Answering the "why" in answer set programming - A survey of explanation approaches

Jorge Fandinno, Claudia Schulz

Answering the "why" in Answer Set Programming – A Survey of Explanation Approaches

JORGE FANDINNO
Institut de Recherche en Informatique de Toulouse (IRIT)
Université de Toulouse, CNRS
E-mail: jorge.fandinno@irit.fr

CLAUDIA SCHULZ
Ubiquitous Knowledge Processing (UKP) Lab
Technische Universität Darmstadt
E-mail: schulz@ukp.informatik.tu-darmstadt.de

Abstract

Artificial Intelligence (AI) approaches to problem-solving and decision-making are becoming more and more complex, leading to a decrease in the understandability of solutions. The European Union's new General Data Protection Regulation tries to tackle this problem by stipulating a "right to explanation" for decisions made by AI systems. One of the AI paradigms that may be affected by this new regulation is Answer Set Programming (ASP). Thanks to the emergence of efficient solvers, ASP has recently been used for problem-solving in a variety of domains, including medicine, cryptography, and biology. To ensure the successful application of ASP as a problem-solving paradigm in the future, explanations of ASP solutions are crucial. In this survey, we give an overview of approaches that provide an answer to the question of *why* an answer set is a solution to a given problem, notably off-line justifications, causal graphs, argumentative explanations and why-not provenance, and highlight their similarities and differences. Moreover, we review methods explaining why a set of literals is *not* an answer set or why no solution exists at all. *Under consideration in Theory and Practice of Logic Programming (TPLP)*

KEYWORDS: answer set, explanation, justification, debugging

1 Introduction

With the increasing use of Artificial Intelligence methods in applications affecting all parts of our lives, the need for explainability of such methods is becoming ever more important. The European Union recently put forward a new General Data Protection Regulation (GDPR) (Parliament and Council of the European Union 2016), outlining how personal data may be collected, stored, and – most importantly – processed. The GDPR reflects the current suspicion of the public towards

automatic methods influencing our lives. It states[1] that anyone has the right to reject a "decision based solely on automated processing" that "significantly affects" this person. This new regulation may not come as a surprise since most Artificial Intelligence methods are 'black-boxes', that is, they produce accurate decisions, but without the means for humans to understand *why* a decision was computed. According to Goodman and Flaxman (2016), an implication of the GDPR is that, in the future, automatically computed decisions will only be acceptable if they are explainable in a human-understandable manner. The GDPR states that such an explanation needs to be made of "meaningful information about the logic involved" in the automatic decision-making and should be communicated to the person concerned in a "concise, intelligible and easily accessible form" (Goodman and Flaxman 2016).

A popular Artificial Intelligengce paradigm for decision-making and problem-solving is Answer Set Programming (ASP) (Brewka et al. 2011; Lifschitz 2008). It has proven useful in a variety of application areas, such as biology (Gebser et al. 2011), psychology (Inclezan 2015; Balduccini and Girotto 2010), medicine (Erdem and Öztok 2015), and music composition (Boenn et al. 2011). ASP is a declarative programming language used to specify a problem in terms of general inference rules and constraints, along with concrete information about the application scenario. For example, Ricca et al. (2012) present the problem of allocating employees of the large Gioia Tauro seaport into functional teams. To solve this problem, rules and constraints are formulated concerning, amongst others, team requirements and employees' shift constraints, along with factual knowledge about available employees. The reasoning engine of ASP then infers possible team configurations, or more generally, solutions to the problem. Such solutions are called *stable models* or *answer sets* (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991). Since the computation of answer sets relies on a 'guess and check' procedure, the question as to *why* an answer set is a solution to the given problem can – intuitively – only be answered with "because it fulfils the requirements of an answer set". Clearly, this explanation does not provide "meaningful information about the logic involved", as required by the GDPR.

In ASP, the need for human-understandable explanations as to *why* an answer set was computed, was recognised long before the new GDPR was put forward (Brain and De Vos 2008). Explanation approaches for ASP have thus been developed for the past twenty years, each focusing on different aspects. Some explain *why* a literal is or is not contained in an answer set, using either the dependencies between literals or the (non-) application of rules as an explanation. Other approaches provide explanations of the whole logic program, in other words, the explanation is not specific to one particular answer set. We will here refer to such explanations of logic programs that have some (potentially unexpected) answer set as *justifications*. A different type of explanation is given by *debugging* approaches for ASP, which focus on explaining errors in logic programs. Such errors become apparent either if

[1] Article 22

an unexpected answer set is computed or if the answer set computation fails, i.e. if the logic program is inconsistent. Debugging approaches thus aim to answer the question *why* an unexpected answer set is computed or *why* no answer set exists at all.

In this survey paper, we outline and compare the most prominent justification approaches for ASP, notably, off-line justifications (Pontelli et al. 2009), LABAS justifications (Schulz and Toni 2016), causal justifications (Cabalar et al. 2014; Cabalar and Fandinno 2016), and why-not provenance (Damásio et al. 2013). Further related approaches outlined here are the formal theory of justifications (Denecker and De Schreye 1993; Denecker et al. 2015) and rule-based justifications (Béatrix et al. 2016). We will see that justifications obtained using these approaches significantly differ due to their ideological underpinnings. For example, causal justifications are inspired by causal reasoning, LABAS justifications by argumentative reasoning, why-not provenance by ideas from databases, and off-line justifications by Prolog tabled computations (Roychoudhury et al. 2000). These ideological differences manifest themselves in the construction and layout of justifications, leading to variations in, for instance, the elements used in a justification (e.g. rules versus literals) and the treatment of negation (e.g. assuming versus further explaining negation-as-failure literals).

Besides explanation approaches for consistent logic programs under the answer set semantics, i.e. justification approaches, we review and discuss approaches for explaining inconsistent logic programs under the answer set semantics, i.e. debugging approaches, notably, `spock` (Brain et al. 2007b; Brain et al. 2007a; Gebser et al. 2008), `Ouroboros` (Oetsch et al. 2010), the interactive debugging approach by Shchekotykhin (2015) that is built on top of `spock`, DWASP (Alviano et al. 2013; Alviano et al. 2015), and `stepping` (Oetsch et al. 2018). We will see that these approaches form three groups, which use different strategies for detecting errors in a logic program causing the inconsistency. These strategies also lead to different types of errors being pointed out to the user. `spock`, `Ouroboros` and the interactive `spock` approach use a program transformation to report unsatisfied rules, unsupported atoms, and unfounded atoms. In contrast, DWASP makes use of the solve-under-assumption and unsatisfiable core features of the WASP solver (Alviano et al. 2013; Alviano et al. 2015), indicating faulty rules causing the inconsistency. The `stepping` approach uses the third strategy, namely a step-wise assignment of truth values to literals until a contradiction arises, which is then pointed out to the user.

The paper is structured as follows. We recall some background on logic programs and their semantics in Section 2. We then review ASP justification approaches in Section 3 and ASP debugging approaches in Section 4. In Section 5, we give a brief historical overview of justifications for logic programs and discuss related work. Finally, Section 6 concludes the paper, pointing out some issues with current approaches that provide interesting future work for the ASP community.

2 Syntax and Semantics of Logic Programs

In this section, we review the syntax and notation for disjunctive logic programs. We also review the stable and the well-founded semantics for this class of programs, which will be the basis for the works presented through the rest of the paper.

We assume the existence of some (possibly empty or infinite) set of atoms At and an operator *not*, denoting negation-as-failure (NAF)[2]. $Lit \stackrel{\text{def}}{=} At \cup \{\ not\ a \mid a \in At\ \}$ denotes the set of literals over At. Literals of the form a and $not\ a$ are respectively called *positive* and *negative*. Given a literal $l \in Lit$, by $\overline{l}$, we denote its complement, that is, $\overline{l} \stackrel{\text{def}}{=} not\ a$ iff $l = a$ and $\overline{l} \stackrel{\text{def}}{=} a$ iff $l = not\ a$. A *rule* is an expression of the form

$$h_1 \vee \ldots \vee h_k \ \leftarrow\ b_1 \wedge \ldots \wedge b_n \wedge not\ c_1 \wedge \ldots not\ c_m \tag{1}$$

where each h_i, b_i and c_i is an atom. Given some rule r of the form of (1), by $head(r) \stackrel{\text{def}}{=} \{h_1, \ldots h_k\}$, we denote the set of head atoms of the rule r. Similarly, by $body^+(r) \stackrel{\text{def}}{=} \{b_1, \ldots b_n\}$ and $body^-(r) \stackrel{\text{def}}{=} \{c_1, \ldots c_k\}$, we respectively denote the positive and negative body of r. For a set of atoms $M \subseteq At$ we denote the negative literals corresponding to atoms in M by $not\ M \stackrel{\text{def}}{=} \{\ not\ a \mid a \in M\ \}$. Furthermore, by $body(r) \stackrel{\text{def}}{=} body^+(r) \cup not\, body^-(r)$, we denote the body literals of r. A rule is called *normal* if it satisfies $head(r) = \{h_1\}$ and *positive* if $body^-(r) = \{\}$ holds. A positive normal rule is called *definite*. If $body(r) = \{\}$, the rule is called a *fact*[3] and we usually represent it omitting the symbol $\leftarrow$. We therefore sometimes use the term 'fact' to refer to the literal(s) in a fact's head. When dealing with normal rules, we sometimes denote by $head(r)$ the atom h_1 instead of the singleton set $\{h_1\}$. A rule with $head(r) = \{\}$ is called *constraint*.

A (logic) program P is a set of rules of the form of (1). A program is called *normal* (resp. *positive* or *definite*) iff all its rules are *normal* (resp. positive or definite).

Given a set of atoms $M \subseteq At$, we write $\overline{M} \stackrel{\text{def}}{=} At \backslash M$ for the set containing all atoms not belonging to M. We say that an atom a is *true* or *holds* w.r.t. $M \subseteq At$ when $a \in M$, we say that it is *false* otherwise. Similarly, we say that a negative literal $not\ a$ is *true* or *holds* w.r.t. $M \subseteq At$ when $a \notin M$ and that it is *false* otherwise. A rule $r \in P$ is *applicable* w.r.t. $M \subseteq At$ iff $body^+(r) \subseteq M$ and $body^-(r) \cap M = \{\}$, that is, when all body literals are true w.r.t. M. A rule r is *satisfied* by M iff $head(r) \cap M \neq \{\}$ whenever r is applicable. $M \subseteq At$ is *closed* under P iff every rule $r \in P$ is satisfied by M.

Answer set semantics. Intuitively, for an atom a, the literal $not\ a$ expresses that a is false by default, i.e. unless it is proven to be true. The following definition of reduct and answer set (Gelfond and Lifschitz 1988) capture this intuition.[4] The

[2] sometimes called 'default negation' in the literature

[3] This includes disjunctive facts of the form $h_1 \vee \ldots \vee h_k$.

[4] Gelfond and Lifschitz (1988) define 'stable models' rather than answer sets. Later, Gelfond and Lifschitz (1991) extended this definition to logic programs with explicit negation and with disjunction in the head, introducing the terms 'answer set'. Since then, both terms are frequently used interchangeably. We will here use the term answer set.

reduct of a program P w.r.t. a set of atoms $M \subseteq At$, in symbols P^M, is the result of applying the following two steps:

1. removing all rules r such that $a \in M$ for some $a \in body^-(r)$,
2. removing all negative literals from the remaining rules.

The result is a positive program P^M. Then, a set of atoms $M \subseteq At$ is an *answer set* of a program P iff it is a $\subseteq$-minimal closed set under P^M. A logic program is called *consistent* if it has at least one answer set, and *inconsistent* otherwise. Intuitively, a set of atoms is an answer set if all atoms in it are justified by the rules of the program under the assumption that all negative literals are evaluated w.r.t. this answer set.

Example 1
Let P_1 be the logic program consisting of the following rules:

$$p \leftarrow q \wedge not\, r \qquad s \leftarrow t \qquad q$$
$$r \leftarrow not\, p \qquad t \leftarrow s$$

and let M_1 be the set of atoms $\{p, q\}$. Then, the reduct of P_1 w.r.t. M_1 is the program $P_1^{M_1}$:

$$p \leftarrow q \qquad s \leftarrow t \qquad q$$
$$t \leftarrow s$$

whose $\subseteq$-minimal closed set is precisely $\{p, q\}$. Hence, M_1 is an answer set of P_1. Intuitively, q is in the answer since it is a fact in the program, while p is in the answer set due to the rule $p \leftarrow q \wedge not\, r$ and the fact that q is true and r is assumed to be false w.r.t. M_1. Note that s and t mutually depend on each other, so there is no reason to believe either of them, and consequently neither is contained in the answer set. It is easy to check that program P_1 has a second answer set $\{q, r\}$. □

Well-founded model semantics. We introduce a definition of the well-founded model semantics for normal logic programs in terms of the least fixpoint of a Γ_P operator (Van Gelder 1989) which is, though equivalent, slightly different from the original definition by Van Gelder et al. (1988) and Van Gelder et al. (1991). Given a normal logic program P, let Γ_P be the function mapping each set of atoms M to the $\subseteq$-minimal closed set of the program P^M and let Γ_P^2 be the operator mapping each set M to $\Gamma_P(\Gamma_P(M))$. Then, Γ_P and Γ_P^2 are antimonotonic and monotonic, respectively, and, consequently, the latter has a least and greatest fixpoint, which we respectively denote by $\mathtt{lfp}(\Gamma_P^2)$ and $\mathtt{gfp}(\Gamma_P^2)$. We also respectively denote by $WF_P^+ \stackrel{\text{def}}{=} \mathtt{lfp}(\Gamma_P^2)$ and $WF_P^- \stackrel{\text{def}}{=} (At \backslash \mathtt{gfp}(\Gamma_P^2))$ the set of true and false atoms in the well-founded model of P. The well-founded model of P can then be defined as the set of literals: $WF_P \stackrel{\text{def}}{=} WF_P^+ \cup not\, WF_P^-$. The well-founded model is said to be *complete* iff $WF_P^+ \cup WF_P^- = At$. We say that an atom a is *true* w.r.t. the well-founded model if $a \in WF_P$, *false* if $not\, a \in WF_P$, and *undefined* otherwise.

It is easy to see that, by definition, the answer sets of any normal program P coincide with the fixpoints of Γ_P and, thus, every stable model is also a fixpoint

of Γ_P^2. Hence, every stable model M satisfies: $WF_P^+ \subseteq M$ and $WF_P^- \cap M = \{\}$. In other words, the well-founded model semantics is more sceptical than the answer set semantics in the sense that all atoms that are true (resp. false) in the well-founded model are also true (resp. false) in all answer sets.

Example 2 (*Ex. 1 continued*)
Continuing with our running example, it is easy to see that $P_1^{\{\}}$ is:

$$p \leftarrow q \qquad s \leftarrow t \qquad q$$
$$r \leftarrow \qquad t \leftarrow s$$

and that its $\subseteq$-minimal model is $\{p, q, r\}$. Hence, we have that $\Gamma_{P_1}(\{\}) = \{p, q, r\}$. In a similar way, it can be checked that $\Gamma^2_{P_1}(\{\}) = \Gamma^4_{P_1}(\{\}) = \{q\}$ is the least fixpoint of the $\Gamma^2_{P_1}$ operator. Hence, we have that $WF_{P_1} = \{q, not\, s, not\, t\}$. As expected, q is true in all answer sets of P_1 while s and t are false in all of them. Furthermore, p and r are true in one answer set but not in the other and are left undefined in the well-founded model. Note that it is possible that an atom is true in all answer sets, but undefined in the well-founded model. For instance, $M_1 = \{p, q\}$ is the unique answer set of $P_1 \cup \{u \leftarrow r \wedge not\, u\}$, but p is still undefined in its well-founded model. □

Explicit negation. In addition to negation-as-failure, we use the operator $\neg$ to denote *explicit negation*. For an atom a, $\neg a$ denotes the contrary of a. By $\neg S \stackrel{\text{def}}{=} \{ \neg a \mid a \in S \}$ we denote the explicitly negated atoms of a set $S \subseteq At$ and, by $At_{ext} \stackrel{\text{def}}{=} At \cup \neg At$ we denote the set of *extended atoms* consisting of atoms and explicitly negated atoms. By $Lit_{ext} \stackrel{\text{def}}{=} At_{ext} \cup \{ not\, a \mid a \in At_{ext} \}$, we denote the set of *extended literals* over At. As for logic programs without explicit negation, extended literals $\neg a$ and $not\, \neg a$ are respectively called *positive* and negative.

An *extended rule* is an expression of the form (1) where each h_i, b_i and c_i is an extended atom. An *extended* (logic) program is a set of extended rules. The notions of head, body, etc. directly carry over from rules without explicit negation. Note that we say that a program is positive when it does not contain negation-as-failure, even if it contains explicit negation.

The definition of answer sets and well-founded model[5] are easily transferred to extended logic programs by replacing $M \subseteq At$ with $M \subseteq At_{ext}$. If an answer set (resp. the well-founded model) contains both an atom a and its contrary $\neg a$, the answer set is called *contradictory* (Gelfond and Lifschitz 1991; Gelfond 2008). In some works (Gelfond and Lifschitz 1991), a contradictory answer set is only an answer set if the program has no other answer set and is, by definition, At_{ext}.

[5] Even though this simply transfer is sufficient for the purpose of this paper, for the well-founded model semantics the property ensuring that the explicit negation of a formula implies its default negation is lost. For a detailed study and solution of this problem we refer to the work of Pereira and Alferes (1992).

Example 3
Let P_2 be the logic program consisting of the following rules:

$$\begin{array}{lll} p \leftarrow q \wedge not\, r & & \neg p \\ r \leftarrow not\, p & & q \end{array}$$

and let M_2 be the set of extended atoms $\{\neg p, q, r\}$. Then, the reduct of P_2 w.r.t. M_2 is the program $P_2^{M_2}$:

$$\begin{array}{lll} & & \neg p \\ r \leftarrow & & q \end{array}$$

whose $\subseteq$-minimal closed set is precisely $\{\neg p, q, r\}$. Hence, M_2 is an answer set of P_2. Note that there is a second answer set $\{p, \neg p, q\}$ which is contradictory. According to the definition of Gelfond and Lifschitz (1991), M_2 is thus the only answer set. □

3 Justifications of Consistent Logic Programs

In this section, we review the most prominent approaches for explaining *consistent* logic programs under the answer set semantics. All approaches reviewed here, except for the formal theory of justifications (Section 3.5.2), aim to provide concise structures called justifications that provide a somewhat minimal explanation as to why a literal in question belongs to an answer set.

We start by introducing off-line (Section 3.1; Pontelli et al. 2009; Pontelli and Son 2006), LABAS (Section 3.2; Schulz and Toni 2016; Schulz and Toni 2013) and causal justifications (Section 3.3; Cabalar et al. 2014; Cabalar and Fandinno 2016). In these three approaches, justifications are represented as different kinds of dependency graphs between literals and/or rules. Next, we review why-not provenance justifications (Section 3.4; Damásio et al. 2013), which represent justifications as propositional formulas instead of graph structures. It is interesting to note that why-not provenance and causal justifications share a multivalued semantic definition based on a lattice. Finally, we sketch the main idea of rule-based justifications (Béatrix et al. 2016) and the formal theory of justifications (Denecker and De Schreye 1993; Denecker et al. 2015) in Section 3.5.

3.1 Off-line Justifications

Off-line justifications (Pontelli et al. 2009; Pontelli and Son 2006) are graph structures that describe the reason for the truth value of an atom with respect to a given answer set. In particular, each off-line justification describes the derivation of the truth value (that is, true or false) of an atom using the rules in the program. Each vertex of such a graph represents an atom and each edge the fact that the two vertices that it joins are related by some rule in the program, with the edge pointing from the head of the rule to some atom in its body. Atoms that are true with respect to a given answer set are labelled '+', whereas atoms that are false with respect to it are labelled '−' (see condition 3 in Definition 1 below). Similarly,

edges labelled '+' represent positive dependencies while those labelled '−' represent negative ones. This is reflected in conditions 5a (a true atom is supported by a true atom through a positive dependency and by a false atom through a negative dependency) and condition 8 of Definition 1 below (a false atom is supported by a false atom through a positive dependency and by a true atom through a negative dependency).

Before we technically describe off-line justifications, we need the following notation: for any set of atoms $S \subseteq At$, the sets of *annotated atoms* are defined as $S^p \stackrel{\text{def}}{=} \{ a^+ \mid a \in S \}$ and $S^n \stackrel{\text{def}}{=} \{ a^- \mid a \in S \}$. Furthermore, given an annotated atom $a^\pm$ (that is, $a^\pm = a^+$ or $a^\pm = a^-$), by $atom(a^\pm) = a$ we denote the atom associated with $a^\pm$. Given a set of annotated atoms S, by $atoms(S) \stackrel{\text{def}}{=} \{ atom(a^\pm) \mid a^\pm \in S \}$, we denote the set of atoms associated with the annotated atoms in S.

Definition 1 (*Off-line Explanation Graph*)
Let P be a normal logic program, let $M, U \subseteq At$ be two sets of atoms, and let $a^\pm \in (At^p \cup At^n)$ be an annotated atom[6]. An *off-line explanation graph* of $a^\pm$ w.r.t. P, M and U is a labelled, directed graph $G = \langle V, E \rangle$ with a set of vertices $V \subseteq (At^p \cup At^n \cup \{assume, \top, \bot\})$ and a set of edges $E \subseteq (V \times V \times \{+, -\})$, which satisfies the following conditions:

1. $a^\pm \in V$ and every $b \in V$ is reachable from $a^\pm$,
2. the only sinks in the graph are: $assume$, $\top$ and $\bot$,
3. $atoms(V \cap At^p) \subseteq M$ and $atoms(V \cap At^n) \subseteq (\overline{M} \cup U)$,
4. The set of edges E satisfies the following two conditions:
   (a) $\{ c \mid (b^+, c^-, +) \in E \} \cup \{ c \mid (b^+, c^+, -) \in E \} = \{\}$ and
   (b) $\{ c \mid (b^-, c^+, +) \in E \} \cup \{ c \mid (b^-, c^+, -) \in E \} = \{\}$,
5. every $b^+ \in V$ satisfies that there is a rule $r \in P$ with $head(r) = b$ s.t.
   (a) $body(r) = \{ c \mid (b^+, c^+, +) \in E \} \cup \{ not\ c \mid (b^+, c^-, -) \in E \}$, or
   (b) both $body(r) = \{\}$ and $(b^+, \top, +)$ is the unique edge in E with source b^+,
6. every $b^- \in V$ with $b \in U$ satisfies that $(b^-, assume, -)$ is the only edge with source b^-,
7. every $b^- \in V$ with $b \notin U$ and no rule $r \in P$ with $head(r) = b$ satisfies that $(b^-, \bot, +)$ is the only edge with source b^-,
8. every $b^- \in V$ with $b \notin U$ and some rule $r \in P$ with $head(r) = b$ satisfies that $S = \{ c \mid (b^-, c^-, +) \in E \} \cup \{ not\ c \mid (b^-, c^+, -) \in E \}$ is a minimal set of literals such that every rule $r' \in P$ with $head(r') = b$ satisfies $body(r') \cap S \neq \{\}$. □

Intuitively, M represents some answer set and U represents a set of assumptions with respect to M. These assumptions derive from the inherent 'guessing' process involved in the definition and algorithmic construction of answer sets. In this sense,

[6] Off-line justifications were defined without using explicit negation, so we here stick to logic programs without explicit negation. However, it is easy to see that they can be applied to extended logic program by replacing At by At_{ext}.

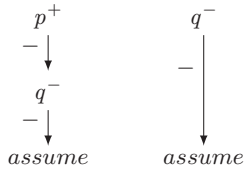


Fig. 1: Off-line justifications of p^+ and q^- w.r.t. $M_3 = \{p\}$ in Example 4. The assumption is $\{q\}$.

the truth value of assumed atoms has no further justification while non-assumed atoms must be justified by the rules of the program. This is reflected in condition 6 of Definition 1. Note also that this condition ensures that true elements are not treated as assumptions, which follows from the intuition that any true atom in an answer set must be justified. Condition 4 ensures that a labelled atom is not supported by the wrong type of relation.

The following example illustrates how assumptions are used to justify atoms that are false w.r.t. an answer set in question.

Example 4
Let P_3 be the program containing the following two rules:

$$p \leftarrow not\ q \qquad\qquad q \leftarrow not\ p$$

Program P_3 has two answer sets, namely $M_3 = \{p\}$ and $M_4 = \{q\}$. Figure 1 depicts the off-line explanation graphs justifying the truth of p (annotated atom p^+) and the falsity of q (annotated atom q^-) with respect to the program P_3, the answer set M_3 and the set of assumptions $\{q\}$. Note that the falsity of q is assumed in both justifications. □

To ensure that the set of assumptions is meaningful with respect to the answer set being explained, it needs to be restricted. In particular, it will be restricted to a subset of atoms that are false w.r.t. the answer set and undefined w.r.t. the well-founded model. As mentioned above, assumptions are restricted to be false atoms to follow the intuition that any true atom in an answer set must be justified. Restricting the set of assumptions further to only those that are undefined w.r.t. the well-founded model ensures that false atoms that are also false w.r.t. to the well-founded model are justified by the constructive process of the well-founded model rather than being assumed. The following notation is needed to achieve this restriction:

Definition 2
Given a normal program P, by $NANT(P) \stackrel{\text{def}}{=} \{\ b \in At \mid \exists r \in P \text{ s.t. } b \in body^-(r)\ \}$, we denote the set of atoms that occur negated in P. □

Definition 3 (*Negative Reduct*)
Given a normal program P, by $NR(P, U) \stackrel{\text{def}}{=} \{\ r \in P \mid head(r) \notin U\ \}$, we denote the *negative reduct* of P w.r.t. some set of atoms $U \subseteq At$. □

Fig. 2: Off-line explanation graphs of p^+ and q^+ w.r.t. {}, which are not off-line justifications.

Definition 4 (*Assumptions*)
Let P be a normal program and M an answer set of P. Let us denote by

$$\mathcal{TA}_P(M) \stackrel{\text{def}}{=} \{\ a \in NANT(P) \mid a \in \overline{M} \text{ and } a \notin (WF_P^+ \cup WF_P^-)\ \}$$

the *tentative assumptions* of P w.r.t. M. Then, an *assumption* w.r.t M is a set of atoms $U \subseteq \mathcal{TA}_P(M)$ such that $WF^+_{NR(P,U)} = M$. The set of all possible assumptions of P w.r.t. M is denoted by $Assumptions(P, M)$. □

An interesting observation to make is that $\mathcal{TA}_P(M)$ is always an element of the set $Assumptions(P, M)$ and, therefore, the latter is never empty. Intuitively, an assumption is a set of atoms that are false w.r.t. the considered answer set and that, when 'forced to be false' in the program, produces a complete well-founded model that coincides with this answer set. The negative reduct (see Definition 3), removing all rules whose head belongs to the assumption, can be interpreted as 'forcing atoms to be false' since it results in all atoms in the assumption being false in the well-founded model. Then, since the computation of the well-founded model is deterministic, no guessing is necessary. Justifications relative to the well-founded model can thus be used for the explanation w.r.t. an answer set by adding edges that point out which atoms in the assumption were used to obtain the answer set. This is formalised as follows:

Definition 5 (*Off-line Justification*)
Let P be a normal program, M an answer set of P, $U \in Assumptions(P, M)$ an assumption w.r.t M and P, and $a^{\pm} \in (At^p \cup At^n)$ an annotated atom. Then, an *off-line justification* of $a^{\pm}$ w.r.t. P, M and U is an off-line explanation graph w.r.t. P, M and U (Definition 1), which satisfies that for all $b \in At$, (b^+, b^+) does not belong to the transitive closure of $\{\ (c, e) \mid (c, e, +) \in E\ \}$. □

The last condition of Definition 5 ensures that true atoms are not justified through positive cycles, thus ensuring that justifications of true atoms are rooted in some rule without positive body, that is, either facts or rules whose body is a conjunction of negative literals. We may also interpret the latter type of rules as a kind of 'facts by default'.

Example 5
Let P_4 be the program containing the following two rules:

$$p \leftarrow q \qquad\qquad q \leftarrow p$$

It has a unique answer set that coincides with its complete well-founded model:

$M_5 = WF^+_{P_4} = \{\}$. Figure 2 depicts two cyclic off-line explanation graphs of p^+ and q^+, which, as can be expected, are not off-line justifications since p and q are false w.r.t. M_5 and since positive cycles are allowed in explanation graphs, but not in off-line justifications. Figure 3 depicts two cyclic off-line justifications

Fig. 3: Off-line justifications of p^- and q^- w.r.t. $M_5 = \{\}$ and assumption $\{\}$.

explaining that p and q are false w.r.t. M_5 because they positively depend on each other. Note that cycles between negatively annotated atoms are allowed in off-line justifications. □

The following example illustrates how off-line justifications are built for a more complex program that has a complete well-founded model, in which case the unique assumption is the empty set. Example 4 is continued later, in Example 8, where it is shown that the off-line explanation graphs in Figure 1 are in fact off-line justifications. Note that the program discussed in Example 4 has a non-complete well-founded model and, thus, some atoms will need to be assumed to build the off-line justifications.

Example 6
Let P_5 be the program consisting of the following rules:

$$p \leftarrow q \qquad q \leftarrow r \wedge s \qquad r \leftarrow not\ t \qquad s$$

This program has a unique answer set $M_6 = \{p, q, r, s\}$, which coincides with its complete well-founded model. As a result, we have an empty set of tentative assumptions $\mathcal{TA}_{P_5}(M_6) = \{\}$ and the empty set as the only valid assumption, that is, $Assumptions(P_5, M_6) = \{\{\}\}$. Figure 4a depicts the unique off-line justification of p^+ w.r.t. program P_5 and answer set M_6. Intuitively, the edge $(t^-, \bot, +)$ points out that t is false because there is no rule in P_5 with t in the head. Then, as a consequence of the closed world assumption, t is considered to be false. Similarly, edge $(s^+, \top, +)$ indicates that s is true because it is a fact. Edge $(p^+, q^+, +)$ (resp. $(r^+, t^-, -)$) indicates that p (resp. r) is true because it positively (resp. negatively) depends on q (resp. t) which is true (resp. false). Finally, edges $(q^+, r^+, +)$ and $(q^+, s^+, +)$ together point out that q is true because it positively depends on both r and s, which are true. It is also worth noting that the subgraphs of this off-line justification rooted in q^+, r^+ and s^+ constitute the off-line justifications of q, r and s being true w.r.t. P_5 and M_6. Similarly, the subgraph rooted in t^- represents the off-line justification for the atom t being false. □

In the above example, there is a unique off-line justification for each true or false atom. The following examples show that several justifications may exist for a given atom w.r.t. a given answer set.

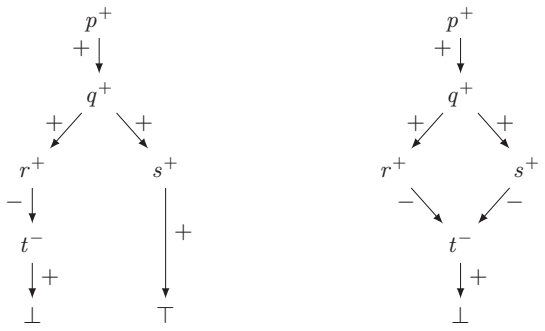


Fig. 4: Off-line justifications of p^+ w.r.t. P_6, M_6, and assumption $\{\}$. Figure 4a is also an off-line justification w.r.t. P_5, M_6, and $\{\}$ (see Examples 6 and 7).

Example 7 (*Ex. 6 continued*)
Let P_6 be the result of adding rule $s \leftarrow \mathit{not}\, t$ to program P_5. It is easy to check that M_6 is also the unique answer set of P_6 (and $\{\}$ the unique assumption), but now there is a second way to justify the truth of s, namely in terms of the falsity of t. As a result, there are two off-line justification of p^+, respectively depicted in Figures 4a and 4b. □

Example 8 (*Ex. 4 continued*)
In contrast to P_5 and P_6, program P_3 does not have a complete well-founded model. In fact, its well-founded model leaves all atoms undefined. Thus, $q \in \mathit{NANT}(P_3)$ implies that $\mathcal{TA}_{P_3}(M_3) = \{q\}$ which, in turn, implies $\mathit{Assumptions}(P_3, M_3) = \big\{\ \{q\}\ \big\}$. Note that $\{\}$ is not a valid assumption because the well-founded model of $\mathit{NR}(P, \{\})$ is not complete. Then, since there is no cycle in Figure 1, it follows that these two off-line explanation graphs are also off-line justifications. Note that edge $(q^-, \mathit{assume}, -)$ captures that atom q is false because of the inherent guessing involved in the definition of answer sets. □

In Example 5, we already illustrated the difference between off-line explanation graphs and off-line justifications. The following example shows this difference in a program without cycles.

Example 9
Let P_7 be the program containing the single rule $p \leftarrow \mathit{not}\, q$. Program P_7 has a complete well-founded model, which consequently coincides with the unique answer set: $M_7 = \mathit{WF}^+_{P_7} = \{p\}$. As in Example 4, it easy to see that graphs depicted in Figure 1 (also depicted in Figure 5a to ease the comparison) are off-line explanation graphs of p^+ and q^- with respect to the program P_7, the answer set M_7 and the assumption $\{q\}$. Moreover, since the well-founded model is complete, there are no tentative assumptions, that is, $\mathcal{TA}_{P_7}(M_7) = \{\}$ and $\mathit{Assumptions}(P_7, M_7) = \big\{\ \{\}\ \big\}$.

(a) (b)

Fig. 5: Off-line justifications of p^+ and q^- w.r.t. $M_3 = M_7 = \{p\}$ in Examples 4 and 9, respectively. Note that the assumption is respectively $\{q\}$ and $\{\}$ in subfigures 5a and in 5b.

Therefore, the off-line explanation graphs in Figure 5a are not valid off-line justifications. Figure 5b depicts the off-line justifications of p^+ and q^- with respect to program P_7, the answer set M_7 and the assumption $\{\}$. Note that, since there is no rule with q in the head, the falsity of q can be justified without assumptions. □

By adding the rule $q \leftarrow not\, p$ to program P_7 (Example 9) we create an even-length negative dependency cycle, that is, not only p is dependent on q being false, but also q is dependent on p being false (note that this is exactly program P_3 from Example 4). This has the effect of replacing the edge $(q^-, \bot, -)$ by $(q^-, assume, -)$ in the off-line justifications of p^+ and q^- (see Figure 5). In other words, rather than q being false by default, it is now *assumed* to be false w.r.t. the answer set $\{p\}$. As shown by the following example this change from default to assuming is not always the case when creating an even-length negative dependency cycle: for some programs, this may have the effect of introducing additional justifications.

