Catching the Ouroboros: On Debugging Non-ground Answer-Set Programs

JOHANNES OETSCH, JÖRG PÜHRER, AND HANS TOMPITS*

Technische Universität Wien,
Institut für Informationssysteme 184/3,
Favoritenstrasse 9-11, A-1040 Vienna, Austria,
(*e-mail:* {oetsch,puehrer,tompits}@kr.tuwien.ac.at)

submitted 7 February 2010; revised (n/a); accepted 20 March 2010

Abstract

An important issue towards a broader acceptance of answer-set programming (ASP) is the deployment of tools which support the programmer during the coding phase. In particular, methods for *debugging* an answer-set program are recognised as a crucial step in this regard. Initial work on debugging in ASP mainly focused on propositional programs, yet practical debuggers need to handle programs with variables as well. In this paper, we discuss a debugging technique that is directly geared towards non-ground programs. Following previous work, we address the central debugging question why some interpretation is not an answer set. The explanations provided by our method are computed by means of a meta-programming technique, using a uniform encoding of a debugging request in terms of ASP itself. Our method also permits programs containing comparison predicates and integer arithmetics, thus covering a relevant language class commonly supported by all state-of-the-art ASP solvers.

KEYWORDS: answer-set programming, program analysis, debugging

1 Introduction

During the last decade, answer-set programming (ASP) has become a well-acknowledged paradigm for declarative problem solving. Although there exist efficient solvers (see, e.g., Denecker et al. (2009) for an overview) and a considerable body of literature concerning the theoretical foundations of ASP, comparably little effort has been spent on methods to support the development of ASP programs. Especially novice programmers, tempted by the intuitive semantics and expressive power of ASP, may get disappointed and discouraged soon when some observed program behaviour diverges from his or her expectations. Unlike for other programming languages like Java or C++, there is currently little support for *debugging* a program in ASP, i.e., methods to *explain* and *localise* unexpected observations. This is a clear shortcoming of ASP and work in this direction has already started (Brain and De Vos 2005; Syrjänen 2006; Brain et al. 2007; Mikitiuk et al. 2007; Caballero et al. 2008; Gebser et al. 2008; Pontelli et al. 2009; Wittocx et al. 2009).

* This work was partially supported by the Austrian Science Fund (FWF) under grant P21698.

Most of the current debugging approaches for ASP rely on declarative strategies, focusing on *conceptual errors* of programs, i.e., mismatches between the intended meaning and the actual meaning of a program. In fact, an elegant realisation of declarative debugging is to use ASP itself to debug programs in ASP. This has been put forth, e.g., in the approaches of Brain et al. (2007) and Gebser et al. (2008). While the former uses a "tagging" method to decompose a program and applying various debugging queries, the latter is based on a meta-programming technique, i.e., using a program over a meta-language to manipulate a program over an object language (in this case, both the meta-language and the object language are instances of ASP). Such techniques have the obvious benefits of allowing (i) to use reliable state-of-the-art ASP solvers as back-end reasoning engines and (ii) to stay within the same paradigm for both the programming and debugging process. Indeed, both approaches are realised by the system `spock` (Gebser et al. 2009). However, like most other ASP debugging proposals, `spock` can deal only with propositional programs which is clearly a limiting factor as far as practical applications are concerned.

In this paper, we present a debugging method for non-ground programs following the methodology of the meta-programming approach of Gebser et al. (2008) for propositional programs. That is to say, we deal with the problem of finding reasons why some interpretation is *not* an answer set of a given program. This is addressed by referring to a model-theoretic characterisation of answer sets due to Lee (2005): An interpretation I is not an answer set of a program P iff (i) some rule in P is not classically satisfied by I or (ii) I contains some loop of P that is unfounded by P with respect to I. Intuitively, Item (ii) states that some atoms in I are not justified by P in the sense that no rules in P can derive them or that some atoms are in I only because they are derived by a set of rules in a circular way—like the *Ouroboros*, the ancient symbol of a dragon biting its own tail that represents cyclicality and eternity. This characterisation seems to be quite natural and intuitive for *explaining* why some interpretation is not an answer set. Furthermore, a particular benefit is that it can ease the subsequent *localisation* of errors since the witnesses why an interpretation is not an answer set, like rules which are not satisfied, unfounded atoms, or cyclic rules responsible for unfounded loops, can be located in the program or the interpretation.

Although, at first glance, one may be inclined to directly apply the original approach of Gebser et al. (2008) to programs with variables by simply grounding them in a preprocessing step, one problem in such an endeavour is that then it is not immediate clear how to relate explanations for the propositional program to the non-ground program. The more severe problem, however, is that the grounding step requires exponential space and time with respect to the size of the problem instance which yields a mismatch of the overall complexity as checking whether an interpretation is an answer set of some (non-ground) program is complete for Π_2^P (Eiter et al. 2004), and thus the complementary problem why some interpretation is not an answer set is complete for Σ_2^P—our method to decide this problem accounts for this complexity bound and avoids exponential space requirements. Indeed, we devise a *uniform* encoding of our basic debugging problem in terms of a *fixed* disjunctive logic program Γ and an efficient reification of a problem instance as a set $\Delta(P, I)$ of facts, where P is the program to be debugged and I is the interpretation under consideration. Explanations why I is not an answer set of P are then obtained by the answer sets of $\Gamma \cup \Delta(P, I)$.

We stress that the definition of Γ is non-trivial: while the meta-program in the approach of Gebser et al. (2008) for debugging propositional disjunctive programs could be achieved in terms of a normal non-ground program, *by uniformly encoding a Σ_2^P property, we reach the very limits of disjunctive ASP* and have to rely on advanced saturation techniques that inherently require disjunctions in rule heads (Eiter et al. 1997).

Currently, our approach handles disjunctive logic programs with constraints, integer arithmetic, comparison predicates, and strong negation, thus covering a practically relevant program class. Further language constructs, in particular aggregates and weak constraints, are left for future work.

