The XAI System for Answer Set Programming xASP2

Mario Alviano[1 [0000−0002−2052−2063]], Ly Ly Trieu[2 [0000−0003−1482−9453]], Tran Cao Son[2 [0000−0003−3689−8433]], and Marcello Balduccini[3 [0000−0001−5445−3054]]

[1] DEMACS, University of Calabria, Via Bucci 30/B, 87036 Rende (CS), Italy
mario.alviano@unical.it
[2] New Mexico State University, USA
lytrieu@nmsu.edu, tson@cs.nmsu.edu
[3] Saint Joseph's University, USA
mbalducc@sju.edu

Abstract. Explainable artificial intelligence (XAI) aims at addressing complex problems by coupling solutions with reasons that justify the provided answer. In the context of Answer Set Programming (ASP) the user may be interested in linking the presence or absence of an atom in an answer set to the logic rules involved in the inference of the atom. Such explanations can be given in terms of directed acyclic graphs (DAGs). This article reports on the advancements in the development of the XAI system `xASP` by revising the main foundational notions and by introducing new ASP encodings to compute minimal assumption sets, explanation sequences, and explanation DAGs. DAGs are shown to the user in an interactive form via the `xASP navigator` application, also introduced in this work.

Keywords: Answer Set Programming · eXplainable Artificial Intelligence · Knowledge Representation and Reasoning.

1 Introduction

The interest in explainable artificial intelligence (XAI) has grown substantially in recent years. The reasons for this trend are obvious: while intelligent systems capable of solving complex problems are useful, confidence in their results is limited unless users can query them about the reasons that lead to the solutions produced. The *right to an explanation* law, extensively discussed in the USA, EU and UK, and partly enacted in some countries, increases the need for XAI systems. In this paper, we focus on the development of an XAI system for Answer Set Programming (ASP) [19, 22]. ASP is a knowledge representation and reasoning (KR&R) approach to problem solving using logic programs under answer set semantics [15], an extension of Datalog with a strong connection with well-founded semantics [23]. In this setting, we are mainly interested in the question "*given an answer set A of a program Π and an atom α, why does $\alpha \in A$ (or $\alpha \notin A$)?*"

As a logic program Π is a set of rules, the question can be answered by providing the subset of Π that supports the presence (or the absence) of α given Π and A. If Π is a Datalog program, then its models are easily explainable by the derivation procedure implemented by Datalog engines. Essentially, each atom in the model is

explained by the support provided by a rule whose body is true and contains only already explained atoms. If Π is a logic program under the well-founded semantics, then the fact that α belongs (or does not belong) to the well-founded model of Π can be explained similarly, with the addition of some atoms that are concluded to be false because they belong to some unfounded set. Generally speaking, explanations for logic programs under the answer set semantics can also be produced in a similar way *under the assumption provided by the answer sets themselves for the interpretation of false atoms*. However, taking all false atoms as an assumption would likely result in a *faint* explanation, actually in an explanation by faith for all such false atoms. Therefore, two main issues need to be tackled in explaining the assignment of α in A: (*i*) how to compute a hopefully small set of assumptions capable of explaining the assignment of α in A; and (*ii*) how to support sophisticated linguistic constructs such as choice rules and aggregates, which can be involved in explaining the falsity of some atoms in easily understandable terms.

An XAI system providing the reasons for the presence or absence of a given atom in an answer set finds another important application in the identification of the cause of unexpected results. This is a feature that can be particularly useful to the designers of complex systems confronted with unexpected inferences. In fact, identifying the root causes of those inferences can be daunting due to the many possible interactions in large knowledge bases. We found ourselves faced with such a challenge during the recent development of a commercial application. The ASP program that powered the decision-making component comprised a number of modules that could be enabled or disabled depending on needs. During development, we noticed that certain combinations of modules yielded unexpected results. After carefully checking each module, individually, for errors, we began to suspect that rules from different modules were interacting with each other in unexpected ways. Investigating those interactions proved to be a very time-consuming task that took approximately 3 *Full-Time Equivalent* weeks and considerably slowed down the project at a critical time. While XAI-inspired research conducted by the ASP community had already produced a number of tools related to this problem, such as `xclingo` [8], `DiscASP` [17], `xASP` [28], and `s(CASP)` [3], none of them could be used for our problem, due to their inability to process a program of the size of ours in an acceptable amount of time and the lack of support for certain advanced language features, and in some cases due to shortcomings in the type of information produced.

In this paper, we present research that takes inspiration from the approach used in `xASP` [28] and extends from [2], but that aims at yielding substantially increased scalability and breadth of supported language features, while producing information more immediately and consistently useful to users. The ultimate goal is to produce a system that can be applied to programs of size and complexity found in commercial-grade applications. Our main contributions are the following:

– A notion of explanation for the presence or absence of an atom in an answer set in terms of easy-to-understand inferences originating from a hopefully small set of atoms assumed false (Section 3).

- A representation of explanations in terms of directed acyclic graphs, restricted to the atoms involved in the explanation (Section 3), and a proof of existence for the explanations according to the given definition (Section 4).
- The implementation of a system for producing explanations powered by ASP and its empirical evaluation (Sections 5–6).
- The implementation of a web application for visualizing and interacting with the generated explanations.
- Suggestions on how to rewrite input programs in order to obtain richer explanations.

The supported fragment of ASP includes uninterpreted function symbols, common aggregation functions, comparison expressions, strong negation, constraints, normal rules, and choice rules. Aggregates are expected to be stratified, to not involve default negation, and to have a single atomic condition. Choice rules are expected to be unconditional, or otherwise to have exactly one conditional atom with a self-explanatory condition (as for example a range expression or an extensional predicate). Additionally, to ease the presentation, in Section 2 we only consider *sum* aggregates, and completely omit uninterpreted function symbols, comparison expressions, strong negation, and conditions in choice rules.

2 Background

All sets and sequences considered in this paper are finite. Let $\mathbf{P}$, $\mathbf{C}$, $\mathbf{V}$ be fixed nonempty sets of *predicate names*, *constants* and *variables*. Predicates are associated with an *arity*, a non-negative integer. A *term* is any element in $\mathbf{C} \cup \mathbf{V}$. An *atom* is of the form $p(\overline{t})$, where $p \in \mathbf{P}$, and $\overline{t}$ is a possibly empty sequence of terms. A *literal* is an atom possibly preceded by the default negation symbol *not*; they are referred to as positive and negative literals.

An *aggregate* is of the form

$$sum\{t_a, \overline{t'} : p(\overline{t})\} \odot t_g \tag{1}$$

where $\odot$ is a binary comparison operator, $p \in \mathbf{P}$, $\overline{t}$ and $\overline{t'}$ are possibly empty sequences of terms, and t_a and t_g are terms.

A *choice* is of the form

$$t_1 \leq \{atoms\} \leq t_2 \tag{2}$$

where *atoms* is a possibly empty sequence of atoms, and t_1, t_2 are terms. Let $\bot$ be syntactic sugar for $1 \leq \{\} \leq 1$.

A *rule* is of the form

$$head \leftarrow body \tag{3}$$

where *head* is an atom or a choice, and *body* is a possibly empty sequence of literals and aggregates. For a rule r, let $H(r)$ denote the atom or choice in the head of r; let $B^{\Sigma}(r)$, $B^{+}(r)$ and $B^{-}(r)$ denote the sets of aggregates, positive and negative literals in the body of r; let $B(r)$ denote the set $B^{\Sigma}(r) \cup B^{+}(r) \cup B^{-}(r)$.

A variable X occurring in $B^+(r)$ is a *global variable*. Other variables occurring among the terms $\overline{t}$ of some aggregate in $B^\Sigma(r)$ of the form (1) are *local variables*. And any other variable occurring in r is an *unsafe variable*. A *safe rule* is a rule with no *unsafe variables*. A *program* Π is a set of safe rules. Additionally, we assume that aggregates are stratified, that is, the *dependency graph* $\mathcal{G}_\Pi$ having a vertex for each predicate occurring in Π and an edge pq whenever there is $r \in \Pi$ with p occurring in $H(r)$ and q occurring in $B^+(r)$ or $B^\Sigma(r)$ is acyclic.

Example 1. Given a connected undirected graph G encoded by predicate *edge*/*2*, source and sink nodes encoded by predicates *source*/*1* and *sink*/*1*, the following program assigns a direction to each edge so that source nodes can still reach all sink nodes:

$$1 \leq \{arc(X,Y);\ arc(Y,X)\} \leq 1 \leftarrow edge(X,Y) \tag{4}$$
$$reach(X,X) \leftarrow source(X) \tag{5}$$
$$reach(X,Y) \leftarrow reach(X,Z),\ arc(Z,Y) \tag{6}$$
$$\bot \leftarrow source(X),\ sink(Y),\ not\ reach(X,Y) \tag{7}$$

If failures on the reachability condition are permitted up to a given threshold encoded by predicate *threshold*/*1*, the program comprising rules (4)–(6) and

$$fail(X,Y) \leftarrow source(X),\ sink(Y),\ not\ reach(X,Y) \tag{8}$$
$$\bot \leftarrow threshold(T), sum\{1,X,Y : fail(X,Y)\} > T \tag{9}$$

can be used. Note that X and Y are local variables in rule (9), and all other variables are global. ■

A substitution σ is a partial function from variables to constants; the application of σ to an expression E is denoted by $E\sigma$. Let *instantiate*(Π) be the program obtained from rules of Π by substituting global variables with constants in $\mathbf{C}$, in all possible ways; note that local variables are still present in *instantiate*(Π). The Herbrand base of Π, denoted *base*(Π), is the set of ground atoms (i.e., atoms with no variables) occurring in *instantiate*(Π).

