Justifications for Logic Programs under Answer Set Semantics

Enrico Pontelli, Tran Cao Son

Dept. Computer Science
New Mexico State University
epontell|tson@cs.nmsu.edu

Abstract. The paper introduces the notion of *off-line justification* for Answer Set Programming (ASP). Justifications provide a graph-based explanation of the truth value of an atom w.r.t. a given answer set. The notion of justification accounts for the specifics of answer set semantics. The paper extends also this notion to provide justification of atoms *during* the computation of an answer set (*on-line justification*), and presents an integration of on-line justifications within the computation model of SMODELS. Justifications offer a basic data structure to support methodologies and tools for *debugging* answer set programs. A preliminary implementation has been developed in $\mathbb{ASP} - \mathbb{PROLOG}$.

1 Introduction

Answer set programming (ASP) is a programming paradigm [13, 19] based on logic programming under answer set semantics [9]. ASP is *highly declarative*; to solve a problem P, we specify it as a logic program $\pi(P)$ whose answer sets correspond one-to-one to solutions of P, and can be computed using an answer set solver. ASP is also attractive because of its numerous building block results (see, e.g., [4]).

A source of difficulties in ASP lies in the lack of *methodologies* for program understanding and debugging. The declarative and the hands-off execution style of ASP leave a programmer with nothing that helps in explaining the behavior of the programs, especially for unexpected outcomes of the computation (e.g., incorrect answer sets).

Although ASP is syntactically close to Prolog, the execution model and the semantics are sufficiently different to make debugging techniques developed for Prolog impractical. E.g., traditional *trace-based* debuggers [17] (e.g., Prolog four-port debuggers), used to trace the entire proof search tree (paired with execution control mechanisms, like spy points and step execution), are cumbersome in ASP, since:

- Trace-based debuggers provide the entire search sequence, including failed paths, which are irrelevant in understanding specific elements of an answer set.
- The process of computing answer sets is bottom-up, and the determination of the truth value of one atom is intermixed with the computation of other atoms; a direct tracing makes it hard to focus on what is relevant to one particular atom.
- Tracing repeats previously performed executions, degrading debugging performance.

In this paper, we address these issues by elaborating the concept of *off-line justification* for ASP. This notion is an evolution of the concept of *justification*, proposed to justify

truth values in tabled Prolog [17, 14]. Intuitively, an off-line justification of an atom w.r.t. an answer set is a graph encoding the reasons for the atom's truth value. This notion can be used to explain the presence or absence of an atom in an answer set, and provides the basis for building a *justifier* for answer set solvers.

The notion of off-line justification is helpful when investigating the content of one (or more) answer sets. When the program does not have answer sets, a different type of justification is needed. We believe it is impractical to rely on a single justification structure to tackle this issue; we prefer, instead, to provide the programmer with a *dynamic* data structure that will help him/her discover the sources of inconsistencies. The data structure we propose is called *on-line justification*, and it provides justifications with respect to a *partial* and/or *inconsistent* interpretation. The intuition is to allow the programmer to interrupt the computation (e.g., at the occurrence of certain events, such as assignment of a truth value to a given atom) and to use the on-line justification to explore the motivations behind the content of the partial interpretation (e.g., why a given atom is receiving conflicting truth values). We describe a *generic* model of on-line justification and a version specialized to the execution model of SMODELS [19]. The latter has been implemented in $\mathbb{ASP} - \mathbb{PROLOG}$ [8].

Related work: Various approaches to logic program debugging have been investigated (a thorough comparison is beyond the limited space of this paper). As discussed in [14], 3 main phases can be considered in understanding/debugging a logic program. *(1) Program instrumentation and execution:* assertion-based debugging (e.g., [16]) and algorithmic debugging [18] are examples of approaches focused on this first phase. *(2) Data Collection:* focuses on *extracting* from the execution data necessary to understand it, as in event-based debugging [3] and explanation-based debugging [7, 12]. *(3) Data Analysis:* focuses on reasoning on data collected during the execution. The proposals dealing with automated debugging (e.g., [3]) and execution visualization (e.g., [21]) are approaches focusing on this phase of program understanding.

The notion of *Justification* has been introduced in [17, 14, 20] to support understanding and debugging of Prolog programs. Justification is the process of generating evidence, in terms of high-level proofs based on the answers (or models) produced during the computation. Justification plays an important role in manual and automatic verification, by providing a *proof description* if a given property holds; otherwise, it generates a *counter-example*, showing where the violation/conflict occurs in the system. The justification-based approach focuses on the last two phases of debugging—collecting data from the execution and presenting them in a meaningful manner. Justifications are focused only on parts of the computation relevant to the justified item. Justifications are fully automated and do not require user interaction (as in declarative debugging).

Our work shares some similarities with the proposals that employ graph structures to guide computation of answer sets (e.g., [1, 6]), although they use graphs for program representation, instead of using graphs to justify an execution.

2 Preliminary Definitions

In this paper, we focus on a logic programming language with negation as failure—e.g., the language of SMODELS without weight constraints [19].

The Language: Let $\Sigma_P = \langle \mathcal{F}, \Pi \rangle$ be a signature, where $\mathcal{F}$ is a finite set of constants and Π is a finite set of predicate symbols. In particular, we assume that $\top$ (stands for $true$) and $\bot$ (stands for $false$) are zero-ary predicates in Π. A *term* is a constant of $\mathcal{F}$. An atom is of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi$, and $t_1, \ldots, t_n$ are terms. In this paper, we deal with normal logic programs, i.e., logic programs that can make use of both positive and negation-as-failure literals. A literal is either an atom (*Positive Literal*) or $not\ a$ where a is an atom (*NAF Literal*). We will identify with $\mathcal{A}$ the set of all atoms, and with $\mathcal{L}$ the set of all literals. Our focus is on ground programs, as current ASP engines operate on ground programs. *Nevertheless*, programmers can write non-ground programs, and each rule represents the set of its ground instances.[1]

A rule is of the form $h \ :-\ b_1, \ldots, b_n$ where h is an atom and $\{b_1, \ldots, b_n\} \subseteq \mathcal{L}$. Given a rule r, we denote h with $head(r)$ and we use $body(r)$ to denote $\{b_1, \ldots, b_n\}$. We denote with $pos(r) = body(r) \cap \mathcal{A}$ and with $neg(r) = \{a \mid (not\ a) \in body(r)\}$. $NANT(P)$ denotes the atoms which appear in NAF literals in P—i.e., $NANT(P) = \{a \in \mathcal{A} \mid \exists r \in P.\, a \in neg(r)\}$.

