# Debugging ASP Programs by Means of ASP*

Martin Brain[1], Martin Gebser[2], Jörg Pührer[3], Torsten Schaub[2,**], Hans Tompits[3], and Stefan Woltran[3]

[1] Department of Computer Science, University of Bath, Bath, BA2 7AY, United Kingdom
mjb@cs.bath.ac.uk
[2] Institut für Informatik, Universität Potsdam, August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de
[3] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits,stefan}@kr.tuwien.ac.at

**Abstract.** Answer-set programming (ASP) has become an important paradigm for declarative problem solving in recent years. However, to further improve the usability of answer-set programs, the development of software-engineering tools is vital. In particular, the area of debugging provides a challenge in both theoretical and practical terms. This is due to the purely declarative nature of ASP that, on the one hand, calls for solver-independent methodologies and, on the other hand, does not directly apply to tracing techniques. In this paper, we propose a novel methodology, which rests within ASP itself, to sort out errors on the conceptual level. Our method makes use of *tagging*, where the program to be analyzed is rewritten using dedicated control atoms. This provides a flexible way to specify different types of debugging requests and a first step towards a dedicated (meta level) debugging language.

## 1 Introduction

Answer-set programming (ASP) has become a popular approach to declarative problem solving. The highly declarative semantics of the language decouples the problem specification from the computation of a solution. As a consequence, there is no general handle on the solving process whenever the output is in question. This deprives us of applying standard, procedural debugging methodologies and has led to a significant lack of methods and tools for debugging logic programs in ASP. However, the semantics itself allows for debugging methodologies that explain *why*, rather than *how*, a program is wrong. Another challenge is that the specification of a problem and its solutions are expressed in different fragments of the underlying language. While an encoding is usually posed in terms of predicate and variable symbols, a solution is free of variables and consists of ground atomic formulas.

We address this gap by proposing a novel debugging methodology that allows for debugging logic programs by appeal to ASP itself. To this end, we exploit and further

* This work was partially supported by the Austrian Science Fund (FWF) under project P18019.
** Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

C. Baral, G. Brewka, and J. Schlipf (Eds.): LPNMR 2007, LNAI 4483, pp. 31–43, 2007.
© Springer-Verlag Berlin Heidelberg 2007

extend the *tagging technique* introduced by Delgrande, Schaub, and Tompits [1] for compiling ordered logic programs into standard ones. The idea is to compile a program in focus (once) and to subsequently accomplish different types of debugging requests by appeal to special debugging modules using dedicated control atoms, called *tags*. Tags allow for controlling the formation of answer sets and reflect different properties (like the applicability status of a rule, for instance) and therefore can be used for manipulating the evaluation of the program (like the actual application of a rule).

The basic tagging technique is then used to conceive a (meta level) *debugging language* providing dedicated debugging statements. The idea here is to first translate a program into its tagged form and then to analyze it by means of debugging statements. More specifically, starting with a program $\Pi$ over an alphabet $\mathcal{A}$, $\Pi$ is translated into a tagged kernel program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$, and a debugging request $\Delta$, formulated in the debugging language, is then compiled into a tagged program $\mathcal{D}[\Delta]$ over $\mathcal{A}^+$. The debugging results are eventually read off the answer sets of the combined tagged program $\mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{D}[\Delta]$. In this paper, we focus on the basic constituents of such a debugging language, confining ourselves to a detailed account of the tagging method.

Our approach has several advantageous distinct features: Firstly, it is based on meta-programming techniques that keep it within the realm of ASP. The dedicated debugging language offers an easy and modular way of specifying debugging requests. Notably, it allows the users to pose their requests with variables, which provides a tight connection to an encoding at hand. Secondly, the different debugging techniques are derived from semantic principles and relate to different characterizations of answer set formation. As a consequence, we can ascribe meaning to different debugging outcomes as well as the underlying compilation techniques. This is nicely demonstrated by our extrapolation techniques that allow for debugging incoherent logic programs. Finally, our approach has been implemented within the tool `spock`, which is publicly available at [2].

Our approach is not meant to be universal. For one thing, it aims at exploring the limits of debugging within the realm of ASP. For another, it nicely complements the majority of existing approaches, all of which are external to ASP [3,4,5,6]. Most of them rely on graph-based characterizations, in the simplest case dependency graphs, and use specific algorithms for analyzing such graphs.

## 2 Background

Given an alphabet $\mathcal{A}$, a (*normal*) *logic program* is a finite set of rules of form

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_{m+1}, \ldots, \mathit{not}\ c_n\ , \tag{1}$$

where $a, b_i, c_j \in \mathcal{A}$ are *atoms* for $0 \leq i \leq m \leq j \leq n$. A *literal* is an atom $a$ or its negation $\mathit{not}\ a$. For a rule $r$ of form (1), let $\mathit{head}(r) = a$ be the *head* of $r$ and $\mathit{body}(r) = \{b_1, \ldots, b_m, \mathit{not}\ c_{m+1}, \ldots, \mathit{not}\ c_n\}$ be the *body* of $r$. Furthermore, we define $\mathit{body}^+(r) = \{b_1, \ldots, b_m\}$ and $\mathit{body}^-(r) = \{c_{m+1}, \ldots, c_n\}$. The set of atoms occurring in a program $\Pi$ is given by $\mathit{At}(\Pi)$. For regrouping rules sharing the same head $a$, we use $\mathit{def}(a, \Pi) = \{r \in \Pi \mid \mathit{head}(r) = a\}$. For uniformity, we assume that any integrity constraint $\leftarrow \mathit{body}(r)$ is expressed as a rule $w \leftarrow \mathit{body}(r), \mathit{not}\ w$, where

$w$ is a globally new atom. Moreover, we allow nested expressions of form $not\ not\ a$, where $a$ is some atom, in the body of rules. Such rules are identified with normal rules in which $not\ not\ a$ is replaced by $not\ a^\star$, where $a^\star$ is a globally new atom, together with an additional rule $a^\star \leftarrow not\ a$. We also take advantage of (singular) *choice rules* of form $\{a\} \leftarrow body(r)$ [7], which are an abbreviation for $a \leftarrow body(r), not\ not\ a$.