Example 2. Let Π_{run} comprise rules (4)–(6), (8)–(9) and the facts (i.e., rules with an empty body) $edge(a,b)$, $edge(a,d)$, $edge(d,c)$, $source(a)$, $source(b)$, $sink(c)$, and $threshold(0)$ (see Figure 1).

Fig. 1: The undirected graph used as running example. Source vertices in blue, sink vertex in red. The goal is to assign directions to edges so that all source nodes still reach all sink nodes.

Hence, *instantiate*(Π_{run}) contains, among others, the rules

$$1 \leq \{arc(a,b);\ arc(b,a)\} \leq 1 \leftarrow edge(a,b)$$
$$\bot \leftarrow threshold(0), sum\{1,X,Y : fail(X,Y)\} > 0$$

and *base*(Π_{run}) contains $fail(a,c)$, $fail(b,c)$, and so on. ■

A *(two-valued) interpretation* is a set of ground atoms. For a two-valued interpretation I, relation $I \models \cdot$ is defined as follows: for a ground atom $p(\overline{c})$, $I \models p(\overline{c})$ if $p(\overline{c}) \in I$, and $I \models \mathit{not}\ p(\overline{c})$ if $p(\overline{c}) \notin I$; for an aggregate α of the form (1), the aggregate set of α w.r.t. I, denoted $\mathit{aggset}(\alpha, I)$, is $\{\langle t_a, \overline{t'}\rangle\sigma \mid p(\overline{t})\sigma \in I$, for some substitution $\sigma\}$, and $I \models \alpha$ if $(\sum_{\langle c_a, \overline{c'}\rangle \in \mathit{aggset}(\alpha,I)} c_a) \odot t_g$ is a true expression over integers; for a choice α of the form (2), $I \models \alpha$ if $t_1 \leq |I \cap \mathit{atoms}| \leq t_2$ is a true expression over integers; for a rule r with no global variables, $I \models B(r)$ if $I \models \alpha$ for all $\alpha \in B(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \mathit{instantiate}(\Pi)$.

For a rule r of the form (3) and an interpretation I, let $\mathit{expand}(r, I)$ be the set $\{p(\overline{c}) \leftarrow \mathit{body} \mid p(\overline{c}) \in I$ occurs in $H(r)\}$. The *reduct* of Π w.r.t. I is the program comprising the expanded rules of $\mathit{instantiate}(\Pi)$ whose body is true w.r.t. I, that is, $\mathit{reduct}(\Pi, I) := \bigcup_{r \in \mathit{instantiate}(Pi),\ I \models B(r)} \mathit{expand}(r, I)$. An *answer set* of Π is an interpretation A such that $A \models \Pi$ and no $I \subset A$ satisfies $I \models \mathit{reduct}(\Pi, A)$.

Example 3. The only answer set A_{run} of program Π_{run} contains, among others, the atoms $\mathit{arc}(b,a)$, $\mathit{arc}(a,d)$, $\mathit{arc}(d,c)$, no other instance of $\mathit{arc}/2$, and no instance of $\mathit{fail}/2$ (see Figure 2). Hence, $A_{run} \models 1 \leq \{\mathit{arc}(a,b); \mathit{arc}(b,a)\} \leq 1$ and $A_{run} \not\models \mathit{sum}\{1, X, Y : \mathit{fail}(X,Y)\} > 0$.

Fig. 2: The directed graph solution of the running example. All source nodes (in blue) reach all sink nodes (in red).

■

A *three-valued interpretation* is a pair (L, U), where L, U are sets of ground atoms such that $L \subseteq U$; also let $(L,U)_1$ denote the lower bound L of (L,U) and $(L,U)_2$ denote the upper bounds U of (L,U), so atoms in L are true, atoms in $U \setminus L$ are undefined, and all other atoms are false. The *evaluation function* $[\![\cdot]\!]_L^U$ associates literals and aggregates with a truth value among **u**, **t** and **f** as follows: $[\![\alpha]\!]_L^U = \mathbf{u}$ if α is a literal whose atom is $p(\overline{c})$ and $p(\overline{c}) \in U \setminus L$, or α is an aggregate of the form (1) and $\mathit{aggset}(\alpha, U \setminus L) \neq \emptyset$, or α is a choice of the form (2) and $(U \setminus L) \cap \mathit{atoms} \neq \emptyset$; $[\![\alpha]\!]_L^U = \mathbf{t}$ if $[\![\alpha]\!]_L^U \neq \mathbf{u}$ and $L \models \alpha$; and $[\![\alpha]\!]_L^U = \mathbf{f}$ if $[\![\alpha]\!]_L^U \neq \mathbf{u}$ and $L \not\models \alpha$. The evaluation function extends to rule bodies as follows: $[\![B(r)]\!]_L^U = \mathbf{f}$ if there is $\alpha \in B(r)$ such that $[\![\alpha]\!]_L^U = \mathbf{f}$; $[\![B(r)]\!]_L^U = \mathbf{t}$ if $[\![\alpha]\!]_L^U = \mathbf{t}$ for all $\alpha \in B(r)$; otherwise $[\![B(r)]\!]_L^U = \mathbf{u}$.

Example 4. For α being $\mathit{sum}\{1, X, Y : \mathit{fail}(X,Y)\} > 0$, $[\![\alpha]\!]_{\emptyset}^{\{\mathit{fail}(a,c)\}} = \mathbf{u}$, $[\![\alpha]\!]_{\{\mathit{fail}(a,c)\}}^{\{\mathit{fail}(a,c)\}} = \mathbf{t}$, and $[\![\alpha]\!]_{\emptyset}^{\emptyset} = \mathbf{f}$. ■

Mainstream ASP systems compute answer sets of a given program Π by applying several inference rules on (a subset of) $\mathit{instantiate}(\Pi)$, the most relevant ones for this work summarized below. Let (L,U) be a three-valued interpretation, and $p(\overline{c})$ be a ground atom such that $[\![p(\overline{c})]\!]_L^U = \mathbf{u}$. Atom $p(\overline{c})$ in $H(r)$ is *inferred true by support* if $[\![B(r)]\!]_L^U = \mathbf{t}$. (Actually, if $H(r)$ is a choice of the form (2), inference by support additionally requires that $|\mathit{atoms} \cap U| = t_1$, that is, undefined atoms in $\mathit{atoms} \cap U$ are required to reach the bound t_1. Such extra condition is not relevant for our work,

and will not be used, because our explanations aim at associating true atoms with rules with true bodies.) Atom $p(\overline{c})$ is *inferred false by lack of support* if each rule $r \in \mathit{instantiate}(\Pi)$ with $p(\overline{c})$ occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$. Atom $p(\overline{c})$ is *inferred false by a constraint-like rule* $r \in \mathit{instantiate}(\Pi)$ if $p(\overline{c}) \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{p(\overline{c})\}]\!]_L^U = \mathbf{t}$. Atom $p(\overline{c})$ is *inferred false by a choice rule* $r \in \mathit{instantiate}(\Pi)$ if $H(r)$ has the form (2), $p(\overline{c}) \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. Atom $p(\overline{c})$ is *inferred false by well-founded computation* if it belongs to some *unfounded set* X for Π w.r.t. (L,U), that is, a set X such that for all rules $r \in \mathit{instantiate}(\Pi)$ at least one of the following conditions holds: (i) no atom from X occurs in $H(r)$; (ii) $[\![B(r)]\!]_L^U = \mathbf{f}$; (iii) $B^+(r) \cap X \neq \emptyset$.

Example 5. Given the program $\mathit{instantiate}(\Pi_{run})$, and the three-valued interpretation $(\emptyset, \mathit{base}(\Pi_{run}))$, atom $\mathit{edge}(a,a)$ is inferred false by lack of support, atom $\mathit{source}(a)$ is inferred true by support, and the set $\{\mathit{edge}(a,a), \mathit{arc}(a,a)\}$ is unfounded. Given $(\{\mathit{arc}(d,c)\}, \mathit{base}(\Pi_{run}) \setminus \{\mathit{reach}(a,c)\})$, atom $\mathit{reach}(a,d)$ is inferred false by the constraint-like rule (6), and $\mathit{arc}(c,d)$ is inferred false by the choice rule (4). ■

3 Explanations

Let Π be a program, and A be one of its answer sets. A *well-founded derivation* for Π w.r.t. A, denoted $\mathit{wf}(\Pi, A)$, is obtained from the interpretation $(\emptyset, \mathit{base}(\Pi))$ by iteratively (i) adding to its lower bound atoms of A that are inferred true by support, and (ii) removing from its upper bound atoms belonging to some unfounded set. Note that $\mathit{wf}(\Pi, A)$ is computed as a preprocessing step.