Answer Set Semantics and Well-Founded Semantics: A *possible interpretation* (or *p-interpretation*) I is a pair $\langle I^+, I^- \rangle$, where $I^+ \cup I^- \subseteq \mathcal{A}$. For a p-interpretation I, we will use the notation I^+ and I^- to denote its two components. A *(three-valued) interpretation* I is a possible interpretation $\langle I^+, I^- \rangle$ where $I^+ \cap I^- = \emptyset$. I is a *complete interpretation* if $I^+ \cup I^- = \mathcal{A}$. For two p-interpretations I and J, $I \sqsubseteq J$ iff $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. A positive literal a is satisfied by I ($I \models a$) if $a \in I^+$. A NAF literal $not\ a$ is satisfied by I ($I \models not\ a$) if $a \in I^-$. A set of literals S is satisfied by I ($I \models S$) if I satisfies each literal in S. The notion of satisfaction is extended to rules and programs as usual.

For an interpretation I and a program P, the *reduct* of P w.r.t. I (P^I) is the program obtained from P by deleting *(i)* each rule r such that $neg(r) \cap I^+ \neq \emptyset$, and *(ii)* all NAF literals in the bodies of the remaining clauses. A complete interpretation I is an *answer set* [9] of P if I^+ is the least Herbrand model of P^I [2].

We will denote with $WF_P = \langle WF_P^+, WF_P^- \rangle$ the (unique) *well-founded model* [2] of program P (we omit its definition for lack of space).

Interpretations and Explanations: Let P be a program and I be an interpretation. An atom a is *true* (*false*, or *unknown*) in I if $a \in I^+$, ($a \in I^-$, or $a \notin I^+ \cup I^-$). $not\ a$ is true (false, unknown) in I if $a \in I^-$, ($a \in I^+$, $a \notin I^+ \cup I^-$). We will denote with $atom(\ell)$ the atom on which the literal ℓ is constructed.

We will now introduce some notations that we will use in the rest of the paper. The graphs used to explain will refer to the truth value assigned to an atom; furthermore, as we will see later, we wish to encompass those cases where an atom may appear as being both true and false (e.g., a conflict during construction of an answer set). For an atom a, we write a^+ to denote the fact that the atom is true, and a^- to denote the the fact that a is false. We will call a^+ and a^- the *annotated* versions of a; furthermore, we will define $atom(a^+) = a$ and $atom(a^-) = a$. For a set of atoms S, $S^p = \{a^+ \mid a \in S\}$,

[1] The visual representations of justifications (produced by the `draw/1` predicate—Sect. 5.3) show the original non-ground rules.

$S^n = \{a^- \mid a \in S\}$, and $not\ S = \{\ not\ a \mid a \in S\}$. In building the notion of justification, we will deal with labeled, directed graphs, called e-graphs.

Definition 1 (Explanation Graph). *For a program P, a labeled, directed graph (N, E) is called an* Explanation Graph *(or* e-graph*) if*

- $N \subseteq \mathcal{A}^p \cup \mathcal{A}^n \cup \{assume, \top, \bot\}$ *and*
- E *is a set of tuples of the form (p, q, s), with $p, q \in N$ and $s \in \{+, -\}$;*
- *the only sinks in the graph are: $assume$, $\top$, and $\bot$;*
- *for every $b \in N \cap \mathcal{A}^p$, $(b, assume, -) \notin E$ and $(b, \bot, -) \notin E$;*
- *for every $b \in N \cap \mathcal{A}^n$, $(b, assume, +) \notin E$ and $(b, \top, +) \notin E$;*
- *for every $b \in N$, if $(b, l, s) \in E$ for some $l \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$ then (b, l, s) is the only outgoing edge originating from b.*

Edges labeled $'+'$ are called *positive* edges, while those labeled $'-'$ are called *negative* edges. A path in an e-graph is *positive* if it contains only positive edges, while a path is negative if it contains at least one negative edge. We will denote with $(n_1, n_2) \in E^{*,+}$ the fact that there is a positive path from n_1 to n_2 in the given e-graph. The above definition allows us to define the notion of a support set of a node in an e-graph.

Definition 2. *Given an e-graph $G = (N, E)$ and a node $b \in N \cap (\mathcal{A}^p \cup \mathcal{A}^n)$,*

- $support(b, G) = \{atom(c) \mid (b, c, +) \in E\} \cup \{\ not\ atom(c) \mid (b, c, -) \in E\}$*, if for every $\ell \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$, $(b, \ell, s) \notin E$;*
- $support(b, G) = \{\ell\}$ *if $(b, \ell, s) \in E$ if $\ell \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$.*

The *local consistent explanation*[2] describes one step of justification for a literal. It describes the possible local reasons for the truth/falsity of a literal. If a is true, the explanation contains those bodies of the rules for a that are satisfied by I. If a is false, the explanation contains sets of literals that are false in I and they falsify all rules for a.

Definition 3 (Local Consistent Explanation). *Let b be an atom, J a possible interpretation, A a set of atoms (*assumptions*), and $S \subseteq \mathcal{A} \cup\ not\ \mathcal{A} \cup \{assume, \top, \bot\}$ a set of literals. We say that*

- S *is* a *local consistent explanation (LCE) of b^+ w.r.t. (J, A), if $S \cap \mathcal{A} \subseteq J^+$ and $\{c \mid\ not\ c \in S\} \subseteq J^- \cup A$, $b \in J^+$, and*
  - $S = \{assume\}$*, or*
  - *there is a rule r in P such that $head(r) = b$ and $S = body(r)$; for convenience, we write $S = \{\top\}$ to denote the case where $body(r) = \emptyset$.*
- *S is a local consistent explanation of b^- w.r.t. (J, A) if $S \cap \mathcal{A} \subseteq J^- \cup A$ and $\{c \mid\ not\ c \in S\} \subseteq J^+$, $b \in J^- \cup A$, and*
  - *$S = \{assume\}$; or*
  - *S is a minimal set of literals such that for every rule $r \in P$, if $head(r) = b$, then $pos(r) \cap S \neq \emptyset$ or $neg(r) \cap \{c \mid\ not\ c \in S\} \neq \emptyset$; for convenience, we write $S = \{\bot\}$ to denote the case $S = \emptyset$.*

[2] Note that our notion of local consistent explanation is similar in spirit, but different in practice from the analogous definition used in [17, 14].

We will denote with $LCE^p_P(b, J, A)$ the set of all the LCEs of b^+ w.r.t. (J, A), and with $LCE^n_P(b, J, A)$ the set of all the LCEs of b^- w.r.t. (J, A).