We define answer sets following the approach of Lin and Zhao [8]. Given a program $\Pi$, let $PF(\Pi) \cup CF(\Pi)$ be the *completion* of $\Pi$ [9], where

$$PF(\Pi) = \{body(r) \rightarrow head(r) \mid r \in \Pi\} \text{ and}$$
$$CF(\Pi) = \{a \rightarrow \bigvee_{r \in def(a,\Pi)} body(r) \mid a \in \mathcal{A}\}.^1$$

A *loop* is a (non-empty) set of atoms that circularly depend upon each other in a program's positive atom dependency graph [8]. Programs having an acyclic positive atom dependency graph are *tight* [10]. The *loop formula* associated with a loop $L$ is

$$LF(\Pi, L) = \neg(\bigvee_{r \in R(\Pi,L)} body(r)) \rightarrow \bigwedge_{a \in L} \neg a\ ,$$

where $R(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L = \emptyset\}$. We denote the set of all loops in $\Pi$ by $loop(\Pi)$. The set of all loop formulas of $\Pi$ is $LF(\Pi) = \{LF(\Pi, L) \mid L \in loop(\Pi)\}$. A set $X$ of atoms is an *answer set* of a logic program $\Pi$ iff $X$ is a model of $PF(\Pi) \cup CF(\Pi) \cup LF(\Pi)$. If $\Pi$ is tight, then the answer sets of $\Pi$ coincide with the models of $PF(\Pi) \cup CF(\Pi)$ (models of the latter are also referred to as the *supported models* of $\Pi$). The set $\Pi_X$ of *generating rules* of a set $X$ of atoms from program $\Pi$ is defined as $\{r \in \Pi \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$.

As an example, consider $\Pi_1 = \{a \leftarrow;\ c \leftarrow not\ b, not\ d;\ d \leftarrow a, not\ c\}$ and its two answer sets $\{a, c\}$ and $\{a, d\}$. The completion of $\Pi_1$ is logically equivalent to $a \wedge \neg b \wedge (c \leftrightarrow \neg b \wedge \neg d) \wedge (d \leftrightarrow a \wedge \neg c)$; its models coincide with the answer sets of $\Pi_1$. Adding $\{b \leftarrow e;\ e \leftarrow b\}$ to $\Pi_1$ induces the loop $\{b, e\}$ but leaves the set of answer sets of $\Pi_1$ intact. Unlike this, the completion of $\Pi_1$ becomes $a \wedge (b \leftrightarrow e) \wedge (c \leftrightarrow \neg b \wedge \neg d) \wedge (d \leftrightarrow a \wedge \neg c)$ and admits an additional model $\{a, b, d, e\}$. This supported model violates the loop formula $LF(\Pi_1, \{b, e\}) = \top \rightarrow \neg b \wedge \neg e$, which denies it the status of an answer set.

## 3 Debugging Modules

Our approach relies on the *tagging technique* introduced by Delgrande, Schaub, and Tompits [1] for compiling ordered logic programs back into normal programs. The idea is to rewrite a program by introducing so-called *tags* that allow for controlling the formation of answer sets. More formally, given a logic program $\Pi$ over $\mathcal{A}$ and a set $\mathcal{N}$ of names for all rules in $\Pi$, we consider an enriched alphabet $\mathcal{A}^+$ obtained from $\mathcal{A}$ by adding new pairwise distinct propositional atoms such as $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, $\mathsf{ko}(n_r)$, etc., where $n_r \in \mathcal{N}$ for each $r \in \Pi$. Intuitively, $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ express whether a rule $r$ is applicable or blocked, respectively, while $\mathsf{ok}(n_r)$ and $\mathsf{ko}(n_r)$ are used for manipulating the application of $r$. Further tags are introduced in the sequel.

[1] Strictly speaking, $CF(\Pi)$ should take $\mathcal{A}$ as additional argument; for simplicity, we leave this implicit. Moreover, $body(r)$ is understood as a conjunction of (classical) literals within $PF(\Pi)$, $CF(\Pi)$, and $LF(\Pi, L)$ in what follows.

### 3.1 Kernel Debugging Module

Our *kernel translation*, $\mathcal{T}_{\mathsf{K}}$, decomposes rules of a given program such that they can be accessed by tags for controlling purposes.

**Definition 1.** *Let $\Pi$ be a logic program over $\mathcal{A}$. Then, the program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for $r \in \Pi$, $b \in body^+(r)$, and $c \in body^-(r)$:*

$$head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r)\ , \qquad \mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ b\ ,$$
$$\mathsf{ap}(n_r) \leftarrow \mathsf{ok}(n_r), body(r)\ , \qquad \mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ not\ c\ ,$$
$$\mathsf{ok}(n_r) \leftarrow not\ \overline{\mathsf{ok}}(n_r)\ .$$

An auxiliary atom $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, or $\mathsf{ok}(n_r)$, respectively, occurs in an answer set $X$ of $\mathcal{T}_{\mathsf{K}}[\Pi]$ only if $r \in \Pi$. Also, for any $r \in \Pi$, there is a priori no atom $\mathsf{ko}(n_r)$ contained in an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$, whereas $\mathsf{ok}(n_r)$ is contained in any answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$ by default. The role of $\overline{\mathsf{ok}}(n_r)$ is to implement potential changes of this default behavior.

The following proposition collects more interesting relations.

**Proposition 1.** *Let $\Pi$ be a logic program over $\mathcal{A}$ and $X$ an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$. Then, for any $r \in \Pi$ and $a \in \mathcal{A}$:*

1. $\mathsf{ap}(n_r) \in X$ *iff* $r \in \Pi_X$ *iff* $\mathsf{bl}(n_r) \notin X$;
2. *if* $a \in X$, *then* $\mathsf{ap}(n_r) \in X$ *for some* $r \in def(a, \Pi)$;
3. *if* $a \notin X$, *then* $\mathsf{bl}(n_r) \in X$ *for all* $r \in def(a, \Pi)$.

The relation between auxiliary atoms and original atoms from $\mathcal{A}$ is described next.