Example 6. Given Π_{run} and A_{run} from Examples 2–3, the lower bound of $\mathit{wf}(\Pi_{run}, A_{run})$ contains head atoms in Example 2, $\mathit{arc}(b,a)$, $\mathit{arc}(a,d)$, $\mathit{arc}(d,c)$, $\mathit{reach}(a,a)$, $\mathit{reach}(b,b)$, $\mathit{reach}(a,d)$, $\mathit{reach}(a,c)$, $\mathit{reach}(b,a)$, $\mathit{reach}(b,c)$, and $\mathit{reach}(b,d)$. Indeed, according to our definition of well-founded derivation for Π w.r.t. A, if the body of a rule is inferred true, then all head atoms belonging to A are inferred true because they are supported in $\mathit{reduct}(\Pi, A)$. The upper bound additionally contains $\mathit{arc}(a,b)$, $\mathit{arc}(d,a)$, $\mathit{arc}(c,d)$, and several instances of $\mathit{reach}/2$ and $\mathit{fail}/2$. ■

An *explaining derivation* for Π and A from (L,U) is obtained by iteratively (i) adding to L atoms of A that are inferred true by support, and (ii) removing from U atoms that are inferred false by lack of support, constraint-like rules and choice rules. An *assumption set* for Π and A is a set $X \subseteq \mathit{base}(\Pi) \setminus A$ of ground atoms such that the explaining derivation for Π and A from $(\emptyset, \mathit{wf}(\Pi, A)_2 \setminus X)$ terminates with A (in words, A is reconstructed from the false atoms of the well-founded derivation extended with X). Let $\mathit{AS}(\Pi, A)$ be the set of assumption sets for Π and A. A *minimal assumption set* for Π, A and a ground atom α is a set $X \in \mathit{AS}(\Pi, A)$ such that $X' \subset X$ implies $X' \notin \mathit{AS}(\Pi, A)$, and $\alpha \in X$ implies $\alpha \in X'$ for all $X' \in \mathit{AS}(\Pi, A)$. (In other words, we prefer assumption sets not including the atom to explain. When all assumption sets include the atom to explain, we opt for the singleton comprising the atom to explain alone.) Let $\mathit{MAS}(\Pi, A, \alpha)$ be the set of minimal assumption sets for Π, A and α.

Example 7. Set $base(\Pi_{run}) \setminus A_{run}$ is an assumption set for Π_{run} and its answer set A_{run}. It can be checked that also $\emptyset \in AS(\Pi_{run}, A_{run}, \alpha)$, and it is indeed the only minimal assumption set in this case, for any atom in $base(\Pi_{run})$. ■

Given an assumption set X and an explaining derivation from $(\emptyset, wf(\Pi, A)_2 \setminus X)$, a directed acyclic graph (DAG) can be obtained as follows: The vertices of the graph are the atoms in $base(\Pi)$ and the aggregates occurring in $instantiate(\Pi)$. (The vertex $p(\overline{c})$ is also referred to as *not* $p(\overline{c})$.) Any aggregate of the form (1) is linked to instances of $p(\overline{t})$. Atoms inferred true by support due to a rule $r \in instantiate(\Pi)$ are linked to elements of $B(r)$. Any atom α inferred false by lack of support is linked to an element of $B(r)$ that is inferred false before α, for each rule $r \in instantiate(\Pi)$ such that α occurs in $H(r)$. Any atom α inferred false by a constraint-like rule $r \in instantiate(\Pi)$ is linked to the atoms occurring in $H(r)$ and the elements of $B(r) \setminus \{\alpha\}$. Any atom α inferred false by a choice rule $r \in instantiate(\Pi)$ is linked to the atoms occurring in $H(r)$ that are true in A, and to the elements of $B(r)$. A portion of an example DAG is reported in Figure 3.

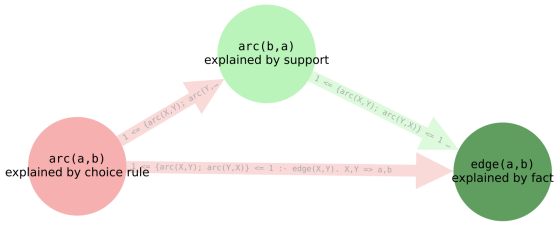

Fig. 3: Induced DAG on the vertices reachable from $arc(a,b)$ for the minimal assumption set $\emptyset$ for Π_{run}.

4 Existence of Minimal Assumption Sets

This section is devoted to formally show that the existence of minimal assumption sets is guaranteed, and so are DAGs as defined in the previous section.

Theorem 1 (Main Theorem). *Let* Π *be a program,* A *one of its answer sets, and* α *a ground atom in* $base(\Pi)$. *Set* $MAS(\Pi, A, \alpha)$ *is nonempty.*

To prove the above theorem, we introduce some additional notation and claims. Let Π be a program, and (L,U) be a three-valued interpretation. We denote by $\Pi, L, U \vdash \alpha$ the fact that $\alpha \in base(\Pi)$ is inferred true by support, which is the case when $[\![\alpha]\!]^U_L = \mathbf{u}$, and there is $r \in instantiate(\Pi)$ such that α occurs in $H(r)$ and $[\![B(r)]\!]^U_L = \mathbf{t}$, as defined in Section 2. Similarly, we denote by $\Pi, L, U \vdash not\ \alpha$ the fact that $\alpha \in base(\Pi)$ is inferred false by lack of support, constraint-like rules and choice rules, which is the case when $[\![\alpha]\!]^U_L = \mathbf{u}$, and one of the following conditions holds: each rule $r \in instantiate(\Pi)$ with α occurring in $H(r)$ is such that $[\![B(r)]\!]^U_L = \mathbf{f}$; there is $r \in instantiate(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]^U_L = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]^U_L = \mathbf{t}$; there is $r \in instantiate(\Pi)$ with $H(r)$ of the form (2), $\alpha \in atoms$, $|atoms \cap L| \geq t_2$ and $[\![B(r)]\!]^U_L = \mathbf{t}$.

The *explaining derivation operator* D_Π is defined as

$$D_\Pi(L,U) := (L \cup \{\alpha \in \mathit{base}(\Pi) \mid \Pi, L, U \vdash \alpha\}, U \setminus \{\alpha \in \mathit{base}(\Pi) \mid \Pi, L, U \vdash \mathit{not}\ \alpha\}).$$

Let $(L,U) \sqsubseteq (L',U')$ denote the fact that $L \subseteq L' \subseteq U' \subseteq U$, i.e., everything that is true w.r.t. (L,U) is true w.r.t. (L',U'), and everything that is false w.r.t. (L,U) is false w.r.t. (L',U').

Lemma 1. *Operator D_Π is monotonic w.r.t. $\sqsubseteq$.*

Proof. For $(L,U) \sqsubseteq (L',U')$, we shall show that $D_\Pi(L,U) \sqsubseteq D_\Pi(L',U')$ holds. For $\alpha \in D_\Pi(L,U)_1 \setminus L$ such that $\alpha \notin L'$, we have $\Pi, L, U \vdash \alpha$, that is, there is $r \in \mathit{instantiate}(\Pi)$ such that α occurs in $H(r)$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, we have that $[\![B(r)]\!]_{L'}^{U'} = \mathbf{t}$, that is, $\Pi, L', U' \vdash \alpha$ holds, and therefore $\alpha \in D_\Pi(L',U')_1 \setminus L$.

For $\alpha \in U \setminus D_\Pi(L,U)_2$ such that $\alpha \in U'$, we have $\Pi, L, U \vdash \mathit{not}\ \alpha$, and therefore we have three cases:

1. Each rule $r \in \mathit{instantiate}(\Pi)$ with α occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$. As $(L,U) \sqsubseteq (L',U')$, $[\![B(r)]\!]_{L'}^{U'} = \mathbf{f}$ holds.
2. There is $r \in \mathit{instantiate}(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, $[\![H(r)]\!]_{L'}^{U'} = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_{L'}^{U'} = \mathbf{t}$.
3. There is $r \in \mathit{instantiate}(\Pi)$ with $H(r)$ of the form (2), $\alpha \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$. As $(L,U) \sqsubseteq (L',U')$, $|\mathit{atoms} \cap L'| \geq t_2$ and $[\![B(r)]\!]_{L'}^{U'} = \mathbf{t}$.

In any case, $\Pi, L', U' \vdash \alpha$ holds, and therefore $\alpha \in U' \setminus D_\Pi(L',U')_2$. □

Lemma 2. *$L \subseteq A \subseteq U$ implies $D_\Pi(L,U)_1 \subseteq A \subseteq D_\Pi(L,U)_2$.*

Proof. For $\alpha \in D_\Pi(L,U)_1 \setminus L$ we have $\Pi, L, U \vdash \alpha$, that is, there is $r \in \mathit{instantiate}(\Pi)$ such that $[\![B(r)]\!]_L^U = \mathbf{t}$. Hence, $A \models B(r)$, and therefore $\mathit{expand}(r, A) \subseteq \mathit{reduct}(\Pi, A)$. In particular, $\alpha \leftarrow B(r)$ belongs to the reduct, and therefore $\alpha \in A$.

For $\alpha \in U \setminus D_\Pi(L,U)_2$ we have $\Pi, L, U \vdash \mathit{not}\ \alpha$ and we have to show that $\alpha \notin A$. Three cases:

1. Each rule $r \in \mathit{instantiate}(\Pi)$ with α occurring in $H(r)$ is such that $[\![B(r)]\!]_L^U = \mathbf{f}$.
2. There is $r \in \mathit{instantiate}(\Pi)$ with $\alpha \in B^+(r)$, $[\![H(r)]\!]_L^U = \mathbf{f}$ and $[\![B(r) \setminus \{\alpha\}]\!]_L^U = \mathbf{t}$.
3. There is $r \in \mathit{instantiate}(\Pi)$ with $H(r)$ of the form (2), $\alpha \in \mathit{atoms}$, $|\mathit{atoms} \cap L| \geq t_2$ and $[\![B(r)]\!]_L^U = \mathbf{t}$.

