predicate `program/2` in Lines 14 and 15, implicitly capturing the reference and foil programs P_r and P_f. The model membership function σ_1 is represented using the predicate `model/2` in Lines 17–18, including both atom and rule nodes. Finally, Line 20 encodes the set $FC_{\mathbb{G}}$.

The foil-finding setup $\mathcal{S}_1 = \langle P_1, (7), \{(6), (5)\}, \{p\}\rangle$ is realized by the tagged program shown in Listing 2. The first line corresponds to rules (1)–(4). Rules (5), (6), and (7) shown in Lines 3, 5,and 7, respectively, are marked as optional via special tag comments `% @removable` and `% @addable` in their preceding line.

```
{d}.        :- p, c.     p :- t.       p :- d, not a.
% @removable
a.
% @removable
c.
% @addable
t.
```

Listing 2: Tagged encoding for the example foil-finding setup $\mathcal{S}_1$

We employ *tags* as a mechanism for attaching additional information to atoms and rules in a program. Their main use is to identify optional rules and to associate natural language descriptions with rules and atoms. More generally, they provide a flexible mechanism for program annotation and explanation customization. Tags are specified as comments in the encoding that start with `% @`. These comments are internally translated into special theory atoms, following an approach similar to the one employed by *xclingo* [7]. Tags can also be associated with atoms, which is particularly useful for generating natural language explanations. The reification of these theory atoms is translated by an ASP encoding into facts that extend the standard reification of the program with the predicate `tag/2` for both rules and atoms. In addition, the tag mechanism provides several automatically generated tags, such as a first-order representation of a rule.

Before obtaining the reification of the program, we must account for stable-models-preserving simplifications that may occur during grounding, as these can alter the original program structure and compromise the correctness of explanations. First, to avoid the removal of body literals or rules due to atoms whose truth value is established during the grounding phase, the user must ensure that

all relevant atoms appear in a rule head or under a `#external` directive. Second, we rely on built-in tags to ensure that all rules are associated with at least one tag. Since tags are treated as theory atoms whose truth value is not evaluated during grounding, they prevent unwanted simplifications, thereby preserving a distinct body for each rule. Finally, to preserve the symbol table, in terms of instances of predicate `output/2` provided by *clingo*'s reification, we use the `-preserve-facts` flag during grounding and remove all `#show` directives from the input program. With these settings, we obtain unique literal identifiers for each atom in the program. The user may provide a program with variables as input, provided that grounding remains within the supported syntax of the system.

After these transformations, the annotated program is reified into a fact-based representation using *clingo*. Each grounded rule gives rise to a `rule/2` fact, together with `atom_tuple/2` and `literal_tuple/2` facts that encode its head and body, where literals are represented by integer identifiers. From this reified form, we construct the program graph $PG_{\mathcal{R}_\mathcal{S}}$ using the encoding in Listing 3. The auxiliary predicate `symbol_literal/2` (Line 1) maps atom symbols to their internal literal identifiers, while an external Python function (Line 2) implements the labeling function Lb. Based on this information, the encoding introduces graph nodes for atoms and rules, assigning types $atom$, $choice$, or $disj$ (Lines 4–6). Directed edges are then added from atoms to rules to represent body literals (positive or negative), and from rules to atoms to represent rule heads (Lines 8–14).

```
1  symbol_literal(A,L):- output(A,B), literal_tuple(B,L).
2  label(rule(H,B), @label(rule(H,B))) :- rule(H,B).

4  node(A)                   :- symbol_literal(A,_).
5  node(R,rule(disjunction)) :- label(rule(disjunction(H),B), R).
6  node(R,rule(choice))      :- label(rule(choice(H),B), R).

8  edge((A, R),1):- label(rule(H, normal(B)), R),
9                   literal_tuple(B,L), L>0, symbol_literal(A,L).
10 edge((A, R),0):- label(rule(H, normal(B)), R),
11                  literal_tuple(B,L), L<0, symbol_literal(A,-L).
12 edge((R, A),1):- label(rule(disjunction(H), B), R),
13                  atom_tuple(H,L), symbol_literal(A,L).
14 edge((R, A),1):- label(rule(choice(H), B), R),
15                  atom_tuple(H,L), symbol_literal(A,L).
```