**Theorem 1.** *Let $\Pi$ be a logic program over $\mathcal{A}$. We have a one-to-one correspondence between the answer sets of $\Pi$ and $\mathcal{T}_{\mathsf{K}}[\Pi]$ satisfying the following conditions:*

1. *If $X$ is an answer set of $\Pi$, then*

$$X \cup \{\mathsf{ok}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \Pi_X\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \Pi_X\}$$

   *is an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$.*
2. *If $Y$ is an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$, then $(Y \cap \mathcal{A})$ is an answer set of $\Pi$.*

### 3.2 Extrapolating Non-existing Answer Sets

Whenever a program $\Pi$ has no answer set, this means in terms of the characterization by Lin and Zhao [8] that there is no interpretation jointly satisfying $PF(\Pi)$, $CF(\Pi)$, and $LF(\Pi)$. In other words, each interpretation $X$ over $\mathcal{A}$ causes at least one of the following problems:

- If $X$ falsifies $PF(\Pi)$, then there is some rule $r$ in $\Pi$ such that $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$, but $head(r) \notin X$.
- If $X$ falsifies $CF(\Pi)$, then there is some atom $a$ in $X$ that lacks a supporting rule, that is, $body^+(r) \not\subseteq X$ or $body^-(r) \cap X \neq \emptyset$ for each $r \in def(a, \Pi)$.
- If $X$ falsifies $LF(\Pi)$, then $X$ contains some loop $L$ in $\Pi$ that is unfounded with respect to $X$, that is, $X \not\models LF(\Pi, L)$.

This intuition is captured in the debugging model described below. It aims at analyzing incoherent situations by figuring out which rules or atoms cause some of the aforementioned problems. The names of the translations, $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, reflect their respective purpose, indicating problems originating from the program, its completion, or its (non-trivial) loop formulas. We use abnormality atoms with a corresponding naming schema to indicate the respective problem: $\mathsf{ab}_p(n_r)$ signals that rule $r$ is falsified under some interpretation, $\mathsf{ab}_c(a)$ points out that atom $a$ is true but has no support, and $\mathsf{ab}_l(a)$ aims at indicating an unfounded atom $a$.

**Definition 2.** *Let $\Pi$ be a logic program over $\mathcal{A}$ and $A$ a set of atoms. Then:*

1. *The logic program $\mathcal{T}_{\mathsf{P}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for all $r \in \Pi$:*

$$\{\, head(r) \,\} \leftarrow \mathsf{ap}(n_r)\,, \qquad \mathsf{ab}_p(n_r) \leftarrow \mathsf{ap}(n_r), not\ head(r)\,,$$
$$\mathsf{ko}(n_r) \leftarrow\ .$$

2. *The logic program $\mathcal{T}_{\mathsf{C}}[\Pi, A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$, where $\{r_1, \ldots, r_k\} = def(a, \Pi)$:*

$$\{\, a \,\} \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\,, \qquad \mathsf{ab}_c(a) \leftarrow a, \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\,.$$

3. *The logic program $\mathcal{T}_{\mathsf{L}}[A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$:*

$$\{\, \mathsf{ab}_l(a) \,\} \leftarrow not\ \mathsf{ab}_c(a)\,, \qquad a \leftarrow \mathsf{ab}_l(a)\,.$$

The purpose of adding facts $(\mathsf{ko}(n_r) \leftarrow)$ in $\mathcal{T}_{\mathsf{P}}[\Pi]$ is to avoid the application of the rule $(head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r))$ in $\mathcal{T}_{\mathsf{K}}[\Pi]$ (rather than to enforce a re-compilation in conjunction with $\mathcal{T}_{\mathsf{K}}[\Pi]$). Regarding $\mathcal{T}_{\mathsf{C}}[\Pi, A]$, note that $def(a, \Pi)$ might be empty, in which case we obtain for $a$ the choice rule $(\{\, a \,\} \leftarrow)$ along with $(\mathsf{ab}_c(a) \leftarrow a)$. Observe that $\mathcal{T}_{\mathsf{L}}[A]$ allows us to add $\mathsf{ab}_l(a)$ to an answer set if $a$ is supported. In contrast to $\mathsf{ab}_p(n_r)$ in $\mathcal{T}_{\mathsf{P}}[\Pi]$ and $\mathsf{ab}_c(a)$ in $\mathcal{T}_{\mathsf{C}}[\Pi, A]$, indicating violations of $PF(\Pi)$ or $CF(\Pi)$, respectively, the presence of $\mathsf{ab}_l(a)$ in an answer set does not necessarily indicate the violation of any loop formula in $LF(\Pi)$. In fact, as the number of loops for $\Pi$ can be exponential, we cannot reasonably check loop formula violations within $\mathcal{T}_{\mathsf{L}}[A]$ (via an additional argument $\Pi$ and tagged rules for analyzing loop formulas). Rather, as discussed below, we filter occurrences of $\mathsf{ab}_l(a)$ in answer sets by *minimization*.

Next, we put things together. The answer sets of the subsequent translation are thought of as extrapolations of putative yet non-existing answer sets of the original program $\Pi$. That is, an atom $\mathsf{ab}_p(n_r)$ signals that an answer set could be obtained if rule $r$ was not contained in $\Pi$. Dually, $\mathsf{ab}_c(a)$ indicates that an answer set could be obtained if atom $a$ would be supported, that is, if it would be derivable by some rule. Finally, $\mathsf{ab}_l(a)$ points to the violation of a loop formula that involves $a$. Moreover, we provide further possibilities to parametrize a debugging request by two additional arguments, $\Pi'$ and $A$. Hereby, $\Pi'$ restricts the set of rules (from program $\Pi$) whose violation is tolerated for debugging purposes, while $A$ restricts the atoms that can be assumed true though being unsupported or belonging to a (non-trivial) unfounded set.