In the first case, $A \setminus \{\alpha\} \models \mathit{reduct}(\Pi, A)$, and therefore $A \setminus \{\alpha\} = A$ because A is an answer set of Π. In the other two cases, $\alpha \notin A$ because $A \models \Pi$ by assumption. □

The explaining derivation from (L,U) is obtained as the fix point of the sequence $(L_0,U_0) := (L,U)$, $(L_{i+1},U_{i+1}) := D_\Pi(L_i,U_i)$ for $i \geq 0$. Note that the fix point is reached in at most $|\mathit{base}(\Pi)|$ steps because of Lemma 1 and each application of D_Π reduces the undefined atoms (or is a fix point). Thus, the system eventually terminates in at most $|\mathit{base}(\Pi)|$ steps.

Lemma 3. *For any answer set A of Π, set $\mathit{base}(\Pi) \setminus A$ is an assumption set for Π and A.*

Proof. Let (L,U) be the explaining derivation from $(\emptyset, base(\Pi)\setminus A)$. Thanks to Lemma 2, it is sufficient to show that $p(\overline{c}) \in A$ implies $p(\overline{c}) \in L$. Due to the assumption of stratified aggregates, let us consider a topological ordering $C_1,\ldots,C_n$ $(n \geq 1)$ for the strongly connected components of $\mathcal{G}_\Pi$, and let $p \in C_i$. We use induction on i. Since $p(\overline{c}) \in A$, there must be $r \in reduct(\Pi, A)$ such that $H(r) = p(\overline{c})$ and $A \models B(r)$. Hence, $[\![B^-(r)]\!]_L^U = \mathbf{t}$. Moreover, $[\![B^\Sigma(r)]\!]_L^U = \mathbf{t}$, either because $i = 1$ and $B^\Sigma(r) = \emptyset$, or because of the induction hypothesis. Therefore, to have $\alpha \notin L$, it must be the case that $[\![B^+(r)]\!]_L^U \neq \mathbf{t}$ for all such rules, but in this case $L \models reduct(\Pi, A)$, a contradiction with the assumption that A is an answer set of Π. □

Given Lemma 3, the proof of Main Theorem is immediate by the definition of $MAS(\Pi, A, \alpha)$ as following:

Proof (Proof of Main Theorem.). By definition, a minimal assumption set for Π, A and α is a set $X \in AS(\Pi, A)$ such that $X' \subset X$ implies $X' \notin AS(\Pi, A)$, and $\alpha \in X$ implies $\alpha \in X'$ for all $X' \in AS(\Pi, A)$. Lemma 3 guarantees the existence of an assumption set for Π and A. Existence of a minimal assumption set for Π, A and α is therefore guaranteed. □

5 Generation via Meta-Programming

By leveraging ASP systems, the concepts introduced in Section 3 can be computed. A meta-programming approach is presented in this section, where the full language of ASP is used, including constructs omitted in the previous sections, like weak constraints, uninterpreted functions, conditional literals and @-terms. The reader is referred to [11] for details. We will use the name *ASP programs* for encodings using the full language of ASP, in contrast to the name *program* that we use for encodings using the restricted syntax introduced in Section 2.

Program Π, answer set A and the atom to explain are encoded by a set of facts obtained by computing the unique answer set of the ASP program $serialize(\Pi, A, \alpha)$, defined next. Each atom $p(\overline{c})$ in $base(\Pi)$ is encoded by a fact `atom(`$p(\overline{c})$`)`; moreover, the encoding includes a fact `true(`$p(\overline{c})$`)` if $p(\overline{c}) \in A$, and `false(`$p(\overline{c})$`)` otherwise; additionally, if $p(\overline{c})$ is false in $wf(\Pi, A)$, the encoding includes a fact `explained_by(`$p(\overline{c})$`, initial_well_founded)`. As for α, the encoding includes a fact `explain(`α`)`. Each rule r of $instantiate(\Pi)$ is encoded by

`rule(`$id(\overline{X})$`) :- atom(`$p_1(\overline{t_1})$`), ..., atom(`$p_n(\overline{t_n})$`).`

where id is an identifier for r, $\overline{X}$ are the global variables of r, and $B^+(r) = \{p_i(\overline{t_i}) \mid i = 1,\ldots,n\}$; moreover, the encoding includes

`head(`$id(\overline{X})$`,`$p(\overline{t})$`) :- rule(`$id(\overline{X})$`).`
`pos_body(`$id(\overline{X})$`,`$p'(\overline{t'})$`) :- rule(`$id(\overline{X})$`).`
`neg_body(`$id(\overline{X})$`,`$p''(\overline{t''})$`) :- rule(`$id(\overline{X})$`).`

for each $p(\overline{t})$ occurring in $H(r)$, $p'(\overline{t'}) \in B^+(r)$ and $p''(\overline{t''}) \in B^-(r)$; additionally, for each aggregate α of the form (1) in $B^\Sigma(r)$, an identifier agg for α is introduced, and the encoding includes

`pos_body(`$id(\overline{X})$`,`$agg(\overline{X})`$`) :- rule(`$id(\overline{X})$`).`
`aggregate(`$agg(\overline{X})$`) :- rule(`$id(\overline{X})$`).`
`true(`$agg(\overline{X})$`) :- rule(`$id(\overline{X})$`), #sum{`$t_a, \overline{t'}$` : true(`$p(\bar{t})$`)} ⊙ `t_g`.`
`false(`$agg(\overline{X})$`):- rule(`$id(\overline{X})$`), not true(`$agg(\overline{X})$`).`

`rule(`$agg(\overline{X})$`):- aggregate(`$agg(\overline{X})$`),true(`$agg(\overline{X})$`).`
`head(`$agg(\overline{X})$`,`$agg(\overline{X})$`) :- rule(`$agg(\overline{X})$`).`
`pos_body(`$agg(\overline{X})$`,`$p(\bar{t})$`):- rule(`$agg(\overline{X})$`),true(`$p(\bar{t})$`).`
`neg_body(`$agg(\overline{X})$`,`$p(\bar{t})$`):-rule(`$agg(\overline{X})$`),false(`$p(\bar{t})$`).`

`rule((`$agg(\overline{X})$`,`$p(\bar{t})$`)) :- aggregate(`$agg(\overline{X})$`), false(`$agg(\overline{X})$`), atom(`$p(\bar{t})$`).`
`head((`$agg(\overline{X})$`,`$p(\bar{t})$`),`$agg(\overline{X})$`):- rule((`$agg(\overline{X})$`,`$p(\bar{t})$`)).`
`pos_body((`$agg(\overline{X})$`,`$p(\bar{t})$`),`$p(\bar{t})$`) :- rule((`$agg(\overline{X})$`,`$p(\bar{t})$`)), false(`$p(\bar{t})$`).`
`neg_body((`$agg(\overline{X})$`,`$p(\bar{t})$`),`$p(\bar{t})$`) :- rule((`$agg(\overline{X})$`,`$p(\bar{t})$`)), true(`$p(\bar{t})$`).`

finally, if $H(r)$ is a choice of the form (2), the encoding includes

`choice(`$id(\overline{X})$`,`t_1`,`t_2`) :- rule(`$id(\overline{X})$`).`

Note that a true ground aggregate of the form (1) identified by $agg(\overline{c})$ is associated with a single rule whose body becomes true after all instances of $p(\bar{t})$ are assigned the truth value they have in the answer set A; on the other hand, a false aggregate is associated with one rule for each instance of $p(\bar{t})$, whose bodies becomes false when instances of $p(\bar{t})$ are assigned the truth value they have in the answer set A.

Example 8. Recall Π_{run} and A_{run} from Examples 2–3. The ASP program $serialize(\Pi_{run}, A, arc(a,b))$ includes

```
atom(edge(a,b)).  atom(arc(b,a)).  atom (arc(a,b)).
explain(arc(a,b)).
true(edge(a,b)).  true(arc(b,a)).  false(arc(a,b)).

rule(r4(X,Y)) :- atom(edge(X,Y)).
choice(r4(X,Y),1,1) :- rule(r4(X,Y)).
head(r4(X,Y), arc(X,Y)) :- rule(r4(X,Y)).
head(r4(X,Y), arc(Y,X)) :- rule(r4(X,Y)).
pos_body(r4(X,Y), edge(X,Y)):- rule(r4(X,Y)).

aggregate(agg1(T)) :- rule(r9(T)).
true(agg1(T)) :- rule(r9(T)), #sum{1,X,Y : true(fail(X,Y))} > T.
```

and several other rules. The answer set of $serialize(\Pi_{run}, A, arc(a,b))$ includes, among other atoms, `aggregate(agg1(0))` and `false(agg1(0))`. ■