Listing 3: Encoding to generate $PG_{\mathcal{R}_\mathcal{S}}$

The encoding in Listing 4 extracts the sets R_{rem} and R_{add} from the tagged program. Lines 1 and 2 employ the `tag/2` predicate added to the standard reification to obtain instances of predicate `optional/2` for the removable and addable rules, respectively.

```
1  optional(remove, L) :- label(R, L), tag(R, removable).
2  optional(add,    L) :- label(R, L), tag(R, addable).
```

Listing 4: Encoding to generate R_{add} and R_{rem}

The program membership function $\boldsymbol{\theta}_\mathcal{S}$ corresponding to the reference program is constructed in Listing 5 by including all rule nodes except those marked as addable. If the reference program P_r is satisfiable, a reference model M_r is provided by the user via facts of the form `model(a,ref).` for each atom $a \in M_r$.

```
1  program(N, ref) :- node(N,_), not optional(add, N).
```

Listing 5: Encoding to generate $\theta_\mathcal{S}$ for the reference program

Listing 6 presents the main encoding used for foil finding. Lines 1 and 2 select which rules from R_{rem} are removed from $PG_{\mathcal{R}_\mathcal{S}}$ and which rules from R_{add} are added to construct the foil program P_f. Based on these choices, Lines 4–5 build the program membership function for P_f, capturing both the rules inherited from P_r and those introduced during the foil-finding process. The membership function σ is constructed in Lines 7–12. The first rule follows the definition of $\sigma^{\mathbb{G}}$ from Section 4, adding $foil$ to σ for a rule node r whenever all positive incoming body atoms satisfy $foil \in \sigma(a)$ (Line 8) and all negative body atoms satisfy $foil \notin \sigma(a)$ (Line 9) [2]. Lines 11 and 12 handle disjunctive and choice rules,

[2]To extend σ to rule nodes for the reference program, a corresponding rule analogous to Lines 7–9 is included for ref.

respectively, propagating *foil* to the corresponding head atoms via outgoing edges. Finally, Lines 14 and 15 enforce that the foil model satisfies the query Q.

```
1  {removed(R)} :- optional(remove, R),       program(R, ref).
2  {added(R)}   :- optional(add, R),      not program(R, ref).

4  program(N, foil):- program(N, ref), not removed(N).
5  program(R, foil):- added(R).

7  model(R,foil) :-    node(R, rule(_)), program(R, foil),
8                      model(A,foil) : edge((A,R), 1);
9                  not model(A,foil) : edge((A,R), 0).

11   model(A,foil) : edge((R,A), _)   :- model(R,foil), node(R, rule(disjunction)).
12 { model(A,foil) : edge((R,A), _) } :- model(R,foil), node(R, rule(choice)).

14 :- query(A,1), not model(A, foil).
15 :- query(A,0),     model(A, foil).
```

Listing 6: Encoding for foil-finding, generating $\theta_{\mathcal{S}}$ and $\sigma_{\mathcal{S}}$ for the foil program

To compute the set $FC_{\mathbb{G}}$, we use the encoding shown in Listing 7. The auxiliary predicate `some_model/2` collects atoms that occur in at least one model as well as atoms that are absent from at least one model. Based on this information, the rule in Lines 3–4 identifies fired constraints by requiring that each negative body literal is missing from some model and each positive body literal is present in some model.

```
1  some_model(1, A) :- node(A,atom), #count{W: model(A,W)}>=1.
2  some_model(0, A) :- node(A,atom), #count{W: model(A,W)}<=1.
3  fired(R):- node(R, rule(disjunction)), #false : edge((R,A),_);
4             some_model(T, A): edge((A,R),T).
```