**Definition 3.** *Let $\Pi$ be a logic program over $\mathcal{A}$, $\Pi' \subseteq \Pi$, and $A \subseteq At(\Pi)$. Then:*

$$\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A] = \mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{T}_{\mathsf{P}}[\Pi'] \cup \mathcal{T}_{\mathsf{C}}[\Pi, A] \cup \mathcal{T}_{\mathsf{L}}[A] \,.$$

*Moreover, let $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi'] = \mathcal{T}_{\mathsf{E}}[\Pi, \Pi', At(\Pi')]$ and $\mathcal{T}_{\mathsf{E}}[\Pi] = \mathcal{T}_{\mathsf{E}}[\Pi, \Pi, At(\Pi)]$.*

Note that $\mathcal{T}_{\mathsf{L}}[A]$ can be omitted in the definition of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A]$ if $\Pi$ is tight.

We list some basic properties first.

**Proposition 2.** *Let $\Pi$ be a logic program over $\mathcal{A}$ and $X$ an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$. Then, for each $r \in \Pi$:*

1. $\mathsf{ab}_p(n_r) \in X$ *iff* $\mathsf{ap}(n_r) \in X$, $\mathsf{bl}(n_r) \notin X$, *and* $head(r) \notin X$;
2. $\mathsf{ab}_p(n_r) \notin X$ *if* $\mathsf{ab}_c(head(r)) \in X$ *or* $\mathsf{ab}_l(head(r)) \in X$.

*Moreover, for every $a \in At(\Pi)$, it holds that:*

1. $\mathsf{ab}_c(a) \in X$ *and* $\mathsf{ab}_l(a) \notin X$ *iff* $a \in X$ *and* $(X \cap \mathcal{A}) \not\models (\bigvee_{r \in def(a,\Pi)} body(r))$;
2. $\mathsf{ab}_c(a) \notin X$ *if* $a \in X$ *and* $(X \cap \mathcal{A}) \models (\bigvee_{r \in def(a,\Pi)} body(r))$;
3. $\mathsf{ab}_c(a) \notin X$ *and* $\mathsf{ab}_l(a) \notin X$ *if* $a \notin X$;
4. $\mathsf{ab}_c(a) \notin X$ *if* $\mathsf{ab}_l(a) \in X$.

The next result shows that abnormality-free answer sets of the translated program correspond to the answer sets of the original program.[2] To this end, we introduce, for a program $\Pi$, $\mathsf{AB}(\Pi) = (\{\mathsf{ab}_p(n_r) \mid r \in \Pi\} \cup \{\mathsf{ab}_c(a), \mathsf{ab}_l(a) \mid a \in At(\Pi)\})$.

**Theorem 2.** *Let $\Pi$ be a logic program over $\mathcal{A}$. Then, it holds that:*

1. *If $X$ is an answer set of $\Pi$, then*

   $$X \cup \{\mathsf{ok}(n_r), \mathsf{ko}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \Pi_X\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \Pi_X\}$$

   *is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$.*
2. *If $Y$ is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ such that $(Y \cap \mathsf{AB}(\Pi)) = \emptyset$, then $(Y \cap \mathcal{A})$ is an answer set of $\Pi$.*

The interesting case, however, is when the original program is incoherent. For illustrating this, let us consider three simple examples. To begin with, consider:

$$\Pi_2 = \{\; n_{r_1} : a \leftarrow,\; n_{i_1} : \leftarrow a \;\} \,.$$

The program $\mathcal{T}_{\mathsf{K}}[\Pi_2]$ consists of the following rules:

$$\begin{array}{ll}
a \leftarrow \mathsf{ap}(n_{r_1}), not\ \mathsf{ko}(n_{r_1}) \,, & \leftarrow \mathsf{ap}(n_{i_1}), not\ \mathsf{ko}(n_{i_1}) \,, \\
\mathsf{ap}(n_{r_1}) \leftarrow \mathsf{ok}(n_{r_1}) \,, & \mathsf{ap}(n_{i_1}) \leftarrow \mathsf{ok}(n_{i_1}), a \,, \\
 & \mathsf{bl}(n_{i_1}) \leftarrow \mathsf{ok}(n_{i_1}), not\ a \,, \\
\mathsf{ok}(n_{r_1}) \leftarrow not\ \overline{\mathsf{ok}}(n_{r_1}) \,, & \mathsf{ok}(n_{i_1}) \leftarrow not\ \overline{\mathsf{ok}}(n_{i_1}) \,.
\end{array}$$

[2] Due to relaxing $\Pi$ by tolerating abnormalities, $\mathcal{T}_{\mathsf{E}}[\Pi]$ always admits (abnormal) answer sets.

We obtain $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ by adding $(\mathcal{T}_{\mathsf{P}}[\{r_1\}] \cup \mathcal{T}_{\mathsf{C}}[\Pi_2, \{a\}] \cup \mathcal{T}_{\mathsf{L}}[\{a\}])$, given next:

$$\begin{array}{rr} \mathsf{ko}(n_{r_1}) \leftarrow \ , & \{\, a \,\} \leftarrow \mathsf{bl}(n_{r_1}) \ , \\ & \mathsf{ab}_c(a) \leftarrow a, \mathsf{bl}(n_{r_1}) \ , \\ \{\, a \,\} \leftarrow \mathsf{ap}(n_{r_1}) \ , & \{\, \mathsf{ab}_l(a) \,\} \leftarrow \mathit{not}\ \mathsf{ab}_c(a) \ , \\ \mathsf{ab}_p(n_{r_1}) \leftarrow \mathsf{ap}(n_{r_1}), \mathit{not}\ a \ , & a \leftarrow \mathsf{ab}_l(a) \ . \end{array}$$

The unique answer set of $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ is:[3]

$$\{\underline{\mathsf{ab}_p(n_{r_1})}, \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{i_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{i_1}), \mathsf{ko}(n_{r_1})\} \ . \tag{2}$$

Note that applying the modules from Definition 2 only to subprogram $\{r_1\}$ makes us focus on answer set candidates satisfying the residual program $\{i_1\}$. The abnormality tag $\mathsf{ab}_p(n_{r_1})$ signals that, in order to obtain an answer set, rule $r_1$ must not be applied.