The ASP program Π_{MAS} reported in Figure 4, coupled with a fact for each atom in the answer set of $serialize(\Pi, A, \alpha)$, has optimal answer sets corresponding to cardinality-minimal elements in $MAS(\Pi, A, \alpha)$. Intuitively, line 1 guesses the assumption set,

```
1 {assume_false(Atom)} :- false(Atom), not aggregate(Atom).
2 :~ false(Atom), assume_false(Atom), not explain(Atom). [1@1, Atom]
3 :~ false(Atom), assume_false(Atom), explain(Atom). [1@2, Atom]

4 has_explanation(Atom) :- explained_by(Atom,_).
5 :- atom(X), #count{Reason: explained_by(Atom,Reason)} != 1.

6 explained_by(Atom, assumption) :- assume_false(Atom).

7 {explained_by(Atom, (support, Rule))} :- head(Rule,Atom), true(Atom);
8     true (BAtom) : pos_body(Rule,BAtom); has_explanation(BAtom) :
          pos_body(Rule,BAtom);
9     false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) :
          neg_body(Rule,BAtom).

10 {explained_by(Atom, lack_of_support)} :- false(Atom); false_body(Rule) :
       head(Rule,Atom).
11 false_body(Rule) :- rule(Rule); pos_body(Rule,BAtom), false(BAtom),
       has_explanation(BAtom).
12 false_body(Rule) :- rule(Rule); neg_body(Rule,BAtom), true(BAtom),
       has_explanation(BAtom).

13 {explained_by(Atom, (required_to_falsify_body, Rule))} :- false(Atom),
       not aggregate(Atom);
14     pos_body(Rule,Atom), false_head(Rule); true(BAtom) :
          pos_body(Rule,BAtom), BAtom != Atom;
15     has_explanation(BAtom) : pos_body(Rule,BAtom), BAtom != Atom;
16     false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) :
          neg_body(Rule,BAtom).
17 explained_head(Rule) :- rule(Rule); has_explanation(HAtom) :
       head(Rule,HAtom).
18 false_head(Rule) :- explained_head(Rule), not choice(Rule,_,_);
19     false(HAtom) : head(Rule,HAtom).
20 false_head(Rule) :- explained_head(Rule), choice(Rule, LowerBound,
       UpperBound);
21     not LowerBound <= #count{HAtom' : head(Rule,HAtom'), true(HAtom')}
          <= UpperBound.

22 {explained_by(Atom, (choice_rule, Rule))} :- false(Atom);
23     head(Rule,Atom), choice(Rule, LowerBound, UpperBound);
24     true(BAtom) : pos_body(Rule,BAtom); has_explanation(BAtom) :
          pos_body(Rule,BAtom);
25     false(BAtom) : neg_body(Rule,BAtom); has_explanation(BAtom) :
          neg_body(Rule,BAtom);
26     #count{HAtom : head(Rule, HAtom), true(HAtom),
          has_explanation(HAtom)} = UpperBound.
```

Fig. 4: ASP program Π_{MAS} for computing a minimal assumption set

```
1 link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule));
      pos_body(Rule, BAtom).
2 link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule));
      neg_body(Rule, BAtom).

3 {link(Atom, A) : pos_body(Rule,A), false(A), explained_by(I,A,_), I <
      Index;
4  link(Atom, A) : neg_body(Rule,A), true (A), explained_by(I,A,_), I <
       Index} = 1 :-
5     explained_by(_, Atom, lack_of_support); head(Rule, Atom).

6 link(Atom,A) :- explained_by(_, Atom, (required_to_falsify_body, Rule));
      head(Rule,A).
7 link(At,A) :- explained_by(_,At,(required_to_falsify_body, Rule));
      pos_body(Rule,A), A!=At.
8 link(Atom,A) :- explained_by(_,Atom,(required_to_falsify_body, Rule));
      neg_body(Rule,A).

9 link(Atom, HAtom) :- explained_by(_,Atom,(choice_rule, Rule));
      head(Rule,HAtom), true(HAtom).
10 link(Atom, BAtom) :- explained_by(_, Atom, (choice_rule, Rule));
      pos_body(Rule, BAtom).
11 link(Atom, BAtom) :- explained_by(_, Atom, (choice_rule, Rule));
      neg_body(Rule, BAtom).
```

Fig. 5: ASP program Π_{DAG} for computing a directed acyclic graph associated with an explaining derivation

line 2–3 minimizes the size of the assumption set (preferring to not assume the falsity of the atom to explain), and lines 4–5 impose that each atom must have exactly one explanation. The other rules encode the explaining derivation for Π and A from $wf(\Pi, A) \setminus X$, where X is the guessed assumption set.

Given a minimal assumption set encoded by predicate `assume_false/1`, an explaining derivation can be computed by removing lines 1–3 from the ASP program Π_{MAS}. Let Π_{EXP} be such an ASP program. Finally, given an explaining derivation encoded by `explained_by(Index,Atom,Reason)`, with the additional `Index` argument encoding the order in the sequence, a DAG linking atoms according to the derivation can be computed by the ASP program Π_{DAG} reported in Figure 5.

Example 9. Let Π_S have a fact for each atom in the answer set of $serialize(\Pi_{run}, A_{run}, arc(a,b))$. $\Pi_{MAS} \cup \Pi_S$ generates the empty assumption set. $\Pi_{EXP} \cup \Pi_S \cup \emptyset$ generates an explaining derivation, for example one including $explained\_by(edge(a,b),(support, r6))$, $explained\_by(arc(b,a),(support, r1(a,b)))$ and $explained\_by(arc(a,b),(choice\_rule, r1(a,b)))$. Let Π_E have a fact for each instance of $explained\_by/3$ in the explaining derivation. $\Pi_{DAG} \cup \Pi_S \cup \Pi_E$ generates a DAG, for example one including $link(arc(b,a), edge(a,b))$, $link(arc(a,b), arc(b,a))$ and $link(arc(a,b), edge(a,b))$. ■

6 Implementation and Experiment

We deployed an XAI system for ASP named `xASP2` , which is powered by the `clingo python api` [16]. By taking an ASP program Π, one of its answer sets A, and an atom α as input, `xASP2` is capable of producing minimal assumption sets, explaining derivations, and DAGs as output to assist the user in determining the assignment of α. The source code is available at `https://github.com/alviano/xasp` and an example DAG is given at `https://xasp-navigator.netlify.app/`.

The pipeline implemented by `xASP2` starts with the serialization of the input data, which is obtained by means of an ASP program crafted from the abstract syntax tree of Π and whose answer set identifies the relevant portion of *instantiate*(Π) and *base*(Π). In a nutshell, ground atoms provided by the user, $A \cup \{\alpha\}$, are part of *base*(Π) and used to instantiate rules of Π (by matching positive body literals), which in turn may extend *base*(Π) with other ground atoms occurring in the instantiated rules; possibly, some atoms of *base*(Π) of particular interest can be explicitly provided by the user. Aggregates are also processed automatically by means of an ASP program, and so is the computation of false atoms in the well-founded derivation $wf(\Pi, A)$.

Obtained *serialize*(Π, A, α), `xASP2` proceeds essentially as described in Section 5, by computing a minimal assumption set, an explaining derivation and an explanation DAG. As an additional optimization, the explaining derivation is shrunk to the atoms reachable from α, utilizing an ASP program. Finally, the user can opt for a few additional steps: obtain a graphical representation by means of the `igraph` network analysis package (`https://igraph.org/`); obtain an interactive representation in `https://xasp-navigator.netlify.app/`; ask for different minimal assumption sets, explaining derivations and DAGs.

We assessed `xASP2` empirically on the commercial application that we mentioned in the introduction. The ASP program of the commercial application can be found in the Github repository. The ASP program comprises 420 rules and 651 facts. After grounding, there are 4261 ground rules and 4468 ground atoms. The program was expected to have a unique answer set, but two answer sets were actually computed. Our experiment was run on an Intel Core i7-1165G7 @2.80 GHz and 16 GB of RAM. `xASP2` computed a DAG for the unexpected true atom, `behaves_inertially (testing_posTestNeg,121)`, in 14.85 seconds on average, over 10 executions. The DAG comprises 87 links, 45 internal nodes and 20 leaves, only one of which is explained by assumption; only 30 of the 420 symbolic rules and 11 of the 651 facts are involved in the DAG; at the ground level, only 48 of the 4261 ground rules and 65 of the 4468 ground atoms are involved. Additionally, we repeated the experiment on 10 randomly selected atoms with respect to two different answer sets, repeating each test case 10 times. We measured an average runtime of 14.79 seconds, with a variance of 0.004 seconds.

As a second experiment, we considered the Latin Square instances reported in Figure 6, whose encodings are shown in Figures 7–8, and queries for each part of the computed solution, for a total of 97 queries. We recall that a Latin Square is a NxN grid with values from the integer interval 1..N and no repeated entries in any row or column. Tests were run on an AMD EPYC 7313 3GHz with 2TB of RAM, allowing 600

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

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

Fig. 6: Latin Square instances used in our experiment. Given clues in black. Computed solution in blue.

```
given((1,1), 3). given((2,4), 2). given((3,1), 1). given((4,4), 1).

assign((Row, Col), Value) :- Row = 1..4; Col = 1..4; Value = 1..4;
    not assign'((Row, Col), Value).
assign'((Row, Col), Value) :- Row = 1..4; Col = 1..4; Value = 1..4;
    not assign((Row, Col), Value).
:- assign(Cell, Value), assign(Cell, Value'), Value < Value'.
:- Row = 1..4; Col = 1..4; Cell = (Row, Col); assign'(Cell, 1);
     assign'(Cell, 2); assign'(Cell, 3); assign'(Cell, 4).
:- given(Cell, Value), assign'(Cell, Value).

:- block(Block, Cell); block(Block, Cell'), Cell != Cell';
    assign(Cell, Value), assign(Cell', Value).
at_least_one(Block, Value) :- block(Block, Cell); assign(Cell, Value).
at_least_one'(Block, Value) :- block(Block, Cell); Value = 1..4;
    not at_least_one(Block, Value).
:- block(Block, Cell); Value = 1..4; at_least_one'(Block, Value).

block((row, Row), (Row, Col)) :- Row = 1..4, Col = 1..4.
block((col, Col), (Row, Col)) :- Row = 1..4, Col = 1..4.
```