2 Preliminaries

We deal with *disjunctive logic programs* which are finite sets of rules of form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \mathrm{not}\ a_{m+1}, \ldots, \mathrm{not}\ a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, and all a_i are literals over a function-free first-order language Ł. A literal is an atom possibly preceded by the *strong negation* symbol $\neg$. In the sequel, we assume that Ł will be implicitly defined by the considered programs. For a rule r as above, we define the *head* of r as $H(r) = \{a_1, \ldots, a_l\}$, the *positive body* as $B^+(r) = \{a_{l+1}, \ldots, a_m\}$, and the *negative body* as $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. If $n = l = 1$, r is a *fact*; if r contains no disjunction, r is *normal*; and if $l = 0$ and $n > 0$, r is a *constraint*. For facts, we will omit the symbol $\leftarrow$. A literal, rule, or program is *ground* if it contains no variables. Furthermore, a program is normal if all rules in it are normal. Finally, we allow arithmetic and comparison predicate symbols $+$, $*$, $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$ in programs, but these may appear only positively in rule bodies.

Let C be a set of constants. A *substitution over* C is a function ϑ assigning each variable an element of C. We denote by $e\vartheta$ the result of applying ϑ to an expression e. The *grounding* of a program P relative to its Herbrand universe, denoted by *grd*(P), is defined as usual.

An *interpretation* I (over some language Ł) is a finite and consistent set of ground literals (over Ł) that does not contain any arithmetic or comparison predicates. Recall that consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a. The satisfaction relation, $I \models \alpha$, between I and a ground atom, a literal, a rule, a set of literals, or a program α is defined in the usual manner. Note that the presence of arithmetic and comparison operators implies that the domain of our language will normally include natural numbers as well as a linear ordering, $\preceq$, for evaluating the comparison relations (which coincides with the usual ordering in case of constants which are natural numbers).

For any ground program P and any interpretation I, the *reduct*, P^I, of P with respect to I (Gelfond and Lifschitz 1991) is defined as $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P, I \cap B^-(r) = \emptyset\}$. An interpretation I is an *answer set* of a program P iff I is a minimal model of *grd*(P).

We will base our subsequent elaboration on an alternative characterisation of answer sets following Lee (2005), described next. Given a program P, the *positive dependency graph* is a directed graph (V, A), where (i) V equals the Herbrand base of the considered language Ł and (ii) $(a, b) \in A$ iff $a \in H(r)$ and $b \in B^+(r)$, for some rule $r \in \mathit{grd}(P)$. A non-empty set L of ground literals is a *loop*[1] of a program P iff, for each pair $a, b \in L$, there is a path π of length greater than or equal to 0 from a to b in the positive dependency graph of P such that each literal in π is in L.

Let P be a program and I and J interpretations. Then, J is *externally supported by P with respect to I* iff there is a rule $r \in \mathit{grd}(P)$ such that (i) $I \models B^+(r)$ and $I \cap B^-(r) = \emptyset$, (ii) $H(r) \cap J \neq \emptyset$, (iii) $(H(r) \setminus J) \cap I = \emptyset$, and (iv) $B^+(r) \cap J = \emptyset$.

Intuitively, Items (i)–(iii) express that J is supported by P with respect to I, in the sense that the grounding of P contains some rule r whose body is satisfied by I (Item (i)) and which is able to derive some literal in J (Item (ii)), while all head atoms of r not contained in J are false under I. Moreover, Item (iv) ensures that this support is external as it is without reference to the set J itself.

Answer sets are now characterised thus:

Proposition 1 (*Lee 2005*)
Let P be a program and I an interpretation. Then, I is an answer set of P iff (i) $I \models P$ and (ii) every loop of P that is contained in I is externally supported by P with respect to I.

We actually make mainly use of the complementary relation of external support: Following Leone et al. (1997), we call J *unfounded by P with respect to I* iff J is not externally supported by P with respect to I.

3 The basic debugging approach

As discussed in the introduction, we view an error as a mismatch between the intended answer sets and the observed actual answer sets of some program. More specifically, our basic debugging question is why a given interpretation I is not answer set of some program P, and thus we deal with finding explanations for I not being an answer set of P. Proposition 1 allows us to distinguish between two kinds of such explanations: (i) instantiations of rules in P that are not satisfied by I and (ii) loops of P in I that are unfounded by P with respect to I. Although our basic debugging question allows for different, multi-faceted, answers, we see two major benefits of referring to this kind of categorisation: First, in view of Proposition 1, these kinds of explanations are always sufficient to explain why I is not an answer set of P, and second, this method provides *concrete witnesses*, e.g., unsatisfied rules or unfounded atoms, that can help to localise the reason for an error in a program or an interpretation in a rather intuitive way.

Before we introduce the details of our approach, we discuss its virtues compared to a method for debugging non-ground programs which can be obtained using the previous meta-programming technique for propositional programs due to Gebser et al. (2008).

[1] Note that loops have first been studied by Lin and Zhao (2004); different definitions of loops for non-ground programs were given by Chen et al. (2006) and Lee and Meng (2008). For our purposes, it suffices to refer to the basic definition for ground programs.

3.1 Prelude: A case for directly debugging non-ground programs

Explaining why some interpretation is not an answer set of some program based on the characterisation of Lee (2005) has been dealt with in previous work for debugging propositional disjunctive logic programs (Gebser et al. 2008). In principle, we could use this method for debugging non-ground programs as well by employing a preparatory grounding step. However, such an undertaking comes at a higher computational cost compared to our approach which respects the inherent complexity of the underlying tasks. We lay down our arguments in what follows.

To begin with, let us recall that Gebser et al. (2008) defined a fixed normal non-ground program γ and a mapping δ from disjunctive propositional programs and interpretations to sets of facts. Given a disjunctive program P without variables and some interpretation I, explanations why I is not an answer set of P can then be extracted from the answer sets of $\gamma \cup \delta(P, I)$. Such a problem encoding is *uniform* in the sense that γ does not depend on the problem instance determined by P and I.

