A Meta-Programming Technique for Debugging Answer-Set Programs[*]

Martin Gebser[1], **Jörg Pührer**[2], **Torsten Schaub**[1], **and Hans Tompits**[2]
[1] Institut für Informatik, Universität Potsdam, Germany, {gebser,torsten}@cs.uni-potsdam.de
[2] Institut für Informationssysteme 184/3, Technische Universität Wien, Austria, {puehrer,tompits}@kr.tuwien.ac.at

Abstract

Answer-set programming (ASP) is widely recognised as a viable tool for declarative problem solving. However, there is currently a lack of tools for *developing* answer-set programs. In particular, providing tools for *debugging* answer-set programs has recently been identified as a crucial prerequisite for a wider acceptance of ASP. In this paper, we introduce a meta-programming technique for debugging in ASP. The basic question we address is why interpretations expected to be answer sets are *not* answer sets of the program to debug. We thus deal with finding *semantical errors* of programs. The explanations provided by our method are based on an intuitive scheme of errors that relies on a recent characterisation of the answer-set semantics. Furthermore, as we are using a meta-programming technique, debugging queries are expressed in terms of answer-set programs themselves, which has several benefits: For one, we can directly use ASP solvers for processing debugging queries. Indeed, our technique can easily be implemented, and we devised a corresponding prototype debugging system. Also, our approach respects the declarative nature of ASP, and the capabilities of the system can easily be extended to incorporate differing debugging features.

Introduction

Answer-set programming (ASP) is a well-known logic-programming paradigm (Baral 2003) that became popular not only because of its fully declarative semantics (Gelfond & Lifschitz 1991) but also in view of the availability of efficient solvers like `DLV` (Leone *et al.* 2006) and `Smodels` (Simons, Niemelä, & Soininen 2002). However, arguably a major reason why ASP has not yet found a more widespread popularity as a problem-solving technique is its lack of suitable *engineering tools* for developing programs. In particular, *debugging* in ASP is an important field that has not been studied thoroughly so far.

The most prominent example of established debugging techniques in logic programming is *tracing* in `PROLOG`, enabling developers to control the evolution of the proof-search tree. Like traditional debugging techniques used in imperative programming, tracing relies on tracking the underlying steps in the execution of a program. However, applying such an approach to ASP results in some decisive drawbacks, undermining the declarativity of answer-set semantics. In particular, establishing canonical tracing-based techniques for debugging answer-set programs requires also a canonical procedure for answer-set computation, and programmers would have to be familiar with the algorithm. This would lead to a shift of perspectives in which an answer-set program is degraded from a declarative problem description to a set of parameters for a static solving algorithm. Therefore, we argue for declarative debugging strategies that are independent of answer-set computation.

Indeed, among the few approaches dealing with debugging of answer-set programs, most rely (fully or largely) on declarative techniques. While Brain & De Vos (2005) essentially outline general considerations on debugging, Syrjänen (2006) focuses on restoring coherence of programs not having any answer set. The latter, among other debugging aspects, is also addressed by Brain *et al.* (2007). In contrast, the technique of Pontelli & Son (2006) supplies justifications for the truth values of atoms with respect to answer sets of a program to debug.

In our approach, we tackle another important question, viz., why interpretations are *not* answer sets of a program under consideration. Thereby, we rely on a *meta-programming technique*, i.e., a program over a meta-language manipulates another program over an object-language. In our case, the program in the object-language, say Π, is a propositional (disjunctive) logic program that we want to debug, whereas the program in the meta-language, denoted $\mathcal{D}(\Pi)$, is also an answer-set program, but non-ground and non-disjunctive. Errors in Π are reflected by special meta-atoms in the answer sets of $\mathcal{D}(\Pi)$. More precisely, each answer set of $\mathcal{D}(\Pi)$ describes the behaviour of Π under an interpretation I that is not an answer set of Π. Conversely, for each interpretation I for Π not being an answer set of Π, there is at least one corresponding answer set of $\mathcal{D}(\Pi)$.

For illustrating our technique, consider program Π_{ex}, consisting of the rules

$$\begin{aligned} r_1 &= \mathit{night} \lor \mathit{day} \leftarrow, \\ r_2 &= \mathit{bright} \leftarrow \mathit{candlelight}, \\ r_3 &= \leftarrow \mathit{night}, \mathit{bright}, \mathit{not}\ \mathit{torch\_on}, \\ r_4 &= \mathit{candlelight} \leftarrow . \end{aligned}$$

Π_{ex} has a single answer set, $\{\mathit{candlelight}, \mathit{day}, \mathit{bright}\}$,

[*]This work was partially supported by the Austrian Science Fund (FWF) under grant P18019.

but the programmer expects also $I = \{candlelight, night, bright\}$ to be an answer set of Π_{ex}. Now, the answer sets of the meta-program $\mathcal{D}(\Pi_{ex})$, projected to the relevant predicates, include $A = \{int(l_{night}), int(l_{bright}), int(l_{candlelight}), violated(l_{r_3})\}$. Here, l_x (for $x \in \{night, bright, candlelight, r_3\}$) is a label for an atom or a rule in the object-language, and the atoms over the predicate $int/1$ describe the considered interpretation I. From the occurrence of $violated(l_{r_3})$ in A, we conclude that r_3 is an integrity constraint violated under I.