Example 1. Let P be the program:

$$\begin{array}{lll} a \ :\!-\ f, not\ b. & b \ :\!-\ e, not\ a. & e \ :\!-\ . \\ f \ :\!-\ e. & d \ :\!-\ c, e. & c \ :\!-\ d, f. \end{array}$$

This program has the answer sets $M_1 = \langle\{f, e, b\}, \{a, c, d\}\rangle$ and $M_2 = \langle\{f, e, a\}, \{c, b, d\}\rangle$. We have: $LCE^n_P(a, M_1, \emptyset) = \{\{\ not\ b\}\}$, $LCE^p_P(b, M_1, \emptyset) = \{\{e,\ \ not\ \ a\}\}$, $LCE^p_P(e, M_1, \emptyset) = \{\top\}$, $LCE^p_P(f, M_1, \emptyset) = \{\{e\}\}$, $LCE^n_P(d, M_1, \emptyset) = \{\{c\}\}$, $LCE^n_P(c, M_1, \emptyset) = \{\{d\}\}$. □

An e-graph is a general structure that can be used to explain the truth value of a, i.e., a positive (negative) e-graph represents a possible explanation for a being true (false). To select an e-graph as an acceptable explanation, we need two additional components: the current interpretation (J) and the collection (A) of elements that have been introduced in the interpretation without any "supporting evidence". An e-graph based on (J, A) is defined next.

Definition 4 ((J, A)-Based Explanation Graph). *Let P be a program, J a possible interpretation, A a set of atoms, and b an element in $\mathcal{A}^p \cup \mathcal{A}^n$. A (J, A)-based explanation graph $G = (N, E)$ of b is an e-graph such that*

- *every node $c \in N$ is reachable from b;*
- *for every $c \in N \setminus \{assume, \top, \bot\}$, $support(c, G)$ is an LCE of c w.r.t. (J, A).*

Definition 5. *A (J, A)-based e-graph (N, E) is* safe *if $\forall b^+ \in N$, $(b^+, b^+) \notin E^{*,+}$.*

Example 2. Consider the e-graphs in Figure 1, for the program of Example 1. We have that none of the e-graphs of a^+ ((i) and (ii)) is a $(M_1, \{c, d\})$-based e-graph of a^+ but both are $(M_2, \{b, c, d\})$-based e-graph of a^+. On the other hand, the e-graph of c^+ (iii) is neither a $(M_1, \{c, d\})$-based nor $(M_2, \{b, c, d\})$-based e-graph of c^+, while the e-graph of c^- (iv) is an a $(M_1, \{c, d\})$-based and a $(M_2, \{b, c, d\})$-based e-graph of c^-.

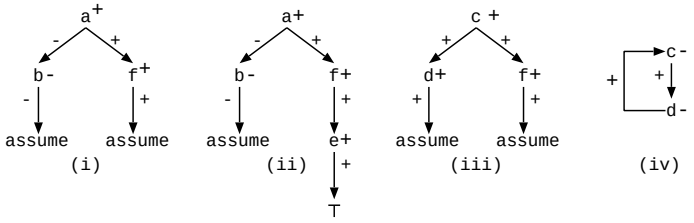

Fig. 1. Sample (J, A)-based Explanation Graphs

3 Off-line Justifications for ASP

Off-line justifications are employed to motivate the truth value of an atom w.r.t. a given (complete) answer set. If M is an answer set and WF_P the well-founded model of P,

then it is known that, $WF_P^+ \subseteq M^+$ and $WF_P^- \subseteq M^-$ [2]. Furthermore, we observe that the content of M is uniquely determined by the truth values assigned to certain atoms in $V = NANT(P) \setminus (WF_P^+ \cup WF_P^-)$, i.e., atoms that appear in negative literals and are not determined by the well-founded model. In particular, we are interested in those subsets of V with the following property: if all the elements in the subset are assumed to be false, then the truth value of all other atoms in $\mathcal{A}$ is uniquely determined. We call these subsets the *assumptions* of the answer set.

Definition 6 (Pre-Assumptions). *Let P be a program and M be an answer set of P. The* pre-assumptions *of P w.r.t. M (denoted by $\mathcal{PA}_P(M)$) are defined as:*

$$\mathcal{PA}_P(M) = \{a \mid a \in NANT(P) \ \wedge \ a \in M^- \ \wedge \ a \notin (WF_P^+ \cup WF_P^-)\}$$

The negative reduct of a program P w.r.t. a set of atoms A is a program obtained from P by forcing all the atoms in A to be false.

Definition 7 (Negative Reduct). *Let P be a program, M an answer set of P, and $A \subseteq \mathcal{PA}_P(M)$ a set of atoms. The* negative reduct *of P w.r.t. A, denoted by $NR(P, A)$, is the set of rules: $P \setminus \{ r \mid head(r) \in A\}$.*

Definition 8 (Assumptions). *Let P be a program and M be an answer set of P. An* assumption *w.r.t. M is a set of atoms A satisfying the following properties: (1) $A \subseteq \mathcal{PA}_P(M)$, and (2) the well-founded model of $NR(P, A)$ is equal to M—i.e., $WF_{NR(P,A)} = M$. We will denote with $Ass(P, M)$ the set of all assumptions of P w.r.t. M. A* minimal assumption *is an assumption that is minimal w.r.t. properties (1) and (2).*

We can observe that the set $Ass(P, M)$ is not empty, since $\mathcal{PA}_P(M)$ is an assumption.

Proposition 1. *Given an answer set M of P, the well-founded model of $NR(P, \mathcal{PA}_P(M))$ is equal to M.*

We will now specialize e-graphs to the case of answer sets, where only false elements can be used as assumptions.

Definition 9 (Off-line Explanation Graph). *Let P be a program, J a partial interpretation, A a set of atoms, and b an element in $\mathcal{A}^p \cup \mathcal{A}^n$. An* off-line explanation graph *$G = (N, E)$ of b w.r.t. J and A is a (J, A)-based e-graph of b satisfying the following conditions: there exists no $p^+ \in N$ such that $(p^+, assume, +) \in E$, and if $(p^-, assume, -) \in E$ then $p \in A$. $\mathcal{E}(b, J, A)$ denotes the set of all off-line explanation graphs of b w.r.t. J and A.*

Definition 10 (Off-line Justification). *Let P be a program, M an answer set, $A \in Ass(P, M)$, and $a \in \mathcal{A}^p \cup \mathcal{A}^n$. An* off-line justification *of a w.r.t. M and A is an element (N, E) of $\mathcal{E}(a, M, A)$ which is safe. $\mathcal{J}_P(a, M, A)$ contains all off-line justifications of a w.r.t. M and A.*

If M is an answer set and $a \in M^+$ ($a \in M^-$), then G is an off-line justification of a w.r.t. M, A iff G is an off-line justification of a^+ (a^-) w.r.t. M, A.

Justifications are built by assembling items from the LCEs of the various atoms and avoiding the creation of positive cycles in the justification of true atoms. Also, the justification is built on a chosen set of assumptions (A), whose elements are all assumed false. In general, an atom may admit multiple justifications, even w.r.t. the same assumptions. The following lemma shows that elements in WF_P can be justified without negative cycles and assumptions.

Lemma 1. *Let P be a program, M an answer set, and WF_P the well-founded model of P. Each atom has an off-line justification w.r.t. M and $\emptyset$ without negative cycles.*

From the definition of assumption and from the previous lemma we can infer that a justification free of negative cycles can be built for every atom.