To find reasons why some interpretation I is not an answer set of a non-ground program P, the above approach can be used by computing the answer sets of $\gamma \cup \delta(\mathit{grd}(P), I)$. However, in general, the size of $\mathit{grd}(P)$ is exponential in the size of P, and the computation of the answer sets of a ground program requires exponential time with respect to the size of the program, unless the polynomial hierarchy collapses. Hence, this outlined approach to compute explanations using a grounding step requires, all in all, *exponential space* and *double-exponential time* with respect to the size of P. But this is a mismatch to the inherent complexity of the overall task, as the following result shows:

Proposition 2
Given a program P and an interpretation I, deciding whether I is not an answer set of P is Π_2^P-complete.

This property is a consequence of the well-known fact that the complementary problem, i.e., checking whether some given interpretation is an answer set of some program, is Σ_2^P-complete (Eiter et al. 2004). Hence, checking whether an interpretation is not an answer set of some program can be computed in *polynomial space*.

Our approach takes this complexity property into account. We exploit the expressive power of disjunctive non-ground ASP by providing a uniform encoding that avoids both exponential space and double-exponential time requirements: Given a program P and an interpretation I, we define an encoding $\Gamma \cup \Delta(P, I)$, where Γ is a fixed disjunctive non-ground program, and $\Delta(P, I)$ is an efficient encoding of P and I by means of facts. Explanations why I is not an answer set of P are determined by the answer sets of $\Gamma \cup \Delta(P, I)$. Since Γ is fixed, the grounding of $\Gamma \cup \Delta(P, I)$ is bounded by a polynomial in the size of P and I. Thus, our approach requires only polynomial space and single-exponential time with respect to P and I.

Note that disjunctions can presumably not be avoided in Γ due to the Π_2^P-hardness of deciding whether an interpretation is not an answer set of some program. One may ask, however, whether Γ could be normal in case P is normal. We have to answer in the negative: answer-set checking for normal programs is complete for D^P, even if no negation is used or negation is only used in a stratified way (Eiter et al. 2004). (We recall that D^P is the class of problems that can be decided by a conjunction of an NP and an independent co-NP property.) Hence, Γ cannot be normal unless NP $=$ co-NP. However, one could use two independent normal meta-programs to encode our desired task.

A further benefit of debugging a program directly at the non-ground level is that we can immediately relate explanations for errors to first-order expressions in the considered program, e.g., to rules or literals with variables instead of their ground instantiations.

In what follows, we give details of Γ and Δ and describe their main properties.

3.2 Construction of the meta-program

3.2.1 Reification of input instances

For realising the encoding $\Delta(P, I)$ for program P and interpretation I, we rely on a reification $\varrho_{\mathit{prg}}(P)$ of P and a reification $\varrho_{\mathit{int}}(I)$ of I. The former is, in turn, constructed from reifications $\varrho_{\mathit{rule}}(r)$ of each individual rule $r \in P$. We introduce the mappings $\varrho_{\mathit{rule}}(\cdot)$, $\varrho_{\mathit{prg}}(\cdot)$, and $\varrho_{\mathit{int}}(\cdot)$ in the following.

To begin with, we need unique names for certain language elements. By an *extended predicate symbol* (EPS) we understand a predicate symbol, possibly preceded by the symbol for strong negation. Let $\cdot'$ be an injective *labelling function* from the set of program rules, literals, EPSs, and variables to a set of labels from the symbols in our language Ł. Note that we do not need labels for constant symbols since they will serve as unique names for themselves.

A single program rule is reified by means of facts according to the following definition.

Definition 1
Let r be a rule. Then,

$$
\begin{aligned}
\varrho_{\mathit{rule}}(r) = {} & \{\mathit{rule}(r')\} \cup \{\mathit{head}(r', a') \mid a \in H(r)\} \cup \\
& \{\mathit{posbody}(r', a') \mid a \in B^{+}(r)\} \cup \{\mathit{negbody}(r', a') \mid a \in B^{-}(r)\} \cup \\
& \{\mathit{pred}(a', L') \mid a = L(x_1, \ldots, x_n) \text{ is a literal in } r, L \text{ is an EPS}\} \cup \\
& \{\mathit{struct}(a', i, \mathit{var}, x_i') \mid a = L(x_1, \ldots, x_n) \text{ is a literal in } r, L \text{ is an EPS}, \\
& \quad i \in \{1, \ldots, n\}, \text{ and } x_i \text{ is a variable}\} \cup \\
& \{\mathit{struct}(a', i, \mathit{const}, x_i) \mid a = L(x_1, \ldots, x_n) \text{ is a literal in } r, L \text{ is an EPS}, \\
& \quad i \in \{1, \ldots, n\}, \text{ and } x_i \text{ is a constant symbol}\} \cup \\
& \{\mathit{var}(r', x') \mid x \text{ is a variable occurring in } r\}.
\end{aligned}
$$

The first fact states that label r' denotes a rule. The next three sets of facts associate labels of the literals in the head, the positive body, and the negative body to the respective parts of r. Then, each label of some literal in r is associated with a label for its EPS. The following two sets of facts encode the positions of variables and constants in the literals of the rule. Finally, the last set of facts states which variables occur in the rule r.

A program is encoded as follows:

Definition 2
Let P be a program. Then,

$$
\begin{aligned}
\varrho_{\mathit{prg}}(P) = {} & \textstyle\bigcup_{r \in P} \varrho_{\mathit{rule}}(r) \cup \{\mathit{dom}(c) \mid c \text{ is a constant symbol in } P\} \cup \\
& \{\mathit{arity}(L', n) \mid \text{a} = L(x_1, \ldots, x_n) \text{ is a literal in } P, L \text{ is an EPS}\}.
\end{aligned}
$$