Fig. 7: ASP encoding associated with the 4x4 instance of Latin Square shown in Figure 6.

```
 1 given((1, 1), 6). given((1, 3), 9). given((1, 4), 8). given((1, 6), 7).
 2 given((2, 4), 6). given((2, 9), 1).
 3 given((3, 2), 3). given((3, 3), 5). given((3, 6), 2). given((3, 8), 7).
 4 given((4, 2), 6). given((4, 3), 8). given((4, 7), 1). given((4, 9), 2).
 5 given((5, 1), 3). given((5, 6), 5).
 6 given((6, 4), 2). given((6, 7), 3). given((6, 8), 6).
 7 given((7, 1), 8). given((7, 2), 5). given((7, 3), 4). given((7, 4), 7).
       given((7, 5), 2). given((7, 7), 6). given((7, 8), 9).
 8 given((8, 4), 5). given((8, 5), 9). given((8, 9), 8).
 9 given((9, 1), 2). given((9, 3), 6). given((9, 4), 4). given((9, 5), 3).
       given((9, 7), 7). given((9, 8), 1). given((9, 9), 5).

10 assign((Row, Col), Value) :- Row = 1..9; Col = 1..9; Value = 1..9;
11     not assign'((Row, Col), Value).
12 assign'((Row, Col), Value) :- Row = 1..9; Col = 1..9; Value = 1..9;
13     not assign((Row, Col), Value).
14 :- assign(Cell, Value), assign(Cell, Value'), Value < Value'.
15 :- Row = 1..9; Col = 1..9; Cell = (Row, Col); assign'(Cell, 1);
16     assign'(Cell,2); assign'(Cell,3); assign'(Cell,4); assign'(Cell,5);
17     assign'(Cell,6); assign'(Cell,7); assign'(Cell,8); assign'(Cell,9).
18 :- given(Cell, Value), assign'(Cell, Value).

19 :- block(Block, Cell); block(Block, Cell'), Cell != Cell';
20     assign(Cell, Value), assign(Cell', Value).
21 at_least_one(Block, Value) :- block(Block, Cell); assign(Cell, Value).
22 at_least_one'(Block, Value) :- block(Block, Cell); Value = 1..9;
23     not at_least_one(Block, Value).
24 :- block(Block, Cell); Value = 1..9; at_least_one'(Block, Value).

25 block((row, Row), (Row, Col)) :- Row = 1..9, Col = 1..9.
26 block((col, Col), (Row, Col)) :- Row = 1..9, Col = 1..9.
```

Fig. 8: ASP encoding associated with the 9x9 instance of Latin Square shown in Figure 6.

seconds and 16GB. The 16 queries associated with the 4x4 instance are answered in around 1.38 seconds on average, using around 68MB of RAM. The produced explanation graphs have around 77 links on average, and use an assumption set of size 1. The 81 queries associated with the 9x9 instance are answered in around 38 seconds on average, using around 438MB of RAM. The produced explanation graphs have around 527 links on average, and use an assumption set of size 2. Details are reported online (`https://asp-chef.alviano.net/s/xasp/jlc2024`) together with links to the produced explanation graphs.

`xASP2` also has the ability to handle explainable planning, meaning it can generate an explanation graph showing why a particular action cannot take place at a certain time. To demonstrate this capability, we will use a popular problem known as Blocksworld. The initial state (left) and goal state (right) of the problem are shown

Table 1: The action preconditions and effects in Blocksworld problem

Action	**Precondition**	**Effects**
$stack(X,Y)$ - stack block X is on block Y	Block Y is clear The agent holds the block X	X is clear X is on Y Y is no longer clear The agent does not hold anything
$unstack(X,Y)$ - unstack block X is on block Y	X is clear X is on Y The agent does not hold anything	The agent holds the block X Y becomes clear X is not clear
$pickup(X)$ - pickup block X from the table	X is clear X is on the table the agent does not hold anything	The agent holds the block X X is no longer on the table and is not clear
$putdown(X)$ - put down block X onto the table	The agent holds the block X	X is clear X is on the table the agent does not hold anything

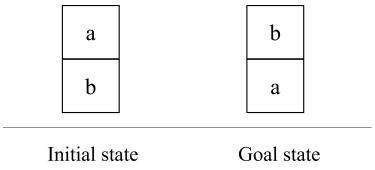

Fig. 9: The initial and goal states of Blocksworld.

in Figure 9. Five fluents are $on(X,Y)$ - block X is on block Y, $onTable(X)$ - block X is on the table, $clear(X)$ - block X is clear, $holding(X)$ - the agent holds the block X, and $handEmpty$ - the agent does not hold anything. Four different actions are $stack$, $unstack$, $pickup$ and $putdown$. The domain description of the problem is shown in Table 1 in which the predictions and effects of four actions are presented.

```
1 h(X,T+1) :- action(action(A)),occurs(A,T), postcondition(action(A),
      effect(unconditional),X,value(X,true)).
2 -h(X,T+1) :- action(action(A)),occurs(A,T), postcondition(action(A),
      effect(unconditional),X,value(X,false)).
3 h(X,T+1) :- h(X,T), not -h(X,T+1).
4 -h(X,T+1) :- -h(X,T), not h(X,T+1).
5 non_exec(A,T) :- action(action(A)), not h(X,T),
      precondition(action(A),X,value(X, true)).
6 non_exec(A,T) :- action(action(A)), not -h(X,T),
      precondition(action(A),X,value(X, false)).
7 :- action(action(A)),occurs(A,T), non_exec(A,T).
```

Fig. 10: ASP program for reasoning about effects of actions [21]

The rules for reasoning about effects of actions, action generation and goal enforcement [21] are utilized as programming input in `xASP2` . Figure 10 shows the ASP program for reasoning about the effects of actions in which an action occurs only when its preconditions are true and then its effects are true in the next time step. Specifically, lines 5 and 6 are used to define states in which an action cannot be executed, and constraint is employed to prevent non-executable actions from occurring (line 7).

For the problem described in Figure 9, executing the actions of $unstack(a,b)$, $putdown(a)$, $pickup(b)$ and $stack(b,a)$ at times 0, 1, 2, and 3 respectively constitutes the optimal plan (assuming time starts at 0). However, if users are in a rush and want to put down block a on the table at time 0, as represented by the atom $occurs(("putdown",constant("a")),0)$, they will encounter a false occurrence of the action $putdown(a)$. Figure 11 shows that atom $occurs(("putdown",constant("a")),0)$ is false because the constraint rule prevents its execution and the prediction of the action holding block a is invalid/false.

Fig. 11: The DAG for atom $occurs(("putdown",constant("a")),0)$.

6.1 xASP Navigator

In order to ease the understanding of the DAG produced by `xASP2` , we designed and implemented a web application called `xASP navigator` (`https://gitlab.com/`

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://xasp-navigator.alviano.net/dag.schema.json",
  "title": "DAG",
  "description": "A DAG generated by xASP2",
  "type": "object",
  "properties": {
    "nodes": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "id": { "type": "number" },
                "label": { "type": "string" },
                "x": { "type": "number" },
                "y": { "type": "number" }
            }
        }
    },
    "links": {
        "type": "array",
        "items": {
            "type": "object",
            "properties": {
                "source": { "type": "number" },
                "target": { "type": "number" },
                "label": { "type": "string" }
            }
        }
    }
  },
  "required": ["nodes", "links"]
}
```

Fig. 12: JSON Schema of DAGs visualized in `xASP navigator`

`mario.alviano/xasp-navigator/`). The application is written in Svelte (`https://svelte.dev/`), a modern Javascript framework and a concrete alternative to broadly adopted frameworks such as Angular [29].

From the usability perspective, `xASP navigator` provides a minimalist user interface showing the DAG as reported in the figures of this article, and some additional information is given on a side panel. Specifically, the list of rules and facts are shown by default, and moving the pointer to a node expand the shown information to include details on the explanation of the pointed node. Moreover, the user interface includes a filter to highlight nodes and arcs of interest, as well as for restricting the list of rules and facts. Finally, the DAG and the list of rules and facts are represented in the URL so that they can be easily shared with other users, essentially by simply sending them a link.

```
{
  "nodes": [
    {
      "id": 0,
      "label": "edge(a,b)\nsupport",
      "x": 0.5,
      "y": 2
    },
    {
      "id": 1,
      "label": "arc(b,a)\nsupport",
      "x": 0,
      "y": 1
    },
    {
      "id": 2,
      "label": "arc(a,b)\nchoice rule",
      "x": 0.5,
      "y": 0
    }
  ],
  "links": [
    {
      "source": 1,
      "target": 0,
      "label": "1 <= {arc(X,Y); arc(Y,X)} <= 1 :- edge(X,Y).\nX,Y => a,b"
    },
    {
      "source": 2,
      "target": 1,
      "label": "1 <= {arc(X,Y); arc(Y,X)} <= 1 :- edge(X,Y).\nX,Y => a,b"
    },
    {
      "source": 2,
      "target": 0,
      "label": "1 <= {arc(X,Y); arc(Y,X)} <= 1 :- edge(X,Y).\nX,Y => a,b"
    }
  ]
}
```

Fig. 13: JSON encoding of the DAG shown in Figure 3

From the development perspective, `xASP navigator` relies on the D3.js library [5] for visualizing the DAG as a *force-directed* graph [18] with preferred points computed by `xASP2` by applying the Sugiyama layout [27]. The DAG is represented in JSON [6] according to the JSON Schema given in Figure 12. For example, the DAG shown in Figure 3 is represented as the JSON reported in Figure 13. Note that labels of nodes also carry the information about the reason of derivation of each node,