In general, our approach provides an intuitive classification of errors for the addressed debugging problem, based on an alternative characterisation of answer sets (Lee 2005; Ferraris, Lee, & Lifschitz 2006). Our approach is not restricted to querying why a single interpretation is not an answer set of a program Π, but also allows for answering why a specified *class* of interpretations for Π does not contain answer sets of Π. This class can be defined by query programs using the meta-atoms of $\mathcal{D}(\Pi)$. Here, various criteria can be combined for choosing the interpretations to be considered, e.g., we can select interpretations in which specific atoms are (not) contained, specific rules are (not) applicable, or specific errors occur. Note, however, that the resulting answer sets of the meta-program refer to the specified interpretations for Π *individually*, not to the class as a whole.

A particular advantage of our method is that it is easily implementable, by using existing ASP solvers. Indeed, we incorporated our technique into the system `spock`, which is a prototype debugging tool originally developed for the debugging approach by Brain *et al.* (2007). We only require a single (linear) transformation of the original program that can then be exploited by highly extensible non-ground modules. Furthermore, our approach is also user-friendly, since debugging queries are formed in the original programming language.

We finally mention that, to the best of our knowledge, our method for debugging in ASP is the first coping with disjunctive programs. A more detailed account of a preliminary version of our technique is discussed by Pührer (2007).

Preliminaries

We assume that the reader is familiar with logic programming and answer-set semantics (cf. Baral (2003) for a comprehensive textbook about ASP) and only briefly recall the necessary concepts.

We consider programs in a function-free first-order language (including at least one constant). As for *terms*, strings starting with uppercase (resp., lowercase) letters denote *variables* (resp., *constants*). An *atom* is an expression of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol and each t_i is a term. A *literal* is either an atom a or an expression of the form $not\ a$, where not denotes *default negation*.

A *disjunctive logic program* (*DLP*), or simply a *program*, is a finite set of rules of the form

$$h_1 \vee \cdots \vee h_l \leftarrow b_1, \dots, b_m, not\ b_{m+1}, \dots, not\ b_n\ , \quad (1)$$

where $l, m, n \geq 0$ and each h_i, b_j is an *atom*. For a rule r of form (1), we call $H(r) = \{h_1, \dots, h_l\}$ the *head* of r and $B(r) = \{b_1, \dots, b_m, not\ b_{m+1}, \dots, not\ b_n\}$ the *body* of r. If $H(r) = \emptyset$, r is an *integrity constraint*. Furthermore, the *positive body* and the *negative body* of r are given by $B^+(r) = \{b_1, \dots, b_m\}$ and $B^-(r) = \{b_{m+1}, \dots, b_n\}$, respectively. Literals (resp., rules, programs) are *ground* if they are variable-free. Non-ground literals (resp., rules, programs) amount to their ground instantiations, i.e., all instances obtained by substituting variables with constants from the (implicit) language. The ground instantiation of a program Π is denoted by $Gr(\Pi)$. By $At(\Pi)$ we denote the set of all (ground) atoms occurring in $Gr(\Pi)$.

The answer-set semantics for DLPs Π is defined as follows (Gelfond & Lifschitz 1991). An *interpretation for* Π is a set $I \subseteq At(\Pi)$ of ground atoms. Whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$, for a rule $r \in Gr(\Pi)$, we say that r is *applicable under* I, and *blocked under* I otherwise. Furthermore, r is *satisfied by* I (symbolically $I \models r$) if r is blocked under I or $H(r) \cap I \neq \emptyset$, otherwise r is *unsatisfied by* I (or *violated under* I). Interpretation I *satisfies* a program Π (symbolically $I \models \Pi$) if $I \models r$, for every $r \in Gr(\Pi)$. If $I \models \Pi$, we call I a *model* of Π. The *reduct*, Π^I, of Π with respect to I is the set of all rules $h_1 \vee \cdots \vee h_l \leftarrow b_1, \dots, b_m$ resulting from a rule $r \in Gr(\Pi)$ of form (1) with $B^-(r) \cap I = \emptyset$. I is an *answer set* of Π if I is a minimal model of Π^I. By $AS(\Pi)$ we denote the set of all answer sets of Π. Whenever $AS(\Pi) = \emptyset$, we say that Π is *incoherent*.