Proposition 2. *Let P be a program and M an answer set. For each atom a, there is an off-line justification w.r.t. M and $M^- \setminus WF_P^-$ which does not contain negative cycles.*

Proposition 2 underlines an important property—the fact that all true elements can be justified in a non-cyclic fashion. This makes the justification more natural, reflecting the non-cyclic process employed in constructing the minimal answer set (e.g., using the iterations of T_P) and the well-founded model (e.g., using the characterization in [5]). This also gracefully extends a similar nice property satisfied by the justifications under well-founded semantics used in [17]. Note that the only cycles possibly present in the justifications are positive cycles associated to (mutually dependent) false elements—this is an unavoidable situation due the semantic characterization in well-founded and answer set semantics (e.g., unfounded sets).

Example 3. Let us consider the program in Example 1. We have that $NANT(P) = \{b, a\}$. The assumptions for this program are: $Ass(P, M_1) = \{\{a\}\}$ and $Ass(P, M_2) = \{\{b\}\}$. The off-line justifications for atoms in M_1 w.r.t. M_1 and $\{a\}$ are shown in Fig. 2.

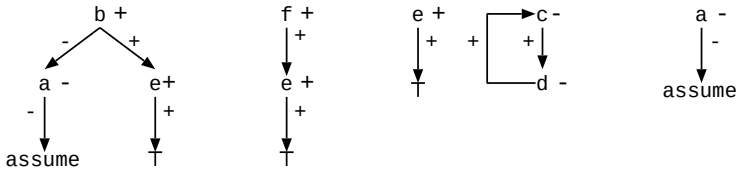

Fig. 2. Off-line Justifications w.r.t. M_1 and $\{a\}$ for b, f, e, c and a (left to right)

4 On-Line Justifications for ASP

In this section, we introduce the concept of on-line justification, which is generated *during* the computation of an answer set and allows us to justify atoms w.r.t. an incomplete interpretation (an intermediate step in the construction of the answer set). The concept of on-line justification is applicable to computation models that construct answer sets in an incremental fashion (e.g., [19, 11, 1])—where we can view the computation as a sequence of steps, each associated to a partial interpretation. We will focus, in particular, on computation models where the progress towards the answer set is monotonic.

Definition 11 (General Computation). *Let P be a program. A* general computation *is a sequence $M_0, M_1, \ldots, M_k$, such that* (i) $M_0 = \langle \emptyset, \emptyset \rangle$, (ii) $M_0, \ldots, M_{k-1}$ *are partial interpretations, and* (iii) $M_i \sqsubseteq M_{i+1}$ *for* $i = 0, \ldots, k-1$. *A* general complete computation *is a computation $M_0, \ldots, M_k$ such that M_k is an answer set of P.*

We do not require M_k to be a partial interpretation, as we wish to model computations that can also fail (i.e., $M_k^+ \cap M_k^- \neq \emptyset$).

Our objective is to associate some form of justification to each intermediate step M_i of a general computation. Ideally, we would like the justifications associated to each M_i to explain truth values in the "same way" as in the final off-line justification. Since the computation model might rely on "guessing" some truth values, M_i might not contain sufficient information to develop a valid justification for each element in M_i. We will identify those atoms for which a justification can be constructed given M_i. These atoms describe a p-interpretation $D_i \sqsubseteq M_i$. The computation of D_i is defined based on the two operators Γ and Δ, which will respectively compute D_i^+ and D_i^-.

Let us start with some preliminary definitions. Let P be a program and I be a p-interpretation. A set of atoms S is called a *cycle w.r.t. I* if for every $a \in S$ and $r \in P$ such that $head(r) = a$, we have that $pos(r) \cap I^- \neq \emptyset$ or $neg(r) \cap I^+ \neq \emptyset$ or $pos(r) \cap S \neq \emptyset$. We can prove that, if I is an interpretation, S is a cycle w.r.t. I and M is an answer set with $I \sqsubseteq M$ then $S \subseteq M^-$. The set of cycles w.r.t. I is denoted by $cycles(I)$. For every element $e \in \mathcal{A}^p \cup \mathcal{A}^n$, let $PE(e, I)$ be the set of LCEs of e w.r.t. I and $\emptyset$.

Let P be a program and $I \sqsubseteq J$ be two p-interpretations. We define

$$\begin{aligned} \Gamma_I(J) &= I^+ \cup \{head(r) \in J^+ \mid I \models body(r)\} \\ \Delta_I(J) &= I^- \cup \{a \in J^- \mid PE(a^-, I) \neq \emptyset\} \cup \bigcup \{S \mid S \in cycles(I), S \subseteq J^-\} \end{aligned}$$

Intuitively, for $I \sqsubseteq J$, we have that $\Gamma_I(J)$ (resp. $\Delta_I(J)$) is a set of atoms that have to be true (resp. false) in every answer set extending J, if J is a partial interpretation. In particular, if I is the set of *"justifiable"* literals (literals for which we can construct a justification) and J is the result of the current computation step, then we have that $\langle \Gamma_I(J), \Delta_I(J) \rangle$ is a new interpretation, $I \sqsubseteq \langle \Gamma_I(J), \Delta_I(J) \rangle \sqsubseteq J$, whose elements are all *"justifiable"*. Observe that it is not necessarily true that $\Gamma_I(J) = J^+$ and $\Delta_I(J) = J^-$. This reflects the practice of guessing literals and propagating these guesses in the computation of answer sets, implemented by several solvers.

We are now ready to specify how the set D_i is computed. Let J be a p-interpretation.

$$\begin{aligned} \Gamma^0(J) &= \Gamma_\emptyset(J) & \Delta^0(J) &= \mathcal{PA}_P(J) \cup \Delta_\emptyset(J) \\ \Gamma^{i+1}(J) &= \Gamma_{I_i}(J) & \Delta^{i+1}(J) &= \Delta_{I_i}(J) \qquad \text{where } I_i = \langle \Gamma^i(J), \Delta^i(J) \rangle \end{aligned}$$

Let

$$\Gamma(J) = \bigcup_{i=0}^{\infty} \Gamma^i(J) \quad \text{and} \quad \Delta(J) = \bigcup_{i=0}^{\infty} \Delta^i(J)$$

Because $\Gamma^i(J) \subseteq \Gamma^{i+1}(J) \subseteq J^+$ and $\Delta^i(J) \subseteq \Delta^{i+1}(J) \subseteq J^-$ (recall that $I \sqsubseteq J$), we know that both $\Gamma(J)$ and $\Delta(J)$ are well-defined. We can prove the following:

Proposition 3. *For a program P, we have that:*

- *Γ and Δ maintain the consistency of J, i.e., if J is an interpretation, then $\langle \Gamma(J), \Delta(J) \rangle$ is also an interpretation;*
- *Γ and Δ are monotone w.r.t the argument J, i.e., if $J \sqsubseteq J'$ then $\Gamma(J) \subseteq \Gamma(J')$ and $\Delta(J) \subseteq \Delta(J')$;*
- *$\Gamma(WF_P) = WF_P^+$ and $\Delta(WF_P) = WF_P^-$; and*
- *if M is an answer set of P, then $\Gamma(M) = M^+$ and $\Delta(M) = M^-$.*

Definition 12 (On-line Explanation Graph). *Let P be a program, A a set of atoms, J a p-interpretation, and $a \in \mathcal{A}^p \cup \mathcal{A}^n$. An* on-line explanation graph *$G = (N, E)$ of a w.r.t. J and A is a (J, A)-based e-graph of a.*

Observe that, if J is an answer set and A a set of assumption, then any off-line e-graph of a w.r.t. J and A is also an on-line e-graph of a w.r.t. J and A.