Example 10
Let P_8 be the program

$$p \leftarrow not\, q \qquad\qquad r \leftarrow not\, p \qquad\qquad s \leftarrow not\, r$$

As in Example 9, this program has a complete well-founded model and, thus, a unique answer set that coincides with the well-founded model: $M_8 = WF^+_{P_8} = \{p, s\}$. Then, we have that $\mathcal{TA}_{P_8}(M_8) = \{\}$ and $Assumptions(P_8, M_8) = \big\{\ \{\}\ \big\}$. Figure 6a depicts the unique off-line justification of s^+ with respect to program P_8, the answer set M_8 and assumption $\{\}$. Let now $P_9 = P_8 \cup \{q \leftarrow not\, p\}$. As in Example 4, this program also has two answer sets, namely $M_9 = \{p, s\}$ and $M_{10} = \{q, r\}$, and an empty well-founded model $WF^+_{P_9} = WF^-_{P_9} = \{\}$. Then, it follows that $\mathcal{TA}_{P_9}(M_9) = \{q, r\}$ and $Assumptions(P_9, M_9) = \big\{\ \{q\}, \{q, r\}\ \big\}$. Figures 6b and 6c depict the two off-line justifications of s^+ with respect to program P_9, M_9 and assumptions $\{q\}$ and $\{q, r\}$, respectively. As opposed to what happens in Example 9, adding the rule $q \leftarrow not\, p$, and thus creating an even-length negative dependency cycle, not only has the effect of replacing the edge $(q^-, \bot, -)$ by $(q^-, assume, -)$, but it also produces a second off-line justification in which r^- is assumed (Figure 6c). This difference disappears if we only take into account off-line justifications with respect to $\subseteq$-minimal assumptions, in which case only Figures 6b would be a justification. □

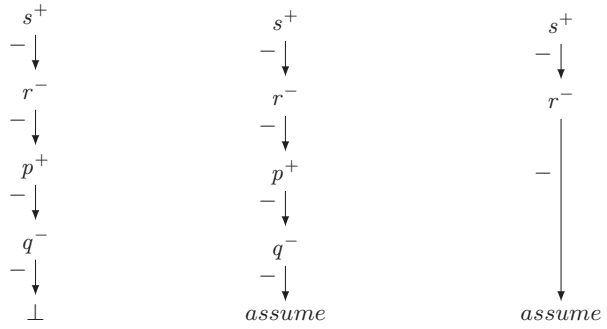


Fig. 6: Off-line justifications of s^+ w.r.t. $M_8 = M_9 = \{p, s\}$ and the assumption $\{\}$ (Subfigure 6a), $\{q\}$ (Subfigure 6b), and $\{q, r\}$ (Subfigure 6c) in Example 10.

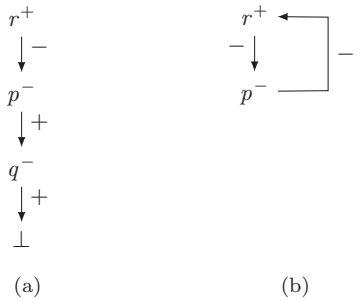


Fig. 7: Off-line justifications of r^+ w.r.t. $M_{11} = \{r\}$ and assumption $\{\}$ in Example 11.

As mentioned above, the last condition of Definition 5 ensures that true atoms are not justified through positive cycles (those in which all edges are labelled '+'). Still, there exist off-line justifications in which true atoms are justified by (non-positive) cycles, as illustrated by the following example.

Example 11
Let P_{10} be the program containing the following two rules:

$$p \leftarrow q \wedge not\, r \qquad\qquad r \leftarrow not\, p$$

This program has a complete well-founded model, which coincides with its unique answer set $WF^+_{P_{10}} = M_{11} = \{r\}$. Then, $Assumptions(P_{10}, M_{11}) = \{\ \{\}\ \}$. Figure 7 depicts the two off-line justifications of r^+ with respect to program P_{10}, the answer set M_{11} and the assumption $\{\}$. □

Though at first sight, cyclic justifications (like the one in Figure 7) may seem to contradict the intuition that the justifications of true atoms must be rooted in a

rule without positive body (facts or rules whose body is a conjunction of negative literals), we note that the existence of an acyclic off-line justification (Figure 7a) in Example 11 is not accidental. In fact, for every true atom, there always exists at least one acyclic justification (Pontelli and Son 2006, Proposition 2).

3.2 LABAS Justifications

LABAS justifications (Schulz and Toni 2016; Schulz and Toni 2013) explain the truth value of an extended literal with respect to a given answer set of an extended normal logic program.[7] They have been implemented in an online platform called `LABAS Justifier`.[8] In contrast to off-line justifications, where every rule application step used to derive a literal is included in a justification, LABAS justifications abstract away from intermediate rule applications in the derivation, only pointing out the literal in question and the facts and negative literals occurring in rules used in the derivation. In addition, the truth of negative literals $not\, l$ is not taken for granted or assumed, but is further explained in terms of the truth value of the respective positive literal l.

LABAS justifications have an *argumentative* flavour as they are constructed from trees of conflicting *arguments*.[9]

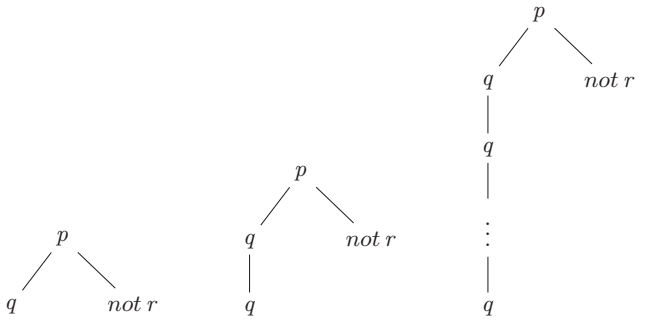
Definition 6 (*Argument*)
Given an extended logic program P, an *argument* for $l \in Lit_{ext}$ is a finite tree, where every node holds a literal in Lit_{ext}, such that

- the root node holds l;
- for every node N
  - — if N is a leaf then N holds either a negative literal or a fact;
  - — if N is not a leaf and N holds the positive literal h, then there is a rule $h \leftarrow b_1 \wedge \ldots \wedge b_n \wedge not\, c_1 \wedge \ldots not\, c_m$ in P and N has $n + m$ children, holding $b_1, \ldots, b_n, not\, c_1, \ldots not\, c_m$ respectively;
- AP is the set of all negative literals held by leaves;
- FP is the set of all facts held by leaves.

An argument is denoted $A : (AP, FP) \vdash l$, where A is a unique name, AP is the set of *assumption premises*, FP the set of *fact premises*, and l the *conclusion*. □

Intuitively, an argument is a derivation where each rule is used and where only negative literals and facts are recorded. Note however, that arguments are not necessarily minimal derivations and that they allow the repeated application of a rule.

[7] For simplicity, we use the term 'literal' instead of 'extended literal' throughout this section.
[8] `http://labas-justification.herokuapp.com/`
[9] Schulz and Toni (2016) define arguments and attack trees with respect to the translation of a logic program into an *Assumption-Based Argumentation (ABA) framework* (Dung et al. 2009). For simplicity, we here reformulate these definitions with respect to a logic program. Due to the semantic correspondence between logic programs and their translation into ABA frameworks (Schulz and Toni 2016; Schulz and Toni 2015), these definitions are equivalent to the original ones.



(a) Argument A_1 (b) Argument A_2 (c) Argument A_n

Fig. 8: Different arguments with conclusion p.

Example 12
Let P_{11} be the following logic program:

$$p \leftarrow q \wedge not\ r \qquad\qquad q \leftarrow q \qquad\qquad q$$

There are *infinitely* many arguments for p (and q) since the second rule can be used infinitely many times before using the fact q. Figure 8a illustrates the argument A_1 where the second rule is not used at all, Figure 8b illustrates the argument A_2 where the second rule is used once, and Figure 8c illustrates arguments where the second rule is applied various times (indicated by the dots). Note that all arguments with conclusion p differ in their name and their tree representation, but they are all denoted $(\{not\ r\}, q) \vdash p$ in the shorthand notation. □

An argument for a literal only exists if all literals in the rules used in the derivation have an argument themselves. That is, for a logic program with only one rule $p \leftarrow q$, there is no argument for either p or q (q is neither a negative literal nor a fact, so it cannot be the leaf of an argument tree).

If the conclusion of an argument is a positive literal l then it *attacks* every argument that has $not\ l$ in its assumption premises. In other words, a derivation for l provides a reason against any derivation using $not\ l$.

Definition 7 (Attack)
An argument $(AP_1, FP_1) \vdash l_1$ *attacks* an argument $(AP_2, FP_2) \vdash l_2$ iff l_1 is a positive literal and $not\ l_1 \in AP_2$. □

Note that attacks do not arise due to the existence of an atom a and its contrary $\neg a$ in two arguments.

Example 13 (Ex. 4 continued, page 9)
Four arguments can be constructed from P_3:

$$A_1 : (\{not\ p\}, \{\}) \vdash not\ p \qquad\qquad A_3 : (\{not\ p\}, \{\}) \vdash q$$
$$A_2 : (\{not\ q\}, \{\}) \vdash not\ q \qquad\qquad A_4 : (\{not\ q\}, \{\}) \vdash p$$

A_3 attacks A_2 and A_4 since its conclusion q is the complement of the assumption premise *not* q in the two attacked arguments. Similarly, A_4 attacks A_1 and A_3. □

3.2.1 Attack Trees

LABAS justifications are constructed from trees of attacking arguments.

Definition 8 (*Attack Tree*)
Given an extended program P, an *attack tree* of an argument $A : (AP, FP) \vdash l$ w.r.t. an answer set M of P, denoted $attTree_M(A)$, is a (possibly infinite) tree such that:

1. Every node in $attTree_M(A)$ holds an argument, labelled '+' or '−'.
2. The root node is A^+ if $\forall\, not\, l' \in AP : l' \notin M$, or A^- otherwise.
3. For every node B^+ and for every argument C attacking argument B, there exists a child node C^- of B^+.
4. Every node B^- has exactly one child node C^+ for some argument $C : (AP_C, FP_C) \vdash l_C$ attacking argument B and satisfying that $\forall\, not\, l' \in AP_C$, $l' \notin M$.
5. There are no other nodes in $attTree_M(A)$ except those given in 1-4. □

The intuition for labelling arguments in an attack tree is as follows: If an argument A is based on some negative literal *not* l (i.e. it has *not* l as an assumption premise) such that $l \in M$, then some rule used to construct A is not applicable w.r.t. M (namely the rule in which *not* l occurs), so argument A does not warrant that its conclusion is in M. Therefore, argument A is labelled '−'. Otherwise, all rules used to construct A are applicable, so the conclusion of argument A is in M. Thus, argument A is labelled '+'.

Example 14 (*Ex. 13 continued*)
The unique attack trees of A_3 and A_4 w.r.t. $M_3 = \{p\}$ are displayed in Figure 9a and 9b, respectively. When inverting all '+' and '−' labels in the trees, the attack trees w.r.t. $M_4 = \{q\}$ are obtained. □

An attack tree is thus made of layers of arguments for literals that are alternately true and false w.r.t. the answer set M. Note the difference in Definition 8 between arguments labelled '+', which have all attackers as child nodes, and arguments labelled '−', which have only one attacker as a child node. This is in line with the definition of answer sets. To prove that a literal l is in M, *all* negative literals *not* l' used in its derivation (i.e. in the argument B in condition 3) need to be true, so for all l' there must not be a derivation that concludes that l' is true. Thus, all such derivations for l' (i.e. all arguments C attacking B in condition 3) are explained in an attack tree. In contrast, to prove that a derivation of a literal l (argument B in condition 4) does not lead to l being true w.r.t. M, it is sufficient that one negative literal *not* l' used in this derivation is false, i.e. l' is in M, so there exists some derivation for l' (argument C in condition 4) that warrants that l' is true w.r.t. M.

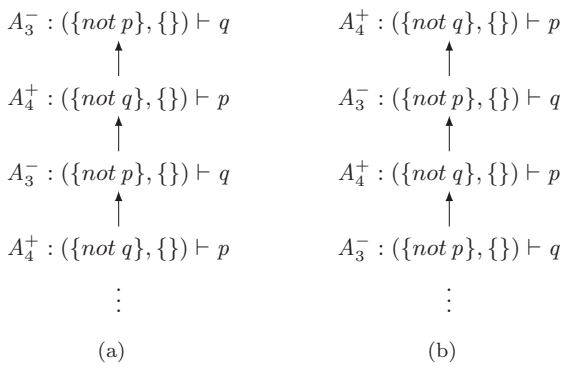


Fig. 9: Attack trees of arguments A_3 and A_4 w.r.t. M_3 of P_3.

Example 15
Let P_{12} be the following logic program:

$$p \leftarrow not\, q \wedge not\, r \qquad q \leftarrow not\, s \qquad s$$
$$r \leftarrow s \wedge not\, p \qquad r \leftarrow not\, s$$

Program P_{12} has two answer sets, namely $M_{12} = \{s, p\}$ and $M_{13} = \{s, r\}$. The argument $A_1 : (\{not\, q, not\, r\}, \{\}) \vdash p$ has one attack tree w.r.t. M_{12} and one w.r.t. M_{13}, depicted in Figures 10a and 10b, respectively. Note that in the attack tree of A_1 w.r.t. M_{13}, A_2 and A_4 cannot be chosen as the child nodes of A_1, even though they attack A_1, since they both have *not* s as an assumption premise, where s is contained in the answer set M_{13} (they thus violate condition 4 in Definition 8). These arguments thus do not provide explanations as to why r is true w.r.t. M_{13} and consequently cannot be used to explain why p is false. □

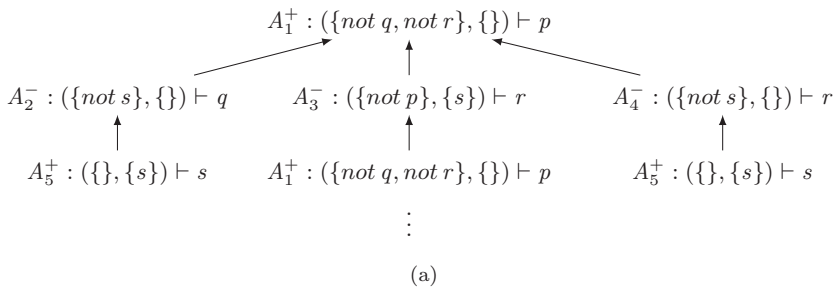
Attack trees are not only used to construct LABAS justifications, as explained in the following, but in fact constitute justifications of literals in their own right.

Definition 9 (*Attack Tree Justification*)
Let M be an answer set of an extended program P, $l \in Lit_{ext}$, and A an argument with conclusion l.

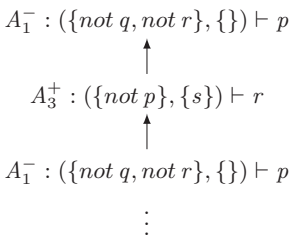
- If l is true w.r.t. M, then an $attTree_M(A)$ is a justification of l if the root node is A^+.
- If l is false w.r.t. M, then an $attTree_M(A)$ is a justification of l if the root node is A^-. □

In fact, in the second case *any* attack tree for an argument with conclusion l will have its root node labelled '$-$' (Schulz and Toni 2016, from Theorem 3 and Lemma 5).

Attack trees justify literals in terms of dependencies between arguments. Next, we explain how dependencies between literals are extracted from attack trees to construct a justification in terms of literals.



(a)



(b)

Fig. 10: Attack trees of argument A_1 w.r.t. M_{12} and M_{13}.

3.2.2 Constructing LABAS Justifications

Labelled ABA-Based Answer Set Justifications ("ABA" stands for "Assumption-Based Argumentation"), short LABAS justifications, are constructed from attack trees by extracting the relations between literals in arguments. That is, literals occurring as assumption or fact premises in an argument of the attack tree are *supporting* the conclusion literal, whereas the conclusion l of an attacking argument *attacks* the negative literal *not* l occurring as an assumption premise of the attacked argument.

As a first step of the LABAS justification construction, an attack tree is transformed into a *labelled justification*. A labelled justification is a set of labelled relations between literals, which can thus be represented as a graph. Each literal in a relation is labelled as '+', meaning that it is true w.r.t. the answer set in question, or '−', meaning that it is false w.r.t. the answer set in question. Support and attack relations are labelled the same as the respective source literals of the relation. The label '+' represents that the source label is able to effectively attack or support the target literal, whereas '−' represents an ineffective relation. In addition, a literal is labelled with *fact* or *asm* if it is a fact or assumption premise, or else with its argument's name.

Definition 10 (*Labelled Justification*)
Let M be an answer set of an extended program P, A an argument and $\Upsilon = attTree_M(A)$ an attack tree of A w.r.t. M. For any node $B^{+/-}$ in Υ, $children(B^{+/-})$

denotes the set of child nodes of $B^{+/-}$ and $conc(B^{+/-})$ the conclusion of argument B. The *labelled justification* of Υ, denoted $just(\Upsilon)$, is obtained as follows:

$$
\begin{aligned}
&just(\Upsilon) \stackrel{\text{def}}{=} \\
&\quad \bigcup_{B^+:(AP,FP)\vdash l \ in \ \Upsilon} \\
&\qquad \{supp\_rel^+(not\ p^+_{asm}, l^+_B) \quad \mid not\ p \in AP\backslash\{l\}\} \cup \\
&\qquad \{supp\_rel^+(f^+_{fact}, l^+_B) \quad \mid f \in FP\backslash\{l\}\} \cup \\
&\qquad \{att\_rel^-(k^-_C, not\ k^+_{asm}) \quad \mid C^- \in children(B^+), conc(C^-) = k\} \cup \\
&\quad \bigcup_{B^-:(AP,FP)\vdash l \ in \ \Upsilon} \\
&\qquad \{supp\_rel^-(not\ p^-_{asm}, l^-_B) \mid not\ p \in AP\backslash\{l\}, children(B^-) = \{C^+\}, \\
&\qquad\qquad\qquad\qquad conc(C^+) = p\} \cup \\
&\qquad \{att\_rel^+(f^+_{fact}, not\ f^-_{asm}) \mid children(B^-) = \{C^+ : (\{\}, \{f\}) \vdash f\}\} \cup \\
&\qquad \{att\_rel^+(k^+_B, not\ k^-_{asm}) \mid children(B^-) = \{C^+ : (AP_C, FP_C) \vdash k\}, \\
&\qquad\qquad\qquad\qquad AP_C \neq \{\} \ or \ FP_C \neq \{k\}\}
\end{aligned}
$$

□

Note that a labelled justification does not extract *all* relations from an attack tree but only those deemed relevant for justifying the conclusion of argument A. For example, for an argument B^- in the attack tree, only one negative literal is extracted as supporting the conclusion, namely the one that is attacked by the child node C^+ of B^-, since this negative literal provides the reason that the conclusion of B is not in the answer set.

Infinite attack trees, as for example shown in Figures 9a and 9b, may be represented by *finite* LABAS justifications as re-occurring arguments in an attack tree are only processed once (note that justifications are sets).

Example 16 (*Ex. 14 continued*)
Since the two attack trees $attTree_{M_3}(A_3)$ and $attTree_{M_3}(A_4)$ (Figures 9a and 9b) comprise the same nodes, their labelled justifications are the same, namely:

$$
\begin{aligned}
&\{supp\_rel^-(not\ p^-_{asm}, q^-_{A_3}), att\_rel^+(p^+_{A_4}, not\ p^-_{asm}), \\
&\ supp\_rel^+(not\ q^+_{asm}, p^+_{A_4}), att\_rel^-(q^-_{A_3}, not\ q^+_{asm})\}
\end{aligned}
$$

□

As illustrated by Example 16, it is not obvious from a labelled justification, which literal is being justified. A LABAS justification thus adds the literal being justified to labelled justifications. It furthermore defines a justification in terms of *one* labelled justification if a literal contained in the answer set is justified and in terms of *all* labelled justifications if a literal not contained in the answer set is justified. This is based on the idea that if a literal can be successfully derived in one way, it is in the answer set, but that it is not in the answer set only if all ways of deriving the literal are unsuccessful.

Definition 11 (*LABAS Justification*)
Let M be an answer set of an extended program P and $l \in Lit_{ext}$.

1. Let l be true w.r.t. M, let $A : (AP, FP) \vdash l$ be an argument, and $attTree_M(A)$ an attack tree with root node A^+. Let $lab(l) \stackrel{\text{def}}{=} l^+_{asm}$ if l is a negative literal,

$lab(l) \stackrel{\text{def}}{=} l^+_{fact}$ if $FP = \{l\}$ and $AP = \{\}$, and $lab(l) = l^+_A$ else. A *(positive) LABAS justification* of l with respect to M is:
$justLABAS^+_M(l) \stackrel{\text{def}}{=} \{lab(l)\} \cup just(attTree_M(A))$.

2. Let l be false w.r.t. M, let $A_1, \ldots, A_n$ be all arguments with conclusion l, and $\Upsilon_{11}, \ldots, \Upsilon_{1m_1}, \ldots, \Upsilon_{n1}, \ldots, \Upsilon_{nm_n}$ all attack trees of $A_1, \ldots, A_n$ with root node labelled '$-$'.
   (a) If $n = 0$, then the *(negative) LABAS justification* of l with respect to M is:
   $justLABAS^-_M(l) \stackrel{\text{def}}{=} \{\}$
   (b) If $n > 0$, then let $lab(l_1) \stackrel{\text{def}}{=} l^-_{asm}, \ldots, lab(l_n) \stackrel{\text{def}}{=} l^-_{asm}$ if l is a negative literal and $lab(l_1) \stackrel{\text{def}}{=} l^-_{A_1}, \ldots, lab(l_n) \stackrel{\text{def}}{=} l^-_{A_n}$ else. Then the *(negative) LABAS justification* of l with respect to M is:
   $justLABAS^-_M(l) \stackrel{\text{def}}{=} \{\{lab(l_1)\} \cup just(\Upsilon_{11}), \ldots, \{lab(l_n)\} \cup just(\Upsilon_{nm_n})\}$. □

Note that there may be various LABAS justifications of a literal that is true w.r.t. the answer set M, but only one LABAS justification of a literal that is false w.r.t. M.

Example 17 (*Ex. 16 continued*)
Since there exists only one argument with conclusion $q \notin M_3$, namely A_3, and since this argument has a unique attack tree $attTree_{M_3}(A_3)$, only the labelled justification from Example 16 has to be taken into account for the LABAS justification of q w.r.t. M_3. That is,

$$justLABAS^-_{M_3}(q) = \{\{q^-_{A_3}, supp\_rel^-(not\, p^-_{asm}, q^-_{A_3}), att\_rel^+(p^+_{A_4}, not\, p^-_{asm}), supp\_rel^+(not\, q^+_{asm}, p^+_{A_4}), att\_rel^-(q^-_{A_3}, not\, q^+_{asm})\}\}$$

Similarly, the only LABAS justification of p w.r.t. M_3 is

$$justLABAS^+_{M_3}(p) = \{p^-_{A_4}, supp\_rel^-(not\, p^-_{asm}, q^-_{A_3}), att\_rel^+(p^+_{A_4}, not\, p^-_{asm}), supp\_rel^+(not\, q^+_{asm}, p^+_{A_4}), att\_rel^-(q^-_{A_3}, not\, q^+_{asm})\}$$

Note that the first is a set of sets, whereas the second is a simple set. □

LABAS justifications can be represented as directed graphs, where the justified literal is depicted as the top node of the graph, and all literals occurring in a relation as the other nodes. Support and attack relations form two different arcs: here, dashed arcs represent support, whereas solid arcs represent attack. Both types of arcs are labelled according to the label in the LABAS justification.

Example 18 (*Ex. 17 continued*)
The graphical representations of the LABAS justifications in Example 17 are respectively illustrated in Figures 11a and 11b. Unsurprisingly, they have the same nodes and arcs. However, the respective orientation of the graph indicates the literal being justified. Note the difference between the LABAS justification graphs and the off-line justifications in Figure 1. In particular, the LABAS justification

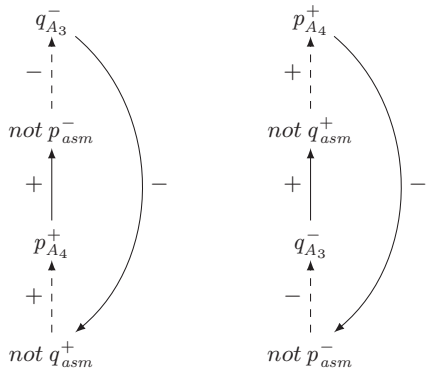


Fig. 11: LABAS justifications of q and p w.r.t. M_3: dashed arcs represent support, whereas solid arcs represent attack.

graphs explain the truth values of non-fact positive literals in terms of negative literals needed to derive the positive literal. Furthermore, the truth values of negative literals, which do not occur in off-line justifications at all, are explained in terms of their complement's truth value. Also note that q being false w.r.t. M_3 is explained as a truth value being assumed in the off-line justifications, whereas its truth value is further explained in terms of the ineffective support by $not\, p$ in the LABAS justifications. □

Example 19 (*Ex. 15 continued*)
Figures 12a and 12b illustrate the LABAS justifications of p w.r.t. M_{12} and M_{13} of P_{12} (see Example 15). The first demonstrates the importance of labelling literals by their arguments for distinction. If these labels did not exist, $r^-_{A_3}$ and $r^-_{A_4}$ would collapse into one node r^-. The resulting graph would give the impression that there is only one derivation for r, which uses both $not\, p$ and $not\, s$. In contrast, the distinction achieved by labelling literals with their argument names (Figure 12a), expresses that there are two derivations for r, one using $not\, p$ and one using $not\, s$. Note that off-line justifications use a non-labelling strategy, leading to the previously explained collapse of the two nodes holding atom r, as shown in Figures 13a and 13b. Figure 12a, and in particular node $r^-_{A_3}$, furthermore shows that for nodes labelled '−' in an attack tree, fact premises are not included in the LABAS justification (A_3 has a fact premise s). In contrast, Figure 12b, and in particular node $r^+_{A_3}$, shows that for nodes labelled '+' in an attack tree, all assumption and fact premises are included in a LABAS justification. Furthermore, for nodes labelled '−' only the assumption premise that is attacked by the child node is included (only assumption premise $not\, r$ of p is included and assumption premise $not\, q$ is neglected). □

Comparing the LABAS justification in Figure 12a and the off-line justification in Figure 13b, we observe various similarities: Deleting the nodes holding negative

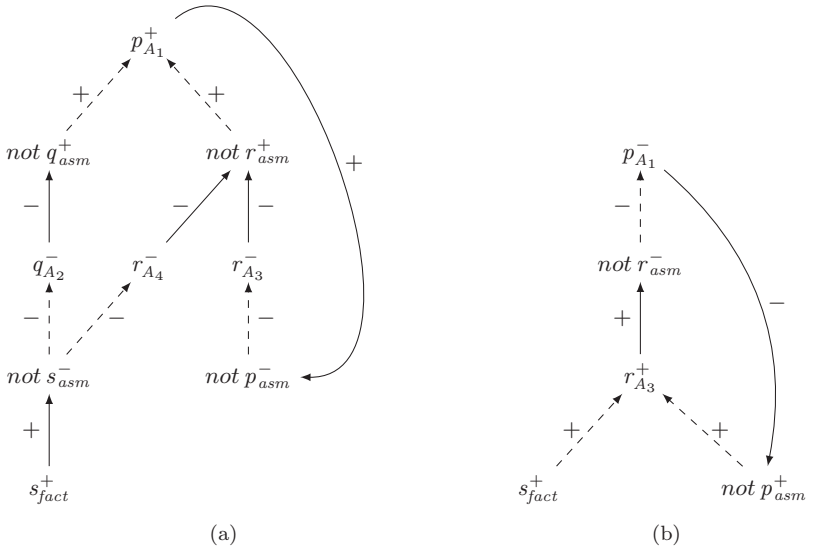


Fig. 12: LABAS justifications of p w.r.t. M_{12} and M_{13}.

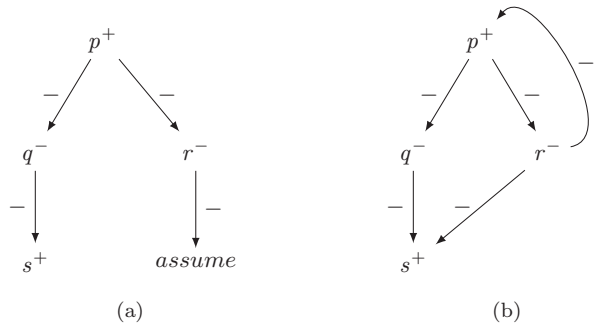


Fig. 13: Off-line justifications of p w.r.t. M_{12}.

literals in the LABAS justification and collapsing the two nodes of atom r results in the same nodes as in the off-line justification. Note that this is because all derivations of atoms are "one-step" derivations, i.e. there is no chaining of rules involved. If the derivation of some atom involved the chaining of various rules, the off-line justification would include more nodes than the LABAS justification, even if nodes holding negative literals were deleted (see for example Figures 17a and 17b). Furthermore, 'rerouting' the attack edges in the LABAS justification from the attacked negative literal to the atom supported by this negative literal (e.g. 'rerouting' the attacking edge from p^+ to $not\, p$ instead to atom r, which is

supported by *not p*) and then reverting them results, in this example, in the same edges as in the off-line justification. Note however that the labelling of edges is different in LABAS and off-line justifications.

The following examples point out some further differences between LABAS and off-line justifications. In particular, LABAS justifications do not explicitly contain information about all rules applied in a derivation and there is no LABAS justification for literals that have no argument, i.e. literals that cannot be successfully derived.

Example 20 (*Ex. 7 continued, page 12*)
Figures 14a and 14b show the LABAS justifications of p w.r.t. M_6 of P_6. The

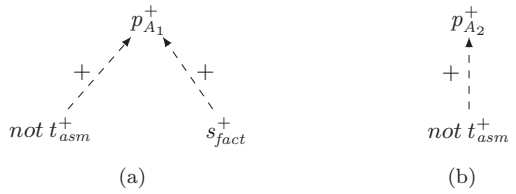


Fig. 14: The two LABAS justifications of p w.r.t. M_6 of P_6.

difference between the two derivations of p is not as explicit as in the off-line justifications illustrated in Figures 4a and 4b (page 12). It is merely indicated by the different argument labels of p. □

Example 21 (*Ex. 11 continued, page 14*)
There are two off-line justifications of r w.r.t. P_{10} and M_{11} (see Figures 7a and 7b on page 14). In contrast, there is only *one* LABAS justification of r, shown in Figure 15. The reason is that there is no argument with conclusion p, since no rule with head p exists. Thus, *not p* is not further explained as there is no way to prove p. □

As previously pointed out, infinite attack trees may be represented by finite LABAS justifications. However, this is only the case if the infinity is due to the repetition of the same arguments. Instead, if the infinity is due to the existence of infinitely many arguments with the same conclusion, a LABAS justification may be infinite too.

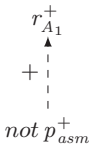
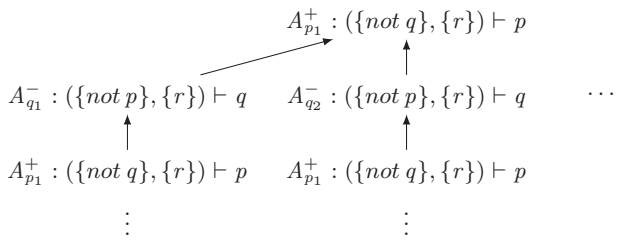
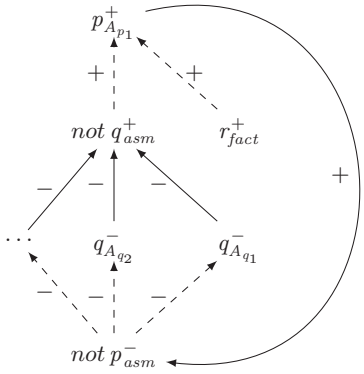


Fig. 15: The unique LABAS justification of r w.r.t. M_{11} of P_{10}.



(a)



(b)

Fig. 16: One of the infinite attack trees and LABAS justifications of p w.r.t. M_{14} of P_{13}.

Example 22
Let P_{13} be the following program with answer sets $M_{14} = \{p, r\}$ and $M_{15} = \{q, r\}$:

$$p \leftarrow not\ q \wedge r \qquad q \leftarrow not\ p \wedge r \qquad r \leftarrow r \qquad r$$

Note first that there are infinitely many arguments with conclusion r of the form $A_{r_i} : (\{\}, \{r\}) \vdash r$, each applying the third rule a different number of times. For the same reason, there are infinitely many arguments with conclusion p, of the form $A_{p_j} : (\{not\ q\}, \{r\}) \vdash p$, and with conclusion q, of the form $A_{q_k} : (\{not\ p\}, \{r\}) \vdash q$. Since there are infinitely many arguments with conclusion p (resp. q), there are also infinitely many attack trees explaining p (resp. q) with respect to either of the two answer sets. Similarly to the attack trees illustrated in Figures 9a and 9b, all attack trees for p and q are infinite in depth. In addition, they are infinite in breadth since any of the A_{p_j} attacks every A_{q_k} and vice versa. This means that whenever an argument for p (resp. q) is labelled '+' in an attack tree, all infinitely many arguments with conclusion q (resp. p) are child nodes labelled '−'. Figure 16a illustrates an attack tree of one of the arguments with conclusion p w.r.t. M_{14}. Note that in this particular attack tree, the argument $A^+_{p_1} : (\{not\ q\}, \{r\}) \vdash p$

is re-used to attack all the arguments with conclusion q attacking the root node. By exchanging any occurrence of $A^+_{p_1} : (\{not\ q\}, \{r\}) \vdash p$ by another argument with conclusion p, e.g. $A^+_{p_2} : (\{not\ q\}, \{r\}) \vdash p$, a different (infinite) attack tree explaining p is obtained. We observe that any of these attack trees yields an infinite LABAS justification. For example, the attack tree from Figure 16a results in a LABAS justification with infinitely many relations of the form $att\_rel^-(q^-_{A_{q_k}}, p^+_{A_{p_j}})$ relations. Assuming that the only argument with conclusion p used in the attack tree in Figure 16a is $A^+_{p_1} : (\{not\ q\}, \{r\}) \vdash p$, we obtain the infinite LABAS justification in Figure 16b. □

This behaviour of infinity is dealt with in the `LABAS Justifier` by disallowing the repeated application of a rule when constructing an argument (Schulz 2017). In Example 22, the `LABAS Justifier` thus only constructs two different arguments for p and q.

3.3 Causal Graph Justifications

In contrast to the two previously discussed approaches (off-line and LABAS justifications), whose main purpose is to explain why a literal is (not) contained in an answer set, the approach outlined in this section – called *causal graph justifications* (Cabalar et al. 2014; Cabalar and Fandinno 2016) – is a reasoning formalism in its own right, which can additionally be used to explain why a literal is contained in an answer set: the main goal of the causal justification approach is to formalise and reason with causal knowledge, so that sentences like "whoever causes the death of somebody else will be imprisoned" can be represented in an *elaboration tolerant*[10] manner (McCarthy 1998). An online tool providing causal justifications and allowing this reasoning with causal knowledge (Fandinno 2016a) is available at `http://kr.irlab.org/cgraphs-solver/nmsolver`.