An alternative characterisation of answer sets is based on the concept of support (Lee 2005). For a program Π, an interpretation I for Π, and $G \subseteq At(\Pi)$, we call a rule $r \in Gr(\Pi)$ a *support for* G with respect to I if r is applicable under I, $H(r) \cap G \neq \emptyset$, and $H(r) \cap I \subseteq G$. A support r for G with respect to I is *external* if $B^+(r) \cap G = \emptyset$. We call G *supported* (resp., *externally supported*) *by* Π with respect to I if there is some support (resp., external support) $r \in Gr(\Pi)$ for G with respect to I. Furthermore, G is *unsupported* (resp., *unfounded*) *by* Π with respect to I if G is not supported (resp., not externally supported) by Π with respect to I. The *positive dependency graph* of Π is given by $(At(\Pi), \{(h, b) \mid r \in Gr(\Pi), h \in H(r), b \in B^+(r)\})$. A non-empty set $\Gamma \subseteq At(\Pi)$ is a *loop* of Π if, for all $a, b \in \Gamma$, there is a path of non-zero length from a to b in the positive dependency graph of Π such that all atoms in the path belong to Γ. As shown by Lee (2005), an interpretation I for Π is an answer set of Π iff I is a model of Π such that each singleton $\{a\} \subseteq I$ is supported and each loop $\Gamma \subseteq I$ of Π is externally supported by Π with respect to I.

Categories of Error

Our interest lies in finding *semantical errors* of programs, i.e., mismatches between the intended meaning and the actual meaning of a program. In ASP, the semantics of a program is given by its answer sets, thus, in this context, errors can be identified as *discrepancies between the expected and the actual answer sets of the program.*

As discussed in the introductory section, we deal with the question why an interpretation I for Π is not an answer set of a program Π to debug. Accordingly, we consider reasons causing this situation as *errors*. We distinguish between four different types of errors:

$$
\begin{aligned}
\pi_{int} = \{&\mathit{int}(A)\leftarrow \mathit{atom}(A), \mathit{not}\ \overline{\mathit{int}}(A),\\
&\overline{\mathit{int}}(A)\leftarrow \mathit{atom}(A), \mathit{not}\ \mathit{int}(A)\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{sat} = \{&\mathit{hasHead}(R)\leftarrow \mathit{head}(R, \_),\\
&\mathit{someHInI}(R)\leftarrow \mathit{head}(R, A), \mathit{int}(A),\\
&\mathit{violated}(C)\leftarrow \mathit{ap}(C), \mathit{not}\ \mathit{hasHead}(C),\\
&\mathit{unsatisfied}(R)\leftarrow \mathit{ap}(R), \mathit{hasHead}(R),\\
&\qquad \mathit{not}\ \mathit{someHInI}(R)\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{supp} = \{&\mathit{otherHInI}(R, A1)\leftarrow \mathit{head}(R, A2), \mathit{int}(A2),\\
&\qquad \mathit{head}(R, A1), A1 \neq A2,\\
&\mathit{supported}(A)\leftarrow \mathit{head}(R, A), \mathit{ap}(R),\\
&\qquad \mathit{not}\ \mathit{otherHInI}(R, A),\\
&\mathit{unsupported}(A)\leftarrow \mathit{int}(A), \mathit{not}\ \mathit{supported}(A)\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{noas} = \{&\mathit{noAnswerSet}\leftarrow \mathit{unsatisfied}(\_),\\
&\mathit{noAnswerSet}\leftarrow \mathit{violated}(\_),\\
&\mathit{noAnswerSet}\leftarrow \mathit{unsupported}(\_),\\
&\mathit{noAnswerSet}\leftarrow \mathit{ufLoop}(\_),\\
&\leftarrow \mathit{not}\ \mathit{noAnswerSet}\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{ap} = \{&\mathit{bl}(R)\leftarrow \mathit{bodyP}(R, A), \overline{\mathit{int}}(A),\\
&\mathit{bl}(R)\leftarrow \mathit{bodyN}(R, A), \mathit{int}(A),\\
&\mathit{ap}(R)\leftarrow \mathit{rule}(R), \mathit{not}\ \mathit{bl}(R)\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{ufloop} = \{&\mathit{ufLoop}(A)\leftarrow \mathit{int}(A), \mathit{supported}(A),\\
&\qquad \mathit{not}\ \overline{\mathit{ufLoop}}(A),\\
&\overline{\mathit{ufLoop}}(A)\leftarrow \mathit{int}(A), \mathit{not}\ \mathit{ufLoop}(A),\\
&\mathit{someBInLoop}(R)\leftarrow \mathit{bodyP}(R, A), \mathit{ufLoop}(A),\\
&\mathit{someHOutLoop}(R)\leftarrow \mathit{head}(R, A), \overline{\mathit{ufLoop}}(A),\\
&\leftarrow \mathit{head}(R, A), \mathit{ufLoop}(A),\\
&\qquad \mathit{ap}(R), \mathit{not}\ \mathit{someHOutLoop}(R),\\
&\qquad \mathit{not}\ \mathit{someBInLoop}(R),\\
&\mathit{dpcy}(A1, A2)\leftarrow \mathit{head}(R, A1), \mathit{bodyP}(R, A2),\\
&\qquad \mathit{ufLoop}(A1), \mathit{ufLoop}(A2),\\
&\qquad \mathit{ap}(R), \mathit{not}\ \mathit{someHOutLoop}(R),\\
&\mathit{dpcy}(A1, A2)\leftarrow \mathit{dpcy}(A1, A3), \mathit{dpcy}(A3, A2),\\
&\leftarrow \mathit{ufLoop}(A1), \mathit{ufLoop}(A2),\\
&\qquad \mathit{not}\ \mathit{dpcy}(A1, A2)\}
\end{aligned}
$$