Observe that $\Gamma^0(J)$ contains the facts of P that belong to J^+ and $\Delta^0(J)$ contains the atoms without defining rules and atoms belonging to positive cycles of P. As such, it is easy to see that for each atom a in $\langle \Gamma^0(J), \Delta^0(J) \rangle$, we can construct an e-graph for a^+ or a^- whose nodes belong to $(\Gamma^0(J))^p \cup (\Delta^0(J))^n$. Moreover, if $a \in \Gamma^{i+1}(J) \setminus \Gamma^i(J)$, an e-graph with nodes (except a^+) belonging to $(\Gamma^i(J))^p \cup (\Delta^i(J))^n$ can be constructed; and if $a \in \Delta^{i+1}(J) \setminus \Delta^i(J)$, an e-graph with nodes belonging to $(\Gamma^{i+1}(J))^p \cup (\Delta^{i+1}(J))^n$ can be constructed. This leads to the following lemma.

Lemma 2. *Let P be a program, J a p-interpretation, and $A = \mathcal{PA}_P(J)$. It holds that*
- *for each atom $a \in \Gamma(J)$ (resp. $a \in \Delta(J)$), there exists a* safe *off-line e-graph of a^+ (resp. a^-) w.r.t. J and A;*
- *for each atom $a \in J^+ \setminus \Gamma(J)$ (resp. $a \in J^- \setminus \Delta(J)$) there exists an on-line e-graph of a^+ (resp. a^-) w.r.t. J and A.*

Let us show how the above proposition can be used in defining a notion called *on-line justification*. To this end, we associate to each partial interpretation J a snapshot $S(J)$:

Definition 13. *Given a p-interpretation J, a snapshot of J is a tuple $S(J) = \langle \mathit{Off}(J), On(J), \langle \Gamma(J), \Delta(J) \rangle \rangle$, where*
- *for each a in $\Gamma(J)$ (resp. a in $\Delta(J)$), $\mathit{Off}(J)$ contains exactly one safe positive (negative) off-line e-graph of a^+ (resp. a^-) w.r.t. J and $\mathcal{PA}_P(J)$;*
- *for each $a \in J^+ \setminus \Gamma(J)$ (resp. $a \in J^- \setminus \Delta(J)$), $On(J)$ contains exactly one on-line e-graph of a^+ (resp. a^-) w.r.t. J and $\mathcal{PA}_P(J)$.*

Definition 14. (On-line Justification) *Given a computation $M_0, M_1, \ldots, M_k$, an* on-line justification *of the computation is a sequence of snapshots $S(M_0), S(M_1), \ldots, S(M_k)$.*

Remark 1. Observe that the monotonicity of the computation allows us to avoid recomputing Γ and Δ from scratch at every step. In particular, when computing the fixpoint we can start the iterations from $\Gamma_{\langle \Gamma(M_i), \Delta(M_i) \rangle}$ and $\Delta_{\langle \Gamma(M_i), \Delta(M_i) \rangle}$ and looking only at the elements of $\langle M_{i+1}^+ \setminus \Gamma(M_i), M_{i+1}^- \setminus \Delta(M_i) \rangle$. Similarly, the computation of $\mathit{Off}(M_{i+1})$ can be made incremental, by simply adding to $\mathit{Off}(M_{i+1})$ the off-line e-graphs for the elements in $\Gamma(M_{i+1}) \setminus \Gamma(M_i)$ and $\Delta(M_{i+1}) \setminus \Delta(M_i)$. Note that these new off-line graphs can be constructed reusing the off-line graphs already in $\mathit{Off}(M_i)$.

Example 4. Let us consider the program P containing

$s \text{ :- } a, not\ t. \qquad a \text{ :- } f, not\ b. \qquad b \text{ :- } e, not\ a. \qquad e \text{ :- } \qquad f \text{ :- } e.$

Two possible general computations of P are

$$M_0^1 = \langle\{e,s\},\emptyset\rangle \qquad M_1^1 = \langle\{e,s,a\},\{t\}\rangle \qquad M_2^1 = \langle\{e,s,a,f\},\{t,b\}\rangle$$
$$M_0^2 = \langle\{e,f\},\emptyset\rangle \qquad M_1^2 = \langle\{e,f\},\{t\}\rangle \qquad M_2^2 = \langle\{e,f,b,a\},\{t,a,b,s\}\rangle$$

The first computation is a complete computation leading to an answer set of P while the second one is not. An on-line justification for the first computation is given next:

$$S(M_0^1) = \langle X_0, Y_0, \langle\{e\},\emptyset\rangle\rangle$$
$$S(M_1^1) = \langle X_0 \cup X_1, Y_0 \cup Y_1, \langle\{e\},\{t\}\rangle\rangle$$
$$S(M_2^1) = \langle X_0 \cup X_1 \cup X_3, \emptyset, M_1^2\rangle$$

where X_0=$\{(\{e^+,\top\},\{(e^+,\top,+)\})\}$, Y_0=$\{(\{s^+,assume\},\{(s^+,assume,+)\})\}$, X_1=$\{(\{t^-,\bot\},\{(t^-,\bot,-)\})\}$, Y_1=$\{\{a^+,assume\},\{(a^+,assume,+)\})\}$, and X_3 is a set of off-line justifications for s, a, f, and b (omitted due to lack of space). □

We can relate the on-line justifications and off-line justifications as follows.

Lemma 3. *Let P be a program, J an interpretation, and M an answer set such that $J \sqsubseteq M$. For each atom a, if (N,E) is a safe off-line e-graph of a^+ (a^-) w.r.t. J and $J^- \cap \mathcal{PA}_P(M)$ then it is an off-line justification of a^+ (a^-) w.r.t. M and $\mathcal{PA}_P(M)$.*

Proposition 4. *Let $M_0, \ldots, M_k$ be a general complete computation and $S(M_0), \ldots, S(M_k)$ be an on-line justification of the computation. Then, for each atom $a \in M_k^+$ (resp. $a \in M_k^-$), the e-graph of a^+ (resp. a^-) in $S(M_k)$ is an off-line justification of a^+ (resp. a^-) w.r.t. M_k and $\mathcal{PA}_P(M)$.*

5 SMODELS On-line Justifications

The notion of on-line justification presented in the previous section is very general, to fit the needs of different models of computation. In this section, we specialize the notion of on-line justification to a specific computation model—the one used in SMODELS [19]. This allows us to define an incremental version of on-line justification—where the steps performed by SMODELS are used to guide the construction of the justification.