Listing 7: Encoding to generate $FC_{\mathbb{G}}$

Users may additionally specify preferences that correspond to the vector of cost functions *costs* to compute the set of best foils *BestFoils* via *clingo*'s optimization features. Preferences are expressed by atoms of the form `cost(Name, Value, Level)`, where `Name` identifies a specific cost instance to be penalized, `Value` specifies its numeric weight, and `Level` denotes its priority in the vector. Costs are optimized using the weak constraint shown in Listing 8, which is instantiated for each cost atom provided by the user. Cost functions may also depend on the model membership function σ, thus enabling preferences over both program and model differences.

```
1  :~ cost(Name, Value, Level). [Value@Level, Name]
```

Listing 8: Encoding to minimize cost

As an example, the program-difference cost function δ^{pdiff} is encoded in Listing 9. Line 1 penalizes the removal of each rule with a cost of 1, while the Line 2 penalizes the addition. The value of 1 assigned to the level indicates the position in the cost vector.

```
1  cost(diff(R), 1, 1):-     program(R, ref), not program(R, foil).
2  cost(diff(R), 1, 1):- not program(R, ref),     program(R, foil).
```

Listing 9: Encoding for program-difference cost function δ^{pdiff}

Both the program graphs and the contrastive explanation graphs can be large, especially when the input program contains variables, since they are constructed from the grounded program. To address this, *asplain* provides three pruning strategies that can be combined freely. ORPHANS removes nodes unreachable from the query Q. PATHS retains only nodes along paths from the query to the rules that changed. CHANGES takes a different approach for cases where such paths are indirect or absent: it retains only nodes that differ between the reference and foil models, as well as fired constraints; this strategy is only meaningful when the query is not satisfied in the reference model, as otherwise both models coincide and the resulting graph would be empty. Users may additionally hide specific rules

and atoms from the final graph using tags. While these strategies effectively reduce graph size, more refined reduction techniques that account for transitive connections [8] remain an open direction.

The *asplain* system provides several additional features to enhance usability and flexibility. These include support for assumptions, enabling interactive exploration of answer sets, and the dynamic generation of tags using ASP rules, avoiding the need for manual annotation. For example, an ASP rule can be added to automatically tag all integrity constraints as removable, which is particularly useful in debugging scenarios.

To facilitate user interaction, *asplain* offers a graphical User Interface (UI) implemented using *clinguin* [9]. The interface allows users to interactively explore the contrastive explanations generated by our approach. Users can iterate over the models of a program, pose queries, and visualize explanations in the form of graphs. Graph visualization is realized using *clingraph* [10], which supports declarative graph specifications in ASP. Figure 3 shows a screenshot of the UI obtaining the desired explanation for our running example.

To aid the interpretation of contrastive explanation graphs, we employ LLMs to generate natural language explanations. The contrastive explanation graph computed by *asplain* is provided to the LLM as part of a prompt. This design cleanly separates roles: *asplain* is responsible for computing accurate contrastive explanations, while the LLM is used solely to condense the graph structure and produce coherent natural language descriptions [11]. We instruct the LLM via in-context learning, supplying a brief description of the task, an example input, and the corresponding phrase expected as an explanation. The prompt is further enriched with labels for atoms and rules, derived from tags added to the encoding [3]. The natural language interpretation shown in Figure 3 was generated from the explanation graph using `GPT-5`. Additional flexibility is provided by allowing different LLM models to be used interchangeably.