The semantics used for causal justifications is a multi-valued extension of the answer set semantics, where each (true) literal in a model is associated with a set of causal values expressing causal reasons for its inclusion in the model. Each of these causal values represents a set of *causal justifications*, each of which, in turn, can be depicted as a *causal graph*. Regarding the causal literature, a *causal graph* can be seen as an extension of Lewis's notion of *causal chain*: "let c, d, e, ... be a finite sequence of actual particular events such that d causally depends on c, e on d, and so on throughout. Then, this sequence is a causal chain." (Lewis 1973; see also Hall 2004 and Hall 2007). The following example illustrates the connection between causal chains and justifications in ASP.

Example 23
Consider a scenario in which Suzy pulls the trigger of her gun, causing the gunpowder to explode. This causes the bullet to leave the gun at a high speed, impacting

[10] We recall that a representation is elaboration tolerant if modifications of it can easily be taken into account.

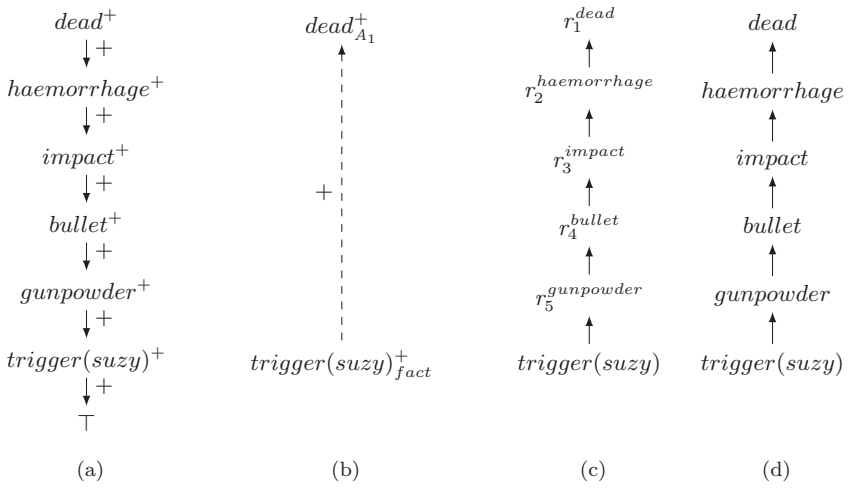


Fig. 17: Off-line justification, LABAS justification, causal justification and causal chain of *dead* in Example 23.

on Billy's chest, provoking a massive haemorrhage and, consequently, Billy's death. We can model this scenario as the following positive logic program P_{14}:

$$dead \leftarrow haemorrhage \quad (2)$$
$$haemorrhage \leftarrow impact \quad (3)$$
$$impact \leftarrow bullet \quad (4)$$
$$bullet \leftarrow gunpowder \quad (5)$$
$$gunpowder \leftarrow trigger(suzy) \quad (6)$$
$$trigger(suzy) \quad (7)$$

Then, $trigger(suzy) \cdot gunpowder \cdot bullet \cdot impact \cdot haemorrhage \cdot dead$ is a causal chain connecting $trigger(suzy)$ with *dead*. □

This example suggests an intuitive correspondence between causal chains and the idea of justification. In particular, the causal chain that connects the fact $trigger(suzy)$ with *dead* can be written as the graph in Figure 17d. It is easy to see the correspondence between this graph and the off-line justification of *dead*, depicted in Figure 17a. For comparison, Figures 17b and 17c depict the LABAS justification and the causal graph (which will be defined later) of *dead*. Recall that LABAS justifications focus on facts and negative literals, precisely abstracting from the causal chain, which will be the focus of causal justifications. In contrast, the causal graph expresses the same information as the causal chain. This is due to the fact that no atom depends on more than one other atom. More generally, causal chains coincide with the paths in causal graphs.

In addition to the idealogical differences between causal justifications, which treat logic programs as causal knowledge, and off-line and LABAS justifications, which

treat logic programs as declarative problem descriptions, causal justifications allow for causal reasoning, as they are based on a causal extension of the answer set semantics. More precisely, causal justifications are defined in terms of the *causal value* that each *causal answer set* associates to atoms (causal answer sets assign causal values instead of truth values to each atom). These causal values form a completely distributive (complete) lattice that serves as the basis for a multi-valued extension of the answer set semantics.

Let us introduce causal terms as a suitable syntax to write causal values.

Definition 12 (*Causal Term*)
Given a set of atoms At and a set of labels Lb, a *(causal) term* t is recursively defined as one of the following expressions

$$t \quad ::= \quad l \;\Big|\; \prod S \;\Big|\; \sum S \;\Big|\; t_1 \cdot t_2$$

where $l \in (At_{ext} \cup Lb)$ is an extended atom or a label, t_1, t_2 are in turn terms, and S is a (possibly empty and possibly infinite) set of terms. □

When $S = \{t_1, \ldots, t_n\}$ is a finite set, we write $t_1 * \ldots * t_n$ and $t_1 + \ldots + t_n$ instead of $\prod S$ and $\sum S$, respectively. The empty sum and empty product are respectively represented as 0 and 1. We assume that *application* '$\cdot$' has higher priority than product '$*$' and, in turn, product '$*$' has higher priority than addition '$+$'. Intuitively, product '$*$' represents conjunction or joint causation, sum '$+$' represents alternative causes, and *application* '$\cdot$' is a non-commutative product that builds causal chains by capturing the successive application of rules.

Associativity

$$t \cdot (u \cdot w) \;=\; (t \cdot u) \cdot w$$

Absorption

$$\begin{array}{rcl} t & = & t + u \cdot t \cdot w \\ u \cdot t \cdot w & = & t * u \cdot t \cdot w \end{array}$$

Identity

$$\begin{array}{rcl} t & = & 1 \cdot t \\ t & = & t \cdot 1 \end{array}$$

Annihilator

$$\begin{array}{rcl} 0 & = & t \cdot 0 \\ 0 & = & 0 \cdot t \end{array}$$

Indempotence

$$l \cdot l = l$$

Addition distributivity

$$\begin{array}{rcl} t \cdot (u + w) & = & (t \cdot u) + (t \cdot w) \\ (t + u) \cdot w & = & (t \cdot w) + (u \cdot w) \end{array}$$

Product distributivity

$$\begin{array}{rcl} c \cdot d \cdot e & = & (c \cdot d) * (d \cdot e) \text{ with } d \neq 1 \\ c \cdot (d * e) & = & (c \cdot d) * (c \cdot e) \\ (c * d) \cdot e & = & (c \cdot e) * (d \cdot e) \end{array}$$

Fig. 18: Properties of the operators: t, u, w are terms, l is a label or an extended atom and c, d, e are terms without addition '$+$'. Addition and product distributivity are also satisfied over infinite sums and products. A kind of absorption over infinite sums and products can also be derived from the finite absorption above and infinite distributivity.

Definition 13 (*Causal Value*)
(Causal) values are the equivalence classes of terms under the axioms for a completely distributive (complete) lattice with meet '$\prod$' and join '$\sum$' plus the axioms in Figure 18. The set of values is denoted by $\mathbf{V}_{Lb}$. Furthermore, by $\mathbf{C}_{Lb}$ we denote the subset of causal values with some representative term without addition '$\sum$'. □

As an example, the causal value $[a] = \{a,\ a * a,\ a + a,\ a{\cdot}a,\ a * (a + b),\ \dots\}$ is the (possibly infinite) set of causal terms that are equivalent to a under the axioms for a completely distributive lattice with meet '$\prod$' and join '$\sum$' plus the axioms in Figure 18. Note that there are no causal terms equivalent to 0 or 1 besides themselves, that is, $[0] = \{0\}$ and $[1] = \{1\}$. By abuse of notation, we will use any causal term belonging to a causal value to represent the value, that is, we write a instead of $[a]$, 0 instead of $[0]$, and so on.

Note that all three operations '$*$', '$+$' and '$\cdot$' are associative. Product '$*$' and addition '$+$' are also commutative, and they satisfy the usual absorption and distributive laws with respect to infinite sums and products of a completely distributive lattice. As usual, the lattice order relation is defined as:

$$t \leq u \qquad \text{iff} \qquad t * u = t \qquad \text{iff} \qquad t + u = u$$

An immediate consequence of this definition is that the $\leq$-relation has the product as greatest lower bound, the addition as least upper bound, 1 as top element and 0 as bottom element. The term 1 represents a value that holds by default, without an explicit cause, and will be assigned to the empty body. The term 0 represents the absence of cause or the empty set of causes, and will be assigned to falsity.

Furthermore, applying distributivity (and absorption) of products and applications over addition, every term can be represented in a *(minimal) disjunctive normal form* in which addition is not in the scope of any other operation and every pair of addends are pairwise $\leq$-incomparable. As we will see in Example 31, this normal form emphasises the intuition that addition '$+$' separates alternative causes. Moreover, applying product distributivity, this normal form can be further rewritten into a *graph normal form* in which the application operator '$\cdot$' is only applied to *pairs* of labels or extended atoms, thus representing the edges of a graph: $v{\cdot}v'$ with $v, v' \in (At_{ext} \cup Lb)$. For instance, applying priority rules, the causal terms $a*(((b{\cdot}c){\cdot}e)+d)$ and $((a*((b{\cdot}c){\cdot}e))+(a*d)$ can be rewritten as $a*(b{\cdot}c{\cdot}e+d)$ and $a * b{\cdot}c{\cdot}e + a * d$, respectively. Furthermore, it is easy to see that these two terms represent the same causal value since the former can be rewritten as the latter by applying distributivity of products over sums. The latter is in disjunctive normal form and can be further rewritten in graph normal form as $a * b{\cdot}c * c{\cdot}e + a * d$ by applying distributivity of application over products.

Given any causal term without sums $c \in \mathbf{C}_{Lb}$ in graph normal form, we can associate a graph $G_c = \langle V, E\rangle$ where V is the set of labels and extended atoms occurring in c and E contains an edge (v, v') for every subterm of the form $v{\cdot}v'$. By $graph(c)$ we denote the transitive and reflexive reduction[11] of G_c. Given this relation between application '$\cdot$' and edges in such graphs it follows that application '$\cdot$' must be non-commutative. For any causal term in normal form t, by $graphs(t)$ we denote the set containing a graph $graph(c)$ for each addend c in t.

[11] Recall that the transitive and reflexive reduction of a graph G is a graph G' whose transitive and reflexive closure is G. A causal graph (see Definition 16), in which every cycle is a reflexive edge, has a unique transitive and reflexive reduction.

Example 24 (*Ex. 23 continued*)
The causal chain of Example 23 is in disjunctive normal form (since it does not contain products nor sums), but not in graph normal form. Using product distributivity, this causal chain can be rewritten in graph normal form as $(trigger(suzy) \cdot gunpowder) * (gunpowder \cdot bullet) * (bullet \cdot impact) * (impact \cdot haemorrhage) * (haemorrhage \cdot dead)$. In this form, every subterm of the form $(v \cdot v')$ corresponds to an edge in Figure 17d. □

So far, we have introduced causal values, which will be the semantic building blocks of causal justifications and the associated causal graphs. In the following, we define how these causal values are assigned to each atom to form causal answer sets and how causal justifications and graphs are obtained.

3.3.1 Causal Semantics for Programs without Negation-as-Failure

Semantics for logic programs usually assign truth values to atoms. In contrast, for the causal semantics of logic programs, causal interpretations assign causal values to atoms. Based on this, causal models and causal answer sets are defined. Causal justifications are then extracted using the causal value of atoms in a causal answer set corresponding to a standard answer set.

A *(causal) interpretation* is a mapping $I : At_{ext} \longrightarrow \mathbf{V}_{Lb}$ assigning a value to each extended atom and satisfying $I(a) = 0$ or $I(\neg a) = 0$ for every atom $a \in At$. By $Atoms(I) \stackrel{\text{def}}{=} \{\ a \in At_{ext} \mid I(a) \neq 0\ \}$ we denote the set of extended atoms in an interpretation I. For any pair of interpretations I and J, we write $I \leq J$ to represent the straightforward causal ordering, that is, $I(a) \leq J(a)$ for every atom $a \in At_{ext}$ and we write $I \sqsubseteq J$ when either $I \leq J$ or $Atoms(I) \subset Atoms(J)$. That is, $I \sqsubseteq J$ is a weaker partial order, since apart from the cases in which $I \leq J$ holds, it also holds when true atoms in I are a strict subset of true atoms in J. As usual, we write $I < J$ (resp. $I \sqsubset J$) iff $I \leq J$ (resp $I \sqsubseteq J$) and $I \neq J$. Note that $Atoms(I) \subset Atoms(J)$ implies $I \neq J$ and so $I \sqsubset J$. We say that an interpretation I is $\leq$-minimal (resp. $\sqsubseteq$-minimal) satisfying some property when there is no $J < I$ (resp. $J \sqsubset I$) satisfying that property. Note that there is a $\leq$-bottom and $\sqsubseteq$-bottom interpretation $\mathbf{0}$ (resp. a $\leq$-top and $\sqsubseteq$-top interpretation $\mathbf{1}$) that stands for the interpretation mapping every extended atom a to the causal value 0 (resp. 1). It is easy to see that $\sqsubseteq$-minimal models are also $\leq$-minimal models, though the converse is not necessarily true, as will be illustrated by Example 30 (see page 33). For every rule r in the program, we assign a label denoted by $label(r)$. We assume that $label(h) = h$ for every definite fact h and that $label(r) \neq label(r')$ for every pair of distinct rules r and r'. We also assume that Lb contains all rule labels.

Definition 14 (*Causal Model*)
An interpretation I *satisfies* a positive rule r of the form (1) (with $m = 0$) iff

$$\big(\, I(b_1) * \ldots * I(b_n) \,\big) \cdot r_i \cdot h_j \quad \leq \quad I(h_j) \tag{8}$$

for some atom $h_j \in head(r)$ and where $r_i = label(r)$ is the label associated with rule r. We say that an interpretation I is a *(causal) model* of a positive extended program P, in symbols $I \models P$, iff I satisfies all rules in P. □

Example 25 (*Ex. 23 continued*)
Let us assume that rules of P_{14} are respectively labelled as r_1, r_2, r_3, r_4, r_5 and $trigger(suzy)$. Then, it is easy to check that the model I of P_{14} must satisfy

$$\begin{array}{lcll} I(trigger(suzy)) & \geq & trigger(suzy) \cdot trigger(suzy) & = \quad trigger(suzy) \\ I(gunpowder) & \geq & trigger(suzy) \cdot r_5 \cdot gunpowder & \end{array}$$

□

Observation 1
If r is a definite fact h, that is, it has the form $(h \leftarrow)$, then $label(r) = h$ and, thus, $I \models r$ iff $I(A) \geq h \cdot h = h$ (by idempotence of application on labels). □

Based on the definitions of causal values and models, the causal extension of the answer set semantics is defined as follows.

Definition 15 (*Causal Answer Set without Negation-as-Failure*)
Let P be a positive extended program. A model I of P is a *causal answer set* iff it is $\sqsubseteq$-minimal among the models of P. □

Example 26 (*Ex. 25 continued*)
Continuing with our running example, note that there is only one rule with atoms $trigger(suzy)$ and $gunpowder$ in the head. Then, any $\sqsubseteq$-minimal model I_1 of P_{14} must satisfy equality instead of $\geq$, that is,

$$\begin{array}{lcll} I_1(trigger(suzy)) & = & trigger(suzy) \cdot trigger(suzy) & = \quad trigger(suzy) \\ I_1(gunpowder) & = & trigger(suzy) \cdot r_5 \cdot gunpowder & \end{array}$$

Note that any $\sqsubseteq$-minimal model must also be a $\leq$-minimal model and, thus, $I_1(A)$ must be equal to the least upper bound of the terms corresponding to all rules with the atom A in the head. Since here we only have one rule for each atom, this least upper bound coincides with the value corresponding to that rule. □

Definition 16 (*Causal Justification and Causal Graph*)
Given a logic program P and an answer set M of P, a term without sums c is a *causal justification* of some atom a w.r.t. P and M if there is some causal answer set I of P such that $Atoms(I) = M$ and c is an addend in the minimal disjunctive normal form of $I(a)$. For any causal justification of a w.r.t. P and M, $graph(c)$ is a *causal graph (justification)*. □

Notation 1
In causal justifications, we will write r_i^a instead of $r_i \cdot a$ when $r_i \in Lb$ is a rule label and $a \in At_{ext}$ is an extended atom occurring in the head of the rule labelled r_i. Similarly, in causal graphs we write a single vertex r_i^a instead of two vertices r_i and a and an edge connecting them. □

Example 27 (*Ex. 26 continued*)
Assuming the above notation, we may rewrite the causal value associated with *gunpowder*, which is also its unique causal justifications, as $I_1(gunpowder) = trigger(suzy) \cdot r_5^{gunpowder}$. Similarly, it is also easy to check that

$$I_1(dead) \;=\; trigger(suzy) \cdot r_5^{gunpowder} \cdot r_4^{bullet} \cdot r_3^{impact} \cdot r_2^{haemorrhage} \cdot r_1^{dead}$$

Figure 17c depicts the causal graph associated with the causal justification $I_1(dead)$. □

Next, we give an example of causal justifications for non-normal programs taken from (Cabalar and Fandinno 2016):

Example 28
Assume that Harvey throws a coin and only shoots when he gets tails. This scenario can be modelled as the following logic program P_{15}:

$$r_1 : \quad dead \leftarrow shoot \tag{9}$$
$$r_2 : \quad shoot \leftarrow tails \tag{10}$$
$$r_3 : \quad head \vee tails \leftarrow harvey \tag{11}$$
$$harvey \tag{12}$$

where r_1, r_2 and r_3 represent the labels associated with the corresponding rules. Then, this logic program has two (standard) answer sets: $M_{16} = \{harvey, head\}$ and $M_{17} = \{harvey, tails, shoot, dead\}$. Similarly, this program also has two causal answer sets satisfying

$$\begin{aligned} I_{16}(harvey) &= harvey \\ I_{16}(head) &= harvey \cdot r_3^{head} \\ I_{16}(tails) &= 0 \\ I_{16}(shoot) &= 0 \\ I_{16}(dead) &= 0 \end{aligned} \qquad \begin{aligned} I_{17}(harvey) &= harvey \\ I_{17}(head) &= 0 \\ I_{17}(tails) &= harvey \cdot r_3^{tails} \\ I_{17}(shoot) &= harvey \cdot r_3^{tails} \cdot r_2^{shoot} \\ I_{17}(dead) &= harvey \cdot r_3^{tails} \cdot r_2^{shoot} \cdot r_1^{dead} \end{aligned}$$

Here, the $I_{17}(dead)$ represents the causal justification of *dead* w.r.t. M_{17} while $I_{16}(dead) = 0$ states that there is no causal justifications for *dead* w.r.t. M_{16}. □

Example 28 illustrates that a causal answer set assigns the value 0 (that is, the absence of a justification) to an atom iff the atom is false in its corresponding standard answer set.

It is also worth to note that, for normal logic programs, there is a one-to-one correspondence between the standard answer sets of a program and their causal answer sets. For programs with disjunctive rules, there also exists a one-to-one correspondence, but in this case it relates each standard answer set with a class of causal answer sets that represent the same truth assignments, but different justifications (see Example 29 below). Furthermore, in the case of disjunctive rules, the superindex of a disjunctive rule's label in the causal answer set indicates the disjunct that has been effectively applied. For instance, in Example 28, term r_3^{tails} points out that the disjunct *tails* in r_3 has been effectively applied. In the case of

normal rules, the superindex is somehow superfluous, as it is fully determined by the rule, and could easily be omitted as in (Cabalar and Fandinno 2016). Nevertheless, we decide to keep them to ease the comparison with the other justification approaches, whose vertices are literals.

Example 29
Consider a program P_{16} consisting of the following rules

$$r_1: \quad a \vee b \leftarrow \qquad\qquad r_2: \quad a \leftarrow b \qquad\qquad r_3: \quad b \leftarrow a$$

which has a unique (standard) answer set $M_{18} = \{a, b\}$, but two causal ones that satisfy:

$$\begin{array}{lll} I_{18}(a) & = & r_1^a \\ I_{18}(b) & = & r_1^a \cdot r_3^b \end{array} \qquad\qquad \begin{array}{lll} I'_{18}(a) & = & r_1^b \cdot r_2^a \\ I'_{18}(b) & = & r_1^b \end{array}$$

As we can see, the true atoms in both models, $Atoms(I_{18}) = Atoms(I'_{18}) = \{a, b\}$, coincide with the unique (standard) answer set M_{18}, but their *justifications differ*. In I_{18}, atom a is a (non-deterministic) effect of the disjunction r_1, while b is derived from a through r_3. Analogously, I'_{18} makes b true because of r_1 and then obtains a from b through r_2. It is interesting to point out that I''_{18} with

$$\begin{array}{lll} I''_{18}(a) & = & r_1^a + r_1^b \cdot r_2^a \\ I''_{18}(b) & = & r_1^b + r_1^a \cdot r_3^b \end{array}$$

is also a model of the program, but not a $\sqsubseteq$-minimal one because we have $I_{18} \sqsubset I''_{18}$. Intuitively, I''_{18} would represent a scenario in which both a and b are justified by rule r_1, which does not fit the intuitive understanding that rule r_1 can only justify one of its head atoms. □

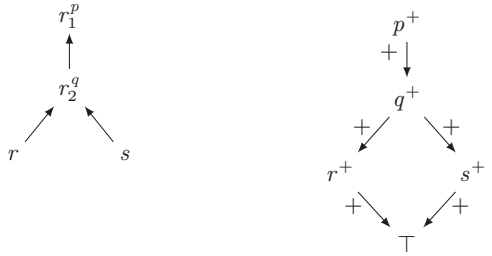
Let us also recall that, for normal programs, (Cabalar et al. 2014) defining causal answer sets as $\leq$-minimal models instead of $\sqsubseteq$-minimal ones. These two definitions agree for normal logic programs (Cabalar and Fandinno 2016) with the former being preferred for its simplicity.[12] On the other hand, for disjunctive programs, there are $\leq$-minimal models that do not correspond to any standard stable model, thus the need for the latter. This is illustrated by the following example.

Example 30
Let P_{17} be the following logic program:

$$r_1: \quad head \vee tails \qquad\qquad head$$

which has two $\leq$-minimal models, one in which $I_{17}(head) = head + r_1^{head}$ and $I_{17}(tails) = 0$, plus another in which $I'_{17}(head) = head$ and $I'_{17}(tails) = r_1^{tails}$. However, only the former is a $\sqsubseteq$-minimal one. Note that this corresponds to the set of atoms $Atoms(I_{17}) = \{head\}$ which is the unique standard answer set of the program. □

[12] This definition is also used in Section 3.3.3 where the syntax is restricted to normal programs.



(a) (b)

Fig. 19: Causal graph and off-line justification of p w.r.t. the unique answer set of P_{18} (see Examples 31 and 31).

The following example illustrates the fact that '$*$' is used to represent joint causation, or in other words, that two or more atoms are needed to justify the conclusion of some rule.

Example 31
Consider the logic program P_{18} consisting of the following rules:

$$r_1 : \ p \leftarrow q \qquad r_2 : \ q \leftarrow r \wedge s \qquad r \qquad s$$

This program has a unique causal answer set I_2 that satisfies:

$$I_2(p) = (r * s)\cdot r_2^q \cdot r_1^p \qquad I_2(r) = r$$
$$I_2(q) = (r * s)\cdot r_2^q \qquad I_2(s) = s$$

As shown in Observation 1, we have $I_2(r) \geq r\cdot r = r$. Then, the value of $I_2(r)$ follows from the fact that causal answer sets are $\leq$-minimal models. Similar reasoning applies for the atom s. Furthermore, from Definition 14, it follows that $I_2(q) \geq (r*s)\cdot r_2^q$ and, by minimality, that $I_2(q) = (r*s)\cdot r_2^q$. In a similar way, we obtain for p that $I_2(p) = I_2(q)\cdot r_1^p = (r*s)\cdot r_2^q \cdot r_1^p$. Figure 19a depicts the causal graph associated with $I_2(p)$. Note that product '$*$' is translated in this causal graph (Figure 19a) as two incoming edges to the vertex r_2^q. The causal graph associated with some causal value can be easily constructed by rewriting the causal value in graph normal form and representing each term of the form $v_1\cdot v_2$ with an edge from v_1 to v_2. In particular, we can obtain the causal graph in Figure 19a by rewriting $(r * s)\cdot r_2^q \cdot r_1^p$ in graph normal form as follows:

$$\begin{aligned}(r * s)\cdot r_2^q \cdot r_1^p &= r\cdot r_2^q \cdot r_1^p * s\cdot r_2^q \cdot r_1^p \\ &= r\cdot r_2^q * r_2^q \cdot r_1^p * s\cdot r_2^q * r_2^q \cdot r_1^p \\ &= r\cdot r_2^q * r_2^q \cdot r_1^p * s\cdot r_2^q\end{aligned}$$

Then, the three edges of the causal graph in Figure 19a correspond to the three subterms of the form $v_1\cdot v_2$ (that is, $r\cdot r_2^q$, $r_2^q\cdot r_1^p$ and $s\cdot r_2^q$) in the above causal term. For comparison, Figure 19b depicts the off-line justification of p^+. It is easy to see that this particular off-line justification can be obtained from the causal graph by

replacing each vertex r_i^a by a, reversing edges, adding the label '+' to each vertex and resulting edge and adding edges of the form $(a, \top, +)$ for each resulting sink a. □

Next, we illustrate that '+' is used to separate alternative causal justifications and the importance of addition distributivity to obtain such behaviour.

Example 32
Consider the logic program P_{19} consisting of the following rules:

$$r_1 : \ p \leftarrow q \qquad r_2 : \ q \leftarrow r \qquad r_3 : \ q \leftarrow s \qquad r \qquad s$$

This program has a unique causal answer set I_3 that satisfies:

$$I_3(p) = r{\cdot}r_2^q{\cdot}r_1^p + s{\cdot}r_3^q{\cdot}r_1^p \qquad I_3(r) = r$$
$$I_3(q) = r{\cdot}r_2^q + s{\cdot}r_3^q \qquad I_3(s) = s$$

As in Example 31, we have that $I_3(r) = r$ and $I_3(s) = s$. Furthermore, in this case, Definition 14 implies $I_3(q) \geq r{\cdot}r_2^q$ and $I_3(q) \geq s{\cdot}r_3^q$. Then, the value of $I_3(q)$ follows from the fact that causal answer sets are $\leq$-minimal models and the fact that '+' is the least upper bound of the $\leq$ relation. Finally, $I_3(p) = I_3(q){\cdot}r_1^p = (r{\cdot}r_2^q + s{\cdot}r_3^q){\cdot}r_1^p$ follows in similar way. The value of $I_3(p)$ shown above is the disjunctive normal form of this term, and it is obtained by applying addition distributivity. Here, both addends in $I_3(p)$, that is $r{\cdot}r_2^q{\cdot}r_1^p$ and $s{\cdot}r_3^q{\cdot}r_1^p$, are causal justifications of p w.r.t. the unique answer set of the program. □

3.3.2 Causal Semantics for Programs with Negation-as-Failure

We now extend the causal answer set semantics to logic programs with negation-as-failure. For this, the *closed world assumption* is directly translated into the language of justifications, assuming that everything that has no justification is false by default. Accordingly, negative literals are assumed to hold by default, without requiring further justification. This contrasts with the previously presented off-line and LABAS justifications, which further explain why negative literals hold. The next section shows how causal justifications can be extended in order to provide such information. Let us start with an example motivating why omitting the justification of negative literals, thus treating them as defaults, may provide intuitive explanation in some scenarios.

Example 33 (*Ex. 23 continued*)
Consider a variation of the scenario of Example 23 in which shooting the victim may fail in several ways: the victim may be wearing a *bulletproof* vest, the gunpowder may be *wet*, etc. This is an instance of the well-known *qualification problem* (McCarthy 1977): any comprehensive knowledge base for general commonsense reasoning may contain hundreds or thousands of exceptions to any rule, which may also be impossible to list in advance. As usual in answer set programming, this problem can be solved by adding abnormality predicates to the body of all rules.

In particular, rules (2-7) are rewritten as follows:

$$r_1: \quad dead \leftarrow haemorrhage \wedge not\, ab_1 \tag{13}$$
$$r_2: \quad haemorrhage \leftarrow impact \wedge not\, ab_2 \tag{14}$$
$$r_3: \quad impact \leftarrow bullet \wedge not\, ab_2 \tag{15}$$
$$r_4: \quad bullet \leftarrow gunpowder \wedge not\, ab_3 \tag{16}$$
$$r_5: \quad gunpowder \leftarrow trigger(suzy) \wedge not\, ab_4 \tag{17}$$
$$trigger(suzy) \tag{18}$$

Then, exceptions can be added in an *elaboration tolerant* manner by adding new rules as follows:

$$r_6: \quad ab_2 \leftarrow bulletproof \tag{19}$$
$$r_7: \quad ab_4 \leftarrow wet \tag{20}$$

Let P_{20} be the program containing rules (13-20). □

For justifications, Example 33 sets out a new challenge: a justification for the lack of all exceptions may be much bigger than the justification for the conclusion without exceptions. Furthermore, from a causal perspective, saying that the lack of an exception is part of a cause (e.g., for *dead*) may seem rather counterintuitive. It is not the case that the victim is *dead* because the gunpowder was not *wet*, or because the victim was not wearing a *bulletproof* vest, or whatever other possible exception might be added in the future.

This is a well-known problem in the causal literature (Maudlin 2004; Hall 2007; Halpern 2008; Hitchcock and Knobe 2009): in particular, Hitchcock and Knobe (2009) provides an extended discussion with several examples showing how people ordinarily understand causes as deviations from a normal or default behaviour. In this sense, by understanding falsity of exceptions as the *default situation*, we obtain that, when no exception is true with respect to the causal answer set, the causal justifications for *dead* in programs P_{14} and P_{20} are the same. This interpretation of negation-as-failure can be captured by the following definitions:

Definition 17 (*Causal Program Reduct*)
The *(causal) reduct* of an extended program P with respect to a causal interpretation I, in symbols P^I, is the result of

1. removing all rules such that $I(b) \neq 0$ for some $b \in body^-(r)$,
2. removing all the negative literals from the remaining rules. □

Definition 18 (*Causal Answer Set*)
We say that a causal interpretation I is a *causal answer set* of an extended program P iff I is a causal answer set of the positive program P^I. □

Example 34 (*Ex. 33 continued*)
Let I_4 be an interpretation such that $I_4(A) = I_1(A)$ for all literals A occurring in the program P_{14}, and $I_4(A) = 0$ for all other literals occurring in program P_{20}. Then,

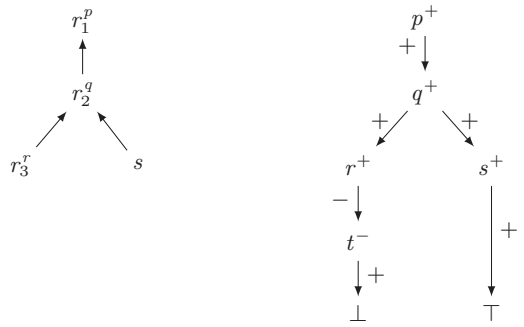


Fig. 20: Causal graph and off-line justification of p w.r.t. the unique answer set of P_5 (see Example 6 on page 11 and Example 35).

it is easy to see that $P_{20}^{I_4} = P_{14} \cup \{(19), (20)\}$ and, thus, that I_4 is the $\sqsubseteq$-minimal model of $P_{20}^{I_4}$. Note that 0 is the bottom value and there are no rules assigning greater values to *bulletproof* or *wet* and, thus, neither to any of the ab_i. That is, the unique answer sets of programs P_{14} and P_{20} agree on the causal values assigned to all literals they have in common. □

We note that the behaviour of causal justifications in Example 33 is similar to LABAS justifications in the sense that, in the latter, the defaults are not further explained either. This happens because there are no derivations for any abnormality atom ab_i. On the other hand, if exceptions could be derived, then the behaviour would be different. For instance, let P_{21} be the program obtained from P_{20} by replacing rule (19) by the following two rules

$$r_6: \quad ab_2 \leftarrow bulletproof \wedge not\, ab_5 \tag{21}$$

$$r_8: \quad ab_5 \leftarrow damaged \tag{22}$$

plus the facts *bulletproof* and *damaged*. In this case, ab_2 is still false, so the causal justification of *dead* remains the same. However, now there is a derivation for ab_2 which is 'attacked' by *damaged*, so a LABAS justification further justifies the falsity of exception ab_2 in terms of *damaged*. The following example illustrates some similarities and differences between causal and off-line justifications.

Example 35 (*Ex. 6 continued, page 11*)
Let us now consider the program P_5 and the following labelling of its rules

$$r_1: \; p \leftarrow q \qquad r_2: \; q \leftarrow r \wedge s \qquad r_3: \; r \leftarrow not\, t \qquad s$$

Then, the unique causal answer set I_5 of program P_5 satisfies:

$$I_5(p) = (r_3^r * s) \cdot r_2^q \cdot r_1^p \qquad I_5(s) = s$$
$$I_5(q) = (r_3^r * s) \cdot r_2^q \qquad I_5(t) = 0$$
$$I_5(r) = r_3^r$$

Figure 20a depicts the causal graph associated with $I_5(p)$, while Figure 20b depicts the off-line justification of p^+ for the sake of comparison. Note that the causal graph can be obtained from the off-line justification by removing the $\bot$, $\top$ and all negatively labelled vertices plus all the edges connected to these vertices (where the edges are inverted). Note that the only change in the causal justification of p in this example with respect to that in program P_{18} is the renaming of the node r as r_3^r, while off-line justifications of the two programs further differ in the subgraph rooted in r^+. □

Example 36
Let us consider a scenario where there is a light bulb that turns *on* whenever the switches a and b are pushed at the same time, and *off* whenever the switches c and d are pushed at the same time. Assume also that the light is currently *off* and the switches a and b are pushed (situation 0). This problem can be easily formalised

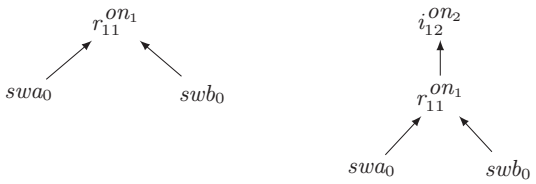


Fig. 21: Causal justifications of the truth of on_1 and on_2.

as a logic program P_{22} consisting of rules[13]:

$$r_{1t+1}: \ on_{t+1} \leftarrow swa_t \wedge swb_t \qquad r_{2t+1}: \ off_{t+1} \leftarrow swc_t \wedge swd_t \tag{23}$$

for $t \geq 0$, plus the facts off_0, swa_0 and swb_0. As usual, inertia is represented by the following pair of rules:

$$i_{1t+1}: \ on_{t+1} \leftarrow on_t \wedge not\ off_{t+1} \tag{24}$$

$$i_{2t+1}: \ off_{t+1} \leftarrow off_t \wedge not\ on_{t+1} \tag{25}$$

for $t \geq 0$. We also have an integrity constraint

$$\leftarrow on_t \wedge off_t \tag{26}$$

ensuring that *on* and *off* cannot hold at the same time. This program has a complete well-founded model and, thus, a unique answer set, in which on_t holds for every time $t > 0$. Figures 21a and 21b respectively depict the causal justifications of on_1 and on_2 w.r.t. that answer set. □

[13] For the sake of simplicity, we avoid introducing a first order language here and indirectly use the propositional logic program that is produced through grounding.