$$\begin{aligned}
\gamma_{\mathit{unsat}}^{\mathit{guess}} = \{ & \mathit{guessRule}(R) \vee \mathit{nguessRule}(R) \leftarrow \mathit{rule}(R),\\
& \mathit{someRule} \leftarrow \mathit{guessRule}(R),\\
& \leftarrow \mathrm{not}\ \mathit{someRule}, \mathit{rule}(R),\\
& \leftarrow \mathit{guessRule}(R_1), \mathit{guessRule}(R_2), R_1 \neq R_2,\\
& \mathit{subst}(X, C) \vee \mathit{nsubst}(X, C) \leftarrow \mathit{guessRule}(R), \mathit{var}(R, X), \mathit{dom}(C),\\
& \mathit{assigned}(X) \leftarrow \mathit{subst}(X, C),\\
& \leftarrow \mathrm{not}\ \mathit{assigned}(X), \mathit{guessRule}(R), \mathit{var}(R, X),\\
& \leftarrow \mathit{subst}(X, C_1), \mathit{subst}(X, C_2), C_1 \neq C_2\}.\\
\gamma_{\mathit{unsat}}^{\mathit{check}} = \{ & \mathit{unsatisfied} \leftarrow \mathit{satBody}, \mathrm{not}\ \mathit{satHead},\\
& \mathit{satBody} \leftarrow \mathrm{not}\ \mathit{unsatPosbody}, \mathrm{not}\ \mathit{unsatNegbody},\\
& \mathit{satHead} \leftarrow \mathit{guessRule}(R), \mathit{head}(R, A), \mathit{true}(A),\\
& \mathit{unsatPosbody} \leftarrow \mathit{guessRule}(R), \mathit{posbody}(R, A), \mathit{false}(A),\\
& \mathit{unsatNegbody} \leftarrow \mathit{guessRule}(R), \mathit{negbody}(R, A), \mathit{true}(A)\}.
\end{aligned}$$

Fig. 1. Modules $\gamma_{\mathit{unsat}}^{\mathit{guess}}$ and $\gamma_{\mathit{unsat}}^{\mathit{check}}$.

The first union of facts stem from the reification of the single rules in the program. The remaining facts represent the Herbrand universe of the program and associate the EPSs occurring in the program with their arities.

The translation from an interpretation to a set of facts is formalised by the next definition.

Definition 3
Let I be an interpretation. Then,

$$\begin{aligned}
\varrho_{\mathit{int}}(I) = & \{\mathit{int}(a') \mid a \in I\} \cup \{\mathit{pred}(a', L') \mid a = L(x_1, \ldots, x_n) \text{ is a literal in } I,\\
& \quad L \text{ is an EPS}\} \cup\\
& \{\mathit{struct}(a', i, \mathit{const}, x_i) \mid a = L(x_1, \ldots, x_n) \text{ is a literal in } I, L \text{ is an EPS},\\
& \quad i \in \{1, \ldots, n\}, \text{ and } x_i \text{ is a constant symbol}\}.
\end{aligned}$$

The first two sets of facts associate the literals in I with their respective labels and EPSs. The last set of facts reifies the internal structure of the literals occurring in I.

Definition 4
Let P be a program and I an interpretation. Furthermore, let N be the the maximum of $|I|$ and the arities of all predicate symbols in P. Then, $\Delta(P, I) = \varrho_{\mathit{prg}}(P) \cup \varrho_{\mathit{int}}(I) \cup \{\mathit{natNumber}(n) \mid n \in \{0, \ldots, N\}\}$.

The literals $\mathit{natNumber}(\cdot)$ are necessary to add sufficiently many natural numbers to the Herbrand universe of $\Delta(P, I)$ to carry out correctly all computations in the subsequent program encodings. Note that the size of $\Delta(P, I)$ is always linear in the size of P and I.

3.2.2 The meta-program Γ

We proceed with the definition of the central meta-program Γ. The complete program consists of more than 160 rules. For space reasons, we only present the relevant parts and omit modules containing simple auxiliary definitions. The full encodings can be found at

`www.kr.tuwien.ac.at/research/projects/mmdasp/encoding.tar.gz`.

The meta-program Γ consists of the following modules: (i) γ_{unsat}, related to unsatisfied rules, (ii) γ_{loop}, related to loops, (iii) γ_{unfd}, for testing unfoundedness of loops, and (iv) γ_{cons}, integrating Parts (i)–(iii) for performing the overall test of whether a given interpretation I is not an answer set of a given program P.

We first introduce the program module γ_{unsat} to identify unsatisfied rules.

Definition 5
By γ_{unsat} we understand the program $\gamma^{\mathit{guess}}_{\mathit{unsat}} \cup \gamma^{\mathit{check}}_{\mathit{unsat}} \cup \gamma^{\mathit{aux}}_{\mathit{unsat}}$, where $\gamma^{\mathit{guess}}_{\mathit{unsat}}$ and $\gamma^{\mathit{check}}_{\mathit{unsat}}$ are given in Figure 1, and $\gamma^{\mathit{aux}}_{\mathit{unsat}}$ defines the auxiliary predicates $\mathit{true}(\cdot)$ and $\mathit{false}(\cdot)$.

Intuitively, for a program P and an interpretation I, $\gamma^{\mathit{guess}}_{\mathit{unsat}}$ guesses a rule $r \in P$, represented by predicate *guessRule*, and a substitution ϑ, represented by *subst*, and $\gamma^{\mathit{check}}_{\mathit{unsat}}$ defines that *unsatisfied* holds if $I \not\models r\vartheta$. Module $\gamma^{\mathit{aux}}_{\mathit{unsat}}$ (omitted for space reasons) defines the auxiliary predicates $\mathit{true}(\cdot)$ and $\mathit{false}(\cdot)$ such that $\mathit{true}(l')$ holds if $I \models l\vartheta$, for some literal l, and $\mathit{false}(l')$ holds if $I \not\models l\vartheta$.

Module γ_{unsat} has the following central property:

Theorem 1
Let P be a program and I an interpretation. Then, $I \not\models P$ iff some answer set of $\gamma_{\mathit{unsat}} \cup \Delta(P, I)$ contains *unsatisfied*. More specifically, for each rule $r \in P$ with $I \not\models r\vartheta$, for some substitution ϑ over the Herbrand universe of P, $\gamma_{\mathit{unsat}} \cup \Delta(P, I)$ has an answer set S such that (i) $\{\mathit{unsatisfied}, \mathit{guessRule}(r')\} \subseteq S$ and (ii) $\mathit{subst}(x', c) \in S$ iff $\vartheta(x) = c$.

We next define module γ_{loop} for identifying loops of a program.