As another example, consider:

$$\Pi_3 = \{\ n_{r_1} : a \leftarrow b,\ n_{i_1} :\ \leftarrow \mathit{not}\ a\ \} \ .$$

Program $\mathcal{T}_{\mathsf{E}}[\Pi_3, \{r_1\}, \{a\}]$ has the unique answer set:

$$\{a, \underline{\mathsf{ab}_c(a)}, \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{i_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{i_1}), \mathsf{ko}(n_{r_1})\} \ , \tag{3}$$

indicating that $a$ lacks supporting rules.

Finally, consider the following program:

$$\Pi_4 = \{\ n_{r_1} : a \leftarrow b,\ n_{r_2} : b \leftarrow a,\ n_{i_1} :\ \leftarrow \mathit{not}\ a\ \} \ .$$

Program $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$ has four answer sets (omitting $\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{i_1})$):

$$\{a, \underline{\mathsf{ab}_p(n_{r_2})}, \underline{\mathsf{ab}_c(a)}, \mathsf{bl}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{4}$$

$$\{a, b, \underline{\mathsf{ab}_l(a)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{5}$$

$$\{a, b, \underline{\mathsf{ab}_l(b)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{6}$$

$$\{a, b, \underline{\mathsf{ab}_l(a)}, \underline{\mathsf{ab}_l(b)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ . \tag{7}$$

The last example illustrates that the relaxation mechanisms underlying translation $\mathcal{T}_{\mathsf{E}}$ can lead to overly involved explanations of the source of incoherence, as manifested by the first and last answer set (cf. (4) and (7)). Therefore, we suggest focusing on answer sets containing a *minimum number* of instances of ab predicates.[4] In the last case, this gives the second and third answer set (cf. (5) and (6)). Indeed, adding only one fact, either $(a \leftarrow)$ or $(b \leftarrow)$, to the (untagged) incoherent program $\Pi_4$ makes it coherent.

The next results shed some more light on the semantic links between the original and the transformed program, whenever the former admits no answer set.

[3] In what follows, we underline abnormality tags.

[4] This can be implemented via `minimize` statements as available in `Smodels`.

**Theorem 3.** *Let $\Pi$ be a logic program over $\mathcal{A}$. Then, it holds that:*

1. *If $Y$ is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ and $\mathsf{ab}_p(n_r) \in Y$, then $(Y \cap \mathcal{A}) \not\models (body(r) \rightarrow head(r))$, where $(body(r) \rightarrow head(r)) \in PF(\Pi)$;*
2. *If $Y$ is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ and $\mathsf{ab}_c(a) \in Y$, then $(Y \cap \mathcal{A}) \not\models (a \rightarrow \bigvee_{r \in def(a,\Pi)} body(r))$, where $(a \rightarrow \bigvee_{r \in def(a,\Pi)} body(r)) \in CF(\Pi)$;*
3. *If $Y$ is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ such that, for some $L \in loop(\Pi)$, we have $L \subseteq (Y \cap \mathcal{A})$, $(Y \cap \mathcal{A}) \not\models LF(\Pi, L)$, and $(Y \cap \mathcal{A}) \models (\bigvee_{r \in def(a,\Pi)} body(r))$ for every $a \in L$, then $\mathsf{ab}_l(a') \in Y$ for some $a' \in L$.*

The same results hold for partial compilations, $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A]$, but are omitted for brevity.

For illustration, let us return to the last three examples. Intersecting the only answer set (2) of $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ with the alphabet of $\Pi_2$ yields the empty set. We obtain $\emptyset \not\models (a \leftarrow)$, as indicated by $\mathsf{ab}_p(n_{r_1})$ in (2). Note that the empty set is the only subset of $At(\Pi_2)$ that satisfies integrity constraint $i_1 \in \Pi_2$. Proceeding analogously with the only answer set (3) of $\mathcal{T}_{\mathsf{E}}[\Pi_3, \{r_1\}, \{a\}]$ yields $\{a\}$, and we obtain $\{a\} \not\models (a \rightarrow b)$, as signaled by $\mathsf{ab}_c(a)$ in (3). In fact, $\{a\}$ is the only subset of the atoms subject to extrapolation that satisfies integrity constraint $i_1 \in \Pi_3$.

Finally, consider the two abnormality-minimum answer sets (5) and (6) of $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$. Both are actually symmetrical since they refer to the same loop $\{a, b\}$ through different elements, as indicated by $\mathsf{ab}_l(a)$ and $\mathsf{ab}_l(b)$, respectively. Hence, both answer sets (5) and (6) of $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$ induce candidate set $\{a, b\}$, which falsifies its own loop formula: $\{a, b\} \not\models (\top \rightarrow \neg a \wedge \neg b)$. Note that $\{a\}$ is actually another candidate subset of $At(\Pi_4)$. However, this candidate suffers from two abnormalities, as indicated by the non-minimum answer set (4) through $\mathsf{ab}_p(n_{r_2})$ and $\mathsf{ab}_c(a)$. In fact, we have $\{a\} \not\models (b \leftarrow a)$, violating $r_2$, and $\{a\} \not\models (a \rightarrow b)$, violating the completion.

The next result captures a more realistic scenario, in which only a subset of a program is subject to extrapolation and only abnormality-minimum answer sets of the translation are considered. From the perspective of an original program $\Pi$, the abnormality-minimum answer sets of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi']$ provide us with the candidate sets among $At(\Pi)$ that satisfy the requirement of being an answer set of $\Pi$ under a minimum number of *repairs* on $\Pi'$. A repair is either the deletion of a rule $r$ or an addition of a fact $(a \leftarrow)$ (which prevents $a$ from being not supported or unfounded). The former repair refers to $\mathsf{ab}_p(n_r)$, which is clearly avoided when $r$ is deleted, and the latter to $\mathsf{ab}_c(a)$ or $\mathsf{ab}_l(a)$, since $(a \rightarrow \bigvee_{r \in def(a, \Pi \cup \{a \leftarrow\})} body(r))$ and $LF(\Pi \cup \{a \leftarrow\}, L)$, for any loop $L$ containing $a$, then amount to $(a \rightarrow \top)$ and $(\bot \rightarrow \bigwedge_{a \in L} \neg a)$.