As illustrated by the above example, understanding negation-as-failure as a default (which does not need to be further explained), allows that causal justifications are 'preserved' by inertia in the following sense: at any situation $t+1$ if nothing happens, then the causal justification of on_{t+1} can be obtained by adding to the causal justification of on_t, an edge from $i_{1t}^{on_t}$ to $i_{1t+1}^{on_{t+1}}$. True persistence of justifications, that is, exactly the same justification preserved by inertia, can be obtained by selecting some rule labels, in this case the labels associated with inertia (i_{1t+1} and i_{2t+1}), as not forming part of the causal justifications, and thus of the causal graphs. In such case, the causal graph for on_t at any situation t would be the one depicted in Figure 21a. In contrast, the number of off-line and LABAS justifications grows exponentially with the number of situations in which nothing happens. This will be discussed in more detail in Section 3.6.

3.3.3 Explaining Negative Literals in Causal Justifications

As we have seen, one major difference between causal justifications and the two previous approaches, off-line and LABAS justifications, is the way in which all negative literals that are true w.r.t. the answer set in question are assumed to hold by default, so they do not need further justification. This behaviour allows to get an important reduction in the number of justifications in examples that involve exceptions or defaults like inertia (as was illustrated in Example 36). On the other hand, there are scenarios in which justifications for negative literals are valuable.[14] Consider, for instance, the following example from (Cabalar and Fandinno 2017):

Example 37
A drug d in James Bond's drink causes his paralysis p provided that he was not given an antidote a that day. We know that Bond's enemy, Dr. No, poured the drug and that Bond is daily administered an antidote by the MI6, unless it is a holiday h:

$$r_1: \quad p \leftarrow d, \mathit{not}\ a \tag{27}$$

$$r_2: \quad a \leftarrow \mathit{not}\ h \tag{28}$$

$$d \tag{29}$$

Then, $\{a, d\}$ is the unique answer set of the program consisting of rules (27-29). Since p is false with respect this answer set, the causal value associated to it is 0, that is, it has its value by default without further explanation. On the other hand, Figures 22a and 22b respectively depict the off-line and LABAS justifications explaining that p does not hold because a is somehow preventing it. The extension of causal justifications, presented in this section, associates the causal value $(\sim r_2^a * d) \cdot r_1^p$ to p in this scenario, pointing out that rule r_2 (and, thus a) is what prevents p from becoming true. A causal reading of this expression is that "a has prevented (through rule r_2) d to cause p (through rule r_1)" or, equivalently, "if it was not for rule r_2 (implying a), d would cause p through rule r_1". Suppose now

[14] A more detailed elaboration of this argument can be found in Section 3.6.

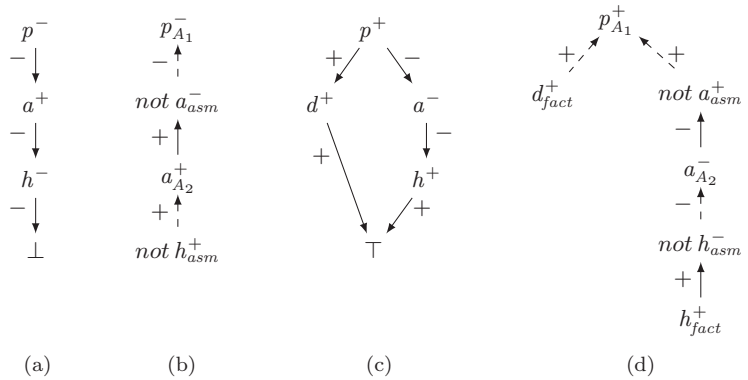


Fig. 22: Off-line and LABAS justifications of p w.r.t. the unique answer set of Example 37.

that it is a holiday, so fact h is added to the program (27)-(28). Then, a is itself disabled and d is free to cause p. The causal justification of p in this case is $d{\cdot}r_1^p$ (which corresponds to the graph with a single edge from d to r_1^p), which reflects the fact that d has caused p, but without keeping any record about the fact that h has also been necessary for this to happen. On the other hand, we can see in Figures 22c and 22d that both off-line and LABAS justifications keep track of this dependency. The extended causal justifications also keep track of this dependency and associate the casual value $(\sim\!\sim h * d){\cdot}r_1^p + (\sim r_2^a * d){\cdot}r_1^p$ with p. Here, the first addend can be informally read as "h has allowed d to cause p (through rule r_1)." Double negation in front of h is introduced to distinguish between the philosophically distinct concepts[15] of *productive cause* (in this case d) and other contingently counterfactual dependencies (in this case h), though this distinction is not of particular relevance in the context of justifications. As before, the second addend can be informally read as "if it was not for rule r_2 (implying a), d would cause p through rule r_1" (even without the presence of h). □

In order to introduce information about negative literals in causal justifications, Cabalar and Fandinno (2017) extended causal justifications with a negation inspired by why-not provenance justifications (see Section 3.4; Damásio et al. 2013). We now review this extension, starting with the introduction of negation in causal terms as follows:

Definition 19 (*Extended Causal Terms*)
Given a set of atoms At and a set of labels Lb, an *extended (causal) term* (*e-term*

[15] A productive cause is an event connected to its effect by a causal chain as explained at the beginning of Section 3.3. For a thorough philosophical explanation about the differences between productive causes and contingently counterfactual dependencies we refer to (Hall 2004; Hall 2007).

Pseudo-complement	*De Morgan*	*Weak excl. middle*	*appl. negation*
$t * {\sim}t = 0$ ${\sim}{\sim}{\sim}t = {\sim}t$	${\sim}(t{+}u) = ({\sim}t * {\sim}u)$ ${\sim}(t * u) = ({\sim}t {+} {\sim}u)$	${\sim}t + {\sim}{\sim}t = 1$	${\sim}(t \cdot u) = {\sim}(t * u)$

Fig. 23: Properties of the '$\sim$' operator.

for short), t is recursively defined as one of the following expressions

$$t \;::=\; l \;\mid\; \prod S \;\mid\; \sum S \;\mid\; t_1 {\cdot} t_2 \;\mid\; {\sim}t_1$$

where $l \in Lb_{ext} \stackrel{\text{def}}{=} \{\; r_i^a \;\big|\; r_i \in Lb \text{ and } a \in At_{ext} \;\}$, t_1, t_2 are in turn terms, and S is a (possibly empty and possibly infinite) set of terms. An e-term is *elementary* if it has the form l, ${\sim}l$ or ${\sim}{\sim}l$ with $l \in Lb_{ext}$ being an extended label. □

Definition 20 (*Extended Causal Values*)
An *extended (causal) value* (*e-value* for short) is each equivalence class of e-terms under axioms for a completely distributive (complete) lattice with meet '$*$' and join '$+$' plus the axioms of Figures 18 and 23. The set of e-values is denoted by $\mathbf{E}_{Lb}$. □

As with causal values, we will use any of the members of the class as representative of the extended casual value. Note that $[0] = \{0, r_1^a * {\sim}r_1^a, \dots,\}$ and $[1] = \{1, {\sim}r_1^a + {\sim}{\sim}r_1^a, \dots\}$ are no longer singleton sets. The definition of disjunctive and graph normal form is now strengthened by requiring that negation '$\sim$' or double negation '$\sim\sim$' only occurs in front of labels and extended atoms. Similarly, the graph normal form also requires now that negation '$\sim$' or double negation '$\sim\sim$' only occurs in front of labels and extended atoms.

Interpretations are extended in a straightforward way: an *e-interpretation* is a mapping $\mathcal{I} : At_{ext} \longrightarrow \mathbf{E}_{Lb}$ assigning an e-value to each extended atom such that $\mathcal{I}(a) = 0$ or $\mathcal{I}(\neg a) = 0$ for every atom $a \in At$. For interpretations $\mathcal{I}$ and $\mathcal{J}$ we say that $\mathcal{I} \leq \mathcal{J}$ when $\mathcal{I}(a) \leq \mathcal{J}(a)$ for each atom $a \in At_{ext}$. As above, there is a $\leq$-bottom e-interpretation $\mathbf{0}$ (resp. a $\leq$-top e-interpretation $\mathbf{1}$) that stands for the e-interpretation mapping each extended atom a to 0 (resp. 1). The value assigned to a negative literal *not* a by an e-interpretation $\mathcal{I}$, denoted as $\mathcal{I}(\mathit{not}\ a)$, is defined as $\mathcal{I}(\mathit{not}\ a) \stackrel{\text{def}}{=} {\sim}\mathcal{I}(a)$, as expected. Similarly, for any e-term t, its valuation $\mathcal{I}(t) \stackrel{\text{def}}{=} [t]$ is the equivalence class of t.

To define the semantics of logic programs for extended causal justifications a slight extension in the syntax is also needed: we allow that $b_1, \dots, b_n$ in (1), are not only extended atoms, but also e-terms. For instance, $p \leftarrow q \wedge (a * {\sim}b)$, with $p, q \in At_{ext}$ and $a, b \in Lb$, is a valid rule in this extended syntax. Furthermore, only normal logic programs are considered.

Definition 21 (*E-Model*)
A e-interpretation $\mathcal{I}$ *satisfies* a rule like (1) with $k = 1$ iff

$$\big(\; \mathcal{I}(b_1) * \ldots * \mathcal{I}(b_n) * \mathcal{I}(\mathit{not}\ c_1) * \ldots * \mathcal{I}(\mathit{not}\ c_m) \;\big) \cdot r_i^{h_1} \;\leq\; \mathcal{I}(h_1) \qquad (30)$$

and $\mathcal{I}$ is an e-model of P, written $\mathcal{I} \models P$, iff $\mathcal{I}$ satisfies all rules in P. □

Definition 22 (*E-Reduct*)
Given a normal program P and an interpretation $\mathcal{I}$, by $P^{\mathcal{I}}$ we denote the positive program containing a rule of the form[16]:

$$h_1 \leftarrow b_1, \ldots, b_n,\ \mathcal{I}(not\ c_1), \ldots,\ \mathcal{I}(not\ c_m) \tag{31}$$

for each rule of the form (1) in P. □

Program $P^{\mathcal{I}}$ is positive and it has a $\leq$-*least e-model*[17]. By $\hat{\Gamma}_P(\cdot)$, we denote the operator[18] mapping each e-interpretation $\mathcal{I}$ to the $\leq$-least e-model of program $P^{\mathcal{I}}$. Furthermore, $\hat{\Gamma}^2_P(\cdot)$ denotes the operator over e-interpretations resulting of applying $\hat{\Gamma}_P$ to the result of is its application to any e-interpretation, that is, $\hat{\Gamma}^2_P(\mathcal{I}) \stackrel{\text{def}}{=} \hat{\Gamma}_P(\hat{\Gamma}_P(\mathcal{I}))$. This operator $\hat{\Gamma}^2_P$ is monotonic and so, by Knaster-Tarski's theorem, it has a least fixpoint $\mathbb{L}_P$ and a greatest fixpoint $\mathbb{U}_P \stackrel{\text{def}}{=} \hat{\Gamma}_P(\mathbb{L}_P)$. These two fixpoints respectively correspond to the justifications for true and for non-false (that is, either true of undefined) extended atoms in the (standard) well-founded model. To capture justifications with respect to answer sets, we use the negative reduct from Definition 3.

Definition 23 (*Extended Causal Answer Sets*)
Given a normal extended program P one of its standard answer sets M, and a set of assumptions $U \subseteq \overline{M}$ such that $WF_{NR(P,U)} = M$, the *extended causal answer set* (*e-answer set*) corresponding to M and U is a function mapping each literal to an e-value as follows:

$$\mathbb{M}_U(a) \stackrel{\text{def}}{=} \mathbb{L}_Q(a) \qquad\qquad \mathbb{M}_U(not\ a) \stackrel{\text{def}}{=} \sim\mathbb{U}_Q(a)$$

with $Q = NR(P, U)$. □

The notion of causal justification is extended as expected.

Definition 24 (*Extended Causal Justification*)
Given a logic program P, an answer set M of P and a set of assumptions $U \subseteq \overline{M}$, a term without sums c is an *extended causal justification* of some literal $l \in \{a, not\ a\}$ w.r.t. P, M and U if c is an addend in the minimal disjunctive normal form of $\mathbb{M}_U(l)$. For any causal justification of l w.r.t. P, M and U $graph(c)$ is an *extended causal graph (justification)*. □

Example 38 (*Ex. 37 continued*)
Let P_{23} be the logic program containing rules (27-28). This program has a complete well-founded model which coincides with its unique answer set: $M_{19} = \{a, d\}$. Then, the possible assumptions with respect to this answer set are those U such that

[16] Note that $\mathcal{I}(not\ c_i)$ is a possibly infinite causal term for each c_i.
[17] Here, we take $\leq$-minimal models instead of $\sqsubseteq$-minimal models as in earlier sections. These two concepts coincide for normal programs, so we use the former for simplicity.
[18] The operator $\hat{\Gamma}_P(\cdot)$ is analogous to the operator $\Gamma_P(\cdot)$ defined in Section 2, but using e-interpretations instead of sets of atoms.

$U \subseteq \{h\}$, that is, $\{\}$ and $\{h\}$. Usually $\subseteq$-minimal assumptions are used and, thus, we have that $P_{23} = NR(P_{23}, \{\})$ and that

$$\begin{array}{lll}
\hat{\Gamma}_{P_{23}}(\mathbf{0})(p) = d{\cdot}r_1^p & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(p) = (\sim r_2^a * d){\cdot}r_1^p & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(p) = (\sim r_2^a * d){\cdot}r_1^p \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(d) = d & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(d) = d & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(d) = d \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(a) = r_2^a & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(a) = r_2^a & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(a) = r_2^a \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(h) = 0 & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(h) = 0 & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(h) = 0
\end{array}$$

Note that $\hat{\Gamma}^2_{P_{23}}(\mathbf{0}) = \hat{\Gamma}^3_{P_{23}}(\mathbf{0})$ also implies that $\hat{\Gamma}^2_{P_{23}}(\mathbf{0}) = \hat{\Gamma}^4_{P_{23}}(\mathbf{0})$ and, thus, $\hat{\Gamma}^2_{P_{23}}(\mathbf{0})$ is the least fixpoint of the $\hat{\Gamma}^2_{P_{23}}(\mathbf{0})$ operator. Note also that $\hat{\Gamma}^2_{P_{23}}(\mathbf{0})(p) = (\sim r_2^a * d){\cdot}r_1^p$ is precisely the causal justification shown in Example 37 to be associated with p in this scenario. Let now $P_{24} = P_{23} \cup \{h\}$, which also has a complete well-founded model and unique answer set: $M_{20} = \{p, d, h\}$. In this case, we have

$$\begin{array}{lll}
\hat{\Gamma}_{P_{23}}(\mathbf{0})(p) = d{\cdot}r_1^p & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(p) = (\sim r_2^a * d){\cdot}r_1^p & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(p) = \ldots \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(d) = d & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(d) = d & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(d) = d \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(a) = r_2^a & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(a) = \sim h{\cdot}r_2^a & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(a) = \sim h{\cdot}r_2^a \\
\hat{\Gamma}_{P_{23}}(\mathbf{0})(h) = h & \hat{\Gamma}^2_{P_{23}}(\mathbf{0})(h) = h & \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(h) = h
\end{array}$$

with $\hat{\Gamma}^4_{P_{23}}(\mathbf{0})(p) = \hat{\Gamma}^3_{P_{23}}(\mathbf{0})(p) = (\sim\sim h * d){\cdot}r_1^p + (\sim r_2^a * d){\cdot}r_1^p$ as also mentioned in Example 37. □

An extended causal justification is said to be *inhibited* when it contains a negated label (non-double negated). Inhibited justifications point out derivations that could have justified the truth value of the atom, but that have been prevented to do so. The negated subterms are the inhibitors of the extended causal justification. *Actual* extended causal justifications are those that only contain non-negated and double negated subterms. In Example 38, the casual term $(\sim\sim h{*}d){\cdot}r_1^p$ represents the actual extended causal justification of p, while $(\sim r_2^a * d){\cdot}r_1^p$ is an inhibited extended causal justification that points out that "had it not been for rule r_2, then d would cause p to be true through rule r_1 (without the need of h)". Note that the presence of the negated subterm $\sim r_2^a$ in the inhibited extended causal justification $(\sim r_2^a * d){\cdot}r_1^p$ is similar to the attack from the argument with conclusion a to the argument with conclusion p in the attack tree used to construct the LABAS justification.

Example 39 (*Ex. 36 continued*)
Continuing with the problem introduced in Example 36 (page 38), we can see that $\hat{\Gamma}^i_{P_{22}}(\mathbf{0})(on_1) = (swa_0 * swb_0){\cdot}r_{11}$ for all $i \geq 1$. That is, the extended causal justification of on_1 has precisely the same graph as the (non-extended) causal justification depicted in Figure 21a (page 38). We also have that $\hat{\Gamma}^i_{P_{22}}(\mathbf{0})(off_1) = (\sim swa_0 * off_0){\cdot}i_{22} + (\sim swb_0 * off_0){\cdot}i_{22} + (\sim r_{11} * off_0){\cdot}i_{22}$ for all $i \geq 2$. This points out that off_1 would be true by inertia (rule i_{22}) if any of the facts swa_0 or swb_0 or the rule r_{11} had not been in the program. It can be checked that $(swa_0 * swb_0){\cdot}r_{11}{\cdot}i_{12}$ is the extended causal justification of on_2. Recall that this is the (non-extended) causal justification of on_2, whose corresponding causal graph is depicted in Figure 21b (page 38). □

Example 40 (*Ex. 38 continued*)
Recall that, in the unique answer set $M_{19} = \{d, a\}$ of program P_{23}, the atom p is false. Extended causal justifications also allow to justify negative literals and we have that *not* p is explained by the causal value $\sim\sim r_2^a + \sim d + \sim r_1^p$. Here, $\sim\sim r_2^a$ is the actual extended causal justification explaining why p is false, while $\sim d$ and $\sim r_1^p$ are inhibited extended causal justifications that point out that p would also be false if either d or r_1 were removed from the program. □

Note that in Example 40 the application operator '$\cdot$' does not appear in the extended causal justification of *not* p. In fact, this is the general case for negative literals and, thus, extended causal justifications for negative literals do not keep track of the derivation order among rules. An algebraic treatment that allows to keep track of this derivation order is still an open topic. It is also an open topic to explain negative literals for disjunctive programs.

3.4 Why-not Provenance Justifications

Why-not provenance (Damásio et al. 2013) is a declarative logical approach, which extracts non-graph based justifications for the truth value of atoms with respect to the (complete) well-founded model of normal logic programs. It can furthermore be used to explain the truth value of atoms with respect to the answer set semantics. The approach has been implemented in a meta-programming tool (Damásio et al. 2015) available at `http://cptkirk.sourceforge.net`. As mentioned in Section 3.3.3, the way extended causal justifications have been defined is inspired by this approach, therefore, we here just introduce the differences between these two approaches, avoiding the overlapping material.

As already mentioned, the first major difference compared to extended causal justifications (and the other justifications approaches reviewed in Section 3) is the non-graph nature of why-not provenance. Instead, why-not provenance justifications are sets of annotations, each one expressing a possible modification of the program to achieve a particular truth value of the justified atom w.r.t. the well-founded model (of the modified program). In other words, why-not provenance computes justifications expressing how the atom can be made true, false, or undefined w.r.t. the well-founded model or the answer set semantics. The justifications for the *actual* truth value of the atom are those that do not imply any modification on the program. This can be achieved by adding the axiom

$$(t \cdot u) \quad = \quad (t * v) \tag{32}$$

to those defining e-values (Definition 20). That is, the non-commutative operator '$\cdot$' is replaced by the commutative one '$*$', effectively removing the order of application of rules from the justifications.

The second difference compared to extended causal justifications is that why-not provenance does not distinguish between productive causes and other counterfactual dependencies, which is achieved by adding the double negation elimination axiom:

$$\sim\sim t \quad = \quad t \tag{33}$$

Definition 25 (*Why-Not Provenance Values*)
A *why-not provenance value* (w-value for short) is each equivalence class of e-terms (Definition 19, page 19) under axioms for a completely distributive (complete) lattice with meet '$*$' and join '$+$' plus the axioms of Figures 18 and 23 and the axioms $(t{\cdot}u) = (t * v)$ and $\sim\!\sim t = t$. The set of w-values is denoted $\mathbf{W}_{Lb}$. □

Due to the addition of axioms (32) and (33), w-values form a free boolean algebra[19] generated by Lb. The definitions of w-interpretation, w-model and reduct are analogous to the ones in Section 3.3.3, but replacing e-values by w-values. We will use $\tilde{\mathcal{I}}$, $\tilde{\mathcal{J}}$ and their variations to denote w-interpretations. By $\tilde{\Gamma}_P(\tilde{\mathcal{I}})$ we denote the least w-model of program $P^{\tilde{\mathcal{I}}}$ and by $\tilde{\Gamma}^2_P(\mathcal{I}) \stackrel{\text{def}}{=} \tilde{\Gamma}_P(\tilde{\Gamma}_P(\mathcal{I}))$ we denote the result of applying $\tilde{\Gamma}_P$ to the result of its application to $\tilde{\mathcal{I}}$. Let us denote by $\mathfrak{T}_P$ and $\mathfrak{TU}_P$, the least and greatest fixpoint of the operator $\tilde{\Gamma}^2_P$.

Notation 2
In order to closely follow the notation used in (Damásio et al. 2013), we will represent the meet as conjunction '$\wedge$' instead of as product '$*$' and the joint as disjunction '$\vee$' instead of '$+$' when representing w-values. We will also write negation as '$\neg$' instead of '$\sim$' to strengthen the fact that it now acts as classical negation and omit the superindex of labels. □

Note that the intuition of the two former operators is as before: conjunction '$\wedge$' indicates joint interaction, disjunction '$\vee$' represents alternative justifications. On the other hand, now negation '$\neg$' denotes hypothetical changes to the program (either removal or addition) that may lead to the literal belonging to the well-founded model.

Example 41 (*Ex. 10 continued*)
Let us label each rule in the program P_8 as follows

$$r_1 :\ p \leftarrow not\ q \qquad r_2 :\ r \leftarrow not\ p \qquad r_3 :\ s \leftarrow not\ r$$

As mentioned in Example 10, this program has a complete well-founded model: $M_8 = \{p, s\}$. We also have that the following extended causal justifications:

$$\begin{array}{lll}
\hat{\Gamma}_{P_8}(\mathbf{0})(p) = r_1^p & \hat{\Gamma}^2_{P_8}(\mathbf{0})(p) = r_1^p & \hat{\Gamma}^3_{P_8}(\mathbf{0})(p) = r_1^p \\
\hat{\Gamma}_{P_8}(\mathbf{0})(q) = 0 & \hat{\Gamma}^2_{P_8}(\mathbf{0})(q) = 0 & \hat{\Gamma}^3_{P_8}(\mathbf{0})(q) = 0 \\
\hat{\Gamma}_{P_8}(\mathbf{0})(r) = r_2^r & \hat{\Gamma}^2_{P_8}(\mathbf{0})(r) = \sim r_1^p{\cdot}r_2^r & \hat{\Gamma}^3_{P_8}(\mathbf{0})(r) = \sim r_1{\cdot}r_2^r \\
\hat{\Gamma}_{P_8}(\mathbf{0})(s) = r_3^s & \hat{\Gamma}^2_{P_8}(\mathbf{0})(s) = \sim r_2^r{\cdot}r_3^s & \hat{\Gamma}^3_{P_8}(\mathbf{0})(s) = \sim\!\sim r_1{\cdot}r_3^s + \sim r_2^r{\cdot}r_3^s
\end{array}$$

and, it can be checked that, $\hat{\Gamma}^4_{P_8}(\mathbf{0}) = \hat{\Gamma}^2_{P_8}(\mathbf{0})$. Then, applying the above two ax-

[19] In fact, the original definition relies on a free boolean algebra instead of causal terms and assumes the notation of logical formulas to represent its values (see Notation 2 below).

ioms (32-33) and the rewriting of Notation 2, we have that

$$\begin{aligned}
\tilde{\Gamma}^4_{P_8}(\mathbf{0})(p) &= r_1 \\
\tilde{\Gamma}^4_{P_8}(\mathbf{0})(q) &= 0 \\
\tilde{\Gamma}^4_{P_8}(\mathbf{0})(r) &= \neg r_1 \wedge r_2 \\
\tilde{\Gamma}^4_{P_8}(\mathbf{0})(s) &= r_1 \wedge r_3 \vee \neg r_2 \wedge r_3
\end{aligned}$$

The intuition behind $r_1 \wedge r_3$ is similar to the one in extended causal justifications, but without derivation order, distinction between productive causes and other contingently counterfactual dependencies: $r_1 \wedge r_3$ means that "s is true because both r_1 and r_3 are in the program". □

In other words, the least fixpoint of $\tilde{\Gamma}^2_P$ can be obtained from the least fixpoint of $\hat{\Gamma}^2_P$ by replacing applications '$\cdot$' by products '$*$', removing every double negation symbols '$\sim\sim$' and, then, applying the rewriting of Notation 2. More formally, let $\lambda : \mathbf{E}_{Lb} \longrightarrow \mathbf{W}_{Lb}$ be this transformation from e-values to w-values, that is, λ is defined in the following recursive way:

$$\lambda(t) \stackrel{\text{def}}{=} \begin{cases} \lambda(u) \wedge \lambda(w) & \text{if } t = u \odot v \text{ with } \odot \in \{*, \cdot\} \\ \lambda(u) \vee \lambda(w) & \text{if } t = u + v \\ \neg\lambda(u) & \text{if } t = \sim u \\ l & \text{if } t = l \text{ with } l \in (Lb \cup At_{ext}) \end{cases}$$

with t in graph normal form.

Note that, similar to LABAS justifications, there are no extended causal justifications for atoms for which there is no derivation. For instance, there is no justification for the atom p w.r.t. to a program consisting of a single rule $p \leftarrow q$. On the other hand, as in off-line justifications, there are why-not provenance justifications for those atoms. In our running example, p is associated with the why-not provenance information $\neg\, not(p) \vee r_1 \wedge \neg\, not(q)$ where r_1 is the label associated to the rule $p \leftarrow q$. This difference is due to the use of an extended program to compute why-not provenance information.

Definition 26 (*Provenance Program*)
Given a normal program P, the why-not provenance program is $\mathfrak{P}(P) \stackrel{\text{def}}{=} P \cup P'$, where P' contains a labelled fact of the form $(\neg not(a) : a)$ for each extended atom $a \in At_{ext}$ not occurring as a fact in P. □

We write $\mathfrak{P}$ instead of $\mathfrak{P}(P)$ when the program P is clear from the context. To compute the why-not provenance information of some normal program P, we will be interested in the least and greatest fixpoints of the $\tilde{\Gamma}^2_{\mathfrak{P}}$ operator with respect to the provenance program $\mathfrak{P}$ (corresponding to P), instead of those of P itself. That is, we will use the least and greatest fixpoints $\mathfrak{T}_{\mathfrak{P}}$ and $\mathfrak{TU}_{\mathfrak{P}}$. It is also worth noting that these fixpoints can be obtained from the fixpoints of extended causal operator with respect to the extended program, that is, $\mathfrak{T}_{\mathfrak{P}} = \lambda(\mathbb{L}_{\mathfrak{P}})$ and $\mathfrak{TU}_{\mathfrak{P}} = \lambda(\mathbb{U}_{\mathfrak{P}})$.

Definition 27 (*Provenance Information*)

Given a normal program P, *why-not provenance information* is defined as a mapping from literals[20] into w-values satisfying:

$$\begin{aligned} Why_P(a) &\stackrel{\text{def}}{=} \mathfrak{T}_{\mathfrak{P}}(a) \\ Why_P(not\ a) &\stackrel{\text{def}}{=} \neg\mathfrak{TU}_{\mathfrak{P}}(a) \\ Why_P(\text{undef}\ a) &\stackrel{\text{def}}{=} \neg\, Why_P(a) * \neg\, Why_P(not\ a) \end{aligned}$$

for each extended atom $a \in At_{ext}$. □

Intuitively, each disjunct in the minimal disjunctive normal form of provenance information corresponds to a justification about to why the atom does or does not have the respective truth value w.r.t. the well-founded model. That is, the disjunct in $Why_P(a)$, $Why_P(not\ a)$, and $Why_P(\text{undef}\ a)$ respectively explain why a is (not) true, false, and undefined w.r.t. the well-founded model. The *actual* truth value of a can be spotted if a disjunct in the respective justification ($Why_P(a)$, $Why_P(not\ a)$, or $Why_P(\text{undef}\ a)$) does not contain any negation $\neg$.

Example 42 (*Ex. 41 continued*)
Continuing with our running example, we have that $\mathfrak{P}_8 = \mathfrak{P}(P_8)$ consists of the following rules:

$$\begin{array}{lll} r_1 :\ p \leftarrow not\ q & \neg not(p) :\ p & \neg not(s) :\ s \\ r_2 :\ r \leftarrow not\ p & \neg not(q) :\ q & \\ r_3 :\ s \leftarrow not\ r & \neg not(p) :\ r & \end{array}$$

Since there is no fact q in P_8, we have that $(\neg not(q) :\ q)$ belongs to $\mathfrak{P}_8$. Furthermore, this is the unique rule in $\mathfrak{P}_8$ with q in the head and, consequently, we have that $\hat{\Gamma}^i_{\mathfrak{P}_8}(\mathbf{0})(q) = \neg not(q)$ for all $i \geq 1$. This implies that $\mathfrak{T}_{\mathfrak{P}}(q) = \mathfrak{TU}_{\mathfrak{P}}(q) = \neg not(q)$ and, thus, that

$$Why_{P_8}(q) = \neg not(q) \tag{34}$$
$$Why_{P_8}(not\ q) = not(q) \tag{35}$$
$$Why_{P_8}(\text{undef}\ q) = 0 \tag{36}$$

Note that $Why_{P_8}(q) = \neg not(q)$ corresponds to the off-line justification of q consisting of a unique edge $(q^-, \bot, -)$. On other hand, since there is no rule in P with q in the head, there is no LABAS nor (extended) causal justification of q. Similarly, to the computation shown in Example 41, we also have that

$$\begin{aligned} \tilde{\Gamma}^i_{P_8}(\mathbf{0})(p) &= \neg not(p) \vee r_1 \wedge not(q) \\ \tilde{\Gamma}^i_{P_8}(\mathbf{0})(r) &= \neg not(r) \vee r_2 \wedge not(p) \wedge \neg r_1 \ \vee\ r_2 \wedge not(p) \wedge \neg not(q) \\ \tilde{\Gamma}^i_{P_8}(\mathbf{0})(s) &= \neg not(s) \vee r_3 \wedge not(r) \wedge \neg r_2 \ \vee\ r_3 \wedge not(r) \wedge \neg not(p) \\ &\quad \vee\ r_3 \wedge not(r) \wedge r_1 \wedge not(q) \end{aligned}$$

[20] In this section, we use a more general notion of 'literal', where an atom a may not only be proceeded by *not*, but also by undef .

for all $i \geq 2$. This implies that $\mathfrak{T}_{\mathfrak{P}}(p) = \mathfrak{TU}_{\mathfrak{P}}(p) = \neg not(p) \vee r_1 \wedge not(q)$ and that

$$\begin{aligned} Why_{P_8}(p) &= \neg not(p) \vee r_1 \wedge not(q) \\ Why_{P_8}(not\, p) &= not(p) \wedge \neg r_1 \vee not(p) \wedge \neg not(q) \\ Why_{P_8}(\text{undef } p) &= 0 \end{aligned}$$

Following a similar procedure, it can be checked that

$$\begin{aligned} Why_{P_8}(r) &= \neg not(r) \vee r_2 \wedge not(p) \wedge \neg r_1 \vee r_2 \wedge not(p) \wedge \neg not(q) \\ Why_{P_8}(not\, r) &= not(r) \wedge \neg r_2 \vee not(r) \wedge \neg not(p) \vee not(r) \wedge r_1 \wedge not(q) \\ Why_{P_8}(\text{undef } r) &= 0 \end{aligned}$$

that $Why_{P_8}(s)$ is

$$\neg not(s) \tag{37}$$
$$\vee\, r_3 \wedge not(r) \wedge \neg r_2 \tag{38}$$
$$\vee\, r_3 \wedge not(r) \wedge \neg not(p) \tag{39}$$
$$\vee\, r_3 \wedge not(r) \wedge r_1 \wedge not(q) \tag{40}$$

and that $Why_{P_8}(not\, s)$ is

$$not(s) \wedge \neg r_3 \tag{41}$$
$$\vee\, not(s) \wedge \neg not(r) \tag{42}$$
$$\vee\, not(s) \wedge r_2 \wedge not(p) \wedge \neg r_1 \tag{43}$$
$$\vee\, not(s) \wedge r_2 \wedge not(p) \wedge \neg not(q) \tag{44}$$

Comparing the conjunction $r_1 \wedge r_3$ obtained in Example 41 with the conjunction (40), we can observe that annotations $not(r)$ and $not(q)$ have been added. This can be informally read as "s is true because both r_1 and r_3 are in the program and facts r and q are not." Note also, that $not(r) \wedge r_1 \wedge not(q)$ is one of the disjuncts of $Why_{P_8}(not\, r)$. This could be read as "r is false because of rule r_1 and the absence of facts r and q in the program." □

The following definitions formalises the notion of why-not provenance *justification*, i.e. a disjunct in the why-not provenance information, and the intuition behind the meaning of each annotation in a justification. In particular, it expresses the idea that each justification describes a modification of the program after which the atom has the truth value of the respective justification.

Definition 28 (*Why-not Provenance Justification*)
Let P be a normal program, let $a \in At_{ext}$ be an extended atom and let $l \in \{a, not\, a, \text{undef } a\}$ such that $Why_P(l) = c_1 \vee \ldots \vee c_n$ is the why-not provenance information of l in minimal disjunctive normal form. Then, we say that each c_i is a *why-not provenance justification* of l w.r.t. P. □

Definition 29
Let P be a normal program, $a \in At_{ext}$ be an extended atom and $l \in \{a, not\, a, \text{undef } a\}$.

Let c be some why-not provenance justification of l w.r.t. P and C a set of annotations such that $\bigwedge C = c$. Then, the following sets are defined, where $b \in At_{ext}$ and $r \in P$:

$$
\begin{array}{lcl}
KeepFacts(c) & \stackrel{\text{def}}{=} & \{\ b \mid\ \ b \in C\ \} \\
RemoveFacts(c) & \stackrel{\text{def}}{=} & \{\ b \mid \neg b \in C\ \} \\
MissingFacts(c) & \stackrel{\text{def}}{=} & \{\ b \mid \neg not(b) \in C\ \} \\
NoFacts(c) & \stackrel{\text{def}}{=} & \{\ b \mid\ \ not(b) \in C\ \} \\
KeepRules(c) & \stackrel{\text{def}}{=} & \{\ r \mid\ \ r_i \in C \text{ and } label(r) = r_i\ \} \\
RemoveRules(c) & \stackrel{\text{def}}{=} & \{\ r \mid \neg r_i \in C \text{ and } label(r) = r_i\ \}
\end{array}
$$

□

Intuitively, any disjunct c_j in the why-not provenance information of some literal l expresses a possible modification of the program such that l belongs to the well-founded model of the resulting program. These modifications are captured by the above sets.