Figure 1: Static Modules of Meta-Program $\mathcal{D}(\Pi)$

1. *Unsatisfied rules*: If a rule $r \in Gr(\Pi)$ such that $H(r) \neq \emptyset$ is unsatisfied by I, the logical implication expressed by r is false under I, and thus I is not a classical model of Π.
2. *Violated integrity constraints*: If an integrity constraint $c \in Gr(\Pi)$ is applicable under I, the fact that $H(c) = \emptyset$ implies $I \not\models c$, and thus I cannot be an answer set of Π.
3. *Unsupported atoms*: If $\{a\} \subseteq I$ is unsupported by Π with respect to I, no rule in $Gr(\Pi)$ allows for deriving a, and thus I is not a minimal model of Π^I.
4. *Unfounded loops*: If a loop $\Gamma \subseteq I$ of Π is unfounded by Π with respect to I, there is no acyclic derivation for the atoms in Γ, and thus I is not a minimal model of Π^I.

Given a program Π and an interpretation I for Π, the meta-program $\mathcal{D}(\Pi)$, detailed in the next section, detects these four errors via occurrences of the following atoms in answer sets of $\mathcal{D}(\Pi)$: $\mathit{unsatisfied}(l_r)$ indicates that a rule $r \in Gr(\Pi)$ is unsatisfied by I; $\mathit{violated}(l_c)$ indicates that an integrity constraint $c \in Gr(\Pi)$ is violated under I; $\mathit{unsupported}(l_a)$ indicates that, for an atom $a \in I$, $\{a\}$ is unsupported by Π with respect to I; and $\mathit{ufLoop}(l_a)$ indicates that an atom a belongs to some loop $\Gamma \subseteq I$ of Π that is unfounded by Π with respect to I. We call the predicates of these meta-atoms *error-indicating predicates*.

A major advantage of this classification scheme is that it is based on a rather intuitive characterisation of answer sets (Lee 2005). In particular, we find the concept of an unfounded loop (exploited in our approach) quite natural, metaphorically speaking corresponding to a serpent biting its tail. Compared to subset minimality relative to the reduct of a program, this notion seems closer to the program written and thus more natural to understand, especially for novice answer-set programmers. Furthermore, the selection of error types is complete in the sense that it suffices to decide whether an interpretation is an answer set and therefore provides explanations for an interpretation not being an answer set, whenever this is the case. Note that the scheme is redundant, as violated integrity constraints could be treated as unsatisfied rules and, since unsupported atoms constitute trivial unfounded sets, also the last two categories could be merged. However, by allowing more types of errors, we get a more differentiated insight into the context of a program Π under an interpretation I. From a developer's point of view, integrity constraints play a rather different role than other rules, as they are used to restrict answer sets rather than to generate them. Therefore, we find it useful to handle their violations separately. Moreover, by this differentiation, our debugging technique allows for restricting the search for errors to interpretations that do not violate any integrity constraint. Finally, unsupported atoms and unfounded loops are usually corrected in a different way, since for correcting the latter, we typically need to investigate multiple rules of Π involved in connecting the loop, which is unnecessary when coping with unsupported atoms.

Basic Approach

We now provide the details of the meta-program $\mathcal{D}(\Pi)$, for a given program Π, and its central properties. The program $\mathcal{D}(\Pi)$ is composed in a modular fashion, consisting of an input module, $\pi_{in(\Pi)}$, representing the program Π to debug, and several fixed modules. We make use of a standard meta-programming encoding of Π, often found in the literature (Eiter & Polleres 2006; Sterling & Shapiro 1994). Thereby, we assume that all rules $r \in \Pi$ and all atoms $a \in At(\Pi)$ have unique labels, $l(r)$ and $l(a)$, respectively. For readability, we denote $l(x)$ by l_x.

We start with the program $\pi_{in(\Pi)}$ translating a program Π to debug into the meta-language, serving as an input module for our meta-program.