We begin with an overview of the algorithms employed by SMODELS. The choice of SMODELS was dictated by availability of its source code and its elegant design. The following description has been adapted from [10, 19]; although more abstract than the concrete implementation, and without various optimizations (e.g., heuristics, lookahead), it is sufficiently faithful to capture the spirit of our approach, and to guide the implementation (see Sect. 5.3).

5.1 An Overview of SMODELS' Computation

We propose a description of the SMODELS algorithms based on a composition of state-transformation operators. In the following, we say that an interpretation I does not satisfy the body of a rule r (or $body(r)$ is false in I) if $(pos(r) \cap I^-) \cup (neg(r) \cap I^+) \neq \emptyset$.

ATLEAST **Operator:** The $AtLeast$ operator is used to expand a partial interpretation I in such a way that each answer set M of P that "agrees" with I (i.e., the elements in I have the same truth value in M) also agrees with the expanded interpretation.

Given a program P and a partial interpretation I, we define the following operators $AL_P^1, \ldots, AL_P^4$:

Case 1. if $r \in P$, $head(r) \notin I^+$, $pos(P) \subseteq I^+$ and $neg(P) \subseteq I^-$ then
$AL_P^1(I)^+ = I^+ \cup \{head(r)\}$ and $AL_P^1(I)^- = I^-$.

Case 2. if $a \notin I^+ \cup I^-$ and $\forall r \in P.(head(r) = a \Rightarrow body(r)$ *is false in* $I)$, then
$AL_P^2(I)^+ = I^+$ and $AL_P^2(I)^- = I^- \cup \{a\}$.

Case 3. if $a \in I^+$ and r is the only rule in P with $head(r) = a$ and whose body is not false in I then, $AL_P^3(I)^+ = I^+ \cup pos(r)$ and $AL_P^3(I)^- = I^- \cup neg(r)$.

Case 4. if $a \in I^-$, $head(r) = a$, and $(pos(r) \setminus I^+) \cup (neg(r) \setminus I^-) = \{b\}$ then,
$AL_P^4(I)^+ = I^+ \cup \{b\}$ and $AL_P^4(I)^- = I^-$ if $b \in neg(r)$
$AL_P^4(I)^- = I^- \cup \{b\}$ and $AL_P^4(I)^+ = I^+$ if $b \in pos(r)$.

Given a program P and an interpretation I, $AL_P(I) = AL_P^i(I)$ if $AL_P^i(I) \neq I$ and $\forall j < i.\ AL_P^j(I) = I$ $(1 \leq i \leq 4)$; otherwise, $AL_P(I) = I$.

ATMOST **Operator:** The $AtMost_P$ operator recognizes atoms that are defined exclusively as mutual positive dependences (i.e., "positive loops")—and falsifies them. Given a set of atoms S, the operator AM_P is defined as $AM_P(S) = S \cup \{head(r) \,|\, r \in P \wedge pos(r) \subseteq S\}$.

Given an interpretation I, the $AtMost_P(I)$ operator is defined as $AtMost_P(I) = \langle I^+, I^- \cup \{p \in \mathcal{A} \mid p \notin \bigcup_{i \geq 0} S_i\}\rangle$ where $S_0 = I^+$ and $S_{i+1} = AM_P(S_i)$.

CHOOSE **Operator:** This operator is used to randomly select an atom that is unknown in a given interpretation. Given a partial interpretation I, $choose_P$ returns an atom of $\mathcal{A}$ such that $choose_P(I) \notin I^+ \cup I^-$ and $choose_P(I) \in NANT(P) \setminus (WF_P^+ \cup WF_P^-)$.

SMODELS COMPUTATION: Given an interpretation I, we define the transitions:

$$
\begin{array}{ll}
I \mapsto_{AL^c} I' & \left\{ \textit{If } I' = AL_P^c(I),\ c \in \{1,2,3,4\} \right. \\
I \mapsto_{atmost} I' & \left\{ \textit{If } I' = AtMost_P(I) \right. \\
I \mapsto_{choice} I' & \left\{ \textit{If } I' = \langle I^+ \cup \{choose_P(I)\}, I^-\rangle \textit{ or } I' = \langle I^+, I^- \cup \{choose_P(I)\}\rangle \right.
\end{array}
$$

We use the notation $I \mapsto I'$ to indicate that there exists $\alpha \in \{AL^1, AL^2, AL^3, AL^4, atmost, choice\}$ such that $I \mapsto_\alpha I'$. A SMODELS computation is a general computation $M_0, M_1, \ldots, M_k$ such that $M_i \mapsto M_{i+1}$.

The SMODELS system imposes constraints on the order of application of the transitions. Intuitively, the SMODELS computation is shown in the algorithms of Figs. 3-4.

Example 5. Consider the program of Example 1. A possible computation of M_1 is:[3]

$$
\begin{array}{llll}
\langle \emptyset, \emptyset\rangle & \mapsto_{AL^1} \langle \{e\}, \emptyset\rangle & \mapsto_{AL^1} \langle \{e,f\}, \emptyset\rangle & \mapsto_{atmost} \\
\langle \{e,f\}, \{c,d\}\rangle & \mapsto_{choice} \langle \{e,f,b\}, \{c,d\}\rangle & \mapsto_{AL^2} \langle \{e,f,b\}, \{c,d,a\}\rangle &
\end{array}
$$

[3] We omit the steps that do not change the interpretation.

function *smodels*(P):
S = $\langle \emptyset, \emptyset \rangle$;
loop
  S = *expand*(P, S);
  if ($S^+ \cap S^- \neq \emptyset$) **then**
    fail;
  if ($S^+ \cup S^- = \mathcal{A}$) **then**
    success(S);
  pick either *% non-deterministic choice*
    S^+ = $S^+ \cup \{choose(S)\}$ **or**
    S^- = $S^- \cup \{choose(S)\}$
endloop;

Fig. 3. Sketch of *smodels*

function *expand*(P, S):
loop
  S' = S;
  repeat
    S = $AL_P(S)$;
  until (S = $AL_P(S)$);
  S = $AtMost$(P, S);
  if (S' = S) **then return** (S);
endloop;