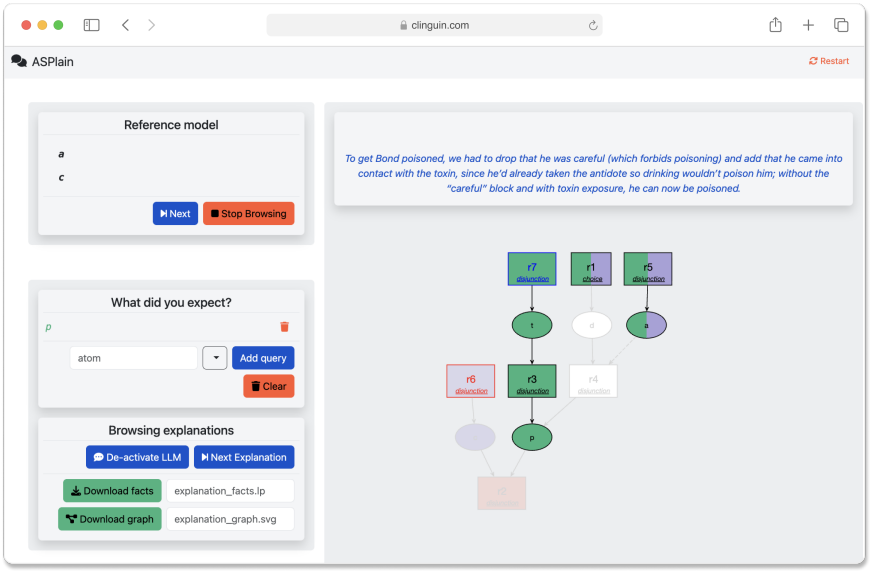


Figure 3: User Interface showing the contrastive explanation graph $\mathbb{G}_1$.

For further details on the implementation, please visit `https://potassco.org/asplain/`.

5.1. Evaluation

We conducted a preliminary evaluation of *asplain* in terms of graph size and computation time, using the Sudoku puzzle with instances of size 4×4 and 9×9. In both cases, the query imposes that cell $(2, 2)$ holds value 2, which is not satisfied by either instance. The problem and instance encodings are included

[3]The complete prompt template can be found in the source code under `https://github.com/potassco/asplain/blob/v1.0.0/src/asplain/llm/templates/prompt_templates/explain_instructions.txt`.

in the Appendix A. For each instance, we compared the effect of applying pruning (CHANGES+ORPHANS) and of using the model-difference cost function δ^{mdiff}. Table 5.1 summarizes the results.

Pruning adds negligible overhead and significantly reduces graph size, especially for the larger instance. The cost function δ^{mdiff} reduces the number of removed rules, which in turn further reduces the size of the contrastive graph when combined with pruning. As expected, program graph construction time is unaffected by pruning and cost settings, since both are applied afterwards. The main overhead of δ^{mdiff} lies in the increased time to compute the optimal foil, which is only significant for the 9×9 instance. Overall, combining pruning and optimization yields considerably smaller explanation graphs, making them easier to interpret both for users and for the LLM. Further user studies are required to assess the usefulness and interpretability of the generated graphs, as well as the practical use for developers to customize the explanations.

			Time (s)			Contrastive Graph Size			
Instance	Cost	Pruning	PG	CG	Prune	# Rule Nodes	# Atoms Nodes	# Edges	# Removed Rules
4×4	–	–	0.43	0.05	0.00	557	105	1573	2
4×4	–	CH+OR	0.43	0.05	0.06	16	10	30	2
4×4	δ^{mdiff}	–	0.42	0.05	0.00	557	105	1573	1
4×4	δ^{mdiff}	CH+OR	0.43	0.05	0.06	8	5	15	1
9×9	–	–	9.97	1.18	0.06	13352	930	39373	26
9×9	–	CH+OR	9.32	1.15	1.20	262	166	498	26
9×9	δ^{mdiff}	–	9.29	12.44	0.07	13352	930	39373	2
9×9	δ^{mdiff}	CH+OR	9.45	12.40	1.22	19	12	36	2

Table 1
Evaluation results for the Sudoku benchmark. Times in seconds shown for computing the program graph (PG), the contrastive graph (CG), and applying pruning.

6. Related work

Our foil-finding process is closely related to abductive logic programming and generalized stable model semantics [12], where a problem is typically given by a logic program P, a set of assumable atoms A, a set of integrity constraints I, and a query Q. The goal is to find subsets $A' \subseteq A$ such that the extended program $P \cup A'$ has a stable model that satisfies both I and Q. That is, the query holds under the resulting interpretation, or in other words A' explains Q.

Although our framework directly manipulates rules through additions and removals, such transformations can often be simulated in abductive settings by using auxiliary atoms as conditions in the body of a rule. Thus, the novelty of our approach does not lie in the abductive mechanism itself, but rather in its use for contrastive explanations in ASP, together with flexible foil preference handling and graph-based representations of differences between reference and foil worlds.