**Theorem 4.** *Let $\Pi$ be a logic program over $\mathcal{A}$ and $(\Pi_1, \Pi_2)$ a partition of $\Pi$ such that $(\{head(r_1) \mid r_1 \in \Pi_1\} \cap At(\Pi_2)) = \emptyset$. Furthermore, let $\mathcal{M}$ be the set of all answer sets $Y$ of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi_2]$ such that the cardinality of $(Y \cap \mathsf{AB}(\Pi_2))$ is minimum among all answer sets of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi_2]$. Then, it holds that:*

1. *If $Y \in \mathcal{M}$, then $(Y \cap \mathcal{A})$ satisfies all formulas in $(PF(\Pi_1) \cup (CF(\Pi_1) \setminus \{a \rightarrow \bot \mid a \in At(\Pi_2)\}) \cup LF(\Pi_1))$ and all formulas in $(PF(\Pi_2) \cup CF(\Pi_2) \cup LF(\Pi_2))$ under a minimum number of repairs on $\Pi_2$;*
2. *If $X \subseteq \mathcal{A}$ satisfies all formulas in $(PF(\Pi_1) \cup (CF(\Pi_1) \setminus \{a \rightarrow \bot \mid a \in At(\Pi_2)\}) \cup LF(\Pi_1))$ and all formulas in $(PF(\Pi_2) \cup CF(\Pi_2) \cup LF(\Pi_2))$ under a minimum number of repairs on $\Pi_2$, then there is a $Y \in \mathcal{M}$ such that $X = (Y \cap \mathcal{A})$.*

### 3.3 Ordering Rule Applications

Our last debugging component allows for imposing an order on the rule applications. To this end, we make use of ordered logic programs following the framework of Delgrande, Schaub, and Tompits [1].

**Definition 4.** *Let $\Pi$ be a logic program over $\mathcal{A}$ and let $<$ be a strict partial order over $\Pi$. Then, $\mathcal{T}_o[\Pi,<]$ consists of the following rules, for all $r,r' \in \Pi$ with $r < r'$ and $\{r_1,\ldots,r_k\} = \{r'' \mid r < r''\}$:*

$$\overline{\mathsf{ok}}(n_r) \leftarrow \ , \qquad \mathsf{rdy}(n_r, n_{r'}) \leftarrow \mathsf{ap}(n_{r'}) \ ,$$
$$\mathsf{ok}(n_r) \leftarrow \mathsf{rdy}(n_r, n_{r_1}), \ldots, \mathsf{rdy}(n_r, n_{r_k}) \ , \qquad \mathsf{rdy}(n_r, n_{r'}) \leftarrow \mathsf{bl}(n_{r'}) \ .$$

*Furthermore, $\mathcal{T}_o[<]$ is a shortcut for $\mathcal{T}_o[\Pi_<,<]$, where $\Pi_< = \{r, r' \in \Pi \mid r < r'\}$.*

The order $<$ on $\Pi$ singles out the answer sets of $\Pi$ whose rule application and/or blockage is compatible with $<$. That is, given the order $r_1 < r_2$, the higher-ranked rule $r_2$ must be applied or found to be blocked before $r_1$. In other words, $<$ selects those answer sets $X$ of $\Pi$ that can be generated by an appropriate sequence of elements of $\Pi_X$. We refer for a detailed formal elaboration to Delgrande, Schaub, and Tompits [1].[5]

### 3.4 Debugging Programs with Variables

With two exceptions, the translations discussed so far carry over to programs with variables simply by using parametrized names. To this end, replace every occurrence of a name $n_r$ by $n_r(\boldsymbol{X})$, where $\boldsymbol{X}$ is the sequence of variables occurring in rule $r$. Accordingly, a rule $(r : p(X,Y) \leftarrow q(1,X), \mathit{not}\ s(Y))$ gets the name $n_r(X,Y)$, for instance, yielding

$$\mathsf{ap}(n_r(X,Y)) \leftarrow \mathsf{ok}(n_r(X,Y)), q(1,X), \mathit{not}\ s(Y) \qquad \text{and}$$
$$\{\, p(X,Y) \,\} \leftarrow \mathsf{ap}(n_r(X,Y)) \ .$$

Similarly, we get for an atomic formula $p(X,Y)$:

$$\{\, \mathsf{ab}_l(p(X,Y)) \,\} \leftarrow \mathit{not}\ \mathsf{ab}_c(p(X,Y)) \ .$$

The final representation (e.g., with or without function symbols) of an atomic formula like $\mathsf{ab}_l(p(X,Y))$ depends on the language capacities of the target ASP solver.

The first exception is resolved by replacing the rules in Item 2 of Definition 2 by

$$\{\, a(\boldsymbol{X}) \,\} \leftarrow \mathit{not}\ \mathsf{apx}(a(\boldsymbol{X})) \quad \text{and} \quad \mathsf{ab}_c(a(\boldsymbol{X})) \leftarrow a(\boldsymbol{X}), \mathit{not}\ \mathsf{apx}(a(\boldsymbol{X})) \ ,$$

and by adding, for each $r \in \mathit{def}(a,\Pi)$, the rule

$$\mathsf{apx}(a(\boldsymbol{X})) \leftarrow \mathsf{ap}(n_r(\boldsymbol{Y})) \ ,$$

[5] For obtaining the precise semantics of [1], predicate bl must actually be replaced by another predicate $\mathsf{bl}^\star$ using no nested expressions like $\mathit{not}\ \mathit{not}\ c$. The definition of $\mathsf{bl}^\star$ is omitted here.

where apx is an auxiliary predicate symbol and $\boldsymbol{X}$ is a subsequence of the variables in $\boldsymbol{Y}$. An atom $\mathsf{apx}(a(\boldsymbol{c}))$ indicates that $a(\boldsymbol{c})$ is not derivable given that all putative rules in $def(a, \Pi)$ are inapplicable.

The second exception concerns translation $\mathcal{T}_o$ which necessitates that the set of dominating rules $\{r'' \mid r < r''\}$ is ground in order to guarantee that all instances have been applied or blocked (cf. Definition 4). Unlike this, the name of $r$, $n_r$, can be parametrized.