```
1 def show_navigator_graph(self, index: int = -1) -> None:
2     self.compute_igraph(index)
3     url = "https://xasp-navigator.netlify.app/#"
4     json_dump = json.dumps(
5         self.navigator_graph(index),
6         separators=(',', ':')
7     ).encode()
8     url += base64.b64encode(zlib.compress(json_dump)).decode() + '%21'
9     webbrowser.open(url, new=0, autoraise=True)
```

Fig. 14: Method `show_navigator_graph` in `xASP2`

while labels of rules comprise a rule in input (as written in the input program) and a substitution for its global variables. A DAG produced by `xASP2` can be shown in `xASP navigator` by calling the method `show_navigator_graph`, whose implementation is given in Figure 14; essentially, the DAG is computed (if not already done; line 2), the URL is composed by compressing its JSON representation (lines 3–8) and opened in the system browser (line 9).

6.2 Usage

The easiest way to use `xASP2` is via the interface provided by the `Explain` class. An instance of `Explain` must be obtained by means of the factory method `the_program`, passing the program Π, the answer set A and the (true or false) atom to explain α. Obtained an instance of `Explain`, the methods `minimal_assumption_set`, `explanation_sequence` and `explanation_dag` can be used to obtain facts representing the artifacts involved in the explanation process. Additionally, method `show_navigator_graph` can be called to open the graph in the `xASP navigator` application (see Figure 14). A code snippet materializing the running example is shown in Figure 15.

An important aspect to take into account in using `xASP2` is that, by design, true atoms are only explained by support, and choice rules with true bodies are considered a support for all true atoms in their heads. Depending on the program in input, the produced explanation can be oversimplified due to such design choices. For example, consider the program reported in Figure 16 – whose unique answer set comprise all atoms in the program but `missing` – and the explanation shown in Figure 17 for the `query` atom. It can be observed that the `query` atom is explained by the support provided by the rule at line 1, and the body atom `direct_support` is explained by the support provided by the choice rule at line 5. While such an explanation justifies the presence of `query` in the answer set, it can be considered oversimplified as the choice rule does not necessarily enforces the truth of its head atoms; again, this is a design choice, and in this work we opted for justifying the presence and absence of atoms, rather than justifying the *mandatory* presence and absence of atoms. Nonetheless, the program can be rewritten to obtain more detailed explanations. First of all, the choice rule at line 5 can be rewritten in terms of cyclic negation as shown below:

```
from xasp.entities import Explain
from dumbo_asp.primitives import Model

explain = Explain.the_program("""
edge(a,b). edge(a,d). edge(d,c).
source(a). source(b). sink(c). threshold(0).

1 <= {arc(X,Y); arc(Y,X)} <= 1 :- edge(X,Y).
reach(X,X) :- source(X).
reach(X,Y) :- reach(X,Z), arc(Z,Y).
fail(X,Y) :- source(X), sink(Y), not reach(X,Y).
:- threshold(T), #sum{1,X,Y : fail(X,Y)} > T.
    """.strip(),
    the_answer_set=Model.of_program("""
edge(a,b). edge(a,d). edge(d,c).
source(a). source(b). sink(c). threshold(0).
reach(a,a). reach(a,c). reach(a,d).
reach(b,a). reach(b,b). reach(b,c). reach(b,d).
arc(b,a). arc(a,d). arc(d,c).
    """),
    the_atoms_to_explain=Model.of_atoms("arc(a,b)"),
)
# print(explain.minimal_assumption_set())
# print(explain.explanation_sequence())
# print(explain.explanation_dag())
explain.show_navigator_graph()
```

Fig. 15: xASP2 generating the DAG for Π_{run}, A_{run} and query `arc(a,b)`

```
query :- direct_support.
:- indirect_support, not direct_support.
indirect_support :- fact, not missing.
fact.
{direct_support}.
```

Fig. 16: A program leading to the oversimplified explanation shown in Figure 17

Fig. 17: The oversimplified explanation generated for the program reported in Figure 16, its unique answer set and the `query` atom

```
direct_support :- not direct_support'.
direct_support' :- not direct_support.
```

Fig. 18: The oversimplified explanation generated for the program reported in Figure 16 with the rewritten choice rule line 5, its unique answer set and the `query` atom

```
1 query :- direct_support.
2 :- indirect_support, direct_support'.
3 indirect_support :- fact, not missing.
4 fact.
5 direct_support :- not direct_support'.
6 direct_support' :- not direct_support .
```

Fig. 19: The program reported in Figure 16 rewritten by eliminating choice rules and negative literals in constraints

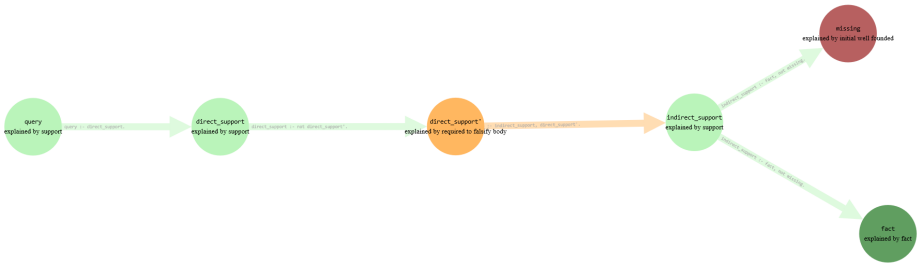

Fig. 20: A more detailed explanation generated using the rewritten program reported in Figure 19, its unique answer set and the `query` atom

This way, the support provided by the choice rule is conditioned by the assumption on the falsity of `direct_support'`, as shown in Figure 18. As a second observation, note that the constraint at line 2 actually enforces truth of `direct_support`, which however we do not capture by design (constraints are only used to justify falsity). In order to capture such an inference, negative literals can be rewritten by introducing (or reusing) auxiliary symbols. In this example, the constraint at line 2 is replaced by

```
:- indirect_support, direct_support'.
```

as in fact `direct_support'` is the complement of `direct_support`. The rewritten program is reported in Figure 19, and the associated explanation is shown in Figure 20.

7 Related Work

As mentioned in the introduction, our work is in the context of XAI, which in turn can be applied to debug by identifying a set of rules that justifies the derivation of a given atom. For example, if an atom α is supposed to be false in all answer sets

Table 2: Summary of compared features

System (if any) and reference	Acyclic explanation	Linguistic extentions	Explanation for false atoms	System availability
`xclingo` [8]	Yes	None	No	Yes
`s(CASP)` [3]	Yes	Constraints	Yes	Yes
`Visual-DLV` [24]	Yes	Constraints	No	Yes
`spock` [7]	Yes	Constraints	No	Yes
`DWASP` [13]	Yes	Constraints	No	Yes
[25]	No	None	Yes	No
[30]	Yes	None	Yes	No
`ASPeRiX` [4]	Yes	Constraints	Yes	Yes
`LABAS` [26]	No	None	Yes	Yes
[20]	No	Aggregates	Yes	No
`xASP2`	Yes	Aggregates and Constraints	Yes	Yes

of a program Π but appears in some answer set A, an explanation graph of α could help to understand which rules are behaving anomalously. Therefore, in this section we consider some debugging tools for ASP, as well as state-of-the-art XAI systems for ASP. Table 2 reports a summary of the compared features: whether the explanation is guaranteed to be acyclic; whether the input program may include aggregates and constraints; whether the query atom can be false in the answer set; and whether the system is available for experimentation.

`xclingo` [8] can generate derivation trees for an atom in an ASP computation. Derivation trees are obtained by adding to the input program `trace_rule` and `trace` annotations, which are then compiled into theory atoms and auxiliary predicates. Then, the explanations are obtained by decoding the answer sets of the modified program. `xclingo` 2.0, the latest release of `xclingo` [9, 10], introduces new annotations: `mute` for atoms and `mute_body` for rules. The annotations serve to prune the edges and exclude nodes explanations, thus aiding in information filtering. While our system maintains edges and nodes in the DAG, it leverages visualization capabilities from `xASP` `navigator` to enhance exploration and navigation. Nonetheless, the possibility to control the granularity of the explanation is an interesting feature that we may include in future releases of the navigator. Differently from our system, `xclingo` and `xclingo` 2.0 do not support some linguistic constructs such as constraint, and cannot include negative literals in their explanations. For example, given a program $\Pi_x = \{\ a.\ \ b \leftarrow a, not\ c.\}$ and its answer set $\{a, b\}$, asking for why b is true, the explanation from `xclingo` 2.0 is as follows:

```
*
|__b
|  |__a
```

The DAG associated with atom b is shown in Figure 21.

`s(CASP)` [3] leverages top-down Prolog computation to generate a justification tree in natural language for Constraint Answer Set programs. Due to Prolog computation, different justifications are produced when the order of atoms in rules or the

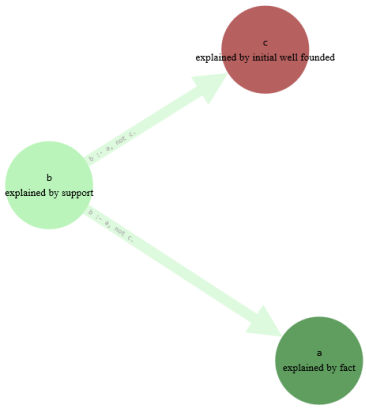

Fig. 21: The DAG for b given the program $\Pi_x = \{a.\ b \leftarrow a, not\ c.\}$ and its answer set $\{a, b\}$.

order of rules in the program is changed [28]. Note that our explanation graphs are not affected by program reordering.

`Visual-DLV` [24] is a GUI for developing and testing `DLV` programs, which in particular provides a command to examine why an atom is true in the latest computed answer set. Such a question is answered by providing the reason that led the solver infer the atom, among them the possibility that the atom is a branching literal (a literal guessed to be true by the backtracking algorithm). Differently from the approach proposed in this paper, in `Visual-DLV` the link with the original program is weak due to several simplifications implemented by the grounder and the solver. Moreover, while our approach minimizes the atoms whose truth value must be assumed, `Visual-DLV` by design does nothing to simplify the amount of data shown to the user to explain the derivation of an atom. For example, the answer set $\{b\}$ of $\Pi_{rw} = \{\ a \leftarrow not\ b.\ \ a \leftarrow b, c.\ \ b \leftarrow not\ a.\ \ c \leftarrow a, b.\}$ may be obtained by branching on *not c*, inferring nothing, and then on *not a*, inferring b. Asking for why c is false, would result in the answer "because *not c* is a branching literal." `xASP2` assumes the falsity of a, from which the truth of b and the falsity of c can be inferred.

`spock` [7] makes use of tagging techniques [12] to translate the input program into a new program whose answer sets can be used to debug the original program. The information reported to the user includes rules whose body is true (applicable rules), rules whose body is false (blocked rules), and abnormality tags associated with completion and loop formulas. For Π_{rw} and the answer set $\{b\}$, `spock` detects the fact that the third rule is applicable, and that the other rules are blocked; the exact reason for which a rule is blocked is not reported. Within this respect, our approach is simpler and focuses on easy-to-understand inference rules that can be clearly visualized via a DAG like the one in Figure 22.

`DWASP` [13] is aimed at identifying a set of rules that are responsible for the absence of an expected answer set. It combines the grounder `gringo` [14] and an extension of the ASP solver `WASP` [1], and introduces the `gringo-wrapper` to "disable" some grounding simplifications. The expected, absent answer set is encoded as a set of constraints, so that its combination with the input program has no answer set at

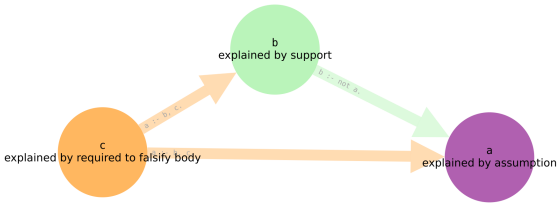

Fig. 22: Induced DAG on the vertices reachable from c for the MAS $\{a\}$ of Π_{rw} (Section 7)

all, and minimal unsatisfiable subsets (MUSes) can be computed. Some questions are asked to the user so to select one MUS that makes more sense to investigate for the absence of the answer set; in fact, at that point the user has a hopefully small set of rules to investigate for bugs.

Explanation graphs can be given in terms of off-line justifications [25, 30], possibly containing cycles among false atoms [25]. For example, given $\Pi_f = \{\ a \leftarrow b.\ b \leftarrow a.\ \}$ and the answer set $\emptyset$, [25] explains the falsity of a by a cycle between a and b; `xASP2` and [30], instead, use the fact that a is false in the well-founded model of Π_f. We also observe that [30] fixes the assumption set to the false atoms that are left undefined by the well-founded model. On-line justifications are produced by `ASPeRiX` [4], which implements a search procedure based on the selection of rules rather than literals. In this case the explanation is produced while searching an answer set, and it is not possible to specify an answer set of interest. Other approaches relying on justifications and resulting in possibly cyclic explanation graphs are based on *assumption-based argumentation*, like `LABAS` [26], or on *trees of systems*, as proposed in [20]. Interestingly, [20] deals with aggregates; however, a system implementing the approach of [20] is not discussed or released.

Finally, comparing `xASP2` with the previous version of `xASP` that lacks support for extended language constructs such as aggregates, we observe that `xASP2` was completely redesigned by replacing several algorithms implemented in procedural programming languages and Prolog with more declarative meta-encoding programming powered by mainstream ASP engines. As can be seen from Table 2, our system is capable of providing explanations for false atoms and does not lead to cyclic argumentation in the explanation. `xASP2` is the only system that tackles a program that includes both aggregates and constraints. Moreover, the explanation DAGs produced by `xASP2` can be visualized in an interactive web user interface that we expect to describe in future publications.

8 Conclusion

We formalized and implemented a system for XAI targeting the ASP language and powered by ASP engines. The presence or absence of an atom in an answer set is explained in terms of easy-to-understand inferences originating from a hopefully small set of atoms assumed false. The explanation is shown as a DAG rooted at the atom to be explained, and can be computed in a few seconds in our test cases. DAGs are shown in the form of an interactive representation in a web browser, and can be eas-

ily shared by sending the URL in the address bar of the browser. The automation of the program rewriting to enrich the generated explanation DAGs constitutes an interesting line of future research.

Acknowledgments

Portions of this publication and research effort are made possible through the help and support of NIST via cooperative agreement 70NANB21H167. Tran Cao Son was also partially supported by NSF awards #1757207, #1914635, and #2151254. Mario Alviano was partially supported by Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23 003420006 under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); and by the LAIA lab (part of the SILA labs). Mario Alviano is member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

1. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in wasp. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 40–54. Springer (2015)
2. Alviano, M., Trieu, L.L., Son, T., Balduccini, M., et al.: Advancements in xasp, an xai system for answer set programming. In: Proceedings of the 38th Italian Conference on Computational Logic, CEUR Workshop Proceedings (2023)
3. Arias, J., Carro, M., Chen, Z., Gupta, G.: Justifications for goal-directed constraint answer set programming. Electronic Proceedings in Theoretical Computer Science **325**, 59–72 (Sep 2020)
4. Béatrix, C., Lefèvre, C., Garcia, L., Stéphan, I.: Justifications and blocking sets in a rule-based answer set computation. In: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
5. Bostock, M., Ogievetsky, V., Heer, J.: D^3 data-driven documents. IEEE Transactions on Visualization and Computer Graphics **17**(12), 2301–2309 (2011). https://doi.org/10.1109/TVCG.2011.185
6. Bourhis, P., Reutter, J.L., Suárez, F., Vrgoč, D.: Json: Data model, query languages and schema specification. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. p. 123–135. PODS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3034786.3056120, https://doi.org/10.1145/3034786.3056120
7. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is illogical captain! the debugging support tool spock for answer-set programs: system description. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). pp. 71–85 (2007)

8. Cabalar, P., Fandinno, J., Muñiz, B.: A system for explainable answer set programming. Electronic Proceedings in Theoretical Computer Science **325**, 124–136 (Sep 2020)
9. Cabalar, P., Muñiz, B.: Explanation graphs for stable models of labelled logic programs. In: Proceedings of the International Conference on Logic Programming 2023 Workshops co-located with the 39th International Conference on Logic Programming (ICLP 2023), London, United Kingdom, July 9th and 10th (2023)
10. Cabalar, P., Muñiz, B.: Model explanation via support graphs. Theory Pract. Log. Program. (2024). https://doi.org/10.1017/S1471068424000048
11. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., Schaub, T.: Asp-core-2 input language format. Theory Pract. Log. Program. **20**(2), 294–309 (2020)
12. Delgrande, J.P., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2), 129–187 (2003)
13. Dodaro, C., Gasteiger, P., Reale, K., Ricca, F., Schekotihin, K.: Debugging non-ground asp programs: Technique and graphical tools. Theory and Practice of Logic Programming **19**(2), 290–316 (2019)
14. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 345–351. Springer (2011)
15. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Warren, D., Szeredi, P. (eds.) Logic Programming: Proc. of the Seventh International Conference. pp. 579–597 (1990)
16. Kaminski, R., Romero, J., Schaub, T., Wanko, P.: How to build your own asp-based system?! Theory and Practice of Logic Programming **23**(1), 299–361 (2023). https://doi.org/10.1017/S1471068421000508
17. Li, F., Wang, H., Basu, K., Salazar, E., Gupta, G.: Discasp: A graph-based asp system for finding relevant consistent concepts with applications to conversational socialbots. arXiv preprint arXiv:2109.08297 (2021)
18. Lin, C.C., Yen, H.C.: A new force-directed graph drawing method based on edge–edge repulsion. Journal of Visual Languages & Computing **23**(1), 29–42 (2012). https://doi.org/https://doi.org/10.1016/j.jvlc.2011.12.001, https://www.sciencedirect.com/science/article/pii/S1045926X11000802
19. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-year Perspective. pp. 375–398 (1999). https://doi.org/10.1007/978-3-642-60085-2_17
20. Marynissen, S., Heyninck, J., Bogaerts, B., Denecker, M.: On nested justification systems (full version). arXiv preprint arXiv:2205.04541 (2022)
21. Nguyen, V., Vasileiou, S.L., Son, T.C., Yeoh, W.: Explainable planning using answer set programming. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. pp. 662–666 (2020). https://doi.org/10.24963/kr.2020/66
22. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3,4), 241–273 (1999). https://doi.org/10.1023/A:1018930122475
23. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory Pract. Log. Program. **7**(3), 301–353 (2007)
24. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing dlv programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). pp. 86–100 (2007)
25. Pontelli, E., Son, T.C., Elkhatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1), 1–56 (2009)

26. Schulz, C., Toni, F.: Justifying answer sets using argumentation. Theory and Practice of Logic Programming **16**(1), 59–110 (2016)
27. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics **11**(2), 109–125 (1981). https://doi.org/10.1109/TSMC.1981.4308636
28. Trieu, L.L., Son, T.C., Balduccini, M.: xasp: An explanation generation system for answer set programming. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 363–369. Springer (2022)
29. Tripon, T.D., Adela Gabor, G., Valentina Moisi, E.: Angular and svelte frameworks: a comparative analysis. In: 2021 16th International Conference on Engineering of Modern Electric Systems (EMES). pp. 1–4 (2021). https://doi.org/10.1109/EMES52337.2021.9484119
30. Viegas Damásio, C., Analyti, A., Antoniou, G.: Justifications for logic programming. In: Logic Programming and Nonmonotonic Reasoning: 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proc. 12. pp. 530–542. Springer (2013)