For instance, $MissingFacts(c_j)$ is a set of facts that would be necessary to add to the program in order to justify l, while $NoFacts(c_j)$ is a set of facts that cannot be added in order to justify l. As a consequence, l will belong[21] to the well-founded model of any program resulting from adding any superset G of $MissingFacts(c_j)$ that does not contain any fact from $NoFacts(c_j)$ (assuming that $RemoveRules(c_j) = RemoveFacts(c_j) = \{\}$).

Example 43 (*Ex. 42 continued*)
Continuing with our running example, we have that *not s* does not belong to the well-founded model of P_8 and that $c = not(s) \wedge \neg not(r)$ is a why-not provenance justification of *not s*, i.e. it is a disjunct (42) of the why-not provenance information of *not s*. Then, we also have $MissingFacts(c) = \{r\}$ and $NoFacts(c) = \{s\}$. This expresses that *not s* would belong to the well-founded model of any program $P' = P_8 \cup G$ with G any set of facts that includes r but does not include s. □

Similarly, $KeepFacts(c_j)$ and $KeepRules(c_j)$ point out facts and rules that need to be kept in the program to justify the literal while $RemoveFacts(c_j)$ and $RemoveRules(c_j)$ state facts and rules that need to be removed from the program. Note that, if a conjunction c_j contains no negation, then it does not imply any change in the program and, thus, constitutes an *actual* justification for the *actual* value of the literal.

Example 44 (*Ex. 43 continued*)
As a further example, let $c' = r_3 \wedge not(r) \wedge r_1 \wedge not(q)$ be a why-not provenance justification of s (the conjunction corresponding to the disjunct (40) of the why-not provenance information of s). Informally, this conjunction expresses that "s is true because both r_1 and r_3 are in the program and facts r and q are not." Note that $KeepRules(c') = \{r_1, r_3\}$ and $NoFacts(c') = \{r, q\}$, indicating that s remains true as long as we keep these two rules and we add neither r nor q,

[21] This has been shown in (Damásio et al. 2013, Theorem 3).

even if we remove other rules or remove or add other facts. Note also that there is no negated annotation in c' and, thus, $RemoveFacts(c') = MissingFacts(c') = RemoveRules(c') = \{\}$. In other words, c' points out a that no modification is required to make s true and, thus, it is an actual justification for the truth of s. □

The following example illustrates how why-not provenance captures justifications of programs with even-length negative dependency cycles:

Example 45 (*Ex. 4 continued*)
Let us define the following labelling for program P_3:

$$r_1 : \ p \leftarrow not\ q \qquad\qquad r_2 : \ q \leftarrow not\ p$$

As we have seen, program P_3 has two answer sets, namely $M_3 = \{p\}$ and $M_4 = \{q\}$, and an empty well-founded model. The computation of the why-not provenance information goes as follows:

$$\begin{array}{ll}
\tilde{\Gamma}^1_{\mathfrak{P}_3}(\mathbf{0})(p) = \neg not(p) \vee r_1 & \tilde{\Gamma}^1_{\mathfrak{P}_3}(\mathbf{0})(not\ q) = not(q) \wedge \neg r_2 \\
\tilde{\Gamma}^2_{\mathfrak{P}_3}(\mathbf{0})(p) = \neg not(p) \vee r_1 \wedge not(q) \wedge \neg r_2 & \tilde{\Gamma}^2_{\mathfrak{P}_3}(\mathbf{0})(not\ q) = not(q) \wedge (\neg r_2 \vee \neg not(p) \vee r_1) \\
\tilde{\Gamma}^3_{\mathfrak{P}_3}(\mathbf{0})(p) = \neg not(p) \vee r_1 \wedge not(q) & \tilde{\Gamma}^3_{\mathfrak{P}_3}(\mathbf{0})(not\ q) = not(q) \wedge (\neg r_2 \vee \neg not(p)) \\
\tilde{\Gamma}^4_{\mathfrak{P}_3}(\mathbf{0})(p) = \neg not(p) \vee r_1 \wedge not(q) \wedge \neg r_2 & \tilde{\Gamma}^4_{\mathfrak{P}_3}(\mathbf{0})(not\ q) = not(q) \wedge (\neg r_2 \vee \neg not(p) \vee r_1)
\end{array}$$

$\tilde{\Gamma}^2_{\mathfrak{P}_3}(\mathbf{0})$ and $\tilde{\Gamma}^3_{\mathfrak{P}_3}(\mathbf{0})$ respectively are the least and greatest fixpoint of $\hat{\Gamma}^2_{\mathfrak{P}_3}$. The case for q and $not\ p$ are symmetric. Then, the why-not provenance information for p is as follows:

$$\begin{array}{lll}
Why_{P_3}(p) & = \neg not(p) & \vee\ r_1 \wedge not(q) \wedge \neg r_2 \\
Why_{P_3}(not\ p) & = not(p) \wedge \neg r_1 & \vee\ not(p) \wedge \neg not(q) \\
Why_{P_3}(\text{undef}\ p) & = not(p) \wedge not(q) \wedge r_1 \wedge r_2 &
\end{array}$$

Note that the only why-not provenance justification without negation $\neg$ occurs in $Why_{wP_3}(\text{undef}\ p)$, indicating that the actual truth value of p w.r.t. the well-founded model is undefined. The conjunction expresses that p is undefined in the well-founded model of P_3 because of the rules r_1 and r_2 and the absence of the facts p and q. □

3.4.1 Answer Set Why-not Provenance

The why-not justifications reviewed so far explain the truth value of literals with respect to the well-founded model. Why-not provenance information of a literal w.r.t. the answer set semantics is defined in terms of the why-not provenance of that literal being true in the well-founded model and the non-existence of undefined atoms in it. In other words, a literal is justified w.r.t. the answer set semantics by referring to modifications that make the literal *true* w.r.t. the complete well-founded model, which implies that it becomes the unique answer set.

Definition 30 (*Answer Set Provenance Information*)
Given a normal program P, the *answer set why-not provenance information* of a literal $l \in Lit_{ext}$ is defined as: $AnsWhy_P(l) \stackrel{\text{def}}{=} Why_P(l) \wedge \bigwedge_{b \in At_{ext}} \neg Why_P(\text{undef}\ b)$. □

Definition 31 (*Answer Set why-not Provenance Justification*)
Let P be a normal program, let $a \in At_{ext}$ be an extended atom and let $l \in \{a, not\ a, \text{undef}\ a\}$ such that $AndWhy_P(l) = c_1 \vee \ldots \vee c_n$ is the answer set why-not provenance information of l in minimal disjunctive normal form. Then, we say that each c_i is an *answer set why-not provenance justification* of l w.r.t. P. □

Note that Definition 30 characterises the major difference between this justification approach and the three previous ones: there is a unique provenance information of a literal with respect to the *whole program*, not with respect to each answer set.

In the case of Example 45 the answer set provenance (Definition 30) for p, q, $not\ p$ and $not\ q$ coincides with their respective provenance information (Definition 27). Note that none of the disjuncts in the why-not provenance information of p (resp. q) is without negation, which seems to point out that p is not true (can only be made true through modifications of the program). The reason is that, even though p (resp. q) is true in *some* answer set, it is not true in the well-founded model (it could also be due to the well-founded model not being complete). The answer set provenance thus points out modifications that would yield a complete well-founded model (and, thus, a *unique* answer set) in which p (resp. q) is true.

The following example illustrates that even if an atom is true in the *unique* answer set, the answer set provenance (as given by Definition 30) may still point out that modifications are needed to make the atom true. This is because a unique answer set may not be a complete well-founded model.

Example 46 (*Ex. 45 continued*)
Let P_{25} be the program

$$r_1 :\ p \leftarrow not\ q \qquad r_2 :\ q \leftarrow not\ p \qquad r_3 :\ s \leftarrow p \wedge not\ s$$

obtained by adding rule r_3 to program P_3. This program has a unique answer set $M_{21} = \{q\}$. Furthermore, adding rule r_3 to program P_3 does not change the why-not provenance information of p or q. The computation of the why-not provenance information for s goes as follows:

$$\begin{aligned}
&\tilde{\Gamma}^1_{\mathfrak{P}_{25}}(\mathbf{0})(s) = \neg not(s) \vee r_3 \wedge \neg not(p) \vee r_3 \wedge r_1 \\
&\tilde{\Gamma}^2_{\mathfrak{P}_{25}}(\mathbf{0})(s) = \neg not(s) \\
&\tilde{\Gamma}^3_{\mathfrak{P}_{25}}(\mathbf{0})(s) = \neg not(s) \vee r_3 \wedge \neg not(p) \vee r_3 \wedge r_1 \wedge not(q) \\
&\tilde{\Gamma}^4_{\mathfrak{P}_{25}}(\mathbf{0})(s) = \neg not(s)
\end{aligned}$$

and we obtain

$$\begin{aligned}
Why_{P_{25}}(s) &= \neg not(s) \\
Why_{P_{25}}(not\ s) &= not(s) \wedge \neg r_3 \vee not(s) \wedge not(p) \wedge \neg r_1 \vee not(s) \wedge not(p) \wedge \neg not(q) \\
Why_{P_{25}}(\text{undef}\ s) &= r_3 \wedge not(s) \wedge \neg not(p) \ \vee \ r_1 \wedge r_3 \wedge not(s) \wedge not(q)
\end{aligned}$$

That is, s is undefined in the well-founded model because of rules r_1 and r_3 and the absence of the facts s and q. It would also be undefined if we added the fact p while keeping the rule r_3 and the absence of s. Furthermore, $AnsWhy_{P_{25}}(\text{undef}\ p) = AnsWhy_{P_{25}}(\text{undef}\ q)$ and, thus, $\neg AnsWhy_{P_{25}}(\text{undef}\ p) \wedge \neg AnsWhy_{P_{25}}(\text{undef}\ q) \wedge$

$\neg AnsWhy_{P_{25}}(\text{undef } s) \;=\; \neg AnsWhy_{P_{25}}(\text{undef } p) \wedge \neg AnsWhy_{P_{25}}(\text{undef } s)$ which corresponds to

$$\neg(not(p) \wedge not(q) \wedge r_1 \wedge r_2) \;\wedge\; \neg(r_3 \wedge not(s) \wedge \neg not(p) \;\vee\; r_1 \wedge r_3 \wedge not(s) \wedge not(q))$$

We also have that

$$Why_{P_{25}}(q) = \neg not(q) \vee r_2 \wedge not(p) \wedge \neg r_1$$

This implies that the answer set provenance information for q is:

$$\begin{aligned} AnsWhy_{P_{25}}(q) \quad = \quad & \neg not(q) \wedge \neg r_3 \\ & \vee \neg not(q) \wedge \neg not(s) \\ & \vee \neg not(q) \wedge not(p) \\ & \vee \neg r_1 \wedge r_2 \wedge not(p) \end{aligned}$$

The disjuncts represent different modifications of the program leading to the existence of a complete well-founded model (and, thus, a unique answer set), in which q is true. □

Example 46 can be used to illustrate how the notion of assumption, as introduced in Section 3.1, can be applied to why-not provenance justifications. In particular, the disjunct $\neg r_1 \wedge r_2 \wedge not(p)$ in $AnsWhy_{P_{25}}(q)$ suggests removing all rules with p in the head (just r_1) and not adding the fact p to the program. This can be understood as "p needs to be assumed to be false" in a similar way as done in off-line or extended causal justifications. In order to make this informal reading about this last disjunct, we need to know that p is actually false in the answer set that we are considering, i.e. $M_{21} = \{q\}$, because $AnsWhy_{P_{25}}(p)$ contains a symmetric disjunct $\neg r_2 \wedge r_1 \wedge not(q)$ whose informal reading does not correspond to an assumption but to an actual modification. This is not a surprise because why-not provenance (as an unsimplified formula) can be computed in polynomial time, while deciding whether some atom is true in some answer set of some normal program is, in general, NP-complete. Hence, unless the polynomial hierarchy collapses, it is obvious that why-not provenance cannot contain information about whether some atom is true or false in some answer set. Note also that, though extended causal justifications (as an unsimplified causal term) can be computed in polynomial time, they are construed w.r.t. a program reduced w.r.t. the set of assumptions corresponding to this answer set. Hence, they assume the information of true atoms in an answer set as a given. The same approach used to define extended causal justifications w.r.t. an answer set could be applied to why-not provenance as well.

3.5 Other Justification Approaches

In this section we informally review two other approaches that deal with justifications in answer set programming, namely *justifications in rule-based answer set computation* (Béatrix et al. 2016) and the *formal theory of justifications* (Denecker and De Schreye 1993; Denecker et al. 2015). Despite sharing a similar purpose with previous approaches, the formal definition of Béatrix et al. (2016) heavily relies on

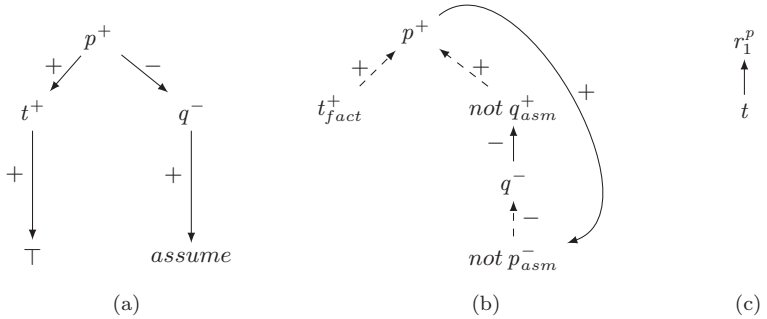


Fig. 24: Off-line, LABAS, and causal justifications of p w.r.t. $\{p, t\}$ and P_{26}.

the concept of `ASPeRiX` *computation* (Lefèvre et al. 2017) and is out of the scope of this survey. On the other hand, the purpose of the works by Denecker and De Schreye (1993) and Denecker et al. (2015) is to study different semantics of logic programming from the point of view of justifications rather than to provide explanations that are "intelligible and easily accessible" by humans, as required by the new GDPR.

3.5.1 Justifications in Rule-Based Answer Set Computation

Béatrix et al. (2016) study the notion of justification from a rule-based point of view of answer set computation, that is, under the assumption that the inherent non-determinism of answer sets is due to the guessing of the application or non-application of rules rather than the guessing of the truth value of literals. Another interesting point to mention is that justifications in this approach, called *reasons*, are sets of rules instead of graphs. The following example illustrates these two differences.

Example 47
Consider the following program P_{26}:

$$r_1 : \ p \leftarrow t \wedge not\ q \qquad r_2 : \ q \leftarrow s \qquad r_3 : \ s \leftarrow not\ p \qquad t : \ t$$

which has two answer sets: $M_{22} = \{p, t\}$ and $M_{23} = \{q, s, t\}$. The rule-based reason for the truth of the atom p with respect to the answer set $\{p, t\}$ of the program P_{26} is the set $\{r_1, t\}$. □

We may use Example 47 to highlight some similarities and differences with previously discussed justification approaches. It can be checked that the causal graph justification (Figure 24c) for p in this example has precisely vertices t and r_1^p, corresponding to the rule-based reason. Correspondences with the off-line justification, shown in Figure 24a, are also easy to see: the application of rule r_1 is represented by the two outgoing edges from p^+ to t^+ and to q^-, where assuming q^- to be

false ensures that r_1 is satisfied. Similarly, the answer set why-not provenance of p includes the disjunct $r_1 \wedge t \wedge not(q) \wedge \neg r_2$, where $not(q) \wedge \neg r_2$ can be understood to mean that q is assumed to be false. The LABAS justification, shown in Figure 24b, further explains that the falsity of q depends on the truth of p, thus also using rules r_2 and r_3 for the explanation. Interestingly, the answer set why-not provenance of p has another disjunct $r_1 \wedge t \wedge not(q) \wedge \neg r_3$, which also uses rule r_3 to justify p.

Note that the rule-based reason for the falsity of q w.r.t. M_{22} is a subset of the reason for p, namely $\{r_1\}$. This contrasts with off-line and causal justifications, in which q is assumed to be false, and LABAS justifications, in which q is explained in the same way as in the justification of p (flipping the justification in Figure 24b so that q is at the top coincides with the LABAS justification of q), i.e. in terms of r_3 (and implicitly r_2) as well as r_1 and t. The answer set why-not provenance of *not* q includes the disjunct $not(q) \wedge \neg r_2$ which, as mentioned before, can be understood as assuming that q is false.

3.5.2 Formal Theory of Justifications

Denecker and De Schreye (1993) and Denecker et al. (2015) present an abstract theory of justifications, suitable for describing the semantics of logics in knowledge representation and computational and mathematical logic. In this theory, each program induces a semantic structure called justification frame, which embodies the potential *reasons why* the program's conclusions are true. Interestingly, the authors show that differences in various semantics can be traced back to a single difference, namely the way in which justifications with infinite branches are handled. For instance, p is justified w.r.t. program $P_3 = \{p \leftarrow not\ q,\ q \leftarrow not\ p\}$ by the following infinite branch:

$$p \rightarrow not\ q \rightarrow p \rightarrow not\ q \rightarrow \ldots$$

This is evaluated as undefined under the well-founded semantics (infinite branches altering positive and negative literals are always evaluated as undefined under the well-founded semantics). In contrast, it takes the value of *not* q under the answer set semantics (under the answer set semantics infinite branches are evaluated to the truth value of the first positive (resp. negative) literal whose predecessors are all negative (resp. positive) literals), which is true w.r.t. answer set $\{p\}$, but false w.r.t. $\{q\}$.

Contrary to the other approaches surveyed here, this work focuses on exploiting justifications as mathematical objects to understand different semantics (and propose new ones) rather than as a means to answer in a compact way, *why* a conclusion has been reached. The *complete justifications* defined in the formal theory of justifications are thus structures that contain information for all literals, even those that are not directly related to the derivation of a literal in question. As an explanation in the sense of the new GDPR, complete justifications are thus not suitable as they are clearly not "concise" and likely not "intelligible and easily accessible", as they comprise information unnecessary for a user's understanding. Studying how concise and intelligible justifications can be obtained from this struc-

Table 1: Comparison of explanation approaches for consistent logic programs under the answer set semantics.

justification approach	**type of logic program**	**explanation in terms of**	**derivation steps included**	**explains**
off-line justifications	normal LP	literal dependency	all	one literal (not) in answer set
LABAS justifications	normal extended LP	literal dependency	some	one literal (not) in answer set
causal justifications	extended LP with nested expressions in the body	rule-literal dependency	all	one literal in answer set
extended causal justifications	normal extended LP	rule-literal dependency	all	one literal (not) in answer set
why-not provenance	normal LP	rule dependency	all	one literal (not) in the complete well-founded model[22]
rule-based justifications	normal LP	rule dependency	all	one literal (not) in answer set
formal theory of justifications	normal LP	literal dependency	all	whole answer set

tures is an interesting open topic as it would be directly applicable to several logics and knowledge representation formalisms like argumentation.

3.6 Summary and Discussion

In Sections 3.1 to 3.5 we have surveyed the most prominent approaches for justifying the solutions to consistent logic programs under the answer set semantics.

Table 2: Comparison of explanation approaches for consistent logic programs under the answer set semantics (continued).

justification approach	**computation uses other models**	**explanation of negative literals**	**infinite explanations**	**infinitely many explanations**
off-line justifications	well-founded model	assumed or further explained	no, if the program is finite	no, if the program is finite
LABAS justifications	no	further explained	yes	yes
causal justifications	no	assumed	no	no, if the program is finite
extended causal justifications	well-founded model	assumed or further explained	no, if the program is finite	no, if the program is finite
why-not provenance	(do not need answer sets)	further explained	no, if the program is finite	no, if the program is finite
rule-based justifications	no	further explained	no, if the program is finite	no, if the program is finite
formal theory	no	further explained	yes	no, if the program is finite

Note that throughout these sections, by referencing an answer set to justify, we implicitly assumed that logic programs are consistent. While explaining the justification approaches, we already pointed out differences and similarities between these approaches. Some of these are reiterated in Tables 1 and 2, which provide a comparative overview of various features of the justification approaches.

Table 1 illustrates for which types of logic programs the different justification approaches are defined, in which terms they explain answer sets (i.e. dependencies between rules or literals), whether all parts of a literal's derivation are included in a justification, and what precisely is being explained, i.e. a literal in an answer set, a literal not contained in an answer set, or a whole answer set. Table 2 complements this comparison, by showing whether the justification approaches make use of logic

[22] The why-not provenance corresponding to each answer set can then be obtained by forcing

programming models other than the answer set in question when constructing a justification, whether negative literals occur in justifications and, if so, how their truth value is explained, whether justifications may be infinite, and whether there may be infinitely many justifications.

In the following, we discuss some of the differences between the justification approaches in more detail and highlight some of their advantages and disadvantages. In particular, we focus on the philosophical ideas underpinning the different justifications approaches (Section 3.6.1), the problem of having exponentially many justifications (Section 3.6.2), how different approaches deal with negation-as-failure (Section 3.6.3), and the issues faced when dealing with large logic programs (Section 3.6.4).

3.6.1 Explanatory Elements

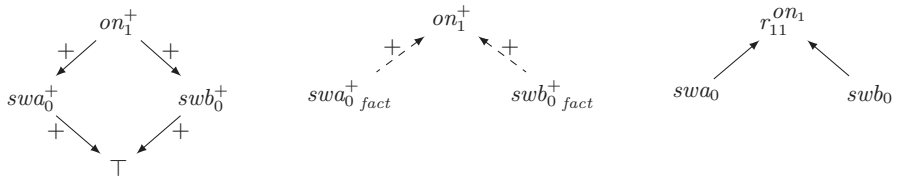
Due to the usage of different definitions of answer set, the different justifications embody distinct ideas. For instance, the intuition of off-line justifications (Section 3.1) can be traced back to Prolog tabled justifications (Roychoudhury et al. 2000), LABAS justifications (Section 3.2) have an argumentative flavour and are based on a correspondence between logic programs and their translation into argumentation frameworks (Schulz and Toni 2015; Schulz and Toni 2016), while causal justifications (Section 3.3) rely on a causal interpretation of rules and the idea of causal chain (Lewis 1973). Despite their differences, these three approaches share the fact that they explain why a literal belongs to some answer set using a "concise" graph structure (in the sense that these graphs do not contain information not related to the literal in question).

The why-not provenance (Section 3.4), which is based on the concept of provenance inherited from the database literature (Green et al. 2007), shares with these approaches the idea of building concise justifications for each literal. However, why-not provenance justifications are set-based (instead of graph-based) and are built without referring to a specific answer set, so justifications are answer set independent. The justifications for a particular answer set can be obtained by "forcing" the appropriate assumptions as done in extended causal justifications.

A similar point of view is also shared by rule-based justifications (Section 3.5.1), which are based on the concept of an `ASPeRiX` *computation* (Lefèvre et al. 2017). Conceptually, the major difference between this and the previously mentioned approaches lies in what is considered as assumptions, i.e. as elements that do not need to be further justified: rules in the case of rule-based justifications and literals in the case of the other approaches.

Finally, the formal theory of justifications (Section 3.5.2) aims to explain the differences between different logic programming semantics by identifying how their conclusions are justified. Contrary to the other approaches, it provides justifications for a whole answer set instead of concise justifications for each literal. This is similar

the atoms not in the answer set as assumptions, similarly as done done for extended causal justifications.



(a) (b) (c)

Fig. 25: Off-line, LABAS, and causal justifications of the truth of on_1.

to debugging systems (which we will overview in Section 4), which explain why a whole set of literals is not an answer set, rather than explaining a specific literal.

3.6.2 The Problem of Exponentially Many Justifications

As mentioned in the introduction, a key point for a human-understandable answer to the question of *why* some conclusion is reached is its conciseness. Most justification approaches reviewed here have tackled this issue and provide justifications that only contain information related to the literal in question. However, a second issue related to conciseness is how many justifications there are. In this section, we show that the number of justifications is in general exponential w.r.t. the size of the program. Let us start by continuing here the discussion about the light bulb scenario introduced in Example 36 (page 38).

Example 48 (*Ex. 36 continued*)
Recall that the program P_{22} representing this scenario consists of the following rules:

$$r_{1t+1}: \ on_{t+1} \leftarrow swa_t \wedge swb_t \qquad i_{1t+1}: \ on_{t+1} \leftarrow on_t \wedge not\ off_{t+1}$$
$$r_{2t+1}: \ off_{t+1} \leftarrow swc_t \wedge swd_t \qquad i_{2t+1}: \ off_{t+1} \leftarrow off_t \wedge not\ on_{t+1}$$

plus the integrity constraint $\leftarrow on_t \wedge off_t$ for $t \geq 0$ and the facts off_0, swa_0 and swb_0. Recall also that this program has a complete well-founded model and, thus, a unique answer set, in which on_t holds for every time $t > 0$. Figures 25a, 25b and 25c respectively depict the off-line, the LABAS and the causal justification explaining why the light is *on* in situation 1. We also have that the answer set why-not provenance of on_1 corresponds to the following propositional formula:

$$AnsWhy_{P_{22}}(on_1) = \neg not(on_1) \ \vee \ \neg not(on_0) \wedge not(off_1) \wedge i_{12} \ \vee \ swa_0 \wedge swb_0 \wedge r_{11}$$

where $swa_0 \wedge swb_0 \wedge r_{11}$ points out that on_1 is true w.r.t. the unique answer set (which, here, coincides with the complete well-founded model) because of facts swa_0 and swb_0 and rule r_{11}. It is easy to see the similarity with Figures 25a, 25b and 25c, in particular that $swa_0 \wedge swb_0 \wedge r_{11}$ is precisely the conjunction of the three vertices in these justifications. Informally, these justification can be read as "because both switches a and b have been pushed in situation 0".

Let us now consider the justifications for the atom on_2, which is true w.r.t. the

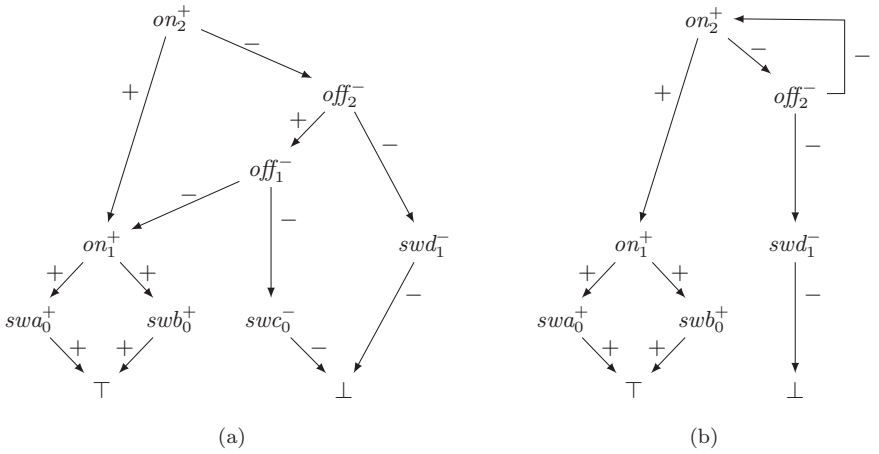


Fig. 26: Off-line justifications of on_2 w.r.t. the unique answer set of Example 48.

unique answer set. Figure 26 depicts two of the six possible off-line justifications for on_2. Furthermore, by replacing swd_1^- with swc_1^- in Figures 26a and 26b, we obtain another two off-line justifications. Similarly, by replacing swc_0^- with swd_0^- in Figure 26a, we obtain another off-line justification and, by replacing both swc_0^- and swd_1^- respectively with swd_0^- and swc_1^-, we obtain the sixth one. Figure 27

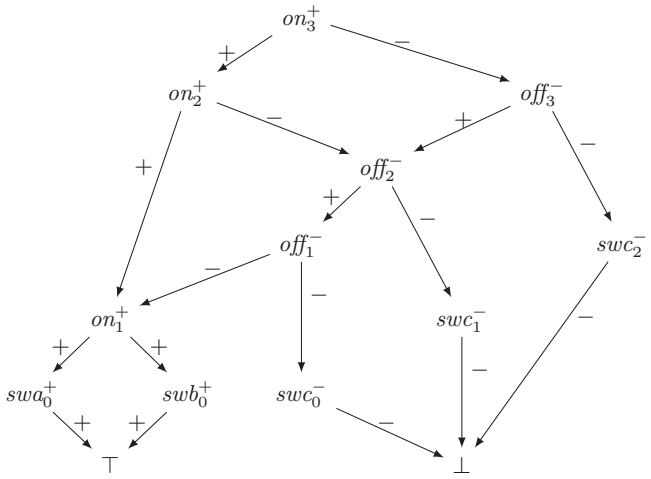


Fig. 27: Off-line justification of on_3 w.r.t. the unique answer set of Example 48.

depicts one of the off-line justifications of on_3^+ and, by replacing any subset of $\{swc_1^-, swc_2^-, swc_3^-\}$ by its corresponding subset of $\{swd_1^-, swd_2^-, swd_3^-\}$, we obtain another 7 alternative off-line justifications. That is, the number of off-line justifications grows exponentially with the number of situations in which nothing happens.

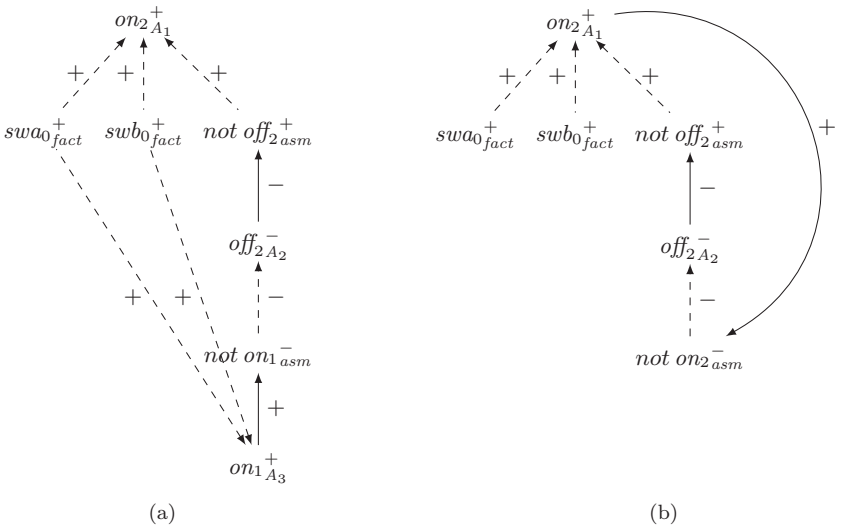


Fig. 28: LABAS justifications of on_2 w.r.t. the unique answer set of Example 48.

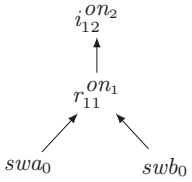


Fig. 29: The unique causal justification of on_2 w.r.t. the unique answer set of Example 48.

Similarly, the number of why-not justifications[23] (i.e. disjuncts in the answer set provenance information) of on_t grows exponentially, because the conjunction of all atoms in an off-line justification plus the rules used to derive those atoms form a why-not justification (Damásio et al. 2013, Theorem 4). The number of LABAS justifications also grows exponentially. There are two LABAS justifications for on_2, displayed in Figures 28a and 28b. The reason for the exponential explosion is that on_t can be justified through any on_i with $i < t$. On the other hand, as explained in Section 3.3 (page 38) (extended) causal justifications are somehow preserved by inertia in the sense that, at any situation $t + 1$, if nothing happens, then the justification of on_{t+1} can be obtained by adding to the justification of on_t an edge from $i_{1t}^{on_t}$ to $i_{1t+1}^{on_{t+1}}$. For instance, Figure 29 shows the unique (extended) causal justification of on_2. □

[23] Why-not information can be obtained in polynomial time and size w.r.t. the program. However, rewriting it as a disjunction of minimal conjuncts may require exponential space.

Despite the fact that understanding negation-as-failure as a default allows to exponentially reduce the number of causal justifications on some knowledge representation scenarios as illustrated by the above example, there still exist logic programs that produce an exponential number of causal justifications:

Example 49
Consider the following logic program adapted from (Cabalar et al. 2014):

$$
\begin{array}{lll}
p_1 \leftarrow q_1 & p_i \leftarrow p_{i-1} \wedge q_i & \text{for } i \in \{2, \ldots, n\} \\
p_1 \leftarrow u_1 & p_i \leftarrow p_{i-1} \wedge u_i & \text{for } i \in \{2, \ldots, n\} \\
 & q_i & \text{for } i \in \{1, \ldots, n\} \\
 & u_i & \text{for } i \in \{1, \ldots, n\}
\end{array}
$$

whose unique answer set is $\{p_1, q_1, u_1 \ldots, p_n, q_n, u_n\}$. Note that p_1 can be justified using the facts q_1 or u_1; the atom p_2 can be justified using the sets of facts $\{q_1, q_2\}$, $\{q_1, u_2\}$, $\{u_1, q_2\}$ or $\{u_1, u_2\}$; and so on. It is easy to see that atom p_n can be justified using 2^n different sets of facts and, thus, that the number of justifications grows exponentially with respect to the size of the program. □

Although this logic program has no deeper knowledge representation meaning, it points out a potential problem regarding the human-readability of the answers provided by current justification approaches. The issue of an exponential number of justifications illustrated by Example 49 holds for any justification approach that records minimal sets of facts used to derive the justified atom, in particular, all justification approaches reviewed here. This does not mean that other kinds of polynomial justifications can be used. For instance, for causal justifications or why-not provenance, a non-simplified formula could be returned and, if we consider such a formula as the justification, then it would be polynomial. In our running example, we would have that p_n is justified by the causal term $(q_1 + u_1) * (q_2 + u_2) * \ldots * (q_n + u_n)$ or the why-not provenance formula $(q_1 \vee u_1 \vee \neg not(p_1)) \wedge (q_2 \vee u_2 \vee \neg not(p_2)) \wedge \ldots \wedge (q_n \vee u_n \vee \neg not(p_n))$. On the other hand, these non-simplified expressions are not minimal and, thus, they do not adhere to the desired conciseness criterion for justifications. Another alternative is to provide simplified justifications, but selecting only some of them in case a some imposed preferences (Cabalar et al. 2014). For instance, approaches in databases (Specht 1993) and Prolog (Roychoudhury et al. 2000) implicitly impose such preferences by selecting only the first negative literal of a rule that fails as its unique justification.

3.6.3 Interpreting Negation-as-Failure

Related to the above exponentiality problem is the way in which different approaches interpret negative literals. The definition of answer sets (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) is inherently non-deterministic: a candidate set is (non-deterministically) selected and then checked against the program to see whether it is the minimal model of the reduct with respect to this candidate. For normal logic programs, the checking part can be done deterministically

in polynomial time, for instance, by iterating the well-known direct consequences operator introduced by van Emden and Kowalski (1976); but the non-determinism is still present in the selection of the candidate. This non-determinism is handled by most justification approaches by considering some part of the justification as assumptions: negative literals in the case of off-line, LABAS and causal justifications; and rules in the case of rule-based justifications (formal theory of justifications takes a different approach, representing this by infinite branches). Regarding the approaches that use negative literals as assumptions, a remarkable difference is how they do or do not justify those negative literals. As the two extremes we have LABAS and causal justifications: the former justifies all negative literals (introducing cycles in the justifications when even-length negative dependency loops are present in the program), while the latter treats all negative literals as assumptions, or rather defaults, that need no further explanation. In the middle, we have off-line and extended causal justifications, which further explain some negative literals, while treating others as assumptions (when the set of assumptions is minimised, these approaches justify all negative literals that can be explained without introducing cycles in the justifications).