Regarding literature on explainable ASP (XASP), as surveyed in [13], only a few works address the problem of answering why-not questions in ASP.

The formalism presented in [14] is probably the first paper introducing contrastive explanations for ASP. This approach shares the same high-level idea of identifying required changes in a reference program to meet the user's expectations. However, our foil-finding framework is more general, allowing their approach to be reconstructed as a special case, as we show below.

In this work, they separate the explanation process into an *explanation frame* $\langle P, S, A \rangle$ and an *explanation problem* $\langle I, E, F \rangle$, where P is the reference program, $S \subseteq P$ is a set of fixed rules that cannot be removed, and A is a set of atoms that can be assumed. The problem is posed with respect to a fixed reference model $I \in AS(P)$, where the *explanandum* E must hold in I and fail in the counterfactual world, conversely, the foil F must hold in the counterfactual world and fail in I. Their solutions are called *counterfactual accounts*, defined by triples $\langle P', I', A' \rangle$ such that:

$$(i)\ S \subseteq P' \subseteq P \quad (ii)\ A' \subseteq A \ \wedge\ I' \in AS(P' \cup A') \quad (iii)\ F \subseteq I' \ \wedge\ E \not\subseteq I'$$

Additional constraints (e.g., subset-maximality of P' and restrictions on A' and F) further restrict the solutions. This approach can be reconstructed as a special case of our foil-finding framework as follows. Given an explanation frame $\langle P, S, A\rangle$ and an explanation problem $\langle I, E, F\rangle$, we define the corresponding foil-finding setup as $\mathcal{S} = \langle P, R_{add}, R_{rem}, Q\rangle$ with

$$R_{rem} = P \setminus S \quad R_{add} = \{\, a \leftarrow \mid a \in A \,\} \quad Q = F \cup \{\neg e \mid e \in E\}$$

Notice that their assumptions A are encoded as addable rules in the form of facts. The conditions imposed on the explanation problem, namely $E \subseteq I$ and $F \cap I = \emptyset$, do not need to be explicitly stated in the foil-finding setup. Nonetheless, the expected behavior of E in the foil can be encoded in the query Q. If conditions are to be imposed on the reference model, they can be encoded as constraints in the reference program P. With this setup, a counterfactual account $\langle P', I', A'\rangle$ computed by their approach, would have a corresponding foil tuple (P_f, M_f) where

$$P_f = P' \cup \{\, a \leftarrow \mid a \in A' \,\}, \qquad M_f = I'.$$

Notice that (P_f, M_f) fulfils the conditions for a foil given by Definition 1, namely, *(i)* since $S \subseteq P' \subseteq P$, we have $P \setminus R_{rem} = S \subseteq P' \subseteq P_f \subseteq P \cup R_{add}$, and *(ii)* since $I' \in AS(P' \cup A')$ and with the representation of A' as facts, it follows that $M_f \in AS(P_f)$, and $(F \subseteq I') \wedge (E \cap I' = \emptyset)$ which imply $M_f \models Q$. Beyond foil finding, both approaches provide additional structure to highlight the differences between the reference and counterfactual worlds, namely: *counterfactual* and *contrastive explanations* in the case of [14], and *contrastive explanation graphs* in our case, where we include the reference model if one is provided.

Overall, there are some key structural differences with [14], that highlight the flexibility of our framework and its ability to capture a wider range of contrastive explanation scenarios. First, we do not fix a reference model I, which also allows handling unsatisfiable reference programs. Second, we do not distinguish between explanandum and foil, but instead use a single query Q consisting of literals. Third, we specify directly the removable rules, and allow additions as rules rather than only facts. Finally, instead of always minimizing the removals over P, we provide a more flexible framework based on customizable foil cost functions.

Since *asplain* can also be used to explain the presence of an atom in an answer set using graphs, it is similar to the systems [4] *xclingo*2 [7, 1], xASP2 [2], and s(CASP) [15] that provide graph-based justifications for atoms in answer sets. These systems' graphs, called *support graph* in *xclingo*2, *explanation graph* in xASP2 and *justification tree* in s(CASP), capture the derivation of an atom in the answer set based on the rules in the program. Unlike them, our contrastive graphs contain nodes explicitly referring to rules — through their labels — while this information is implicitly represented in other systems.