Definition 6
By γ_{loop} we understand the program $\gamma^{\mathit{guess}}_{\mathit{loop}} \cup \gamma^{\mathit{check}}_{\mathit{loop}} \cup \gamma^{\mathit{aux}}_{\mathit{loop}}$, where $\gamma^{\mathit{guess}}_{\mathit{loop}}$ and $\gamma^{\mathit{check}}_{\mathit{loop}}$ are given in Figure 2, and $\gamma^{\mathit{aux}}_{\mathit{loop}}$ defines the auxiliary predicates $\mathit{loopSz}(\cdot)$ and $\mathit{differSeq}(\cdot, \cdot, \cdot)$.

Intuitively, for a program P and an interpretation I, $\gamma^{\mathit{guess}}_{\mathit{loop}}$ guesses a non-empty subset L of I, represented by $\mathit{inLoop}(\cdot)$, as a candidate for a loop, and $\gamma^{\mathit{check}}_{\mathit{loop}}$ defines that *isLoop* holds if L is a loop of P. More specifically, this check is realised as follows. Assume L contains n literals.

1. Guess a set $\mathcal{G}$ of n pairs (r, ϑ), where r is a rule from P and ϑ is a substitution over the Herbrand universe of P.
2. Check, for each $a, b \in L$, whether there is a path π in the positive dependency graph of the ground program consisting of rules $\{r\vartheta \mid (r, \vartheta) \in \mathcal{G}\}$ such that π starts with a and ends with b, and all literals in π are in L. A path π is represented by the binary predicate $\mathit{path}(\cdot, \cdot)$.

Module $\gamma^{\mathit{aux}}_{\mathit{loop}}$ (again omitted for space reasons) defines that (i) $\mathit{loopSz}(n)$ holds if $|L| = n$ and (ii) $\mathit{differSeq}(i, a', b')$ holds if $a\vartheta \neq b\vartheta$, where a, b are literals and ϑ is the substitution stemming from a pair in $\mathcal{G}$ that is associated with an index i by γ_{loop}.

Theorem 2
For any program P and any interpretation I, $L \subseteq I$ is a loop of P iff, for some answer set S of $\gamma_{\mathit{loop}} \cup \Delta(P, I)$, $\mathit{isLoop} \in S$ and $L = \{x \mid \mathit{inLoop}(x') \in S\}$.

$$
\begin{aligned}
\gamma_{\mathit{loop}}^{\mathit{guess}} = \{&\mathit{inLoop}(X) \vee \mathit{outLoop}(X) \leftarrow \mathit{int}(X),\\
&\mathit{someInLoop} \leftarrow \mathit{inLoop}(X),\\
&\leftarrow \mathrm{not}\ \mathit{someInLoop}, \mathit{int}(X)\}.\\
\gamma_{\mathit{loop}}^{\mathit{check}} = \{&\mathit{inRuleSet}(N,R) \vee \mathit{outRuleSet}(N,R) \leftarrow 1 \leq N, N \leq S, \mathit{loopSz}(S), \mathit{rule}(R),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{natNumber}(N),\\
&\mathit{someRule}(N) \leftarrow \mathit{inRuleSet}(N,R),\\
&\leftarrow \mathrm{not}\ \mathit{someRule}(N), 1 \leq N, N \leq S, \mathit{loopSz}(S), \mathit{rule}(R), \mathit{natNumber}(N),\\
&\leftarrow \mathit{inRuleSet}(N,R_1), \mathit{inRuleSet}(N,R_2), R_1 \neq R_2,\\
&\leftarrow \mathit{inRuleSet}(N_1,R_1), \mathit{inRuleSet}(N_2,R_2), N_1 \leq N_2, R_1 > R_2,\\
&\mathit{loopSubst}(N,X,C) \vee \mathit{nloopSubst}(N,X,C) \leftarrow \mathit{var}(R,X), \mathit{dom}(C),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{inRuleSet}(N,R),\\
&\mathit{loopAssigned}(N,X) \leftarrow \mathit{loopSubst}(N,X,C),\\
&\leftarrow \mathrm{not}\ \mathit{loopAssigned}(N,X), \mathit{inRuleSet}(N,R), \mathit{var}(R,X),\\
&\leftarrow \mathit{loopSubst}(N,X,C_1), \mathit{loopSubst}(N,X,C_2), C_1 \neq C_2,\\
&\mathit{isLoop} \leftarrow \mathrm{not}\ \mathit{unreachablePair}, inLoop(X),\\
&\mathit{unreachablePair} \leftarrow \mathit{inLoop}(X), \mathit{inLoop}(Y), \mathrm{not}\ \mathit{path}(X,Y),\\
&\mathit{path}(X,X) \leftarrow \mathit{inLoop}(X),\\
&\mathit{path}(X,Y) \leftarrow \mathit{inLoop}(X), \mathit{inLoop}(Y), \mathit{pred}(X,T_1), \mathit{pred}(Y,T_2), \mathit{loopSz}(S),\\
&\qquad\qquad 1 \leq N, N \leq S, \mathit{head}(R,H), \mathit{inRuleSet}(N,R), \mathit{posbody}(R,B),\\
&\qquad\qquad \mathit{pred}(H,T_1), \mathit{pred}(B,T_2), \mathrm{not}\ \mathit{differSeq}(N,X,H),\\
&\qquad\qquad \mathrm{not}\ \mathit{differSeq}(N,Y,B),\\
&\mathit{path}(X,Z) \leftarrow \mathit{inLoop}(X), \mathit{inLoop}(Z), \mathit{path}(X,Y), \mathit{path}(Y,Z)\}.
\end{aligned}
$$

Fig. 2. Modules $\gamma_{\mathit{loop}}^{\mathit{guess}}$ and $\gamma_{\mathit{loop}}^{\mathit{check}}$.