Fig. 4. Sketch of *expand*

5.2 SMODELS On-line Justifications

We can use knowledge of the specific steps performed by SMODELS to guide the construction of an on-line justification. Let us consider the step $M_i \mapsto_\alpha M_{i+1}$ and let us consider the possible $\mapsto_\alpha$. Let $S(M_i) = \langle E_1, E_2, D \rangle$ and $S(M_{i+1}) = \langle E_1', E_2', D' \rangle$. Obviously, $S(M_{i+1})$ can always be computed by computing $D' = \langle \Gamma(M_{i+1}), \Delta(M_{i+1}) \rangle$ and updating E_1 and E_2. As discussed in Remark 1, D' can be done incrementally. Regarding E_1' and E_2', observe that the e-graphs for elements in $\langle \Gamma^k(M_{i+1}), \Delta^k(M_{i+1}) \rangle$ can be constructed using the e-graphs constructed for elements in $\langle \Gamma^{k-1}(M_{i+1}), \Delta^{k-1}(M_{i+1}) \rangle$ and the rules involved in the computation of $\langle \Gamma^k(M_{i+1}), \Delta^k(M_{i+1}) \rangle$. Thus, we only need to update E_1' with e-graphs of elements of $\langle \Gamma^k(M_{i+1}), \Delta^k(M_{i+1}) \rangle$ which do not belong to $\langle \Gamma^{k-1}(M_{i+1}), \Delta^{k-1}(M_{i+1}) \rangle$. Also, E_2' is obtained from E_2 by removing the e-graphs of atoms that "move" into D' and adding the e-graph $(a^+, assume, +)$ (resp. $(a^-, assume, -)$) for $a \in M_{i+1}^+$ (resp. $a \in M_{i+1}^-$) not belonging to D'.

- $\boxed{\alpha \equiv choice:}$ let p be the atom chosen in this step. If p is chosen to be true, then we can use the graph $G_p = (\{a^+, \textit{assume}\}, \{(a^+, \textit{assume}, +)\})$ and the resulting snapshot is $S(M_{i+1}) = \langle E_1, E_2 \cup \{G_p\}, D \rangle$—$D$ is unchanged, since the structure of the computation (in particular the fact that an *expand* has been done before the choice) ensures that p will not appear in the computation of D. If p is chosen to be false, then we will need to add p to D^-, compute $\Gamma(M_{i+1})$ and $\Delta(M_{i+1})$ (using the optimization as discussed in Remark 1), and update E_1 and E_2 correspondingly; in particular, p belongs to $\Delta(M_{i+1})$ and $G_p = (\{a^-, \textit{assume}\}, \{(a^-, \textit{assume}, -)\})$ is added to E_1.
- $\boxed{\alpha \equiv atmost:}$ in this case, $M_{i+1} = \langle M_i^+, M_i^- \cup AtMost(P, M_i) \rangle$. The computation of $S(M_{i+1})$ is performed as from definition. In particular, observe that if $\forall c \in AtMost(P, M_i)$ we have that $LCE_P^n(c^-, D, \emptyset) \neq \emptyset$ then the computation can be started from $\Gamma(M_i)$ and $\Delta(M_i) \cup AtMost(P, M_i)$.
- $\boxed{\alpha \equiv AL^1:}$ let p be the atom dealt with in this step and let r be the rule employed. We have that $M_{i+1} = \langle M_i^+ \cup \{p\}, M_i^- \rangle$. If $D \models body(r)$ then $S(M_{i+1})$ can

be computed from the definition (and starting from $\Gamma(M_i) \cup \{p\}$ and $\Delta(M_i)$); in particular, an off-line graph for p^+, G_p, will be added to E_1 and such graph will be constructed using the rule r and the e-graphs in E_1. Otherwise, $S(M_{i+1}) = \langle E_1, E_2 \cup \{G^+(p, r, \Sigma)\}, D\rangle$, where $G^+(p, r, \Sigma)$ is the e-graph of p^+ constructed using rule r and using the e-graphs in $\Sigma = E_1 \cup E_2$ (note that all elements in $body(r)$ have an e-graph in $E_1 \cup E_2$).

- $\boxed{\alpha \equiv AL^2\textbf{:}}$ let p be the atom dealt with in this step. In this case $M_{i+1} = \langle M_i^+, M_i^- \cup \{p\}\rangle$. If there exists $\gamma \in LCE_P^n(p, D, \emptyset)$ then $S(M_{i+1})$ can be computed from the definition (starting from $\Gamma(M_i)$ and $\Delta(M_i) \cup \{p\}$; observe that the graph of p^- can be constructed starting with $\{(p^-, a^-, +) \mid a \in \gamma\} \cup \{(p^-, b^+, -) \mid not\ b \in \gamma\}$). Otherwise, given an arbitrary $\psi \in LCE_P^n(p, M_i, \emptyset)$, we can build an e-graph G_p for p^- such that $\psi = support(b, G_p)$ and the graphs $E_1 \cup E_2$ are used to describe the elements of γ, and $S(M_{i+1}) = \langle E_1, E_2 \cup \{G_p\}, D\rangle$.
- $\boxed{\alpha \equiv AL^3\textbf{:}}$ let r be the rule used in this step and let $p = head(r)$. Then $M_{i+1} = \langle M_i^+ \cup pos(r), M_i^- \cup neg(r)\rangle$ and $S(M_{i+1})$ is computed according to the definition. Observe that the e-graph G_p for p^+ (added to E_1 or E_2) for $S(M_{i+1})$ will be constructed using $body(r)$ as $support(p, G_p)$, and using the e-graphs in $E_1 \cup E_2 \cup \Sigma$ for some $\Sigma \subseteq \{(a^+, \textit{assume}, +) \mid a \in pos(r)\} \cup \{(a^-, \textit{assume}, -) \mid a \in neg(r)\}$.
- $\boxed{\alpha \equiv AL^4\textbf{:}}$ let r be the rule processed and let b the atom detected in the body. If $b \in pos(r)$, then $M_{i+1} = \langle M_i^+, M_i^- \cup \{p\}\rangle$ and $S(M_{i+1})$ is computed using the definition. Analogously, if $b \in neg(r)$ then $M_{i+1} = \langle M_i^+ \cup \{b\}, M_i^-\rangle$ and $S(M_{i+1})$ is computed using the definition.

Example 6. Let us consider the computation of Example 5. A sequence of snapshots is (we provide only the edges of the graphs and combine e-graphs of different atoms):

	E_1	E_2	D
$S(M_0)$	$\emptyset$	$\emptyset$	$\emptyset$
$S(M_1)$	$\{(e^+, \top, +)\}$	$\emptyset$	$\langle\{e\}, \emptyset\rangle$
$S(M_2)$	$\{(e^+, \top, +), (f^+, e^+, +)\}$	$\emptyset$	$\langle\{e, f\}, \emptyset\rangle$
$S(M_3)$	$\left\{\begin{array}{c}(e^+, \top, +), \{f^+, e^+, +)\\ (d^-, c^-, +), (c^-, d^-, +)\end{array}\right\}$	$\emptyset$	$\langle\{e, f\}, \{c, d\}\rangle$
$S(M_4)$	$\left\{\begin{array}{c}(e^+, \top, +), \{f^+, e^+, +)\\ (d^-, c^-, +), (c^-, d^-, +)\end{array}\right\}$	$\{(b^+, \textit{assume}, +)\}$	$\langle\{e, f\}, \{c, d\}\rangle$
$S(M_5)$	$\left\{\begin{array}{c}(e^+, \top, +), \{f^+, e^+, +),\\ (d^-, c^-, +), (c^-, d^-, +),\\ (a^-, \textit{assume}, -),\\ (b^+, e^+, +), (b^+, a^-, -)\end{array}\right\}$	$\emptyset$	$\langle\{e, f, b\}, \{c, d, a\}\rangle$

5.3 Discussion

The description of SMODELS on-line justifications we proposed is clearly more abstract than the concrete implementation—e.g., we did not address the use of lookahead, the use of heuristics, and other optimizations introduced in SMODELS. We also did not address the extensions available in SMODELS (e.g., choice rules). All these elements can

be handled in the same spirit of what described here, and they would require more space than available in this paper; all these elements *have been addressed* in the implementation of SMODELS on-line justification.