Definition 1 *Let Π be a ground DLP. Then, the* input program *for Π is the following set of facts:*

$$\begin{aligned}\pi_{in(\Pi)} = \{&atom(l_a) \leftarrow && \mid a \in At(\Pi)\} \cup \\ \{&rule(l_r) \leftarrow && \mid r \in \Pi\} \cup \\ \{&head(l_r, l_a) \leftarrow && \mid r \in \Pi, a \in H(r)\} \cup \\ \{&bodyP(l_r, l_a) \leftarrow && \mid r \in \Pi, a \in B^+(r)\} \cup \\ \{&bodyN(l_r, l_a) \leftarrow && \mid r \in \Pi, a \in B^-(r)\} \,.\end{aligned}$$

Observe that $\pi_{in(\Pi)}$ just consists of facts stating which rules and atoms occur in Π and, for each rule $r \in \Pi$, which atoms are contained in $H(r)$, $B^+(r)$, and $B^-(r)$, respectively.

Given $\pi_{in(\Pi)}$, the meta-program $\mathcal{D}(\Pi)$ is as follows:

Definition 2 *Let Π be a ground DLP. Then, the* meta-program $\mathcal{D}(\Pi)$ *for Π consists of $\pi_{in(\Pi)}$ together with all programs from Figure 1, i.e., $\mathcal{D}(\Pi) = \pi_{in(\Pi)} \cup \pi_{int} \cup \pi_{ap} \cup \pi_{sat} \cup \pi_{supp} \cup \pi_{ufloop} \cup \pi_{noas}$.*

Note that $\mathcal{D}(\Pi)$ is both non-ground and non-disjunctive.

In what follows, we describe the meanings and important properties of the individual modules from Figure 1.

Modules π_{int} and π_{ap} The purpose of module π_{int} is to guess an (arbitrary) interpretation I for the program Π to debug. In fact, each answer set A of $\mathcal{D}(\Pi)$ reveals properties of Π under interpretation $I = \{a \mid int(l_a) \in A\}$. Intuitively, the two rules of π_{int} partition the atoms of Π into two categories, either belonging to I or not.

Based on the interpretation I guessed by π_{int}, for each rule of Π, module π_{ap} checks whether it is applicable or blocked under I. We thus obtain the following behaviour:

Theorem 1 *Let Π be a ground DLP. Then, for any $A \in AS(\mathcal{D}(\Pi))$, $I = \{a \mid int(l_a) \in A\}$ is an interpretation for Π satisfying the following conditions:*

1. *$ap(l_r) \in A$ iff $r \in \Pi$ and r is applicable under I, and*
2. *$bl(l_r) \in A$ iff $r \in \Pi$ and r is blocked under I.*

Modules π_{sat}, π_{supp}, and π_{ufloop} The modules π_{sat}, π_{supp}, and π_{ufloop} are central for debugging, detecting reasons why I is not an answer set of Π. More specifically, module π_{sat} detects rules of Π that are unsatisfied by I, thereby distinguishing integrity constraints. Furthermore, π_{supp} detects atoms $a \in I$ such that $\{a\}$ is unsupported by Π with respect to I. The purpose of π_{ufloop} is to detect loops of Π that are unfounded by Π with respect to I. Since such loops may intersect or be subsets of one another, we cannot provide all of them within a single answer set of $\mathcal{D}(\Pi)$, at least not without using involved enumeration strategies. Rather, we allow multiple answer sets of $\mathcal{D}(\Pi)$, each of which inspects at most one unfounded loop of Π.

Even though many loops of Π might be unfounded with respect to the same interpretation I, not all of them are critical in the sense that they incipiently spoil the minimality of I relative to the reduct Π^I. In order to focus the consideration of unfounded loops, we make use of a recent result by Gebser, Lee, & Lierler (2006). For a ground program Π, an interpretation I for Π, and $G \subseteq At(\Pi)$, let $\Pi_I(G)$ denote the set of all $r \in \Pi$ such that r is a support for G with respect to I. If I is a model of Π such that each singleton $\{a\} \subseteq I$ is supported by Π with respect to I, and if I is still not an answer set of Π, then some loop $\Gamma \subseteq I$ of Π such that Γ is also a loop of $\Pi_I(\Gamma)$ is unfounded by Π with respect to I. Module π_{ufloop} makes use of this fact and focuses on such loops. More precisely, the first two rules of π_{ufloop} permit guessing an unfounded loop among the atoms supported by Π with respect to I. For the guessed atoms $a \in I$, meta-atom $ufLoop(l_a)$ is included in a corresponding answer set A of $\mathcal{D}(\Pi)$. The next three rules of π_{ufloop} make sure that $\Gamma = \{a \mid ufLoop(l_a) \in A\}$ is unfounded by Π with respect to I. Provided that $\Gamma \neq \emptyset$, the last three rules of π_{ufloop} check that Γ is indeed a loop of $\Pi_I(\Gamma)$, which also implies that Γ is a loop of Π.