$$
\begin{aligned}
\gamma_{\mathit{unfd}}^{\mathit{guess}} = \{&\mathit{variable}(X) \leftarrow \mathit{var}(R,X),\\
&\mathit{suppSubst}(X,C) \vee \mathit{nsuppSubst}(X,C) \leftarrow \mathit{variable}(X), \mathit{dom}(C),\\
&\mathit{saturate} \leftarrow \mathit{suppSubst}(X,C_1), \mathit{suppSubst}(X,C_2), C_1 \neq C_2,\\
&\mathit{saturate} \leftarrow \mathit{unassigned}(X),\\
&\mathit{unass}(X,C) \leftarrow \mathit{first}(C), \mathit{nsuppSubst}(X,C),\\
&\mathit{unass}(X,C_2) \leftarrow \mathit{succ}(C_1,C_2), \mathit{unass}(X,C_1), \mathit{nsuppSubst}(X,C_2),\\
&\mathit{unassigned}(X) \leftarrow \mathit{last}(C), \mathit{unass}(X,C)\ \}.\\
\gamma_{\mathit{unfd}}^{\mathit{check}} = \{&\mathit{unfounded} \leftarrow \mathit{unsupp}(R), \mathit{lastR}(R),\\
&\mathit{unsupp}(R) \leftarrow \mathit{firstR}(R), \mathit{unsuppRule}(R),\\
&\mathit{unsupp}(R_2) \leftarrow \mathit{succR}(R_1,R_2), \mathit{unsupp}(R_1), \mathit{unsuppRule}(R_2),\\
&\mathit{saturate} \leftarrow \mathit{unfounded},\\
&\mathit{suppSubst}(X,C) \leftarrow \mathit{variable}(X), \mathit{dom}(C), \mathit{saturate},\\
&\mathit{nsuppSubst}(X,C) \leftarrow \mathit{variable}(X), \mathit{dom}(C), \mathit{saturate}\ \} \cup\\
&\{\mathit{unsuppRule}(R) \leftarrow c_i(R) \mid i \in \{1,\dots,5\}\}.
\end{aligned}
$$

Fig. 3. Modules $\gamma_{\mathit{unfd}}^{\mathit{guess}}$ and $\gamma_{\mathit{unfd}}^{\mathit{check}}$.

We proceed with module γ_{unfd} for checking whether some set J of ground literals is unfounded by P with respect to an interpretation I. We later combine this co-NP check with γ_{loop} to identify unfounded loops, i.e., we will integrate a loop guess with a co-NP check, thus reaching the very limits of disjunctive ASP by uniformly encoding a Σ_2^P property.

Definition 7
By γ_{unfd} we understand the program $\gamma_{\mathit{unfd}}^{\mathit{guess}} \cup \gamma_{\mathit{unfd}}^{\mathit{check}} \cup \gamma_{\mathit{unfd}}^{\mathit{aux}}$, where $\gamma_{\mathit{unfd}}^{\mathit{guess}}$ and $\gamma_{\mathit{unfd}}^{\mathit{check}}$ are given

in Figure 3, and $\gamma_{\mathit{unfd}}^{\mathit{aux}}$ defines the auxiliary predicates $\mathit{succ}(\cdot,\cdot)$, $\mathit{succR}(\cdot,\cdot)$, $\mathit{first}(\cdot)$, $\mathit{last}(\cdot)$, $\mathit{firstR}(\cdot)$, $\mathit{lastR}(\cdot)$, and $c_1(\cdot),\ldots,c_5(\cdot)$.

The intuition behind this definition is as follows. Consider a program P, some set J of ground literals, encoded via $\mathit{inLoop}(\cdot)$, and an interpretation I. Module $\gamma_{\mathit{unfd}}^{\mathit{guess}}$ non-deterministically guesses a binary relation $\mathit{suppSubst}(\cdot,\cdot)$ between the variables and the constant symbols in P. In case this relation is not a function, $\gamma_{\mathit{unfd}}^{\mathit{guess}}$ establishes *saturate*. Module $\gamma_{\mathit{unfd}}^{\mathit{check}}$, in turn, encodes whether, for each substitution ϑ and each rule $r \in P$, some of the conditions from the definition of J being externally supported by P is violated. In fact, *unfounded* is derived if some of these conditions is violated. Moreover, *saturate* holds if *unfounded* holds, and $\gamma_{\mathit{unfd}}^{\mathit{check}}$ saturates the relation defined by predicate $\mathit{suppSubst}(\cdot,\cdot)$ if *saturate* holds. Module $\gamma_{\mathit{unfd}}^{\mathit{aux}}$ (omitted for space reasons) defines $\mathit{succ}(\cdot,\cdot)$ and $\mathit{succR}(\cdot,\cdot)$, which express the immediate successor relation, based on $\preceq$, for the constant symbols and rules in P, respectively, as well as the predicates $\mathit{first}(\cdot)$, $\mathit{firstR}(\cdot)$, $\mathit{last}(\cdot)$, and $\mathit{lastR}(\cdot)$, which mark the first and the last elements in the order defined by $\mathit{succ}(\cdot,\cdot)$ and $\mathit{succR}(\cdot,\cdot)$, respectively. Moreover, the module $\gamma_{\mathit{unfd}}^{\mathit{aux}}$ defines predicates $c_1(\cdot),\ldots,c_5(\cdot)$, expressing failure of one of the conditions for J being externally supported by P with respect to I.

The rough idea behind the encoded saturation technique is to search, via $\gamma_{\mathit{unfd}}^{\mathit{guess}}$, for counterexample substitutions that witness that the set J of ground literals is *not* unfounded. For such a substitution, neither *saturate* nor *unfounded* can become true which implies that no answer set can contain *unfounded* due to the saturation of $\mathit{suppSubst}(\cdot,\cdot)$ and the minimality of answer sets.

Theorem 3
Consider a program P, an interpretation I, and a set J of ground literals. Then, J is unfounded by P with respect to I iff the unique answer set of $\gamma_{\mathit{unfd}} \cup \Delta(P,I) \cup \{\mathit{inLoop}(x') \mid x \in J\}$ contains the literal *unfounded.*

Given the above defined program modules, we arrive at the uniform encoding of the overall program Γ.