*xclingo*2 does not consider explanations for atoms not in an answer set (false atoms) while xASP2 and s(CASP) do. Rather than showing the reasons why an atom is not in the answer set, *asplain* identifies a foil model that supports the atom, and uses it to point out what the original answer set is missing to support it. In addition, *asplain* provides explanations in natural language via LLMs which is a feature present in *xclingo*2 via manual annotations, but not in xASP2 or s(CASP). Interestingly, in [16] the authors also use LLMs to interpret queries, which is not provided by *asplain*.

Along the line of providing a justification for the presence of an atom in an answer set, Eiter and Geibinger propose the notion of justification for programs with abstract constraint atoms and study the complexity of computing such justifications [17]. Extending the present work to deal with more abstract constructs is not the focus of this paper, but it is an interesting direction for future work.

7. Conclusions and Future Work

We presented *asplain*, a framework for computing and exploring contrastive explanations in Answer Set Programming. Our approach formalizes contrastive explanations as graph structures that make

[4] We refer to the latest incarnations of these systems in this paper. Some of these systems have several implementations, e.g., xASP has xASP, xASP2, etc.

explicit the differences between a reference program and a foil, together with their associated models. By combining a flexible foil-finding process, user-defined preferences and graph-based representations, *asplain* supports adaptive and interpretable explanation generation. The implementation demonstrates how these ideas can be realized within existing ASP tooling and extended with interactive visualization and natural language explanations, providing a practical basis for further research on explainability in declarative systems.

A primary direction for future work is extending the approach to support richer ASP constructs, such as aggregates and optimization statements. This could be achieved either by assigning semantics to the auxiliary atoms introduced during grounding, or by extending the program graph to support weighted edges or even more complex constructs. Another promising direction is distinguishing necessary from sufficient changes, prioritizing modifications that ensure query satisfaction in all models through the integration of cautious and brave reasoning within the foil-finding process. This can be paired with solution space navigation [18] techniques to allow users to explore different explanations and their implications.

Beyond expressivity, *asplain* enables the investigation of more refined foil selection criteria. In particular, cost functions that approximate subset-minimal explanations, similar to UNSAT core minimization, could be realized through more expressive preference handling such as in *asprin* [19]. Furthermore, we intend to formalize the relation between *asplain* and formalisms such as abduction and correction sets, paving the way for an analysis on the complexity of our approach. Further work may also specialize *asplain* for particular application scenarios, such as debugging or temporal reasoning, and support user-driven simplification of explanations similarly to *xclingo*.

Finally, the quality of LLM-generated natural language explanations warrants further evaluation in more complex settings, together with studies on usability and versatility, including prompt design for different use cases and natural-language interaction for querying and exploring explanations.

Declaration on Generative AI

The author declare that no generative artificial intelligence (AI) tools were used in the conception, writing, analysis, or preparation of this manuscript. All content presented in this work was produced and reviewed exclusively by the authors.

References

[1] P. Cabalar, B. Muñiz, Model explanation via support graphs, CoRR abs/2310.01626 (2023). doi:10.48550/ARXIV.2310.01626. arXiv:2310.01626.

[2] M. Alviano, L. Trieu, T. Son, M. Balduccini, The XAI system for answer set programming xASP2, Journal of Logic and Computation 34 (2024) 1500–1525.

[3] T. Eiter, T. Geibinger, Z. Saribatur, An xai view on explainable asp: Methods, systems, and perspectives (2026). URL: https://arxiv.org/abs/2601.14764. arXiv:2601.14764.

[4] T. Miller, Explanation in artificial intelligence: Insights from the social sciences, Artificial Intelligence 267 (2019) 1–38. doi:10.1016/j.artint.2018.07.007.

[5] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012. doi:10.1007/978-3-031-01561-8.

[6] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own ASP-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361. doi:10.1017/S1471068421000508.