Module π_{noas} We use module π_{noas} to eliminate potential answer sets of $\mathcal{D}(\Pi)$ that do not explain why an interpretation I is not an answer set of Π, regardless of whether I is (not) an answer set of Π. This is because the absence of atoms over the error-indicating predicates in an answer set A of $(\mathcal{D}(\Pi) \setminus \pi_{noas})$ does not necessarily mean that $I = \{a \mid int(l_a) \in A\}$ is an answer set of Π. In fact, even if I comprises some loop of Π that is unfounded by Π with respect to I, A does not have to contain any meta-atom of the form $ufLoop(l_a)$. Hence, the absence of atoms over error-indicating predicates does, in general, not imply that I is an answer set of Π. However, the filtering by module π_{noas} results in the following central property of our approach:

Theorem 2 *Let Π be a ground DLP and I an interpretation for Π. Then, I is an answer set of Π iff there is no $A \in AS(\mathcal{D}(\Pi))$ such that $I = \{a \mid int(l_a) \in A\}$.*

Knowing that every answer set A of $\mathcal{D}(\Pi)$ expresses that $I = \{a \mid int(l_a) \in A\}$ is not an answer set of Π, we next detail the meaning of atoms over error-indicating predicates contained in A.

Theorem 3 *Let Π be a ground DLP. Then, for any $A \in AS(\mathcal{D}(\Pi))$, $I = \{a \mid int(l_a) \in A\}$ is an interpretation for Π satisfying the following conditions:*

1. *$unsatisfied(l_r) \in A$ iff $r \in \Pi$ such that $H(r) \neq \emptyset$ and r is unsatisfied by I,*
2. *$violated(l_c) \in A$ iff $c \in \Pi$ such that $H(c) = \emptyset$ and c is violated under I,*
3. *$unsupported(l_a) \in A$ iff $a \in I$ such that $\{a\}$ is unsupported by Π with respect to I, and*
4. *if $ufLoop(l_a) \in A$, then $\Gamma = \{a \mid ufLoop(l_a) \in A\} \subseteq I$ is a loop of $\Pi_I(\Gamma)$ such that Γ is unfounded and each $\{a\} \subseteq \Gamma$ is supported by Π with respect to I.*

Note that in the last item there is no "iff"-condition since I may contain several loops of Π satisfying the "then"-part. Such a situation gives rise to multiple answer sets of $\mathcal{D}(\Pi)$.

Finally, the following result describes how interpretations for the program Π to debug induce answer sets of $\mathcal{D}(\Pi)$:

Theorem 4 *Let Π be a ground DLP. Then, for every interpretation I for Π:*

1. *there is exactly one answer set A of $\mathcal{D}(\Pi)$ such that $\{a \mid ufLoop(l_a) \in A\} = \emptyset$ iff some $r \in \Pi$ (resp., $c \in \Pi$) is*

unsatisfied by I (resp., violated under I) or some $\{a\} \subseteq I$ is unsupported by Π with respect to I, and

2. *there is exactly one answer set A of $\mathcal{D}(\Pi)$ such that $\Gamma = \{a \mid ufLoop(l_a) \in A\} \neq \emptyset$ iff $\Gamma \subseteq I$ is a loop of $\Pi_I(\Gamma)$ such that Γ is unfounded and each $\{a\} \subseteq \Gamma$ is supported by Π with respect to I.*

Applying the Meta-Program

Usually, many interpretations for a program Π are not answer sets of Π. Since $\mathcal{D}(\Pi)$ can potentially explain each of them, it is sensible to prune unwanted information. Generally, every meta-atom in $\mathcal{D}(\Pi)$ provides a handle to direct the search for errors. This allows a programmer to specify knowledge about her or his expected results, e.g., which atoms should (not) be contained in expected answer sets or which rules should (not) be applicable. For reducing debugging information, a programmer can thus join $\mathcal{D}(\Pi)$ with a query program Q, specifying intended results.

Example 1 *Consider program Π_1, consisting of the rules*

$$\begin{aligned} &r_1 = active \vee awake \vee sleeping \leftarrow, \\ &r_2 = awake \leftarrow active, \\ &r_3 = tired \vee rested \leftarrow awake, not\ active. \end{aligned}$$

The answer sets of Π_1 are given by $\{sleeping\}$, $\{awake, tired\}$, and $\{awake, rested\}$. Assume that the programmer wonders why $I_1 = \{active, awake\}$ is not an answer set of Π_1. The corresponding meta-program $\mathcal{D}(\Pi_1)$ has 29 answer sets. This number can be reduced to a single answer set by joining $\mathcal{D}(\Pi_1)$ with the following set Q_1 of constraints, pruning all answer sets not associated to I_1:

$$\begin{aligned} &\leftarrow int(l_{sleeping}), \quad \leftarrow int(l_{tired}), \quad \leftarrow int(l_{rested}), \\ &\leftarrow \overline{int}(l_{active}), \quad \leftarrow \overline{int}(l_{awake}). \end{aligned}$$

The single answer set of $\mathcal{D}(\Pi_1) \cup Q_1$, projected to $int/1$ and the error-indicating predicates, is $\{int(l_{active}), int(l_{awake}), unsupported(l_{active})\}$. Therefore, $\{active\}$ is unsupported by Π_1 with respect to I_1.