Definition 8
Let γ_{unsat}, γ_{loop}, and γ_{unfd} be the programs from Definitions 5, 6, and 7, respectively. Then, $\Gamma = \gamma_{\mathit{unsat}} \cup \gamma_{\mathit{loop}} \cup \gamma_{\mathit{unfd}} \cup \gamma_{\mathit{cons}}$, where

$$\begin{aligned}\gamma_{\mathit{cons}} = \{&\mathit{notAnswerSet} \leftarrow \mathit{unsatisfied}, \quad \mathit{notAnswerSet} \leftarrow \mathit{isLoop}, \mathit{unfounded},\\ &\leftarrow \mathrm{not}\ \mathit{notAnswerSet}\}.\end{aligned}$$

Module γ_{cons} encodes that each answer set of Γ witnesses either $I \not\models P$ or that some loop $L \subseteq I$ of P is unfounded by P with respect to I.

We finally obtain our main result, which follows essentially from the semantics of module γ_{cons} and Theorems 1, 2, and 3.

Theorem 4
Given a program P and an interpretation I, $\Pi = \Gamma \cup \Delta(P,I)$ satisfies the following properties:

(i) Π has no answer set iff I is an answer set of P.

(ii) I is not an answer set of P iff, for each answer set S of Π, $\{\mathit{unsatisfied}, \mathit{unfounded}\} \cap S \neq \emptyset$.
(iii) $I \not\models P$ iff $\mathit{unsatisfied} \in S$, for some answer set S of Π. Moreover, for each rule $r \in P$ with $I \not\models r\vartheta$, for some substitution ϑ over the Herbrand universe of P, there is some answer set S of Π such that (a) $\{\mathit{unsatisfied}, \mathit{guessRule}(r')\} \subseteq S$ and (b) $\mathit{subst}(x', c) \in S$ iff $\vartheta(x) = c$.
(iv) A loop $L \subseteq I$ is unfounded by P with respect to I iff some answer set S of Π contains both *isLoop* and *unfounded*, and $L = \{x \mid \mathit{inLoop}(x') \in S\}$.

4 Applying the debugging approach

In this section, we first describe a simple scenario with different debugging tasks and show how the meta-program defined in the previous section can be used to solve them. Afterwards, we discuss some pragmatic aspects relevant for realising a prospective user-friendly debugging system based on our approach.

4.1 A simple debugging scenario

We assume that students have to encode the assignments of papers to members of a program committee (PC) based on some bidding information in terms of ASP. We consider three cases, each of them illustrates a different kind of debugging problem. In the first case, an answer set is expected but the program is inconsistent. In the second case, multiple answer sets are expected but the program yields only one answer set. In the third case, it is expected that a program is inconsistent, but it actually yields some answer set. We illustrate that, in all cases, our approach gives valuable hints how to debug the program in an iterative way.

Assume that $\mathit{pc}(X)$ means that X is a member of the PC, $\mathit{paper}(X)$ means that X is a paper, and $\mathit{bids}(X, Y, Z)$ means that PC member X bids on paper Y with value Z, where Z is a natural number ranging from 0 to 3 expressing a degree of preference for that paper.

To start with, Lucy wants to express that the default bid for a paper is 1. That is, if a PC member does not bid on a paper, then it is assumed that the PC member bids 1 on that paper per default. Lucy's first attempt looks as follows:

$$\begin{aligned} L_1 = \{&\mathit{pc}(m_1), \mathit{pc}(m_2), \mathit{paper}(p_1), \mathit{bid}(m_1, p_1, 2), \mathit{bid}(m_2, p_1, 3),\\ &\mathit{some\_bid}(M, P) \leftarrow \mathit{bid}(M, P, X),\\ &\mathit{bid}(M, P, 1) \leftarrow \mathrm{not}\ \mathit{some\_bid}(M, P), \mathit{pc}(M), \mathit{paper}(P)\}. \end{aligned}$$

Lucy's intention is that $\mathit{some\_bid}(M, P)$ is true if PC member M bids on paper P, and $\mathit{bid}(M, P, 1)$ is true if there is no evidence that PC member M has bid on that paper. Indeed, the unique answer set of L_1 is

$$\begin{aligned} S_1 = \{&\mathit{pc}(m_1), \mathit{pc}(m_2), \mathit{paper}(p_1), \mathit{bid}(m_1, p_1, 2), \mathit{bid}(m_2, p_1, 3),\\ &\mathit{some\_bid}(m_1, p_1), \mathit{some\_bid}(m_2, p_1)\}. \end{aligned}$$

The answer set S_1 is indeed as expected: We have that each PC member bids on some paper in L_1 and the last rule is inactive. Lucy's next step is to delete the fact $\mathit{bid}(m_2, p_1, 3)$ from L_1—let us denote the resulting program by L_2. Lucy expects that the answer set of L_2 contains $\mathit{bid}(m_2, p_1, 1)$. However, it turns out that L_2 yields no answer set at all!

To find out what went wrong, Lucy defines her expected answer set as

$$E_1 = (S_1 \cup \{\mathit{bid}(m_2, p_1, 1)\}) \setminus \{\mathit{some\_bid}(m_2, p_1), \mathit{bid}(m_2, p_1, 3)\}$$

and inspects the answer sets of $\Gamma \cup \Delta(L_2, E_1)$. It turns out that one answer set contains the facts *unsatisfied* and $\mathit{guessRule}(r_1')$, where r_1' is the label for the rule

$$r_1 = \mathit{some\_bid}(M, P) \leftarrow \mathit{bid}(M, P, X).$$

Hence, r_1 is not satisfied by E_1: $\mathit{bid}(m_2, p_1, 1)$ is in E_1 and thus satisfies the body of r_1, but the head of r_1 is not satisfied since E_1 does not contain $\mathit{some\_bid}(m_2, p_1)$.

Now that Lucy sees that L_2's answer set has to contain $\mathit{some\_bid}(m_2, p_1)$, she defines E_2 as E_1 plus the fact $\mathit{some\_bid}(m_2, p_1)$. The answer sets of $\Gamma \cup \Delta(L_2, E_2)$ reveal that E_2 is not an answer set of L_2 because the singleton loop $\mathit{bid}(m_2, p_1, 1)$ is contained in E_2 but it is unfounded by L_2 with respect to E_2. The reason is clear: the only rule that could support $\mathit{bid}(m_2, p_1, 1)$ is

$$r_2 = \mathit{bid}(M, P, 1) \leftarrow \mathrm{not}\ \mathit{some\_bid}(M, P), \mathit{pc}(M), \mathit{paper}(P).$$

However, r_2 is blocked since E_2 contains $\mathit{some\_bid}(m_2, p_1)$.