We have seen that treating negative literals as assumptions may help to (exponentially) reduce the number of justifications of some knowledge representation problems in which negation is used to express defaults. Let us now illustrate the opposite case, with the following example from (Schulz and Toni 2016), where justifications for negative literals are as important as those for positive literals:

Example 50
The logic program P_{27} in Figure 30 represents the decision support system used by an ophthalmologist. It encodes some general world knowledge as well as an ophthalmologist's specialist knowledge about the possible treatments of shortsightedness. P_{27} also captures the additional information that the ophthalmologist has about his shortsighted patient Peter. Program P_{27} has a unique answer set

$$M_{24} = \{\ \mathit{shortSighted}, \mathit{afraidToTouchEyes}, \mathit{student}, \mathit{likesSports}, \mathit{tightOnMoney}, \mathit{correctiveLens}, \mathit{caresAboutPracticality}, \mathit{intraocularLens}\ \}$$

Focusing on the positive dependencies on facts and not considering dependencies on negative literals, we can only say that Peter has been recommend to use an *intraocularLens* because he is *shortSighted*. However, this reasoning could also lead to the recommendation of other treatments that have the same positive dependencies: *glasses*, *contactLens* or *laserSurgey*. Negative dependencies, on the other hand, tell us that *intraocularLens* was recommended because all the other alternatives were discarded for different reasons: *glasses* because Peter *likesSports*, *contactLens* because he is *afraidToTouchEyes* and *laserSurgey* because he is a *student* without *richParents*. □

The informal reading shown in the above example can be extracted from off-line, LABAS, extended causal, why-not provenace and rule-based justifications, but not from (non-extended) causal justifications. A general approach to justifications should be able to effectively combine both interpretations of negation-as-failure,

$$
\begin{aligned}
\mathit{tightOnMoney} &\leftarrow \mathit{student} \wedge \mathit{not\ richParents} \\
\mathit{caresAboutPracticality} &\leftarrow \mathit{likesSports} \\
\mathit{correctiveLens} &\leftarrow \mathit{shortSighted} \wedge \mathit{not\ laserSurgery} \\
\mathit{laserSurgery} &\leftarrow \mathit{shortSighted} \wedge \mathit{not\ tightOnMoney} \wedge \mathit{not\ correctiveLens} \\
\mathit{glasses} &\leftarrow \mathit{correctiveLens} \wedge \mathit{not\ caresAboutPracticality} \wedge \\
&\quad\ \mathit{not\ contactLens} \\
\mathit{contactLens} &\leftarrow \mathit{correctiveLens} \wedge \mathit{not\ afraidToTouchEyes} \wedge \\
&\quad\ \mathit{not\ longSighted} \wedge \mathit{not\ glasses} \\
\mathit{intraocularLens} &\leftarrow \mathit{correctiveLens} \wedge \mathit{not\ glasses} \wedge \mathit{not\ contactLens} \\
\mathit{shortSighted} &\leftarrow \\
\mathit{afraidToTouchEyes} &\leftarrow \\
\mathit{student} &\leftarrow \\
\mathit{likesSports} &\leftarrow
\end{aligned}
$$

Fig. 30: Program P_{27} from Example 50.

something which to the best of our knowledge has not been studied in the literature yet.

3.6.4 Large Programs and Application-Oriented Considerations

Our comparison so far has concentrated on the theoretical, or even philosophical, nature of justification approaches. Another important, and distinguishing, aspect of justification approaches is their applicability when solving real-world problems. In such situations, various challenges arise.

Firstly, representing a real-world problem may result in a large logic program, where literals may have long derivations, i.e. their truth value depends on a large number of rules. It is then not clear, which information a justification should comprise in order to be, on the one hand, succinct enough for humans to understand, but, on the other hand, complete enough to provide all important information. For example, justification approaches where all derivation steps are included in the justification, that is all approaches other than LABAS justifications, may struggle with the succinctness when explaining a large logic program, as explanations grow with longer derivations. In contrast, LABAS justifications are independent of the derivation length. However, a large logic program may also comprise more dependencies on negative literals, thus increasing the size of LABAS justifications. More generally, it is an open problem how to effectively deal with the growing size (as well as the previously mentioned exponential number) of justifications.

In order to use justifications in real-world problems, they need to be automatically constructed. Currently, only LABAS, causal and why-not provenance justifications

have been implemented in working prototypes.[24] A related issue is which type of logic programs can be explained. The only approach able to handle non-normal logic programs, i.e. logic programs with disjunctive heads, is the causal justification approach, which can also deal with nested expressions in the body.[25] Furthermore, in practice logic programs are rarely normal and often use additional language constructs, such as weight constraints, aggregates, and choice rules, which extend the syntax and/or semantics of logic programs under the answer set semantics. Choice rules are handled by off-line justifications and in a limited way by causal justifications (Cabalar and Fandinno 2016). Note that explanations of additional language constructs have not been investigated so far.

As a last challenge, we mention variables. Even though the theory of most justification approaches can easily be applied to programs with variables by considering the complete grounding of the program, it is questionable if this method yields meaningful justifications in practice. The difficulty of handling variables in explanations of inconsistent programs is a further indication that justifications involving variables are non-trivial, and therefore an interesting consideration for future work.

4 Debugging of Inconsistent Logic Programs

In this section, we review the most prominent approaches for explaining *inconsistent* logic programs. i.e. logic programs that have no answer set. Note that various approaches discussed in this section are not only applicable to inconsistent logic programs, but also to consistent ones. More specifically, they can also be used to explain why a set of atoms of a *consistent* logic program is not an answer set, or even why a set of atoms *is* an answer set, and are thus closely related to the previously reviewed justification approaches.

Finding errors that lead to a logic program being inconsistent is often referred to as *debugging*. Errors can be roughly divided into *syntactic* and *semantic* ones.[26] The first category, comprising for example misspelled literals and wrong rule layout, are handled by most IDEs (Integrated Development Environments) for ASP such as `SeaLion` (Busoniu et al. 2013), ASPIDE (Febbraro et al. 2011), and APE (Sureshkumar et al. 2007).

Semantic errors are more difficult to identify due to the inherent declarative nature of the answer set semantics. In procedural programming languages, the cause of wrong program behaviour can be found by investigating the program procedure step-by-step. This cannot be straightforwardly done for logic programs, as answer sets are computed in a 'guess and check' fashion rather than procedurally. Various approaches tackle this problem by searching for known error classes for inconsistent logic programs, for example unfounded loops, unsupported atoms, and unsatisfied

[24] There also used to be an implementation of off-line justifications (El-Khatib et al. 2005), but this is not available anymore.

[25] In this survey, we have limited ourselves to normal extended logic programs. For a the definition of causal justifications for logic programs with nested expressions in the body, we refer to (Fandinno 2016b).

[26] Note that we here use these terms differently than e.g. Syrjänen (2006).

rules. We review these approaches in Sections 4.1 to 4.3. Another approach makes use of the unsatisfiable core feature of the ASP solver WASP, which we review in Section 4.4, and Section 4.5 outlines an approach for finding semantic errors that indeed applies a step-by-step procedure. Finally, Section 4.6 concludes the section with a discussion about similarities and differences between these debugging approaches. Throughout this section, we will use the term 'debugging' to refer to the task of finding and explaining *semantic* errors in logic programs.

4.1 The `spock` *System – Debugging with a Meta Program*

The `spock` system explains why a *potential* answer set, i.e. some set of atoms, is not an answer set of a the given program P. This is achieved by transforming P into a *meta (logic) program*, expressing, for example, conditions for the applicability of rules in P. Each answer set of this meta program contains the atoms of a potential answer set of P along with special atoms indicating reasons why this potential answer set is not an actual answer set of P. Thus, `spock` uses answer sets of a meta logic program for explaining the inconsistency of a given logic program.

The `spock` system is a command line tool[27] usable with either the `DLV` (Leone et al. 2006) or `Smodels` (Syrjänen and Niemelä 2001) ASP solver.[28] It implements two different approaches to transform P into a meta-program, where the second (Gebser et al. 2008) was developed as a successor of the first (Brain et al. 2007a). Both transformations distinguish three types of reasons for explaining why a set of atoms is not an answer set. These reasons are different ways of violating the definition of answer sets as given by Lin and Zhao (2004) and extended by Lee (2005). Note that this definition of answer sets is equivalent to the one given in Section 2.

Definition 32 (*Answer Set*)
A set of atoms $M \subseteq At$ is an answer set of a program P iff

1. each rule $r \in P$ is *satisfied* by M, i.e.
   - $head(r) \cap M \neq \{\}$ if r is applicable;
2. each atom $a \in M$ is *supported* w.r.t. M, i.e.
   - there exists $r \in P$ such that r is applicable w.r.t. M and $head(r) \cap M = \{a\}$;
3. each (positive dependency) loop $L \subseteq M$ is *founded* w.r.t. M, where
   - L is a loop iff for all $a \in L$ there is a chain of rules $r_1, \ldots, r_n \in P$ $(n \geq 1)$ such that $a \in head(r_1) \cap body^+(r_n)$, and if $n > 1$ then it holds for all r_i $(1 \leq i < n)$ that $\exists b_i \in body^+(r_i) \cap head(r_{i+1})$ with $b_i \in L$, and
   - L is founded w.r.t. M iff there exists $r \in P$ such that r is applicable and satisfied w.r.t. M, $head(r) \cap M \subseteq L$, and $body^+(r) \cap L = \{\}$. □

[27] `http://www.kr.tuwien.ac.at/research/systems/debug/index.html`
[28] `Smodels` is not maintained anymore and may thus not work on new systems. However, `spock` should work fine on most systems using `DLV`.

The third condition defines a loop as a set of atoms that positively depend on themselves, possibly via positive dependencies on other atoms in the loop. Such a positive dependency loop is founded w.r.t. M if there exists an applicable and satisfied rule that allows to derive some loop atoms without using other atoms in this loop. An atom contained in an unfounded loop is said to be *unfounded*.

Both transformation approaches of `spock` generate reasons why a set of atoms M is not an answer set in terms of violations of the three conditions in Definition 32. These reasons are:[29]

1. a rule $r \in P$ is not satisfied,
2. an atom $a \in M$ is not supported,
3. there exists an unfounded loop in M.

In the following, we illustrate how the two transformation approaches generate these three reasons and point out some differences between the approaches.

4.1.1 Transformation 1

The first transformation approach (Brain et al. 2007b; Brain et al. 2007a), defined for *normal* logic programs, can be used to explain

1. *why* a set of atoms *is* an answer set, by referring to the applicability and non-applicability of rules, and
2. *why* a set of atoms *is not* an answer set, by referring to the violation (i.e. non-satisfaction) of rules, the unsupportedness of atoms, or the unfoundedness of atoms.

To achieve the first, each rule $r : h \leftarrow b_1 \wedge \ldots \wedge b_n \wedge not\ c_1 \wedge \ldots not\ c_m$ of a normal program P is transformed into two new rules[30]

$$\mathsf{applicable}(r) \leftarrow b_1 \wedge \ldots \wedge b_n \wedge not\ c_1 \wedge \ldots not\ c_m \tag{45}$$

$$h \leftarrow \mathsf{applicable}(r) \tag{46}$$

They respectively express that r is applicable if the body of r is true and that the head of r can be deduced if r is applicable. Similarly, rules expressing conditions under which rule r is 'blocked' are added, namely if one of its positive body literals b or negative body literals $not\ c$ are false ($c^* \notin At$ is a new atom).

$$\mathsf{blocked}(r) \leftarrow not\ b \tag{47}$$

$$\mathsf{blocked}(r) \leftarrow not\ c^* \tag{48}$$

$$c^* \leftarrow not\ c \tag{49}$$

These transformed rules are added for each rule in the given program and each of its body literals.

[29] Lloyd (1987) discusses a similar idea for diagnosing errors in Prolog programs in terms of incorrect rules (analogous to unsatisfied rules) and uncovered atoms (analogous to unsupported atoms).

[30] The transformed rules as originally defined also have body literals $\mathsf{ok}(r)$ and $\mathsf{ko}(r)$ for fine-tuning the debugging process, which we omit as they do not play a role at this point.

The transformation given by rules (45)-(49) is called *kernel transformation* of P and denoted $\mathcal{T}_k[P]$. For a consistent program P, the answer sets of $\mathcal{T}_k[P]$ coincide with the answer sets of P, but additionally contain the new *tagging-atoms* $\mathsf{applicable}(r)$ and $\mathsf{blocked}(r)$ (Brain et al. 2007a). This 'explains' why a set of atoms is an answer set in the sense that it gives an insight into the rules that were used to derive the answer set.

Example 51 (*Ex. 36 continued, page 38*)
The rules of the logic program from Example 36 can be grounded for the first time step as follows, obtaining the logic program P_{28}:

$$\begin{aligned}
r_1 &: \ on_1 \ \leftarrow \ swa_0 \wedge swb_0 \\
r_2 &: \ off_1 \ \leftarrow \ swc_0 \wedge swd_0 \\
r_3 &: \ on_1 \ \leftarrow \ on_0 \wedge not\ off_1 \\
r_4 &: \ off_1 \ \leftarrow \ off_0 \wedge not\ on_1 \\
r_5 &: \ off_0 \ \leftarrow \\
r_6 &: \ swa_0 \ \leftarrow \\
r_7 &: \ swb_0 \ \leftarrow
\end{aligned}$$

The only answer set of P_{28} is $\{swa_0, swb_0, off_0, on_1\}$. In comparison, the only answer set of $\mathcal{T}_k[P_{28}]$ is $\{swa_0, swb_0, off_0, on_1, \mathsf{applicable}(r_1), \mathsf{applicable}(r_5), \mathsf{applicable}(r_6), \mathsf{applicable}(r_7), \mathsf{blocked}(r_2), \mathsf{blocked}(r_3), \mathsf{blocked}(r_4)\}$, pointing out that this answer set was obtained due to the applicability of rules r_1, r_5, r_6, and r_7, whereas the applicability of the other rules was blocked. □

For explaining the inconsistency of a logic program, three additional *extrapolation transformations* are performed (rules (50)-(55)), denoted $\mathcal{T}_{ex}[P]$. They allow to generate potential answer sets, i.e. sets of atoms, that violate Definition 32 and thus provide an explanation of the inconsistency. To generate potential answer sets *choice*-rules are used, which allow to choose whether or not the head of this rule should be true if the rule is applicable. These rules have the form $\{head(r)\} \leftarrow body(r)$ and are shorthand notation for

$$\begin{aligned}
head(r) \ &\leftarrow \ body(r) \wedge not\ x \\
x \ &\leftarrow \ not\ head(r)
\end{aligned}$$

where $x \notin At$ is a new atom.

Concerning the first inconsistency reason – the violation of rules – a new *abnormality* tagging-atom $\mathsf{unsatisfied}(r)$ is introduced and used to transform each rule r, where $head(r) = h$.[31]

$$\{h\} \ \leftarrow \ \mathsf{applicable}(r) \tag{50}$$

$$\mathsf{unsatisfied}(r) \ \leftarrow \ \mathsf{applicable}(r) \wedge not\ h \tag{51}$$

[31] We use the more intuitive naming $\mathsf{unsatisfied}(r)$ instead of the original $\mathsf{ab}_p(r)$ (Brain et al. 2007a) (similarly for the tagging-atoms described in the rest of this section).

When used for explaining inconsistent programs, rule (50) substitutes rule (46) from the kernel transformation. This allows to exclude h from an answer set, even if r is applicable. This choice rule allows to generate potential answer sets and rule (51) derives a respective reason why they may not be actual answer sets. In particular, this is the case if a rule is applicable w.r.t. a potential answer set but it head is not contained in this set.

The second extrapolation transformation is concerned with the supportedness of atoms. It introduces a new abnormality tagging-atom $\mathsf{unsupported}(a)$ for each $a \in At$, used in a transformation as follows:

$$\{a\} \leftarrow \mathsf{blocked}(r_1) \wedge \ldots \wedge \mathsf{blocked}(r_k) \tag{52}$$

$$\mathsf{unsupported}(a) \leftarrow a, \mathsf{blocked}(r_1) \wedge \ldots \wedge \mathsf{blocked}(r_k) \tag{53}$$

where $r_1, \ldots, r_k$ are all the rules with head a. Similarly to the first extrapolation transformation, rule (52) allows to choose if a is or is not included in a potential answer set being explained. Rule (53) derives $\mathsf{unsupported}(a)$ whenever a is in the answer set without any rule to support it.

The third extrapolation transformation deals with unfounded atoms. A new abnormality tagging-atom $\mathsf{unfounded}(a)$ is introduced for each atom $a \in At$ and used as follows:

$$\{\mathsf{unfounded}(a)\} \leftarrow \mathit{not}\ \mathsf{unsupported}(a) \tag{54}$$

$$a \leftarrow \mathsf{unfounded}(a) \tag{55}$$

This transformation gives a choice to include or exclude the abnormality atom $\mathsf{unfounded}(a)$, given that there is no other reason for $a$to be causing the inconsistency, namely being unsupported. This is different from the previous transformations, where abnormality atoms are only derived if there is an actual violation of a condition in Definition 32. Here, the abnormality atom may be derived even if the third condition in Definition 32 is not violated. This means that unfounded loops cannot be identified with certainty.

Example 52
Consider the following inconsistent logic program P_{29}:

$$r_1 : \ a \leftarrow \mathit{not}\ b$$
$$r_2 : \ b \leftarrow \mathit{not}\ b$$

The answer sets of $\mathcal{T}_k[P_{29}] \cup \mathcal{T}_{ex}[P_{29}]$ (where rule (46) is not included since derivability of the head is expressed through rule (50) as previously explained) indicate potential answer sets and explain why these potential answers sets are not *actual* answer sets by pointing out violations concerning the definition of answer sets.

- $M_{25} = \{a, b, \mathsf{unsupported}(a), \mathsf{unsupported}(b), \mathsf{blocked}(r_1), \mathsf{blocked}(r_2)\}$
- $M_{26} = \{b, \mathsf{unsupported}(b), \mathsf{blocked}(r_1), \mathsf{blocked}(r_2)\}$
- $M_{27} = \{a, \mathsf{unfounded}(a), \mathsf{unsatisfied}(r_2), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2)\}$
- $M_{28} = \{a, \mathsf{unsatisfied}(r_2), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2)\}$
- $M_{29} = \{\mathsf{unsatisfied}(r_1), \mathsf{unsatisfied}(r_2), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2)\}$

M_{25} expresses that $\{a, b\}$ is not an answer set because neither of the two atoms are supported by an applicable rule. This is because both r_1 and r_2 are blocked w.r.t. $\{a, b\}$. In contrast M_{29} explains that w.r.t. $\{\}$ both r_1 and r_2 are applicable, but the head of neither rule is included in $\{\}$. M_{27} illustrates the guessing of unfounded atoms. It states that $\{a\}$ is not an answer set because a may be unfounded and because r_2 is violated. Note that this guess is redundant, since answer set M_{28} explains $\{a\}$ by only referring to the violation of r_2. In fact, a is not unfounded here, as it is not part of an unfounded loop w.r.t. $\{a\}$ (it is not part of a loop at all). □

As shown by Example 52, there may be many explanations for the inconsistency of a logic program and some of them may be redundant. It is thus advisable to only consider explanations with a minimal number of abnormality tagging-atoms. This also ensures that $\mathsf{unfounded}(a)$ only occurs if a is indeed unfounded (Brain et al. 2007a). In Example 52, minimisation narrows the explanations down to sets M_{26} and M_{28}.

Example 53
Let P_{30} be the logic program P_{29} with the two additional rules:

$$r_3: \ a \leftarrow b$$
$$r_4: \ b \leftarrow a$$

These rules induce an unfounded loop w.r.t. the set $\{a, b\}$. $\mathcal{T}_k[P_{30}] \cup \mathcal{T}_{ex}[P_{30}]$ has three answer sets explaining why $\{a, b\}$ is not an answer set: one in terms of a being an unfounded atom (comprising $\mathsf{unfounded}(a)$), one in terms of b being an unfounded atom (comprising $\mathsf{unfounded}(b)$), and one in terms of both atoms being unfounded (comprising both $\mathsf{unfounded}(a)$ and $\mathsf{unfounded}(b)$). Similarly to Example 52, the last of these three answer sets provides a redundant explanation compared to the first two. However, here the explanations in terms of unfoundedness of atoms are correct, as there exists an unfounded loop. In addition, $\mathcal{T}_k[P_{30}] \cup \mathcal{T}_{ex}[P_{30}]$ has four answer sets stating the same reasons as $M_{26} - M_{29}$. □

Note that `spock` does not suggest how to *change* an inconsistent logic program to make it consistent. However, based on the abnormality tagging-atoms in an answer set M of $\mathcal{T}_k[P] \cup \mathcal{T}_{ex}[P]$ there is a straightforward way of turning the inconsistent program P into a consistent logic program:

- if $\mathsf{unsatisfied}(r) \in M$, then delete r from P;
- if $\mathsf{unsupported}(a) \in M$ or $\mathsf{unfounded}(a) \in M$, then add $a \leftarrow$ to P.

If this is done for all abnormality-tagging atoms in M, the changed logic program has an answer set $M \cap At$. Note that even though this change results in a consistent program, there is no guarantee that this program captures the originally intended meaning.

Example 54 (*Ex. 52 continued*)
Consider adding $b \leftarrow$ to P_{29}, based on M_{26}. This turns P_{29} into a consistent logic program with answer set $\{b\}$. However, the intended meaning of the program

may have been a choice between answer set $\{a\}$ and $\{b\}$, with the programmer's mistake being that *not* b in r_2 should have been *not* a. In this case, the change does not capture the original meaning. □

In addition to giving explanations of inconsistent programs with respect to automatically generated potential answer sets, the `spock` system also allows for more user-directed explanations. Among others, a user can specify a set of rules and atoms from which the explanations are drawn (Brain et al. 2007b). For example, in P_{29} we may be sure that rule r_2 is correct and thus restrict[32] abnormality tagging-atoms $\mathsf{unsatisfied}(r)$ to rule r_1. This prevents the construction of answer set M_{28}, thus resulting in M_{26} as the only explanation (when using minimisation). Furthermore, an atom a that should be included in an answer set can be specified by adding the constraint $\leftarrow$ *not* a to the kernel transformation of the given logic program.

4.1.2 Transformation 2

In the first transformation approach of `spock`, an ASP solver is merely used to compute the answer sets of the kernel and extrapolation transformations, thus generating explanations. That is, the kernel and extrapolation transformations are constructed externally (from the ASP solver). In contrast, the second transformation approach of `spock` (Gebser et al. 2008) uses an ASP solver to both construct a transformation and compute explanations. This is achieved by using a static non-ground *meta-program* $\mathcal{P}_{meta}$, which expresses violation conditions that can be instantiated with any given logic program. The second transformation approach is furthermore defined for any logic program, i.e. the head of rules is a (possibly empty) disjunction of atoms.

In order to instantiate the meta-program with the rules and atoms of a given logic program P, an *input transformation* $\mathcal{P}_{inp}[P]$ is generated, containing facts that express which rules r and atoms a are contained in P. More specifically, for every atom $a \in At$, every rule $r \in P$ (where r is the label of the rule), and every $h \in head(r)$, $b \in body^+(r)$, and $c \in body^-(r)$ the following facts are included in $\mathcal{P}_{inp}[P]$:

$$\mathsf{atom}(a) \leftarrow \tag{56}$$

$$\mathsf{rule}(r) \leftarrow \tag{57}$$

$$\mathsf{head}(r, h) \leftarrow \tag{58}$$

$$\mathsf{bodyP}(r, b) \leftarrow \tag{59}$$

$$\mathsf{bodyN}(r, c) \leftarrow \tag{60}$$

This input transformation $\mathcal{P}_{inp}[P]$ is combined with the static meta-program $\mathcal{P}_{meta}$ to compute explanations for inconsistent logic programs using an ASP solver. The meta-program uses a more explicit way of constructing potential answers sets than

[32] In the `spock` implementation this is achieved by using flags `exrules` and `exatoms` for specifying rules and atoms to be debugged. This restricts the transformations to these rules and atoms.

the extrapolation transformations, namely, for every $\mathsf{atom}(a)$ there is the choice to include or not include it in a potential answer set.[33]

$$\mathsf{in}(A) \leftarrow \mathsf{atom}(A) \wedge \mathit{not}\, \mathsf{out}(A) \tag{61}$$

$$\mathsf{out}(A) \leftarrow \mathsf{atom}(A) \wedge \mathit{not}\, \mathsf{in}(A) \tag{62}$$

Thus, an answer set of $\mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta}$ comprises for each atom $a \in At$ either $\mathsf{in}(a)$ or $\mathsf{out}(a)$. In contrast, an answer set of $\mathcal{T}_k[P] \cup \mathcal{T}_{ex}[P]$ either does or does not contain $a \in At$.

The other parts of the meta-program $\mathcal{P}_{meta}$ are similar to the kernel and extrapolation transformations. The rule applicability conditions of the kernel transformation (rules (45)-(49)) are expressed in $\mathcal{P}_{meta}$ as follows:

$$\mathsf{blocked}(R) \leftarrow \mathsf{bodyP}(R, B) \wedge \mathsf{out}(B) \tag{63}$$

$$\mathsf{blocked}(R) \leftarrow \mathsf{bodyN}(R, C) \wedge \mathsf{in}(C) \tag{64}$$

$$\mathsf{applicable}(R) \leftarrow \mathit{not}\, \mathsf{blocked}(R) \tag{65}$$

In contrast to the first transformation approach, the applicability of a rule is here expressed in terms of the rule not being blocked.

The following rules of the meta-program $\mathcal{P}_{meta}$ generalise the extrapolation transformations for rule satisfiability from normal rules to rules whose head may be empty or a disjunction of atoms.[34] In contrast to normal rules, here we check if at least *one* of the head atoms of an applicable rule is satisfied.

$$\mathsf{headSatisfied}(R) \leftarrow \mathsf{head}(R, A) \wedge \mathsf{in}(A) \tag{66}$$

$$\mathsf{unsatisfied}(R) \leftarrow \mathsf{applicable}(R) \wedge \mathit{not}\, \mathsf{headSatisfied}(R) \tag{67}$$

For logic programs that are not normal, an atom may be unsupported even if there exists a rule with a in the head that is not blocked. As stated in the second condition of Definition 32, a is supported if some rule is applicable and a is the *only* head atom that is in the potential answer set being explained. Thus, for an atom to be unsupported, this condition has to be false.

$$\mathsf{oHeadTrue}(R, A) \leftarrow \mathsf{head}(R, A2) \wedge A \neq A2 \wedge \mathsf{in}(A2) \tag{68}$$

$$\mathsf{supported}(A) \leftarrow \mathsf{head}(R, A) \wedge \mathsf{applicable}(R) \wedge \mathit{not}\, \mathsf{oHeadTrue}(R, A) \tag{69}$$

$$\mathsf{unsupported}(A) \leftarrow \mathsf{in}(A) \wedge \mathit{not}\, \mathsf{supported}(A) \tag{70}$$

The biggest difference between the first and second transformation approach concerns unfounded loops. Just like the first approach, the second includes a choice as to whether or not an atom that is part of the potential answer set being explained is unfounded (see rules (71) and (72)). The difference is that if an atom is guessed to be unfounded, there is a check (see rule (73)) of the foundedness condition in Definition 32. That is, for an unfounded atom a it is checked if there is an applicable

[33] Throughout this section, we use uppercase letters to denote variables.

[34] The meta-program also contains rules explicitly handling unsatisfied constraints, tagging them with a different abnormality atom. For simplicity, and since rule (67) also applies to constraints, we do not report these constraint rules.

rule r with a in the head (if so, r is also satisfied since $\mathsf{unfounded}(a)$ only holds if $\mathsf{in}(a)$) that has no head atom that is founded (in the same loop) and no positive body atom that is unfounded (in the same loop). If such a rule exists, a is by Definition 32 founded, which is why this check is implemented as a constraint in $\mathcal{P}_{meta}$ (rule (73)). This ensures that $\mathsf{unfounded}(a)$ is only part of an answer set of $\mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta}$, if a is actually unfounded.

$$\mathsf{unfounded}(A) \leftarrow \mathsf{in}(A) \wedge \mathsf{supported}(A) \wedge \mathit{not}\ \mathsf{founded}(A) \tag{71}$$

$$\mathsf{founded}(A) \leftarrow \mathsf{in}(A) \wedge \mathit{not}\ \mathsf{unfounded}(A) \tag{72}$$

$$\leftarrow \mathsf{unfounded}(A) \wedge \mathsf{head}(R, A) \wedge \mathsf{applicable}(R) \wedge \mathit{not}\ \mathsf{headNotinLoop}(R) \wedge \mathit{not}\ \mathsf{BodyPInLoop}(R) \tag{73}$$

$$\mathsf{headNotinLoop}(R) \leftarrow \mathsf{head}(R, A) \wedge \mathsf{founded}(A) \tag{74}$$

$$\mathsf{BodyPInLoop}(R) \leftarrow \mathsf{bodyP}(R, A) \wedge \mathsf{unfounded}(A) \tag{75}$$

Furthermore, there are rules ensuring that only one loop is considered at a time, i.e. $\mathsf{unfounded}(a)$ and $\mathsf{unfounded}(b)$ only hold if a and b are part of the same loop.

Another main difference between the two `spock` approaches is that the second transformation approach only explains sets of atoms that are *not* answer sets of the given logic program, whereas the first also explains *actual* answer sets of the given logic program (if any exist). This is due to the following rules in the meta-program $\mathcal{P}_{meta}$, ensuring that at least one abnormality tagging-atom is part of an answer set:

$$\mathsf{noAS} \leftarrow \mathsf{unsatisfied}(r) \tag{76}$$

$$\mathsf{noAS} \leftarrow \mathsf{unsupported}(r) \tag{77}$$

$$\mathsf{noAS} \leftarrow \mathsf{unfounded}(r) \tag{78}$$

$$\leftarrow \mathit{not}\ \mathsf{noAS} \tag{79}$$

Example 55 (*Ex. 52 continued*)
Applying the second transformation approach to P_{29}, `spock` computes the answer sets of $\mathcal{P}_{inp}[P_{29}] \cup \mathcal{P}_{meta}$, yielding the following:

- $M_{30} = \{\mathsf{in}(a), \mathsf{in}(b), \mathsf{founded}(a), \mathsf{founded}(b), \mathsf{unsupported}(a), \mathsf{unsupported}(b), \mathsf{blocked}(r_1), \mathsf{blocked}(r_2), \mathsf{headSatisfied}(r_1), \mathsf{headSatisfied}(r_2), \mathsf{headNotinLoop}(r_1), \mathsf{headNotinLoop}(r_2)\}$
- $M_{31} = \{\mathsf{in}(b), \mathsf{out}(a), \mathsf{founded}(b), \mathsf{unsupported}(b), \mathsf{blocked}(r_1), \mathsf{blocked}(r_2), \mathsf{headSatisfied}(r_2), \mathsf{headNotinLoop}(r_2)\}$
- $M_{32} = \{\mathsf{in}(a), \mathsf{out}(b), \mathsf{founded}(a), \mathsf{supported}(a), \mathsf{supported}(b), \mathsf{unsatisfied}(r_2), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2), \mathsf{headSatisfied}(r_1), \mathsf{headNotinLoop}(r_1)\}$
- $M_{33} = \{\mathsf{out}(a), \mathsf{out}(b), \mathsf{supported}(a), \mathsf{supported}(b), \mathsf{unsatisfied}(r_1), \mathsf{unsatisfied}(r_2), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2)\}$

Note that all answer sets also comprise the facts in $\mathcal{P}_{inp}[P_{29}]$, such as $\mathsf{atom}(a)$, $\mathsf{head}(r_1, a)$, and $\mathsf{rule}(r_1)$, as well as the atom noAS, which we omitted above for readability. Since the second transformation approach does not generate explanations containing unfoundedness as a reason when an atom is in fact founded, there is no equivalent of answer set M_{27} from the first transformation approach. All other

answer sets of $\mathcal{T}_k[P_{29}] \cup \mathcal{T}_{ex}[P_{29}]$ report the same reasons as the answer sets given above. □

Example 56 (*Ex. 53 continued*)
For the program P_{29}, which comprises an unfounded loop w.r.t. $\{a, b\}$, even more redundant explanations are omitted when using the second transformation approach. More precisely, as for P_{29} there is one explanation for each possible set of atoms, i.e. $\{\}$, $\{a\}$, $\{b\}$, and $\{a, b\}$. The explanation as to why the last set is not an answer set is given by $\mathsf{unfounded}(a)$ and $\mathsf{unfounded}(b)$. The explanations concerning the other three sets are analogue to the explanations of P_{29} in Example 55. □

Similarly to the first transformation approach, the user can specify constraints for debugging. An atom a can, for example, be forced to (not) be a part of an answer set by adding the constraint $\leftarrow \mathsf{out}(a)$ (respectively $\leftarrow \mathsf{in}(a)$) to the input transformation of the given logic program. In the same way, constraints on the abnormality tagging-atoms can be specified, e.g. $\leftarrow \mathsf{unsatisfied}(r)$ enforces that rule r is satisfied.

In conclusion, the second transformation approach requires less processing of the given logic program P performed outside the ASP solver than the first transformation approach. Furthermore, the two transformation approaches differ in the number of explanations given, since the first approach may yield redundant explanations and explanations where unfoundedness is given as a reason even though the atom in question is founded.

4.2 The `Ouroboros` *System – Debugging Non-ground Programs*

The two `spock` approaches do not explicitly deal with variables occurring in the given logic program. However, variables are important to consider for debugging approaches, since, in practice, logic programs under the answer set semantics often contain first-order predicates and variables. Handling variables when debugging thus requires an efficient grounding strategy.

Building upon the second `spock` transformation, Oetsch et al. (2010) develop a meta-program able to construct explanations of inconsistent *extended* logic programs possibly comprising variables. In contrast to the approach taken by `spock`, which constructs various sets of atoms and explains why these are not answer sets, `Ouroboros` requires an *intended answer set*. It thus assumes that the user already has a solution in mind. An explanation is then constructed for this anticipated solution.

Efficiently constructing explanations for logic programs with variables is non-trivial as it requires grounding (i.e. substituting variables with constants). First grounding a given logic program and then constructing explanations, for example using the `spock` approach, requires exponential space and double-exponential time. Instead, the `Ouroboros` approach requires only polynomial space and single-exponential time, as it applies grounding to the input transformation and meta-program during the solving process rather than grounding the given logic program before transforming and solving it.