Example 2 *Consider program Π_2, consisting of the rules*

$$\begin{aligned} r_1 &= \quad goodJob \leftarrow goodAppearance, \\ r_2 &= \quad highIncome \leftarrow goodJob, \\ r_3 &= \quad goodFood \leftarrow highIncome, \\ r_4 &= \quad healthy \leftarrow goodFood, sportive, \\ r_5 &= goodAppearance \leftarrow healthy, \\ r_6 &= \quad sportive \leftarrow, \end{aligned}$$

possessing one answer set, $\{sportive\}$. Let the interpretation of interest be $I_2 = \{sportive, goodJob, highIncome, goodFood, healthy, goodAppearance\}$. Using a query Q_2, defined in a similar fashion as query Q_1 from Example 1, the single answer set of $\mathcal{D}(\Pi_2) \cup Q_2$, projected to the error-indicating predicates, is $A = \{ufLoop(l_{goodJob}), ufLoop(l_{highIncome}), ufLoop(l_{goodFood}), ufLoop(l_{healthy}), ufLoop(l_{goodAppearance})\}$, exposing $\{a \mid ufLoop(l_a) \in A\}$ as a loop of Π_2 that is unfounded by Π_2 with respect to I_2.

Next, we look at a situation where we have multiple loops unfounded with respect to the same interpretation.

Example 3 *Consider program Π_3, consisting of the rules*

$$\begin{aligned} r_1 &= \quad fruity \leftarrow fresh, \\ r_2 &= \quad fresh \leftarrow creamy, \\ r_3 &= creamy \leftarrow tasty, \\ r_4 &= \quad tasty \leftarrow fruity, creamy, \end{aligned}$$

and interpretation $I_3 = \{fruity, fresh, creamy, tasty\}$. Applying our technique as above, we get two answer sets of the meta-program, identifying $\Gamma_1 = \{fruity, fresh, creamy, tasty\}$ and $\Gamma_2 = \{creamy, tasty\}$ as loops of Π_3 being unfounded with respect to I_3. While Γ_1 involves all rules of Π_3, Γ_2 is formed by rules r_3 and r_4 only.

Finally, we consider a specific class of interpretations instead of a single one.

Example 4 *Reconsider program Π_{ex} from the introduction and assume that the programmer wants to know why no interpretation for Π_{ex} containing atom night is an answer set of Π_{ex}. Furthermore, she or he is interested only in interpretations under which integrity constraint r_3 is not violated and no atom is unsupported. We devise the following query Q_4 according to these requirements:*

$$\leftarrow \overline{int}(l_{night}), \quad \leftarrow violated(l_{r_3}), \quad \leftarrow unsupported(\_).$$

Then, $\mathcal{D}(\Pi_{ex}) \cup Q_4$ has two answer sets, whose projections to $int/1$ and the error-indicating predicates are as follows: $\{int(l_{candlelight}), int(l_{night}), unsatisfied(l_{r_2})\}$ and $\{int(l_{night}), unsatisfied(l_{r_4})\}$. They express that rule r_2 is unsatisfied by $\{candlelight, night\}$ and r_4 by $\{night\}$.

Related Work

In their paper, Brain & De Vos (2005) identify general requirements of future debugging systems for ASP. Most importantly, they state the need for declarative debugging techniques and formulate the questions why a set of literals is satisfied by a specific answer set of the program to debug or why a set of literals is not satisfied by any answer set, which cover many debugging tasks. However, the algorithms they provide answer these questions only fragmentarily.

The approach by Syrjänen (2006) addresses the issue of debugging incoherent answer-set programs. It is adapted from the field of symbolic diagnosis (Reiter 1987) and designed to find reasons for the absence of answer sets. The identified reasons for incoherence of a non-disjunctive program are integrity constraints and odd cycles. As shown by You & Yuan (1994), for the considered class of programs, incoherence is indeed always caused by integrity constraints or odd cycles. Odd-cycle detection is performed using a meta-programming technique related to ours. In fact, when considering non-disjunctive programs, our meta-program can be combined with the one used by Syrjänen, and thus its functionality can be extended to odd-cycle detection.