Apart from firmer instantiations, further optimizations are possible by use of domain predicates for names and atoms, like $name$ and $atom$. For instance, a rule $(\mathsf{ok}(n_r) \leftarrow not\ \overline{\mathsf{ok}}(n_r))$ could be represented as $(\mathsf{ok}(X) \leftarrow name(X), not\ \overline{\mathsf{ok}}(X))$.

### 3.5 Implementation

The tool `spock` implements a collection of methods for debugging ASP programs, following the ideas introduced above. The system can be used either with `DLV` [11] or with `Smodels` [7] (together with `Lparse`) and is obtainable at [2], where also some further information about the system is available.

The tool is written in Java 5.0 and published under the GNU General Public License. The input of `spock` is a logic program in the core language of either `DLV` or `Smodels`, read from the system standard input and/or from (multiple) files. The output varies according to the selected functionalities, determined by a set of options. The most important syntax extension of the input programs, however, is the labeling of rules, allowing debugging mechanisms to explicitly refer to certain rules.

## 4 Elements of a Debugging Language

In order to enhance the usability and convenience of our debugging technique, we provide in this section some basic elements for a higher-level debugging language. The idea is to compile a program once and to explore it subsequently by means of debugging statements. To this end, we assume that the program $\Pi$ has been compiled into $\mathcal{T}_\mathsf{K}[\Pi]$. A debugging request, $\Delta$, formulated in the debugging language, is compiled by means of a function $\mathcal{D}$ into a tagged program $\mathcal{D}[\Delta]$ such that the debugging results are read off the answer sets of the combined tagged program $\mathcal{T}_\mathsf{K}[\Pi] \cup \mathcal{D}[\Delta]$. We stress that the subsequent discussion is intended as a starting point only, given at a rather informal level; a detailed elaboration of the full language will be explored elsewhere.

The first type of expressions are referred to as *enforcement statements* since they may alter the original set of answer sets. The following expressions are enforcement statements (in what follows, let $C$ be a set of literals over $At(\Pi)$ and $n_r$ the name of some $r \in \Pi$):

- **block** $n_r$ **if** $C$;
- **apply** $n_r$ **if** $C$;
- **assign** $a_1 = v_1, \ldots, a_k = v_k$ **if** $C$, where $a_i \in At(\Pi)$ and $v_i \in \{\mathbf{t}, \mathbf{f}\}$ for $i = 1, \ldots, k$.

The semantics of enforcement statements is given via a compilation function $\mathcal{D}$:

- $\mathcal{D}[\textbf{block}\ n_r\ \textbf{if}\ C] = \{\mathsf{ko}(n_r) \leftarrow C,\ \underline{\mathsf{bl}}(n_r) \leftarrow C,\ \overline{\underline{\mathsf{ok}}}(n_r) \leftarrow C\}$;
- $\mathcal{D}[\textbf{apply}\ n_r\ \textbf{if}\ C] = \{\mathsf{ap}(n_r) \leftarrow C,\ \overline{\underline{\mathsf{ok}}}(n_r) \leftarrow C\}$;
- $\mathcal{D}[\textbf{assign}\ a_1{=}v_1, \ldots, a_k{=}v_k\ \textbf{if}\ C] = \{a_i \leftarrow C \mid v_i = \mathbf{t}\} \cup \{\leftarrow a_i, C \mid v_i = \mathbf{f}\}$.

For example, $\textbf{block}\ n_r(X)\ \textbf{if}\ not\ \ (X > 5)$ blocks rule $r$ unless instantiated with objects larger than 5. Also, one may use tags within the precondition, as in $\textbf{block}\ n_r(X)\ \textbf{if}\ \mathsf{ap}(n_s(X))$.

Unlike the above, *projection statements* do not alter the original set of answer sets but return a specific subset of the original answer sets:

- $\textbf{blocked}\ n_1, \ldots, n_k\ \textbf{if}\ C$;
- $\textbf{applied}\ n_1, \ldots, n_k\ \textbf{if}\ C$;
- $\textbf{assigned}\ a_1 = v_1, \ldots, a_k = v_k\ \textbf{if}\ C$, where $a_i \in At(\Pi)$ and $v_i \in \{\mathbf{t}, \mathbf{f}\}$ for $i = 1, \ldots, k$.

The semantics of projection statements is again given in terms of a compilation:

- $\mathcal{D}[\textbf{blocked}\ n_1, \ldots, n_k\ \textbf{if}\ C] = \{\leftarrow not\ \mathsf{bl}(n_1), \ldots, not\ \mathsf{bl}(n_k), C\}$;
- $\mathcal{D}[\textbf{applied}\ n_1, \ldots, n_k\ \textbf{if}\ C] = \{\leftarrow not\ \mathsf{ap}(n_1), \ldots, not\ \mathsf{ap}(n_k), C\}$;
- $\mathcal{D}[\textbf{assigned}\ a_1 = v_1, \ldots, a_k = v_k\ \textbf{if}\ C] = \{\leftarrow not\ a_i, C \mid v_i = \mathbf{t}\} \cup \{\leftarrow a_i, C \mid v_i = \mathbf{f}\}$.

For example, $\textbf{blocked}\ n_r(X), n_s(Y)\ \textbf{if}\ X \neq Y$ eliminates answer sets to which rules $r$ and $s$ contribute with different instantiations. Applying the compilation function $\mathcal{D}$, we get

$$\mathcal{D}[\textbf{blocked}\ n_r(X), n_s(Y)\ \textbf{if}\ X \neq Y] = \{\leftarrow not\ \mathsf{bl}(n_r(X)), not\ \mathsf{bl}(n_s(Y)), X \neq Y\}.$$

Observe that the translation of the corresponding enforcement statement would yield multiple rules whose instantiations cannot be controlled in a synchronous fashion.

Next, we introduce expressions for analyzing incoherent situations by extrapolation, correspondingly called *extrapolation statements*:

- $\textbf{extrapolate}\ n_r\ \textbf{if}\ C$;
- $\textbf{extrapolate_}x\ s\ \textbf{if}\ C$, where $x \in \{\mathbf{p}, \mathbf{c}, \mathbf{l}\}$ and $s \in (\{n_r \mid r \in \Pi\} \cup At(\Pi))$;
- $\textbf{minimize}\ X$, where $X \subseteq \{p, c, l\}$.