[7] P. Cabalar, J. Fandinno, B. Muñiz, A system for explainable answer set programming, in: [20], 2020, pp. 124–136. doi:10.4204/EPTCS.325.19.

[8] P. Cabalar, B. Muñiz, Explanation graphs for stable models of labelled logic programs, volume 385, Open Publishing Association, 2023. URL: https://ceur-ws.org/Vol-3437/paper3ASPOCP.pdf. doi:https://doi.org/10.4204/EPTCS.385.

[9] A. Beiser, S. Hahn, T. Schaub, ASP-driven user-interaction with clinguin, in: P. Cabalar, T. Swift (Eds.), Technical Communications of the Fortieth International Conference on Logic Programming (ICLP'24), EPTCS, 2024.
[10] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, Clingraph: A system for ASP-based visualization, Theory and Practice of Logic Programming 24 (2024) 533–559. doi:10.1017/S147106842400005X.
[11] Y. Zhang, H. Jin, D. Meng, J. Wang, J. Tan, A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods, arXiv preprint arXiv:2403.02901 (2024).
[12] A. Kakas, R. Kowalski, F. Toni, Abduction in logic programming, Journal of Logic and Computation 2 (1992) 837–956.
[13] J. Fandinno, C. Schulz, Answering the "why"in answer set programming–a survey of explanation approaches, Theory and Practice of Logic Programming 19 (2019) 114–203. doi:10.1017/S1471068418000534.
[14] T. Eiter, T. Geibinger, J. Oetsch, Contrastive explanations for answer-set programs, in: S. Gaggl, M. Martinez, M. Ortiz (Eds.), Proceedings of the Eighteenth European Conference on Logics in Artificial Intelligence (JELIA'23), volume 14281 of *Lecture Notes in Computer Science*, Springer-Verlag, 2023, pp. 73–89. doi:10.1007/978-3-031-43619-2_6.
[15] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: [20], 2020. doi:10.4204/EPTCS.325.12.
[16] T. Geibinger, T. Kaminski, J. Oetsch, Explanations for guess-and-check asp encodings using an llm (extended abstract), in: A. Tarzariol, K. Strauch, T. Geibinger (Eds.), Proceedings of the Seventh Workshop on Trends and Applications of Answer Set Programming (TAASP'24), 2024, p. 6. URL: https://taasp.at/2024/assets/paper/TAASP_2024_paper_9.pdf.
[17] T. Eiter, T. Geibinger, Explaining answer-set programs with abstract constraint atoms, in: E. Elkind (Ed.), Proceedings of the Thirty-second International Joint Conference on Artificial Intelligence (IJCAI'23), ijcai.org, 2023, pp. 3193–3202. doi:10.24963/IJCAI.2023/356.
[18] J. K. Fichte, S. A. Gaggl, D. Rusovac, Rushing and strolling among answer sets – navigation made easy, 2021. URL: https://arxiv.org/abs/2112.07596. arXiv:2112.07596.
[19] J. Romero, asprin: Answer set programming with preferences, in: B. Mitschang, N. Ritter, H. Schwarz, M. Klettke, A. Thor, O. Kopp, M. Wieland (Eds.), Datenbanksysteme für Business, Technologie und Web (BTW'17), Gesellschaft für Informatik e.V., 2017, pp. 159–162.
[20] F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. Lisi, M. Maratea, A. Mileo, F. Riguzzi (Eds.), Technical Communications of the Thirty-sixth International Conference on Logic Programming (ICLP'20), volume 325 of *EPTCS*, 2020. doi:10.4204/EPTCS.325.