The notions of justification proposed here is meant to represent the basic data structure on which debugging strategies for ASP can be developed. We have implemented both the off-line and the on-line justifications within the $\mathbb{ASP}-\mathbb{PROLOG}$ system [8]. $\mathbb{ASP}-\mathbb{PROLOG}$ allows the construction of Prolog programs (in CIAO Prolog) which include modules written in ASP (the SMODELS flavor of ASP). The SMODELS engine has been modified to extract, during the computation, a compact footprint of the execution, i.e., a trace of the key events (corresponding to the transitions described in Sect. 5) with links to the atoms and rules involved. The modifications of the trace are trailed to support backtracking. Parts of the justification (as described in the previous section) are built on the fly, while others (e.g., certain cases of AL^3 and AL^4) are delayed until the justification is requested.

To avoid imposing the overhead of justification construction on every computation, the programmer has to specify what ASP modules require justifications, using an additional argument (`justify`) in the module import declaration:

`:- use_asp(`⟨`module_name`⟩`, `⟨`file_name`⟩`, `⟨`parameters`⟩` [,justify]).`

On-line justifications are integrated in the ASP debugging facilities of $\mathbb{ASP}-\mathbb{PROLOG}$—which provide predicates to set breakpoints on the execution of an ASP module (e.g., triggered by assignments of a truth value to a certain atom) and to step through execution. Off-line justifications are always available.

$\mathbb{ASP}-\mathbb{PROLOG}$ provides the predicate `model/1` to retrieve answer sets of an ASP module—it retrieves them in the order they are computed by SMODELS, and it returns the current one if the computation is still in progress. The main predicate to access the justification is `justify/1` which retrieves a CIAO Prolog object [15] containing the justification; i.e., `?- my_asp:model(Q), Q:justify(J).` will assign to `J` the object containing the justification relative to the answer set `Q` of the ASP module `my_asp`. Each justification object provides the following predicates: `node/1` which succeeds if the argument is one of the nodes in the justification graph, `edge/3` which succeeds if the arguments correspond to the components of one of the edges in the graph, and `draw/1` which will generate a graphical drawing of the justification for the given atom (using the *uDrawGraph* application). For example,

```
?- my_asp:model(Q),Q:justify(J),findall(e(X,Y),J:edge(p,X,Y),L).
```

will collect in `L` all the edges supporting `p` in the justification graph (for answer set `Q`).

6 Conclusion

In this paper we provided a generalization of the notion of *justification* (originally designed for Prolog with SLG-resolution [17]), to suit the needs of ASP. The notion, named *off-line justification*, offers a way to understand the motivations for the truth value of an atom within a specific answer set, thus making it easy to analyze answer sets for program understanding and debugging. We also introduced *on-line justifications*, which are meant to justify atoms *during* the computation of an answer set. The structure of an on-line justification is tied to the specific steps performed by a computa-

tional model for ASP (specifically, the computation model adopted by SMODELS). An on-line justification allows a programmer to inspect the reasons for the truth value of an atom at the moment such value is determined while constructing an answer set. These data structures provide a foundation for the construction of tools to debug ASP.

The process of computing and presenting justifications has been embedded in the ASP-Prolog system [8], thus making justifications a first-class citizen of the language. This allows the programmer to use Prolog to manipulate justifications as standard Prolog terms. A preliminary implementation can be found at `www.cs.nmsu.edu/~okhatib/asp_prolog.html`.

As future work, we propose to complete the implementation, refine the definition of on-line justification to better take advantage of SMODELS, and develop a complete debugging and visualization environment for ASP based on these data structures.

Acknowledgments

The research has been partially supported by NSF grants CNS-0454066, HRD-0420407, and CNS-0220590.

References

1. C. Anger et al. The nomore++ Approach to Answer Set Solving. *LPAR*, Springer, 2005.
2. K. Apt, R. Bol. Logic Programming and Negation: A Survey. *J. Log. Program.* 19/20, 1994.
3. M. Auguston. Assertion Checker for the C Programming Language. *AADEBUG*, 2000.
4. C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, 2003.
5. S. Brass et al. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001.
6. S. Costantini et al. On the Equivalence and Range of Applicability of Graph-based Representations of Logic Programs. *Information Processing Letters*, 84(5):241–249, 2002.
7. M. Ducassé. Opium: an Extendable Trace Analyzer for Prolog. *J. Logic Progr.*, 39, 1999.
8. O. Elkhatib et al. A System for Reasoning about ASP in Prolog. *PADL*, Springer, 2004.
9. M. Gelfond, V. Lifschitz. The Stable Model Semantics for Logic Programs. *ILPS*, 1988.
10. E. Giunchiglia and M. Maratea. On the Relation between Answer Set and SAT Procedures. In *ICLP*, Springer Verlag, 2005.
11. N. Leone et al. The DLV System. In *JELIA*, Springer Verlag, 2002.
12. S. Mallet, M. Ducasse. Generating Deductive Database Explanations. *ICLP*, MIT, 1999.
13. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*, Springer Verlag, 1999.
14. G. Pemmasani et al. Online Justification for Tabled Logic Programs. *FLOPS*, 2004.
15. A. Pineda. Object-oriented programming library O'Ciao. TR 6/99.0, UPM Madrid, 1999.
16. G. Puebla, F. Bueno, M.V. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *LOPSTR*, Springer Verlag, 1999.
17. A. Roychoudhury et al. Justifying Proofs Using Memo Tables. *PPDP*, ACM Press, 2000.
18. E. Shapiro. Algorithmic Program Diagnosis. In *POPL*, ACM Press, 1982.
19. P. Simons et al. Extending and Implementing the Stable Model Semantics. *Artif. Intell.*, 138(1-2), 2002.
20. G. Specht. Generating Explanation Trees even for Negation in Deductive Databases. *Workshop on Logic Programming Environments*, 8–13, Vancouver, 1993.
21. R. Vaupel et al. A Tool for Visualizing And-Or Parallel Executions. *ICLP*, MIT Press, 1997.