Further constructs, for instance, statements applying to entire sets of rules and/or atoms, are definable in a straightforward way but omitted for space reasons.

For a program $\Pi$, the function $\mathcal{D}$ defines the semantics of extrapolation statements:

- $\mathcal{D}[\textbf{extrapolate}\ n_r\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{E}}[\Pi, \{r\}]$;
- $\mathcal{D}[\textbf{extrapolate_p}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{P}}[\{r\}]$, provided that $s \in \{n_r \mid r \in \Pi\}$;
- $\mathcal{D}[\textbf{extrapolate_c}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{C}}[\Pi, \{s\}]$, provided that $s \in At(\Pi)$;
- $\mathcal{D}[\textbf{extrapolate_l}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{L}}[\{s\}]$, provided that $s \in At(\Pi)$.

The semantics of the **minimize** command depends on the capacities of the underlying ASP solver. For instance, in `Smodels`, a global minimization of abnormalities, using a binary predicate $\mathtt{ab}/2$, could be expressed as $\mathtt{minimize\{ab}(X, Y)\}$.

A pragmatic variant of extrapolation is *variation*:

– **vary** $a_1, \ldots, a_k$ **if** $C$, where $a_i \in At(\Pi)$ for $i = 1, \ldots, k$.

The semantics of variation statements is as follows:

– $\mathcal{D}[\textbf{vary } a_1, \ldots, a_k \textbf{ if } C] = \{\, \{a_i\} \leftarrow C \mid i = 1, \ldots, k \,\} \cup \{\, \mathsf{ko}(n_r) \leftarrow C \mid r \in \Pi, head(r) = a_i, i = 1, \ldots, k \,\}$.

Finally, we introduce some procedural flavor, which allows for imposing a certain order of rule and/or atom considerations. A basic *ordinal statement* is an expression of the following form:

– $s$ **before** $t$, where $s, t \in \{n_r \mid r \in \Pi\}$ or $s, t \in At(\Pi)$.

Its semantics is defined in the following way:

– $\mathcal{D}[s \textbf{ before } t] = \mathcal{T}_o[\{r_t < r_s\}]$, provided that $s = n_{r_s}$ and $t = n_{r_t}$;
– $\mathcal{D}[s \textbf{ before } t] = \mathcal{T}_o[\{r_t < r_s \mid head(r_s) = s, head(r_t) = t\}]$, provided that $s, t \in At(\Pi)$.

The (atom-based) generalization to sets is defined as $A$ **before** $B$ where $A, B \subseteq At(\Pi)$, and it could be used to debug a generate-and-test encoding by specifying the generate atoms' precedence over the test atoms.

## 5 Discussion and Related Work

Although debugging is mentioned as a possible application in many papers on program analysis (often with the implicit assertion that incoherent and erroneous programs are the same thing), there are relatively few papers that are primarily focused on debugging.

The `noMoRe` system [3] uses a graph-based algorithm for computing the answer sets of a program. The interface to the system allows the computation process to be visualized and animated, so that the user can observe certain parts of the computation process. Given some background knowledge on how answer-set computation algorithms work, this is an intuitive and appealing approach to debugging. However, as it works on ground programs, dealing with larger programs is difficult. Also, at the conceptual level, it blurs the distinction between what a program means and how its answer sets are computed.

Brain and De Vos [4] start by characterizing bugs as mismatches between what the programmer expects and the actual answer sets of a program. Two query algorithms, answering why some set $S$ is in some answer set $A$ and why some set $S$ is not contained in any answer set, can then be used to explore these mismatches. The algorithms suggested are procedural and similar to the ones used in ASP solvers, suggesting that an approach using a program-level transformation would be more practical.

Syrjänen [5] discusses a pragmatic approach to debugging, focusing on incoherent ground programs, whose source of incoherence is an active constraint (or an odd negative cycle). Derived from the field of symbolic diagnosis [12], the corresponding system uses an approach similar to ours, using program transformations to find minimal sets of constraints that would be active and to find odd cycles.

Pontelli and Son [6] adopt the concept of *justifications* [13,14,15] to the context of ASP. Roughly, justifications formalize reasons why an atom is in an answer set.

One important area that is not considered by the existing approaches is that of interfaces to debugging systems. Most of the proposed methods produce very large amounts of structured information, and it is difficult to automatically identify which parts of it are of interest to the programmer [4]. Thus, the design of the debugging interface is critical to the utility of the finished system. It must allow the programmers to quickly and easily focus in on areas that they consider to be erroneous without overloading them with information. This is an open and important area of research if ASP is to achieve truly wide-spread use. Other directions of future work include extending the results given here to handle constructs such as cardinality rules, disjunction, aggregates, and functions. Also, there is a potential to use similar tagging systems to allow programs to reason about their own consistency. We mention *consistency restoring rules* [16] as one such example.

## References


1. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2) (2003) 129–187
2. (`http://www.kr.tuwien.ac.at/research/debug`)
3. Bösel, A., Linke, T., Schaub, T.: Profiling answer set programming: The visualization component of the `noMoRe` system. In Proc. JELIA'04. Springer (2004) 702–705
4. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In Proc. ASP'05. (2005) 141–152
5. Syrjänen, T.: Debugging inconsistent answer set programs. In Proc. NMR'06. (2006) 77–83
6. Pontelli, E., Son, T.: Justifications for logic programs under answer set semantics. In Proc. ICLP'06. Springer (2006) 196–210
7. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
8. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
9. Clark, K.: Negation as failure. In Logic and Data Bases. Plenum (1978) 293–322
10. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
12. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (1987) 57–95
13. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying proofs using memo tables. In Proc. PPDP '00. (2000) 178–189
14. Pemmasani, G., Guo, H., Dong, Y., Ramakrishnan, C., Ramakrishnan, I.: Online justification for tabled logic programs. In Proc. FLOPS'04. Springer (2004) 24–38
15. Specht, G.: Generating explanation trees even for negations in deductive database systems. In Proc. LPE'93. (1993) 8–13
16. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In Proc. Commonsense'03. (2003) 9–18