Similarly to the input transformation $\mathcal{P}_{inp}[P]$ of the second `spock` approach, `Ouroboros` constructs an *input transformation* $\varrho_{inp}[P]$ of a given logic program P, expressing which extended literals are part of the head and body of each rule. Additionally, $\varrho_{inp}[P]$ includes facts expressing which predicates occur in P, what the position of variables and constants is in each predicate, and which variables occur in which rules. Since `Ouroboros` requires a given set of atoms $M \subseteq At$ to be explained, this set is also transformed to make it applicable to the input transformation and the meta-program. The *interpretation transformation* $\varrho_{int}[M]$ includes facts $\mathsf{in}(a)$ for each atom $a \in M$ as well as facts stating which predicates occur in M and what the position of constants is in predicates in M.

The *meta-program* ϱ_{meta} of `Ouroboros` follows the same ideas as `spock`, expressing conditions under which a rule is unsatisfied or a loop is unfounded. Note that in contrast to `spock`, `Ouroboros` does not explicitly point out unsupported atoms. Instead, unsupported atoms are handled as singleton loops that are unfounded. The exact encoding of ϱ_{meta} with its more than 160 rules can be found online[35].

When applying an ASP solver to $\varrho_{inp}[P] \cup \varrho_{int}[M] \cup \varrho_{meta}$ to compute explanations as to why M is not an answer set, the automatic grounding of the solver allows for the efficient computation of ground answer sets if P contains variables.

Just like `spock`, `Ouroboros` only gives *explanations* as to why a set of atoms is not an answer set. The subsequent changing of the program to make it consistent is left to the user. In addition to explicit negation, `Ouroboros` can also handle arithmetic operations with integers ($+$ and $*$) and allows for comparison predicates ($=$, $\neq$, $\geq$, $\leq$, $>$, $<$). Polleres et al. (2013) further extend `Ouroboros` to deal with choice rules and cardinality and weight constraints by translating these constructs into normal rules (possibly containing variables). Frühstück et al. (2013) integrate `Ouroboros` into the `SeaLion` IDE.[36]

4.3 Interactive Debugging Based on `spock`

No matter which of the two transformations is used, the `spock` approach may generate many different explanations, since for every set of atoms that is not an answer set at least one explanation is constructed. Even for the small logic program in Example 52, which has only two atoms, four explanations are generated using the second transformation (see Example 55). `Ouroboros` tackles this problem by requiring the user to specify an intended answer set. However, a user may not have a truth assignment for every atom in mind. Shchekotykhin (2015) therefore proposes an interactive method on top of the second `spock` approach, where the user is *queried* whether or not an atom should be contained in an answer set. The user's answer narrows down the sets of atoms for which explanations are constructed to the ones *relevant* to the user and relieves the user of the burden to specify the whole intended answer set upfront.

As mentioned in previous sections, the user can force atoms to be contained or not

[35] `www.kr.tuwien.ac.at/research/projects/mmdasp/encoding.tar.gz`
[36] Note that a special setup of ASP solvers is needed to make this integration work.

contained in explanation answer sets of `spock` (using the second transformation) by adding facts $\mathsf{in}(a)$ or $\mathsf{out}(a)$. In the interactive debugging approach, such statements are explicitly used as *test cases*.

Definition 33 (*Test Case and Background Theory*)
Given a program P, its input transformation $\mathcal{P}_{inp}[P]$, and the meta-program $\mathcal{P}_{meta}$

- $Pos, Neg \subseteq \{\ \mathsf{in}(a), \mathsf{out}(a) \mid a \in At\ \}$ are sets of positive and negative *test cases*,
- $\mathcal{B} \subseteq P$ is a *background theory*. □

Positive test cases are atoms that have to be contained in ($\mathsf{in}(a)$) or excluded from ($\mathsf{out}(a)$) *all* answer sets. In contrast, negative test cases are atoms that have to be contained in ($\mathsf{in}(a)$) or excluded from ($\mathsf{out}(a)$) *some* answer set. A background theory consists of rules in the logic program that are assumed to be satisfied.

In contrast to the `spock` approach, answer sets of $\mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta}$ that contain the same abnormality tagging-atoms are considered as the same explanation, even if the atoms in the respective explained answer sets are different. The aim is to find sets of abnormality tagging-atoms that satisfy all given test cases and the given background theory. In other words, we want to compute all answer sets of $\mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta}$ containing *only* abnormality tagging-atoms satisfying the test cases and the background theory. Sets of abnormality tagging-atoms satisfying this condition are called *diagnoses*.

Definition 34 (*Diagnosis*)
Let $Er(P)$ be the set of all abnormality tagging-atoms over a program P, that is,

$$Er(P) \stackrel{\text{def}}{=} \{\ \mathsf{unsatisfied}(r) \mid r \in P\ \} \cup \{\ \mathsf{unsupported}(a), \mathsf{unfounded}(a) \mid a \in At\ \}$$

A set $\mathcal{D} \subseteq Er(P)$ is a *diagnosis* for the *problem instance* $\langle P, \mathcal{B}, Pos, Neg\rangle$ if

1. $P^* = \mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta} \cup \{\ \leftarrow d \mid d \in Er(P) \setminus \mathcal{D}\ \} \cup \{\ \leftarrow \mathsf{unsatisfied}(r) \mid r \in \mathcal{B}\ \} \cup \{\ p \leftarrow \mid p \in Pos\ \}$ has an answer set and
2. for each $n \in Neg$, $P^* \cup \{n \leftarrow\}$ has an answer set. □

Note that due to the constraints of the form $\leftarrow d$, any answer set of P^* will only contain abnormality tagging-atoms from $\mathcal{D}$.

Diagnoses can be found by computing answer sets of the program $\mathcal{P}_{inp}[P] \cup \mathcal{P}_{meta} \cup \{\ \leftarrow \mathsf{unsatisfied}(r) \mid r \in \mathcal{B}\ \}$ and then verifying whether the respective sets of abnormality tagging-atoms contained in these answer sets satisfy the conditions for being a diagnosis. Usually, only (subset) *minimal* diagnoses will be considered.

Example 57
Consider the logic program P_{28} (see Example 51; page 67) with the additional constraint $r_8 : \leftarrow \mathit{not\ off}_1$. This program, called P_{31}, is inconsistent. Using the second translation approach of `spock`, 256 answer sets are computed for $\mathcal{P}_{inp}[P_{31}] \cup \mathcal{P}_{meta}$, each explaining a different set of atoms that is not an answer set. Let us now specify $\mathcal{B} = \{r_1, r_2, r_6, r_7\}$, in other words, we are sure that the first two rules are correct and that switches a and b are *on* in situation 0. This narrows down the answer sets; program $\mathcal{P}_{inp}[P_{31}] \cup \mathcal{P}_{meta} \cup \{\ \leftarrow \mathsf{unsatisfied}(r_1),\ \leftarrow \mathsf{unsatisfied}(r_2),$

$\leftarrow$ unsatisfied(r_6), $\leftarrow$ unsatisfied$(r_7)\}$ has only 28 answer sets. Given positive test cases $Pos = \{\mathsf{out}(swc_0), \mathsf{out}(swd_0)\}$, only eight out of the 28 answer sets satisfy these, namely :

- $M_{34} = \{\mathsf{out}(on_0), \mathsf{out}(off_0), \mathsf{in}(on_1), \mathsf{out}(off_1)\} \cup \{\mathsf{unsatisfied}(r_5), \mathsf{unsatisfied}(r_8)\}$
- $M_{35} = \{\mathsf{in}(on_0), \mathsf{out}(off_0), \mathsf{in}(on_1), \mathsf{out}(off_1)\} \cup \{\mathsf{unsatisfied}(r_5), \mathsf{unsatisfied}(r_8), \mathsf{unsupported}(on_0)\}$
- $M_{36} = \{\mathsf{out}(on_0), \mathsf{out}(off_0), \mathsf{in}(on_1), \mathsf{in}(off_1)\} \cup \{\mathsf{unsatisfied}(r_5), \mathsf{unsupported}(off_1)\}$
- $M_{37} = \{\mathsf{in}(on_0), \mathsf{out}(off_0), \mathsf{in}(on_1), \mathsf{in}(off_1)\} \cup \{\mathsf{unsatisfied}(r_5), \mathsf{unsupported}(off_1), \mathsf{unsupported}(on_0)\}$
- $M_{38} = \{\mathsf{out}(on_0), \mathsf{in}(off_0), \mathsf{in}(on_1), \mathsf{out}(off_1)\} \cup \{\mathsf{unsatisfied}(r_8)\}$
- $M_{39} = \{\mathsf{in}(on_0), \mathsf{in}(off_0), \mathsf{in}(on_1), \mathsf{out}(off_1)\} \cup \{\mathsf{unsatisfied}(r_8), \mathsf{unsupported}(on_0)\}$
- $M_{40} = \{\mathsf{out}(on_0), \mathsf{in}(off_0), \mathsf{in}(on_1), \mathsf{in}(off_1)\} \cup \{\mathsf{unsupported}(off_1)\}$
- $M_{41} = \{\mathsf{in}(on_0), \mathsf{in}(off_0), \mathsf{in}(on_1), \mathsf{in}(off_1)\} \cup \{\mathsf{unsupported}(off_1), \mathsf{unsupported}(on_0)\}$

Note that each answer set also contains $\mathsf{in}(swa_0)$, $\mathsf{in}(swb_0)$, $\mathsf{out}(swc_0)$, and $\mathsf{out}(swd_0)$, as well as the further tagging-atoms discussed in Section 4.1.2. Taking a closer look at these 8 answer sets, each of them defines a diagnosis when $Neg = \{\}$, namely the second part of each answer set. Only M_{38} and M_{40} induce *minimal* diagnoses. Now consider that $Neg = \{\mathsf{in}(on_0), \mathsf{in}(off_0)\}$. This rules out half of the diagnoses, leaving only the following four:

- $\mathcal{D}_1 = \{\mathsf{unsatisfied}(r_5), \mathsf{unsatisfied}(r_8), \mathsf{unsupported}(on_0)\}$ (cf. M_{35})
- $\mathcal{D}_2 = \{\mathsf{unsatisfied}(r_5), \mathsf{unsupported}(off_1), \mathsf{unsupported}(on_0)\}$ (cf. M_{37})
- $\mathcal{D}_3 = \{\mathsf{unsatisfied}(r_8), \mathsf{unsupported}(on_0)\}$ (cf. M_{39})
- $\mathcal{D}_4 = \{\mathsf{unsupported}(off_1), \mathsf{unsupported}(on_0)\}$ (cf. M_{41})

Even though $\mathsf{in}(off_0) \notin M_{35}$, $\mathcal{D}_1$ is a diagnosis of the given problem instance since there are two answer sets of P^* w.r.t. $\mathcal{D}_1$, namely M_{35} and M_{39}, and $\mathsf{in}(off_0) \in M_{39}$, thus satisfying the negative test case $\mathsf{in}(off_0)$ w.r.t. $\mathcal{D}_1$. □

As illustrated in Example 57, positive and negative test cases can considerably reduce the number of diagnoses and, thus, of explanations as to why sets of atoms are not answer sets of P. If the user does not specify any test cases, it is therefore desirable to produce them automatically by querying the user. That is, the user is asked whether an atom is expected to be contained in or excluded from all or some answer sets. Ideally, the debugging system chooses an atom as a query that helps to reduce the number of diagnoses as much as possible.

Definition 35 (*Query and Diagnosis Splitting*)
Let $\mathbf{D}$ be the set of all diagnoses of the problem instance $\langle P, \mathcal{B}, Pos, Neg\rangle$ and let $\mathcal{Q} \subseteq At$ be a *query*. $\mathcal{Q}$ splits the diagnoses in $\mathbf{D}$ into three sets, where for each $\mathcal{D} \in \mathbf{D}$:

- $\mathcal{D} \in \mathbf{D^P}$ if for all $a \in \mathcal{Q}$, $\mathsf{in}(a)$ is in every answer set of P^*;
- $\mathcal{D} \in \mathbf{D^N}$ if for all $a \in \mathcal{Q}$, $\mathsf{out}(a)$ is in every answer set of P^*;
- $\mathcal{D} \in \mathbf{D^\emptyset}$ if $\mathcal{D} \notin (\mathbf{D^P} \cup \mathbf{D^N})$.

This means that $\mathbf{D^P}$ and $\mathbf{D^N}$ contain all diagnoses that are still diagnoses if the atoms in the query are added as positive test cases so as to force them to be, respectively, included in or excluded from all answer sets. Thus, if the user's reply to a query is that the atoms should be included, then the diagnoses in $\mathbf{D^N}$ can be disregarded. Likewise, if the user replies that the atoms should be excluded, the diagnoses in $\mathbf{D^P}$ can be disregarded.

Example 58 (*Ex. 57 continued*)
Consider the two atoms that are not part of positive or negative test cases yet, namely on_1 and off_1. For $\mathcal{Q}_1 = \{on_1\}$, all four diagnoses are in $\mathbf{D^P}$, so $\mathbf{D^N} = \mathbf{D}^\emptyset = \{\}$. For example, the answer sets of P^* w.r.t. $\mathcal{D}_1$ are M_{35} and M_{39}, and both comprise $\mathsf{in}(on_1)$. This means that if the user replies to the query, that on_1 should be in the desired answer set, then no diagnoses can be disregarded. However, if the user replies that on_1 should not be in the desired answer set, then all diagnoses would be disregarded and therefore no explanations given. This would imply, that the test cases specified could not be satisfied. In contrast, for $\mathcal{Q}_2 = \{off_1\}$ we get $\mathbf{D^P} = \{\mathcal{D}_2, \mathcal{D}_4\}$, $\mathbf{D^N} = \{\mathcal{D}_1, \mathcal{D}_3\}$, and $\mathbf{D}^\emptyset = \{\}$. Note that if one of the negative test cases was used as a query, then $\mathbf{D}^\emptyset \neq \{\}$. For instance, for $\mathcal{Q}_3 = \{off_0\}$ we get $\mathcal{D}_1 \in \mathbf{D}^\emptyset$ since $\mathsf{out}(off_0) \in M_{35}$ but $\mathsf{in}(off_0) \in M_{39}$. □

There may be a large number of queries, so queries that yield a large information gain are desirable, i.e. queries that allow to disregard as many diagnoses as possible, independent of the user's answer, which clearly is not known when generating a query. Thus, a useful query should at least yield a partition with $\mathbf{D^P}, \mathbf{D^N} \neq \{\}$ so that independent of the user's answer, some diagnoses can be disregarded.

A straightforward selection method is the *myopic* strategy, which prefers queries yielding sets $\mathbf{D^P}$ and $\mathbf{D^N}$ that have similar size and where $\mathbf{D}^\emptyset$ is as small as possible. That is, a query that minimises

$$| \, | \, \mathbf{D^P} \, | - | \, \mathbf{D^N} \, | \, | + | \, \mathbf{D}^\emptyset \, |$$

Example 59 (*Ex. 58 continued*)
According to the myopic strategy, $\mathcal{Q}_2$ is preferable to $\mathcal{Q}_1$ since independent of the answer of the user, the number of possible queries is reduced to two. □

The idea of this interactive debugging approach is that queries are generated and presented to the user until only one diagnosis, or a specified maximal number of diagnoses, is left.

4.4 The DWASP *System – Interactive Debugging of Non-ground Programs*

The interactive debugging approach discussed in the previous section only applies to logic programs without variables. Dodaro et al. (2015) and Gasteiger et al. (2016) extend the idea, of querying the user to find relevant explanations of inconsistency, to non-ground programs. Instead of using an elaborate meta-program expressing possible reasons for inconsistencies as in `spock`, they use the solving process of the

ASP solver WASP (Alviano et al. 2013; Alviano et al. 2015) to find inconsistencies in a logic program. Their ASP debugger is thus called DWASP.

Like Shchekotykhin (2015), DWASP allows to define a background theory. If the background theory is not explicitly specified, the set of facts of the given logic program P is used. Instead of applying abnormality tagging-atoms to indicate inconsistencies, the DWASP system adds to each rule in P that is not part of the background theory a *debug atom*, stating the name of the rule and the variables occurring in it.

Definition 36 (*Debugging Program*)
Given a logic program P and a background theory $\mathcal{B} \subseteq P$, the *debugging program* is defined as:

$$\begin{aligned} \mathcal{P}_{deb}[P] = \mathcal{B} \cup \{\ h_1 \vee \ldots \vee h_k \ \leftarrow\ b_1 \wedge \ldots \wedge b_n \wedge \_debug(r, \mathbf{vars}_r) \\ \wedge not\ c_1 \wedge \ldots not\ c_m \ \mid\ r \in P \setminus \mathcal{B}, head(r) = \{h_1 \vee \ldots \vee h_k\}, \\ body(r) = \{b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m\}\ \} \end{aligned} \tag{80}$$

where $\mathbf{vars}_r$ is a tuple consisting of all variables in $body(r)$. □

When applying the WASP solver to the debugging program $\mathcal{P}_{deb}[P]$, atoms can be *assumed* to hold when computing answer sets. That is, these assumed atoms do not need to be derived from rules or facts, they are true by default. Assumed atoms are thus similar to positive test cases in the approach of Shchekotykhin (2015).

If a debugging atom is *not* assumed to hold, this amounts to "blocking" the respective rule specified in the atom, i.e. the rule is no longer applicable when computing answer sets, since a debugging atom cannot be derived using the rules in $\mathcal{P}_{deb}[P]$. If all debugging atoms are assumed to hold, the answer sets of $\mathcal{P}_{deb}[P]$ (minus the debugging atoms) coincide with the answer sets of P. If P is inconsistent, it therefore follows that $\mathcal{P}_{deb}[P]$ is also inconsistent.

To find rules causing the inconsistency of a program, the WASP solver allows to compute *unsatisfiable cores*, i.e. sets of atoms such that if they are assumed to hold, no answer set exists. In the DWASP system, only debugging atoms are considered for unsatisfiable cores. Thus, an unsatisfiable core points out a combination of rules causing the inconsistency.

Definition 37 (*Unsatisfiable Core*)
Let $\mathcal{P}^G_{deb}[P]$ be the grounding of $\mathcal{P}_{deb}[P]$ and let $At_{deb}(P)$ be the set of all (ground) debugging atoms occurring in $\mathcal{P}^G_{deb}[P]$. $C \subseteq At_{deb}(P)$ is an *unsatisfiable core* iff $\mathcal{P}^G_{deb}[P]$ is inconsistent when all debugging atoms in C are assumed to hold. □

Note that this definition does not make any assumptions about other atoms assumed to hold. Therefore, an unsatisfiable core is such that, no matter which other atoms are assumed to hold, $\mathcal{P}^G_{deb}[P]$ is inconsistent.

If P is inconsistent, clearly $At_{deb}(P)$ is an unsatisfiable core. However, there may be other unsatisfiable cores, which are subsets of $At_{deb}(P)$, and thus more useful for identifying the source of inconsistency. Therefore, only (subset) *minimal* unsatisfiable cores are of interest in DWASP.

If there is only one unsatisfiable core, then deleting any of the atoms in the core from the atoms assumed to hold results in the existence of an answer set. However, if there are various unsatisfiable cores, only a combination of atoms from the different cores will lead to the existence of an answer set. DWASP finds such sets of debugging atoms that, when no longer assumed to hold, ensure the existence of an answer set. Such sets thus express which rules need to be "blocked" to obtain an answer set.

Definition 38 (DWASP *Diagnosis*)
Let $\mathcal{P}^G_{deb}[P]$ be the grounding of $\mathcal{P}_{deb}[P]$ and let $At_{deb}(P)$ be the set of all (ground) debugging atoms occurring in $\mathcal{P}^G_{deb}[P]$. $\mathcal{D}_{\text{DWASP}} \subseteq At_{deb}(P)$ is a *diagnosis* iff $\mathcal{P}^G_{deb}[P]$ is consistent when none of the debugging atoms in $\mathcal{D}_{\text{DWASP}}$ is assumed to hold. □

The DWASP system only considers *minimal diagnoses*. Even though the definition of diagnosis does not reference unsatisfiable cores, diagnoses are computed from unsatisfiable cores in DWASP.

Note the difference between the notions of diagnosis used in DWASP and in the approach of Shchekotykhin (2015). In both cases, a diagnosis comprises atoms identifying the reason for inconsistency. The difference is that in DWASP a diagnosis is a set of atoms such that the debugging program is consistent if the atoms are *not contained* in answer sets. In contrast, a diagnosis according to Definition 34 is a set of abnormality tagging-atoms such that the transformed logic program is consistent if these are the only abnormality tagging-atoms *contained* in answer sets.

As in the approach by Shchekotykhin (2015), there may be a large number of diagnoses and not all of them may be relevant to the user. Thus, DWASP uses the same strategy for querying the user as discussed in the previous section for the approach by Shchekotykhin (2015). That is, a *query atom* $q \in At$ is determined, i.e. a ground (non-debugging) atom, which partitions the set of all diagnoses into $\mathbf{D^P}$, $\mathbf{D^N}$, and $\mathbf{D}^\emptyset$, where:

- $\mathcal{D}_{\text{DWASP}} \in \mathbf{D^P}$ if q is in every answer set of $\mathcal{P}^G_{deb}[P]$ when none of the debugging atoms in $\mathcal{D}_{\text{DWASP}}$ is assumed to hold;
- $\mathcal{D}_{\text{DWASP}} \in \mathbf{D^N}$ if q is in no answer set of $\mathcal{P}^G_{deb}[P]$ when none of the debugging atoms in $\mathcal{D}_{\text{DWASP}}$ is assumed to hold;
- $\mathcal{D}_{\text{DWASP}} \in \mathbf{D}^\emptyset$ if $\mathcal{D}_{\text{DWASP}} \notin (\mathbf{D^P} \cup \mathbf{D^N})$.

The only difference in the usage of queries in DWASP as compared to the approach of Shchekotykhin (2015) is that, rather than adding test cases, the user's answer determines if q (in case q should hold) or *not* q (in case q should not hold) is added to the set of assumed atoms.

4.5 Stepping

The debugging approach of Oetsch et al. (2018), which extends previous work by Oetsch et al. (2011) and Pührer (2014), tackles the problem of explaining why a set of atoms is or is not an answer set of a logic program in a *procedural* manner. Inspired by debugging in procedural programming languages, where the step-wise

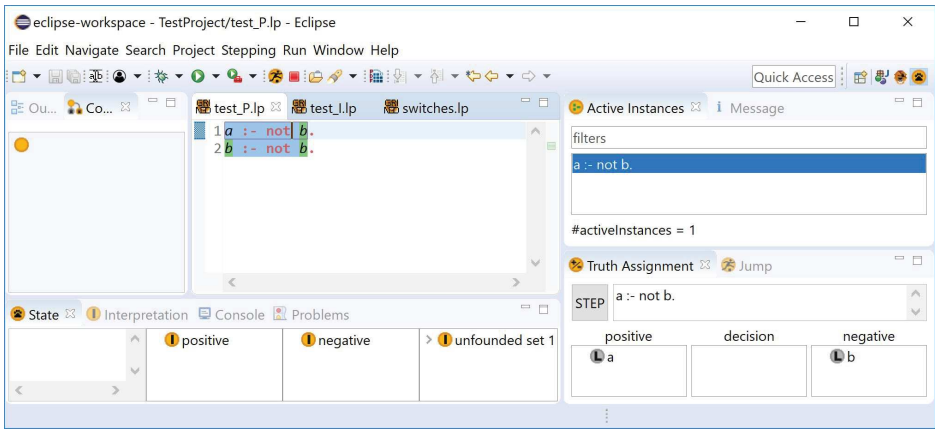


Fig. 31: The first rule of P_{28} is chosen for stepping. The 'truth assignment' tab shows the assignment of truth values to the atoms a and b if a step is performed on the chosen rule.

execution of a program can be traced, the `stepping` approach allows to apply rules and assign literals to be true or false with respect to a potential answer set step by step. In contrast to the execution of a procedural program, the sequence of steps in the execution of a logic program is not predetermined, due to the declarative nature of the answer set semantics. Thus, the *user* chooses the step sequence in the `stepping` approach. This debugging approach has been implemented in the `SeaLion` IDE (Busoniu et al. 2013), a logic programming plugin of the Eclipse platform.

Starting with the empty set as the potential answer set, in each computation step the user is presented with all rules that are applicable w.r.t. the current potential answer set. To satisfy the chosen rule, a head of the rule is then added to the current potential answer set and any atoms that thus cannot be in the potential answer set (because they occur in the negative body of the rule) are recorded as being false w.r.t. the potential answer set.

Example 60 (*Ex. 52 continued, page 68*)
Recall the logic program P_{28}:

$$r_1 : \ a \leftarrow not\ b$$
$$r_2 : \ b \leftarrow not\ b$$

The stepping starts with no atoms recorded as being true or false w.r.t. the potential answer set. Thus, both r_1 and r_2 are applicable since b is not recorded as being in the potential answer set, so *not* b may be true w.r.t. the current potential answer set. The user can therefore choose which of the two rules to apply. Figure 31 illustrates this scenario in the `stepping` component of `SeaLion`, where all applicable rules are marked in blue. The user chooses r_1 to proceed, so r_1 is the only 'active instance' of the chosen rule shown in the respective tab (if r_1 contained variables, all applicable grounded versions would be shown in this tab). The active instance r_1 is then used

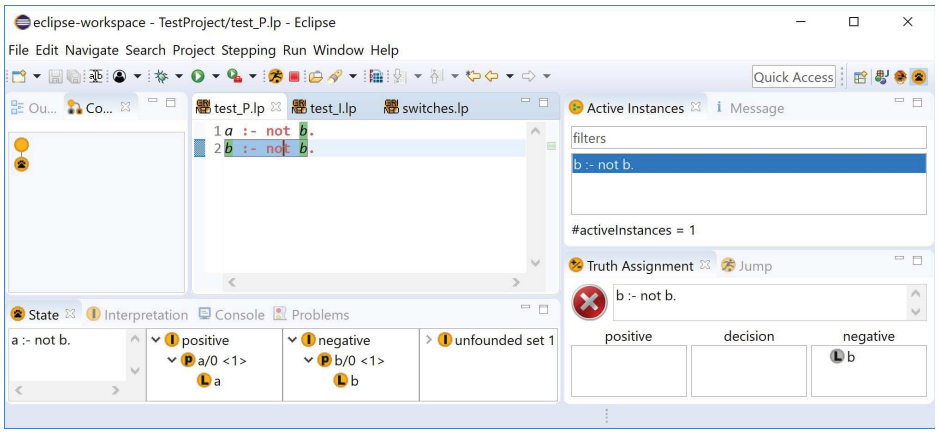


Fig. 32: After the first step, the second rule is active but a step cannot be performed.

for the 'truth assignment', which is performed by clicking the 'step' button. This records a as being true and b as being false w.r.t. the potential answer set M, as illustrated in the 'state' tab at the bottom of Figure 32. After this first step, rule r_2 is still applicable, so it is chosen for the next 'truth assignment'. However, as indicated by the red X in Figure 32, the truth assignment that would satisfy r_2 cannot be performed. Thus, the stepping computation fails before being completed, indicating to the user that the assignment of truth values performed so far does not lead to an answer set. Note that the reason why r_2 cannot be used for the next step is not pointed out to the user explicitly, i.e. that b is recorded as false, but to satisfy r_2 it would also have to be true. If r_2 was chosen in the first step, the stepping would fail straight away, i.e. the scenario from Figure 32 would apply, but without the truth assignments shown in the 'state' tab at the bottom. □

As illustrated in Example 60, the `stepping` approach gives the user an insight into the answer set computation in terms of truth assignments to atoms, rather than providing an explicit explanation of the cause of inconsistency like the previously discussed debugging approaches. It also does not make any suggestions on how to change the logic program to make it consistent. Whereas in `Ouroboros` the user needs to explicitly specify an intended answer set, the `stepping` approach indirectly allows this but does not require it. In other words, if a user expects a certain answer set, but the logic program is inconsistent or has different answer sets, the stepping can be targeted towards the intended answer set, until it becomes clear why certain atoms in the intended answer set are false or why atoms not expected to be in the answer set are true. However, the `stepping` approach can also be applied if a logic program is inconsistent and the user does not know what the answer set should be. In this case, the user can simply step through applicable rules until the stepping computation fails, thus providing an insight into how the inconsistency of the logic program arises. Note that the stepping approach can also be used to find out how consistent answer sets are derived, in line with the approaches discussed in Section 3.

Like `Ouroboros` and DWASP, the `stepping` system can handle logic programs with variables and supports language constructs such as constraints, choice rules, and aggregates. Furthermore, it can easily be used with different ASP solvers.

The theory behind the `stepping` approach is based on an extension of the FLP-semantics (Faber et al. 2011) by Oetsch et al. (2012), which coincides with the answer set semantics. This guarantees that the computation of answer sets using `stepping` is sound and complete, that is, any answer set can be reached through the step-wise application of rules and truth assignment of atoms, and any successfully terminated step-wise computation results in an assignment of truth values to atoms forming an answer set. Thus, if the step-wise computation does not terminate successfully, the current assignment of truth values cannot be extended to an answer set.

To speed up the step-wise computation, especially in large logic programs with variables, where rules have various groundings that can be applied in different steps, the user can perform *jumps*. A jump is the automatic application of various specified rules in such a way that they are satisfied. This is useful if the user is not interested in the exact workings of these rules and their influence on a potential answer set. Note that it only makes sense to use a jump if the chosen rules can be satisfied given the current truth assignment, so the user should be sure that the chosen jumping rules do not pose a problem.

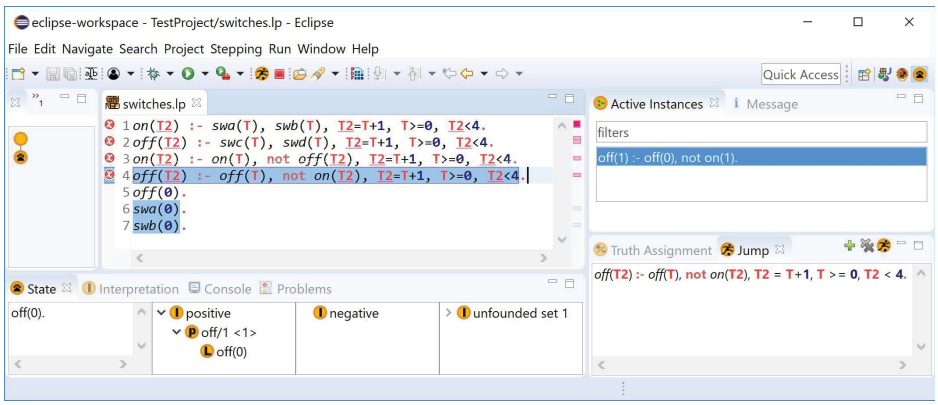


Fig. 33: The user chooses r_4 as a rule for jumping.

Example 61 (*Ex. 36 continued, page 38*)
Consider again the logic program about a light bulb and the four switches to turn the light on and off. We encode this in P_{32} for the time steps $t = 0 \dots 3$. Figure 33 illustrates P_{32} and the scenario where the user chose the fact $off(0)$ in the first step and now decides to perform a jump on r_4 (see the 'jump' tab). Since the jump only considers the current assignment of truth values and the chosen rule(s), it makes $off(1)$, $off(2)$, and $off(3)$ true and $on(1)$, $on(2)$, and $on(3)$ false by repeatedly applying r_4. This automatic assignment is shown in the 'state' tab in Figure 34, along with the grounded rules used in the automatic steps of the jump. As illustrated

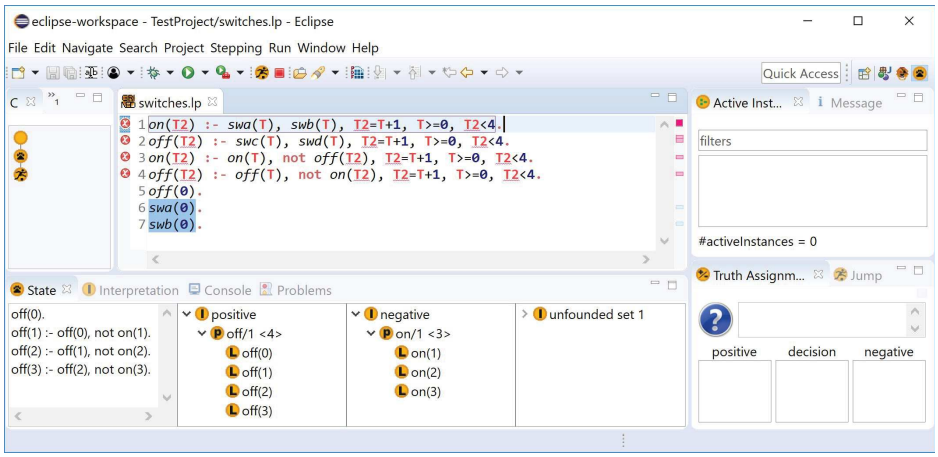


Fig. 34: Truth assignment and applicable facts (highlighted blue) after the jump.

by the blue highlighting, at this point only facts $swa(0)$ and $swb(0)$ are applicable. Performing steps on these two facts results in r_1 being applicable, but the rule

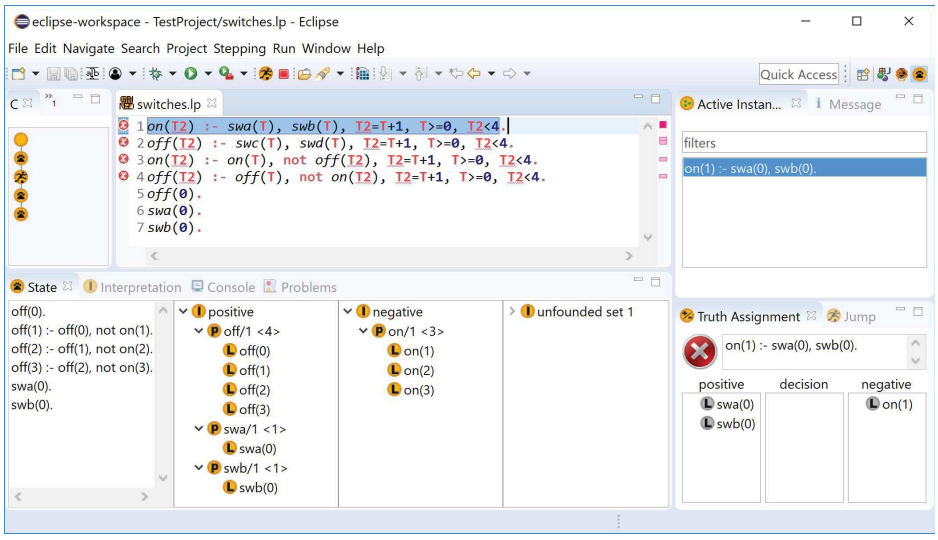


Fig. 35: Failure of the `stepping` computation.

cannot be satisfied w.r.t. the current truth assignment, as shown in Figure 35. The failure provides insights as to why there is no answer set in which the bulb is turned *off* at $t \geq 0$. Namely, the reason it may be turned *off* is inertia (application of rule r_4), however, since switches *swa* and *swb* are pushed, it follows that the light bulb must be turned on at $t = 1$. This conflicts with the previous inertia assumption that the light is not turned on (*not* $on(1)$ in r_4 when deriving $off(1)$). □

4.6 Summary and Discussion

In Sections 4.1 to 4.5, we outlined the most prominent approaches to ASP debugging, i.e. the explanation of non-existence of answer sets in terms of semantic errors. In contrast to the justification approaches discussed Section 3, where the truth value of literals is explained in detail by referring to truth values of other literals used in their derivation, the explanations provided by debugging approaches can seem rather minimalistic. Indeed, debugging aims at providing a *pointer* to the cause of inconsistency rather than a full-fledged explanation. Furthermore, we have seen that these approaches follow different ideas as to what an explanation should encompass and that they use different methodologies to achieve this. Tables 3 and 4 provide a comparative overview of the differences and similarities of the surveyed debugging approaches. In particular, Table 3 compares debugging approaches concerning the type of logic programs that can be debugged, whether or not logic programs with variables as well as with language constructs such as aggregates or arithmetic terms can be debugged, and whether the approach can also be used to explain consistent logic programs. Table 4 complements this by illustrating whether the debugging approaches require an intended answer set, or rather, whether they detect mistakes with respect to potentially intended answer sets, which types of errors in a logic program the debugging approaches distinguish, and whether the user can or has to interact with the debugger.