A. Sudoku Example

Listing 10 shows the Sudoku encoding used in our evaluation. Rules are annotated with `@label` tags to provide human-readable descriptions, and selected rules are hidden from the final graph via `@hide`. Specifically, auxiliary rules irrelevant to the explanation—such as the definition of `defined/2`—are hidden, as is the integrity constraint enforcing that each cell is assigned exactly one value, since this is intuitive and its inclusion would unnecessarily increase graph size. The instances used in the evaluation are shown in Figures 4 and 5, with their corresponding encodings in Listings 11 and 12. The input predicates in these files are tagged as removable.

```
1 % @label("The solution has value {} in position ({},{})",(N,X,Y,)) :: sudoku(X,Y,N)
2 % @label("The problem has the initial value {} at position ({},{})",(N,X,Y,)) :: initial(X,Y,N)
3 % @label("The cell ({},{}) is part of subgrid {}",(X1,Y1,S)) :: subgrid(X1,Y1,S)
4 % @label("{} is a number",(N,)) :: number(N)

6 number(1..dim*dim).
```

```
% @label("Initals assign cell ({},{}) the value {}",(X,Y,N,))
sudoku(X,Y,N) :- initial(X,Y,N).
% @label("Choose a value for cell ({},{})",(X,Y,))
{sudoku(X,Y,N): number(N)} :- number(X), number(Y).
% @hide
:- number(X), number(Y), sudoku(X,Y,N), sudoku(X,Y,N2), N>N2.
% @hide
defined(X,Y):- sudoku(X,Y,V).
% @label("Cell ({},{}) has to be defined",(X,Y,))
% @hide
:- number(X), number(Y), not defined(X,Y).

% @label("Number {} cannot be multiple times in the row {}, but appears in columns {} and {}",(N,Y,X1,X2,))
:- sudoku(X1,Y,N), sudoku(X2,Y,N), X1>X2.
% @label("Number {} cannot be multiple times in the column {}, but appears in rows {} and {}",(N,X,Y1,Y2,))
:- sudoku(X,Y1,N), sudoku(X,Y2,N), Y1>Y2.

% @label("Cell ({},{}) is in subgrid {}",(X,Y,S,))
subgrid(X,Y,S) :- number(X), number(Y), S=(((X-1)/dim)*dim+((Y-1)/dim)).
% @label("Subgrid {} cannot have value {} in both cell ({},{}) and ({},{})",(S,V,X,Y,X2,Y2,))
:- sudoku(X,Y,V), sudoku(X2,Y2,V), subgrid(X,Y,S), subgrid(X2,Y2,S), (X,Y)<(X2,Y2).
```

Listing 10: Sudoku encoding for explanation

3	4	2	1
2	1	4	3
4	3	1	2
1	2	3	4

Figure 4: Sudoku 4x4 instance

```
#const dim=2.

% @removable
initial(1,1,1).
% @removable
initial(4,2,2).
% @removable
initial(2,3,1).
% @removable
initial(3,4,2).
```

Listing 11: Sudoku 4x4 instance

2	8	4	5	1	3	6	9	7
5	7	1	6	9	4	3	8	2
6	9	3	2	7	8	1	5	4
9	1	8	7	3	5	2	4	6
4	3	2	8	6	9	7	1	5
7	6	5	4	2	1	9	3	8
8	2	9	3	4	6	5	7	1
3	4	6	1	5	7	8	2	9
1	5	7	9	8	2	4	6	3

Figure 5: Sudoku 9x9 instance

```
#const dim=3.

% @removable
initial(5,1,8).
% @removable
initial(6,2,7).
% @removable
initial(7,1,4).
% @removable
initial(8,2,2).
% @removable
initial(8,1,6).
% @removable
initial(9,2,9).
% @removable
initial(3,3,9).
% @removable
initial(4,3,3).
% @removable
initial(6,3,6).
% @removable
initial(7,3,5).
% @removable
initial(8,3,7).
% @removable
initial(1,4,7).
% @removable
initial(2,4,6).
% @removable
initial(4,4,4).
% @removable
initial(7,4,9).
% @removable
initial(8,4,3).
% @removable
initial(2,5,3).
% @removable
initial(3,5,2).
% @removable
initial(4,5,8).
% @removable
initial(7,5,7).
% @removable
initial(9,5,5).
% @removable
initial(1,6,9).
% @removable
initial(2,6,1).
% @removable
initial(3,6,8).
% @removable
initial(8,6,4).
% @removable
initial(9,6,6).
% @removable
initial(1,7,6).
% @removable
initial(5,7,7).
% @removable
initial(6,7,8).
```

Listing 12: Sudoku 9x9 instance