Precursing our present method, Brain *et al.* (2007) introduce a debugging technique in ASP by augmenting non-disjunctive programs with auxiliary atoms, called tags, for manipulating the applicability of rules. In its basic version, the method gives information about the applicability of rules with respect to an answer set of the program to debug. This task could also be performed by a modified version of our meta-program. In an extension of their technique, Brain *et al.* aim at "extrapolating" non-existing answer sets, i.e., identifying a minimal number of operations on a program needed to make it coherent. It shares the classification of errors in terms of rule violation, unsupportedness of atoms, and unfoundedness of loops with our new approach.

Pontelli & Son (2006) adopt the concept of justification (Roychoudhury, Ramakrishnan, & Ramakrishnan 2000) to the context of ASP. Here, the general debugging question addressed is why an atom is true or false with respect to an answer set of the program to debug. Answers to this question are given in the form of graphs, called justifications, explaining the truth values of atoms with respect to a given answer set and, in case of so-called online justifications, also with respect to a partial interpretation. The approach nicely complements our technique, as it investigates interrelations within actual answer sets, whereas we focus on the remaining interpretations.

A completely different approach towards debugging in ASP is based on translating program rules into sentences of natural language (Mikitiuk, Moseley, & Truszczyński 2007). The intention is to ease the reading of a program and thereby to help a programmer to detect errors.

Discussion

In our approach, we express debugging queries by answer-set programs, which allows for restricting debugging information to relevant parts. Moreover, they can be refined by using optimisation techniques of answer-set solvers, e.g., weak constraints (Leone *et al.* 2006). Applications include finding error-minimal interpretations within a class of erroneous interpretations and detecting minimal or maximal unfounded loops of a program. Generally, forming queries and analysing the output of the debugging system does not require expert knowledge of ASP beyond the basic understanding needed for programming. As meta-programming is very powerful and flexible, our method is highly extensible, e.g., minor modifications allow for checking standard properties like whether a program is tight or head-cycle-free.

Our approach is easy to implement as it requires only a fixed non-ground program and a trivial translation of the input program to a set of facts. Indeed, we incorporated this functionality into the debugging system `spock` (Brain *et al.* 2007), which is written in Java 5.0 and publicly available at

`www.kr.tuwien.ac.at/research/debug`.

An important issue for future work concerns debugging of non-ground programs, as these typically evolve from real-world applications. Clearly, debugging strategies for ground programs can be applied to the ground instantiations of programs with variables. Here, an important task is relating debugging information about ground programs to their non-ground counterparts. Furthermore, since ground instantiations can be of huge size, keeping the debugging process efficient is a difficulty. As a next step, we plan an empirical evaluation of our technique in such settings. Another open task is investigating the specifics of restricting debugging to certain program modules. Finally, future tools for debugging should provide easy-to-use interfaces and be embedded into integrated development environments.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Brain, M., and De Vos, M. 2005. Debugging Logic Programs under the Answer-Set Semantics. In *Proc. ASP'05*, `CEUR-WS.org`.

Brain, M.; Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2007. Debugging ASP Programs by Means of ASP. In *Proc. LPNMR'07*, 31–43. Springer.

Eiter, T., and Polleres, A. 2006. Towards Automated Integration of Guess and Check Programs in Answer-Set Programming: A Meta-Interpreter and Applications. *Theory and Practice of Logic Programming* 6(1-2):23–60.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2006. A Generalization of the Lin-Zhao Theorem. *Annals of Mathematics and Artificial Intelligence* 47(1-2):79–101.

Gebser, M.; Lee, J.; and Lierler, Y. 2006. Elementary Sets for Logic Programs. In *Proc. AAAI'06*. AAAI Press.

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

Lee, J. 2005. A Model-Theoretic Counterpart of Loop Formulas. In *Proc. IJCAI'05*, 503–508. Professional Book Center.

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

Mikitiuk, A.; Moseley, E.; and Truszczyński, M. 2007. Towards Debugging of Answer-Set Programs in the Language PSpb. In *Proc. ICAI'07*, 635–640. CSREA Press.

Pontelli, E., and Son, T. C. 2006. Justifications for Logic Programs under Answer Set Semantics. In *Proc. ICLP'06*, 196–210. Springer.

Pührer, J. 2007. On Debugging of Propositional Answer-Set Programs. Master's thesis, Technische Universität Wien.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artificial Intelligence* 32(1):57–95.

Roychoudhury, A.; Ramakrishnan, C. R.; and Ramakrishnan, I. V. 2000. Justifying Proofs Using Memo Tables. In *Proc. PPDP'00*, 178–189. ACM.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence* 138(1):181–234.

Sterling, L., and Shapiro, E. 1994. *The Art of PROLOG: Advanced Programming Techniques*. MIT Press.

Syrjänen, T. 2006. Debugging Inconsistent Answer-Set Programs. In *Proc. NMR'06*, 77–83.

You, J.-H., and Yuan, L. Y. 1994. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences* 49(2):334–361.