In the following, we discuss some of the distinguishing features in more detail, to facilitate users to choose the appropriate debugging approach for their application.

4.6.1 Knowledge Representation versus Programming

As discussed by Cabalar (2011), logic programs under the answer set semantics are seen as a pure knowledge representation and reasoning formalism by some and as a programming language by others. It is therefore not surprising that explanation and debugging approaches reflect this difference. Seeing ASP as a knowledge representation formalism, a user represents knowledge in terms of a logic program and uses the answer set semantics to find out which conclusions can be drawn from this knowledge. The user may also represent a problematic situation and compute answer sets to find a solution to the problem. Especially in the latter of these two cases, the user most likely has no idea what the solution may be, in other words, there is no answer set intended by the user. On the other hand, if ASP is seen as a programming language, the user may well have an idea as to what the solution, i.e. the answer sets, should look like.

Taking these considerations into account, the `spock` approach (Section 4.1) may be more suitable for knowledge representation applications, as it does not require that the user specifies an intended answer set. Sets of literals are generated automatically as potential answer sets, which are then justified as to *why* they are

[37] The earlier version of the `stepping` approach (Oetsch et al. 2011) uses extended normal programs.

Table 3: Comparison of explanation approaches for inconsistent logic programs.

debugging approach	**type of logic program**	**variables supported**	**additional language constructs**	**explains consistent LPs**
`spock` transformation 1	normal LP	no	no	yes
`spock` transformation 2	LP	no	no	only non answer sets
`Ouroboros`	extended LP	yes	arithmetic, comparison	only non answer sets
interactive `spock`	LP	no	no	only non answer sets
DWASP	LP	yes	no	no
`stepping`	LP[37]	yes	aggregates, weight constraints, external atoms	yes

not actual answer sets. Similarly, the `stepping` approach (Section 4.5) does not require the user to have an answer set in mind as applicable rules are automatically determined and the user can then freely choose which one to use. However, both approaches allow the user to guide the explanation towards specific literals that may be expected in an answer set.

The interactive debugging approaches (Sections 4.3 and 4.4) take a programming language rather than a knowledge representation view on ASP, as they assume that the user has at least some idea as to what an answer set should look like, querying the user about the expected truth values of some literals. The user can certainly choose these truth values at random, making the interactive approaches applicable even if the user has no answer set in mind. However, this is not their intended usage. Note also that in order to know the truth value of a literal chosen by the debugging approach, the user essentially has to have an answer set in mind, as the user does not know upfront which literal will be chosen as a query.

The `Ouroboros` approach (Section 4.2) is clearly on the programming language end of the spectrum as it requires the user to specify a complete intended answer set. The user could of course choose an 'intended' answer set at random, but, again, this is not the usage envisaged by this approach.

Table 4: Comparison of explanation approaches for inconsistent logic programs (continued).

debugging approach	**intended answer set**	**error types**	**user interaction**
`spock` transformation 1	possible but not required (automatically generated)	unsatisfied rule, unsupported atom, unfounded atom	possible
`spock` transformation 2	possible but not required (automatically generated)	unsatisfied rule/constraint, unsupported atom, unfounded atom	possible
`Ouroboros`	required	unsatisfied rule/constraint, unfounded atom	required for intended answer set
interactive `spock`	possible but not required	unsatisfied rule/constraint, unsupported atom, unfounded atom	required
DWASP	possible but not required	minimal unsatisfiable core	required
`stepping`	not required but (indirectly) possible	unsatisfiability of rules, conflicting truth value of atoms	required

4.6.2 Error Classification

As in the case of justifications for consistent logic programs, the debugging approaches also differ regarding the elements used for explaining the inconsistency. More precisely, they identify different types of 'errors' causing a set of literals to not be an answer set. Broadly speaking, two different ideas towards errors can be distinguished: the classification of errors into different classes or the reduction of all errors two one 'class'.

DWASP and the `stepping` approach do not use any named error classes, thus following the latter idea. In DWASP errors are sets of rules that, when blocked, make the program consistent. However, there is no further explanation as to *why* this is the case. On the other hand, errors in the `stepping` approach are only indirectly specified. They are indicated by (partial) truth assignments to literals, which lead to a contradiction. Again, there is no further explanation, other than

the rule causing the contradiction. In contrast, the other approaches reviewed here distinguish different classes of errors.

The `spock` system and the two approaches based on it (interactive debugging and `Ouroboros`) use mostly the same classes of errors. As previously explained, these are violations of the definition of answer sets by Lin and Zhao (2004) and Lee (2005) (see Definition 32 on page 65), namely unsatisfied rules, unsupported atoms, and unfounded atoms.

Interestingly, one reason for inconsistency of logic programs often discussed in the literature (You and Yuan 1994; Syrjänen 2006; Costantini 2006; Schulz et al. 2015) is not explicitly pointed out by `spock`, namely *odd-length (negative dependency) cycles*. In Examples 52 and 55 (see pages 68 and 72), the odd-length cycle in r_2 of P_{29} is only indirectly pointed out: M_{26} expresses that $\{b\}$ is not an answer set of P_{29} since all rules with head b are blocked by $\{b\}$. Taking a closer look at P_{29}, we realise that the only rule with head b is r_2 and that the reason for it being blocked is that *not* b is in the body of r_2. However, if P_{29} was a large logic program, it would be infeasible to check all rules with head b to find out that one of them may comprise an odd-length cycle, causing the rule to be blocked. Similarly, M_{28} indirectly points out the odd-length cycle by stating that r_2 is applicable but its head is not contained in the set $\{a\}$. We then realise that the reason for r_2 not being satisfied is the odd-length cycle.

Example 62
Let P_{33} be the inconsistent logic program with:

$$r_1 : \ a \leftarrow b \qquad\qquad r_2 : \ b \leftarrow \mathit{not}\ a \tag{81}$$

The answer sets of $\mathcal{T}_k[P_{33}] \cup \mathcal{T}_{ex}[P_{33}]$ (when using minimisation) are:

- $M_{42} = \{a, b, \mathsf{unsupported}(b), \mathsf{applicable}(r_1), \mathsf{blocked}(r_2)\}$
- $M_{43} = \{a, \mathsf{unsupported}(a), \mathsf{blocked}(r_1), \mathsf{blocked}(r_2)\}$
- $M_{44} = \{b, \mathsf{unsatisfied}(r_1), \mathsf{applicable}(r_1), \mathsf{applicable}(r_2)\}$
- $M_{45} = \{\mathsf{unsatisfied}(r_2), \mathsf{blocked}(r_1), \mathsf{applicable}(r_2)\}$

None of the answer sets captures the fact that there is an odd-length cycle $a \leftarrow \mathit{not}\ a$. For a similarly structured logic program with more rules and derivation steps between a and *not* a it would therefore be difficult to identify that the reason of the inconsistency is an odd-length cycle. □

A debugging approach related to `spock` (Syrjänen 2006) explicitly points out inconsistencies due to odd-length cycles. The approach also uses the input transformation $\mathcal{P}_{inp}[P]$ of a logic program together with a meta-encoding of two types of errors: odd-length cycles and violated constraints. However, *all* odd-length cycles are considered as faulty, even though some odd-length cycles do not cause a logic program to be inconsistent. In contrast to the `spock` system, faults are pointed out independent of intended or potential answer sets.

Another class of 'errors' not considered in any of the debugging approaches are those of contradictory answer sets. In fact, none of the debugging approaches reviewed here deals with contradictory atoms in an answer set. Schulz et al. (2015)

show that logic programs with contradictory answer sets include different types of semantic errors than inconsistent logic programs. This is also taken into account in the inconsistency measurements of Ulbricht et al. (2016).

4.6.3 Large and Real-World Logic Programs

We already hinted at the fact that the different debugging approaches require various levels of user interaction to obtain an explanation. In particular, some approaches require the user to specify an intended answer set before starting the debugging process, especially the `Ouroboros` system. This can be difficult if faced with a large logic program, potentially comprising hundreds of atoms. Furthermore, using the `stepping` approach, the user has to step through every single applicable rule, unless being sure that some rules are not problematic, in which case the jumping feature can be used. Assuming that the user does not have any idea why the logic program is inconsistent, thus ruling out jumping, the stepping approach can take a long time and also be prone to errors for these large programs.

In contrast, for approaches requiring only little user interaction, first and foremost the `spock` system, the amount of interaction does not increase when dealing with large logic programs. However, note that the more literals occur in a program, the more explanations are computed by `spock`, namely one for each potential answer set. The user interaction is thus implicitly required after explanations are computed, since the user then has to decide which explanations to take into account. It follows, that, just like the `Ouroboros` and `stepping` approaches, using spock with large logic programs may take a long time.

The two interactive approaches (the one based on `spock` and the DWASP system) are the ones that require least user interaction when handling large logic programs. This is because queries are determined in such a way that the user's answer provides maximal information gain. Consequently, the total number of queries generated is as small as possible. From a user's point of view, answering a query on the expected truth value of a single literal may furthermore be easier than specifying the truth value of all literals at once or choosing a meaningful explanation from all the ones generated.

When using ASP in practice, logic programs often include additional language constructs, make use of variables, and are seldom limited to normal rules. These are important consideration when choosing a debugging approach. Currently, `Ouroboros` and the `stepping` approach are the only ones to handle both negation-as-failure and explicit negation, variables, and additional language constructs, where the `stepping` approach supports more constructs than `Ouroboros`. DWASP supports variables, but to the best of our knowledge no explicit negation or additional language constructs. Nevertheless, is to be assumed that these will be supported in the future since DWASP is implemented in terms of the ASP solver WASP, which is able to handle these.

5 Related Work

In this survey, we focussed on justification and debugging approaches for logic programs under the *answer set* semantics. Historically, the concept of justifications can be traced back to the works of Shapiro (1983) and Sterling and Lalee (1986), where they have been used as a means for identifying bugs in programs. Later, Lloyd (1987) introduced the notions of uncovered atoms and incorrect rules under the completion semantics (Clark 1978) while Sterling and Yalçinalp (1989) explained Prolog expert systems using a meta-interpreter.

An important notion for understanding errors in ASP is the concept of a supported set of atoms, which was introduced by Pereira et al. (1991) and further elaborated by Pereira et al. (1993). Another important concept is the notion of assumptions, which was introduced for truth maintenance systems by de Kleer (1986) and developed for logic programming by Pereira et al. (1993). Specht (1993) presented one of the first techniques to compute complete proof trees for bottom-up evaluation of database systems by means of a program transformation. Further techniques for computing justifications or explanations for Prolog by means of meta-interpreters or program transformations can be found in (Sterling and Shapiro 1994) and (Bratko 2001). Furthermore, explanation approaches have been developed for knowledge representation paradigms related to ASP. For instance, Arora et al. (1993) present explanations for deductive databases and Ferrand et al. (2006) for constraint logic programs and constraint satisfaction problems.

Regarding justifications for logic programs under the answer set semantics, Brain and De Vos (2005) were one of the first to tackle this issue, by presenting two algorithms for producing natural language explanations as to why a (set of) literal(s) is or is not part of an answer set. In the first case, applicable rules are provided as an explanation, whereas in the second case contradictions (concerning the truth values of atoms) are pointed out.

Off-line justifications (Pontelli and Son 2006; Pontelli et al. 2009), as reviewed in Section 3.1, use graphs as justifications, expressing why an atoms is (not) contained in a given answer set. This approach can be traced back to tabled justifications for Prolog (Roychoudhury et al. 2000; Pemmasani et al. 2003). Albrecht et al. (2013) further show how off-line explanation graphs can be constructed from a graphical representation of logic programs called extended dependency graph. The root of causal justifications can be traced back to (Cabalar 2011), where an extension of the stable semantics with causal proofs was introduced, and (Cabalar and Fandinno 2013), where an algebraic characterisation of this semantics was developed. Argumentation-based answer set justifications (Schulz et al. 2013) are a predecessor of LABAS justifications. They share the argumentative flavour of LABAS justifications but use a slightly different way of constructing arguments and justifications.

Erdem and Öztok (2015) use ASP to construct explanations for biomedical queries. These explanations have a tree structure expressing derivations of a literal in question and have a close relationship with off-line justifications. Lifschitz (2017) introduces a methodology that facilitates the design of encodings that are

easy to understand and provably correct. In addition to the implementations of justification and debugging approaches reviewed here, Perri et al. (2007) integrate an explanation and debugging component into the DLV solver.

As we saw throughout this survey, many justification approaches construct a graphical explanation. Graph representations of logic programs have also been extensively studied for other purposes (Costantini et al. 2002; Costantini and Provetti 2010). Graphs can for instance be useful for the computation of answer sets, as is the purpose of attack graphs (Dimopoulos and Torres 1996), rule graphs (Dimopoulos 1996), and block graphs (Linke 2001) and their extensions (Linke and Sarsakov 2004; Konczak et al. 2006). Furthermore, Costantini (2001) and Costantini and Provetti (2011) study desirable properties of graphs representing logic programs and Costantini (2006) uses cycle graphs to prove conditions for the existence of answer sets.

Various IDEs for ASP also make use of graphical representations of logic programs or visualise dependencies between literals to help the user understand a problem represented as a logic program. For example, for the DLV solver a visual computation tracing feature (Calimeri et al. 2009) as well as a dependency graph feature (Febbraro et al. 2011) have been developed. Furthermore, the VIDEAS system (Oetsch et al. 2011) uses entity relationship graphs of logic programs for model-driven engineering in ASP and, in the 'Visual ASP' system (Febbraro et al. 2010), the user can draw a graph, which is then translated into a logic program.

6 Conclusion

Lifschitz (2010) lists thirteen different definitions of the concept of answer set (and points out that even more exist). These definitions are equivalent (at least for normal programs), but provide alternative points of view on the intuitive meaning of logic programs or lead to different algorithms for generating answer sets. In this sense, it is not surprising that there exist several ways of explaining the solutions to consistent programs and the errors in inconsistent ones. In this survey, we have reviewed and compared the most prominent explanation approaches for both consistent and inconsistent logic programs under the answer set semantics and pointed out their differences and similarities. These approaches try to answer important 'why'-questions regarding answer sets, namely *why* a set of literals is or is not an answer set, or *why* a logic program is inconsistent. Approaches aiming at answering the first question for consistent logic programs are referred to as *justification* approaches, while explanation approaches trying to answer the second question for inconsistent logic programs are referred to as *debugging* approaches. The latter take a more global view than justification approaches: in debugging approaches the explanation is w.r.t. a whole *set* that is not an answer set, whereas in most justification approaches the explanation is w.r.t. one *literal* that is (not) in an answer set.

As we have seen in Sections 3.6 and 4.6, the different justification and debugging approaches suffer from various issues. Building upon these observations, in the following we suggest some considerations for future research that are mainly

independent of philosophical choices made by different approaches. These are particularly important in the light of the European Union's new General Data Protection Regulation (GDPR), which states that explanations should consist of "meaningful information about the logic involved" and be "concise, intelligible and easily accessible" (Goodman and Flaxman 2016). Since the approaches discussed here construct explanations based on the logical connection between rules and literals leading to the existence of a particular answer set or to inconsistency, at least the first part of the first GDPR condition, i.e. "information about the logic involved", can be deemed satisfied by these approaches. The proposed directions of research are as follows:

- Number of explanations (tackling the conciseness and intelligibility required by the GDPR): As previously discussed, most justification and debugging approaches suffer from a large number of possible explanations when dealing with large programs with, potentially, many (and long) dependencies between literals. This is not feasible in practice, so a method for choosing the most suitable explanation(s) is needed. This could for example be tackled by querying the user as in DWASP and the interactive `spock` approach.
- Size of explanations (tackling meaningfulness of information, conciseness, intelligibility, and easy accessibility required by the GDPR): A related problem is the growth in size, from which many of the justification approaches suffer. Large explanations are infeasible in many practical applications, since they make it difficult for the user to understand the explanation. The development of techniques for collapsing less important parts of an explanation provides a challenging topic for the future.
- Language constructs and variables: We have seen that, especially among the justification approaches, there is little support for logic programs that contain language constructs such as aggregates, weight constraints, etc. Many approaches are not even able to efficiently handle variables. In order to apply explanations in practice, these issues will have to be addressed.
- Cross-fertilisation of justification and debugging: Most current approaches either focus on justifying consistent programs or debugging inconsistent programs. A first step towards the cross-fertilisation of the two was made by Damásio et al. (2015), who combine the second `spock` transformation approach with why-not provenance justifications.
- Going beyond debugging: Current debugging approaches merely point out errors in a program, leaving the fixing of these errors to the user. The automatic revision of inconsistent logic programs is thus an interesting, and challenging, topic for future investigations. A first step in this direction was made by Li et al. (2015), who use inductive logic programming to achieve a semi-automatic revision of logic programs.

Meeting the requirements of the GDPR will be a challenging task, especially since conditions like meaningfulness and intelligibility of information may have to be realised differently for ASP experts and non-experts. Applications of ASP explanation approaches will thus determine whether or not they meet the required conditions. In

this sense, an exciting prospect for the future is the combination of the advantages and minimisation the disadvantages of all the different approaches for answering a 'why'-question in answer set programming.

Acknowledgements We are thankful to the anonymous reviewers for their valuable feedback, which helped to improve the paper.

References

Albrecht, E., Krümpelmann, P., and Kern-Isberner, G. 2013. Construction of Explanation Graphs from Extended Dependency Graphs for Answer Set Programs. In *Revised Selected Papers of the Kiel Declarative Programming Days (KDPD'13)*. 1–16.

Alviano, M., Dodaro, C., Faber, W., Leone, N., and Ricca, F. 2013. WASP: A Native ASP Solver Based on Constraint Learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. 54–66.

Alviano, M., Dodaro, C., Leone, N., and Ricca, F. 2015. Advances in WASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. 40–54.

Arora, T., Ramakrishnan, R., Roth, W. G., Seshadri, P., and Srivastava, D. 1993. Explaining Program Execution in Deductive Systems. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases (DOOD'93)*. 101–119.

Balduccini, M. and Girotto, S. 2010. Formalization of Psychological Knowledge in Answer Set Programming and its Application. *Theory and Practice of Logic Programming 10*, 4-6, 725–740.

Béatrix, C., Lefèvre, C., Garcia, L., and Stéphan, I. 2016. Justifications and Blocking Sets in a Rule-Based Answer Set Computation. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP'16)*. 6:1–6:15.

Boenn, G., Brain, M., De Vos, M., and Fitch, J. P. 2011. Automatic Music Composition Using Answer Set Programming. *Theory and Practice of Logic Programming 11*, 2-3, 397–427.

Brain, M. and De Vos, M. 2005. Debugging Logic Programs under the Answer Set Semantics. In *Proceedings of the 3rd Workshop on Answer Set Programming, Advances in Theory and Implementation (ASP'05)*.

Brain, M. and De Vos, M. 2008. Answer Set Programming - a Domain in Need of Explanation: A Position Paper. In *Proceedomgs of the 3rd International Workshop on Explanation-aware Computing (ExaCt'08)*. 37–48.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. 2007a. Debugging ASP Programs by Means of ASP. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. 31–43.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. 2007b. "That is illogical captain!" - The debugging support tool spock for answer-set programs: System description. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*. 71–85.

Bratko, I. 2001. *Prolog programming for artificial intelligence*. Pearson education.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM 54*, 12, 92–103.

Busoniu, P.-A., Oetsch, J., Pührer, J., Skocovsky, P., and Tompits, H. 2013. SeaLion: An Eclipse-Based IDE for Answer-Set Programming with Advanced Debugging Support. *Theory and Practice of Logic Progrmming 13,* 4-5, 657–673.

Cabalar, P. 2011. Answer Set; Programming? In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*. 334–343.

Cabalar, P. and Fandinno, J. 2013. An algebra of causal chains. *CoRR abs/1312.6134*.

Cabalar, P. and Fandinno, J. 2016. Justifications for programs with disjunctive and causal-choice rules. *Theory and Practice of Logic Programming 16,* 5-6, 587–603.

Cabalar, P. and Fandinno, J. 2017. Enablers and Inhibitors in Causal Justifications of Logic Programs. *Theory and Practice of Logic Programming 17,* 1, 49–74.

Cabalar, P., Fandinno, J., and Fink, M. 2014. Causal Graph Justifications of Logic Programs. *Theory and Practice of Logic Programming 14,* 4-5, 603–618.

Cabalar, P., Fandiño, J., and Fink, M. 2014. A complexity assessment for queries involving sufficient and necessary causes. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA'14)*. Lecture Notes in Computer Science, vol. 8761. Springer, 297–310.

Calimeri, F., Leone, N., Ricca, F., and Veltri, P. 2009. A Visual Tracer for DLV. In *Proceedings of the 2nd International Workshop on Software Engineering for Answer Set Programming (SEA'09)*. 79–93.

Clark, K. L. 1978. Negation as failure. In *Logic and data bases*. Springer, 293–322.

Costantini, S. 2001. Comparing Different Graph Representations of Logic Programs under the Answer Set Semantics. In *Proceedings of the 1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning (ASP'01)*.

Costantini, S. 2006. On the Existence of Stable Models of Non-Stratified Logic Programs. *Theory and Practice of Logic Programming 6,* 1-2, 169–212.

Costantini, S., D'Antona, O., and Provetti, A. 2002. On the Equivalence and Range of Applicability of Graph-based Representations of Logic Programs. *Information Processing Letters 84,* 5, 241–249.

Costantini, S. and Provetti, A. 2010. Graph Representations of Logic Programs: Properties and Comparison. In *Proceedings of the 6th Latin American Workshop on Non-Monotonic Reasoning*. 1–14.

Costantini, S. and Provetti, A. 2011. Conflict, Consistency and Truth-Dependencies in Graph Representations of Answer Set Logic Programs. In *Revised Selected Papers of the 2nd International Workshop on Graph Structures for Knowledge Representation and Reasoning (GKR'11)*. 68–90.

Damásio, C. V., Analyti, A., and Antoniou, G. 2013. Justifications for Logic Programming. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. 530–542.

Damásio, C. V., Moura, J., and Analyti, A. 2015. Unifying Justifications and Debugging for Answer-Set Programs. In *Technical Communications of the 31st International Conference on Logic Programming (ICLP'15)*.

Damásio, C. V., Pires, J. M., and Analyti, A. 2015. Unifying justifications and debugging for answer-set programs. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP'15)*, M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, Eds. CEUR Workshop Proceedings, vol. 1433. CEUR-WS.org.

de Kleer, J. 1986. An assumption-based tms. *Artificial Intelligence 28,* 2, 127 – 162.

Denecker, M., Brewka, G., and Strass, H. 2015. A Formal Theory of Justifications. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. 250–264.

Denecker, M. and De Schreye, D. 1993. Justification Semantics: A Unifiying Framework for the Semantics of Logic Programs. In *Proceedings of the 2nd International Workshop on Logic Programming and Non-monotonic Reasoning (LPNMR'93)*. 365–379.

Dimopoulos, Y. 1996. On Computing Logic Programs. *Journal of Automated Reasoning 17,* 3, 259–289.

Dimopoulos, Y. and Torres, A. 1996. Graph Theoretical Structures in Logic Programs and Default Theories. *Theoretical Computer Science 170,* 1-2, 209–244.

Dodaro, C., Gasteiger, P., Musitsch, B., Ricca, F., and Shchekotykhin, K. M. 2015. Interactive Debugging of Non-ground ASP Programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. 279–293.

Dung, P. M., Kowalski, R. A., and Toni, F. 2009. Assumption-Based Argumentation. In *Argumentation in Artificial Intelligence*, G. R. Simari and I. Rahwan, Eds. Springer US, 199–218.

El-Khatib, O., Pontelli, E., and Son, T. C. 2005. Justification and Debugging of Answer Set Programs in ASP - Prolog. In *Proceedings of the 6th International Workshop on Automated Debugging (AADEBUG'05)*. 49–58.

Erdem, E. and Öztok, U. 2015. Generating Explanations for Biomedical Queries. *Theory and Practice of Logic Programming 15,* 1, 35–78.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and Complexity of Recursive Aggregates in Answer Set Programming. *Artificial Intelligence 175,* 1, 278–298.

Fandinno, J. 2016a. Deriving conclusions from non-monotonic cause-effect relations. *Theory and Practice of Logic Programming 16,* 5-6, 670–687.

Fandinno, J. 2016b. Towards deriving conclusions from cause-effect relations. *Fundamenta Informaticae 147,* 1, 93–131.

Febbraro, O., Reale, K., and Ricca, F. 2010. A Visual Interface for Drawing ASP Programs. In *Proceedings of the 25th Italian Conference on Computational Logic (CILC'10)*.

Febbraro, O., Reale, K., and Ricca, F. 2011. ASPIDE: Integrated Development Environment for Answer Set Programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. 317–330.

Ferrand, G., Lesaint, W., and Tessier, A. 2006. Explanations and Proof Trees. *Computers and Informatics 25,* 2-3, 105–122.

Frühstück, M., Pührer, J., and Friedrich, G. 2013. Debugging Answer-Set Programs with Ouroboros - Extending the SeaLion Plugin. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. 323–328.

Gasteiger, P., Dodaro, C., Musitsch, B., Reale, K., Ricca, F., and Schekotihin, K. 2016. An integrated Graphical User Interface for Debugging Answer Set Programs. In *Proceedings of the Workshop on Trends and Applications of Answer Set Programming (TAASP'16)*.

Gebser, M., Pührer, J., Schaub, T., and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'18)*, D. Fox and C. P. Gomes, Eds. AAAI Press, 448–453.

Gebser, M., Schaub, T., Thiele, S., and Veber, P. 2011. Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. *Theory and Practice of Logic Programming 11,* 2-3, 323–360.

Gelfond, M. 2008. Answer Sets. In *Handbook of Knowledge Representation*. 285–316.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the 5th International Conference and Symposium (Volume 2)*.

Gelfond, M. and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9,* 3/4, 365–386.

Goodman, B. and Flaxman, S. 2016. European union regulations on algorithmic decision-making and a "right to explanation". *arXiv preprint arXiv:1606.08813*.

Green, T. J., Karvounarakis, G., and Tannen, V. 2007. Provenance semirings. In *Proceedings of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, L. Libkin, Ed. ACM, 31–40.

Hall, N. 2004. Two concepts of causation. In *Causation and counterfactuals*, J. Collins, N. Hall, and L. A. Paul, Eds. Cambridge, MA: MIT Press, 225–276.

Hall, N. 2007. Structural equations and causation. *Philosophical Studies 132,* 1, 109–136.

Halpern, J. Y. 2008. Defaults and normality in causal structures. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, G. Brewka and J. Lang, Eds. AAAI Press, 198–208.

Hitchcock, C. and Knobe, J. 2009. Cause and norm. *Journal of Philosophy 11*, 587–612.

Inclezan, D. 2015. An Application of Answer Set Programming to the Field of Second Language Acquisition. *Theory and Practice of Logic Programming 15,* 01, 1–17.

Konczak, K., Linke, T., and Schaub, T. 2006. Graphs and Colorings for Answer Set Programming. *Theory and Practice of Logic Programming 6,* 1-2, 61–106.

Lee, J. 2005. A Model-Theoretic Counterpart of Loop formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*. 503–508.

Lefèvre, C., Béatrix, C., Stéphan, I., and Garcia, L. 2017. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming 17,* 3, 266–310.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Lewis, D. K. 1973. Causation. *The journal of philosophy 70,* 17, 556–567.

Li, T., De Vos, M., Padget, J., Satoh, K., and Balke, T. 2015. Debugging ASP using ILP. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP'15)*.

Lifschitz, V. 2008. What Is Answer Set Programming? In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'08)*. 1594–1597.

Lifschitz, V. 2010. Thirteen definitions of a stable model. In *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, A. Blass, N. Dershowitz, and W. Reisig, Eds. Lecture Notes in Computer Science, vol. 6300. Springer, 488–503.

Lifschitz, V. 2017. Achievements in answer set programming. *Theory and Practice of Logic Programming 17,* 5-6, 961–973.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence 157,* 1-2, 115–137.

Linke, T. 2001. Graph Theoretical Characterization and Computation of Answer Sets. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI'01)*. 641–648.

Linke, T. and Sarsakov, V. 2004. Suitable Graphs for Answer Set Programming. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*. 154–168.

Lloyd, J. W. 1987. Declarative error diagnosis. *New Generation Computing 5,* 2 (Jun), 133–154.

Maudlin, T. 2004. Causation, counterfactuals, and the third factor. In *Causation and Counterfactuals*, J. Collins, E. J. Hall, and L. A. Paul, Eds. MIT Press.

McCarthy, J. 1977. Epistemological problems of Artificial Intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. MIT Press, Cambridge, MA, 1038–1044.

McCarthy, J. 1998. Elaboration tolerance. In *Proceedings of the 4th Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense'98)*. London, UK, 198–217. Updated version at `http://www-formal.stanford.edu/jmc/elaboration.ps`.

Oetsch, J., Pührer, J., Seidl, M., Tompits, H., and Zwickl, P. 2011. VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. 382–387.

Oetsch, J., Pührer, J., and Tompits, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming 10,* 4-6, 513–529.

Oetsch, J., Pührer, J., and Tompits, H. 2011. Stepping through an Answer-Set Program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. 134–147.

Oetsch, J., Pührer, J., and Tompits, H. 2012. An FLP-Style Answer-Set Semantics for Abstract-Constraint Programs with Disjunctions. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*. 222–234.

Oetsch, J., Pührer, J., and Tompits, H. 2018. Stepwise Debugging of Answer-Set Programs. *Theory and Practice of Logic Programming 18,* 1, 30–80.

Parliament and Council of the European Union. 2016. *Regulation (EU) 2016/679: General Data Protection Regulation.*

Pemmasani, G., Guo, H., Dong, Y., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2003. Online justification for tabled logic programs. In *Proceedings of the 19th International Conference on Logic Programming (ICLP'03)*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, 500–501.

Pereira, L. M. and Alferes, J. J. 1992. Well founded semantics for logic programs with explicit negation. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*. 102–106.

Pereira, L. M., Alferes, J. J., and Aparício, J. N. 1991. Contradiction removal within well founded semantics. In *Proceedings of the 1st International Workshop on Logic Programming and Non-monotonic Reasonin (LPNMR'91)*.

Pereira, L. M., Aparcio, J. N., and Alferes, J. J. 1993. Non-monotonic reasoning with logic programming. *The Journal of Logic Programming 17,* 2, 227 – 263. Special Issue: Non-Monotonic Reasoning and Logic Programming.

Pereira, L. M., Damásio, C. V., and Alferes, J. J. 1993. Debugging by diagnosing assumptions. In *International Workshop on Automated and Algorithmic Debugging*. Springer, 58–74.

Perri, S., Ricca, F., Terracina, G., Cianni, D., and Veltri, P. 2007. An Integrated Graphic Tool for Developing and Testing DLV Programs. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*. 86–100.

Polleres, A., Frühstück, M., Schenner, G., and Friedrich, G. 2013. Debugging Non-ground ASP Programs with Choice Rules, Cardinality and Weight Constraints. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. 452–464.

Pontelli, E. and Son, T. C. 2006. Justifications for Logic Programs Under Answer Set Semantics. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP'06)*. 196–210.

Pontelli, E., Son, T. C., and El-Khatib, O. 2009. Justifications for Logic Programs under Answer Set Semantics. *Theory and Practice of Logic Programming 9,* 1, 1–56.

Pührer, J. 2014. Stepwise Debugging in Answer-Set Programming: Theoretical Foundations and Practical Realisation. Ph.D. thesis.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. 2012. Team-Building with Answer Set Programming in the Gioia-Tauro Seaport. *Theory and Practice of Logic Programming 12,* 3, 361–381.

Roychoudhury, A., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2000. Justifying proofs using memo tables. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'00)*. 178–189.

Schulz, C. 2017. Developments in abstract and assumption-based argumentation and their application in logic programming. Ph.D. thesis, Imperial College London.

Schulz, C., Satoh, K., and Toni, F. 2015. Characterising and Explaining Inconsistency in Logic Programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. 467–479.

Schulz, C., Sergot, M., and Toni, F. 2013. Argumentation-Based Answer Set Justification. In *Proceedings of the 11th International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense'13)*.

Schulz, C. and Toni, F. 2013. ABA-Based Answer Set Justification. *Theory and Practice of Logic Programming 13,* 4-5-Online-Supplement.

Schulz, C. and Toni, F. 2015. Logic Programming in Assumption-Based Argumentation Revisited - Semantics and Graphical Representation. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*. 1569–1575.

Schulz, C. and Toni, F. 2016. Justifying Answer Sets using Argumentation. *Theory and Practice of Logic Programming 16,* 01, 59–110.

Shapiro, E. Y. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA.

Shchekotykhin, K. M. 2015. Interactive Query-Based Debugging of ASP Programs. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*. 1597–1603.

Specht, G. 1993. Generating explanation trees even for negations in deductive database systems. In *Proceedings of the 5th Workshop on Logic Programming Environments (LPE'93)*, M. Ducassé, B. L. Charlier, Y. Lin, and L. Ü. Yalçinalp, Eds. IRISA, Campus de Beaulieu, France, 8–13.

Sterling, L. and Lalee, M. 1986. An explanation shell for expert systems. *Computational Intelligence 2,* 1, 136–141.

Sterling, L. and Shapiro, E. Y. 1994. *The art of Prolog: advanced programming techniques*. MIT press.

Sterling, L. and Yalçinalp, L. Ü. 1989. Explaining prolog based expert systems using a layered meta-interpreter. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'89)*. 66–71.

Sureshkumar, A., De Vos, M., Brain, M., and Fitch, J. 2007. APE: An AnsProlog* environment. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*. 101–115.

Syrjänen, T. 2006. Debugging Inconsistent Answer Set Programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR'06)*. 77–84.

Syrjänen, T. and Niemelä, I. 2001. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. 434–438.

Ulbricht, M., Thimm, M., and Brewka, G. 2016. Measuring Inconsistency in Answer Set Programs. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA'16)*. 577–583.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23,* 4, 733–742.

Van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 1–10.

Van Gelder, A., Ross, K., and Schlipf, J. S. 1988. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 221–230.

Van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM (JACM) 38,* 3, 619–649.

You, J.-H. and Yuan, L. Y. 1994. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences 49,* 2, 334–361.