Lucy concludes that, to make r_2 work as expected, $\mathit{some\_bid}(m_2, p_1)$ must not be contained in the answer set. To achieve this, Lucy changes r_1, the only rule with predicate *some_bid* in the head, into

$$\mathit{some\_bid}(M, P) \leftarrow \mathit{bid}(M, P, X), X \neq 1.$$

The resulting program works as expected and contains $\mathit{bid}(m_2, p_1, 1)$ in its answer set.

The next student who is faced with a mystery is Linus. He tried to formalise that each paper is non-deterministically assigned to at least one member of the PC. His program looks as follows:

$$\begin{aligned} P_1 = \{ & \mathit{pc}(m_1), \mathit{pc}(m_2), \mathit{paper}(p_1), \mathit{paper}(p_2), \mathit{bid}(m_1, p_1, 2), \\ & \mathit{bid}(m_1, p_2, 3), \mathit{bid}(m_2, p_1, 1), \mathit{bid}(m_2, p_2, 1), \\ & \mathit{assigned}(P, M) \vee \neg \mathit{assigned}(P, M) \leftarrow \mathit{paper}(P), \mathit{pc}(M), \\ & \leftarrow \mathit{paper}(P), \mathit{pc}(M), \mathrm{not}\ \mathit{assigned}(P, M)\}. \end{aligned}$$

Linus expects that the disjunctive rule realises the non-deterministic guess, and then the constraint prunes away all answer set candidates where a paper is not assigned to some PC member. Now, poor Linus is desperate since the non-deterministic guess seems not to work correctly; the only answer set of P_1 is

$$\begin{aligned} S_3 = \{ & \mathit{paper}(p_1), \mathit{paper}(p_2), \mathit{pc}(m_1), \mathit{pc}(m_2), \mathit{bid}(m_1, p_1, 2), \mathit{bid}(m_1, p_2, 3), \\ & \mathit{bid}(m_2, p_1, 1), \mathit{bid}(m_2, p_2, 1), \mathit{assigned}(p_1, m_1), \mathit{assigned}(p_1, m_2), \\ & \mathit{assigned}(p_2, m_1), \mathit{assigned}(p_2, m_2)\}, \end{aligned}$$

although Linus expected one answer set for each possible assignment. In particular, Linus expected

$$E_3 = (S_3 \cup \{\neg \mathit{assigned}(p_1, m_2)\}) \setminus \{\mathit{assigned}(p_1, m_2)\}$$

to be an answer set as well. Hence, Linus inspects the answer sets of $\Gamma \cup \Delta(P_1, E_3)$ and learns that the constraint in P_1 is not satisfied by E_3. In particular, it is the substitution that maps the variable P to p_1 and M to m_2 that is responsible for the unsatisfied constraint, which can be seen from the *subst*$(\cdot)$ atoms in each answer set that contains *unsatisfied*.

Having this information, Linus observes that the constraint in its current form is unsatisfied if some paper is not assigned to *each* PC member. However, he intended it to be unsatisfied only when a paper is assigned to *no* PC member. Hence, he replaces the constraint by the two rules

$$\leftarrow \mathit{paper}(P), \mathit{pc}(M), \mathrm{not}\ \mathit{at\_least\_one}(P) \quad \text{and} \quad \mathit{at\_least\_one}(P) \leftarrow \mathit{assigned}(P, M).$$

The resulting program yields the nine expected answer sets.

Meanwhile, Peppermint Patty encounters a strange problem. Her task was to write a program that expresses the following issue: If a PC member M bids 0 on some paper P, then this means that there is a conflict of interest with respect to M and P. In any case, there is a conflict of interest if M (co-)authored P. A PC member can only be assigned to some paper if there is no conflict of interest with respect to that PC member and that paper. This is Peppermint Patty's solution:

$$\begin{aligned} Q_1 = \{&\mathit{pc}(m_1), \mathit{paper}(p_1), \mathit{bid}(m_1, p_1, 2), \mathit{assigned}(p_1, m_1), \mathit{author}(p_1, m_1),\\ &\mathit{conflict\_of\_interest}(M, P) \leftarrow \mathit{bid}(M, P, 0),\\ &\mathit{conflict\_of\_interest}(M, P) \leftarrow \mathit{pc}(M), \mathit{paper}(P), \mathit{author}(M, P),\\ &\mathit{bid}(M, P, 0) \leftarrow \mathit{pc}(M), \mathit{paper}(P), \mathit{conflict\_of\_interest}(M, P),\\ &\leftarrow \mathit{assigned}(P, M), \mathit{bid}(M, P, 0)\}. \end{aligned}$$

The facts in Q_1 should model a scenario where a PC member authored a paper and is assigned to that paper. According to the specification from above, this should not be allowed. Since Patty is convinced that her encoding is correct, she expects that Q_1 has no answer sets. But Q_1 has the unique answer set

$$S_4 = \{\mathit{assigned}(p_1, m_1), \mathit{pc}(m_1), \mathit{paper}(p_1), \mathit{author}(p_1, m_1), \mathit{bid}(m_1, p_1, 2)\}.$$

What Peppermint Patty finds puzzling is that S_4 does not contain any atoms signalling a conflict of interest. Hence, she decides to analyse why

$$E_4 = S_4 \cup \{\mathit{conflict\_of\_interest}(m_1, p_1), \mathit{bid}(m_1, p_1, 0)\}$$

is not an answer set of Q_1. If Q_1 was correct, then the only reason why E_4 is not an answer set of Q_1 would be that the (only) constraint in Q_1 is unsatisfied.

As expected, some answer sets of $\Gamma \cup \Delta(Q_1, E_4)$ contain *unsatisfied* and *guessRule*(r'), where r' is the label of the constraint in Q_1. However, some answer sets contain the atom *unfounded* as well—a surprising observation. Patty learns, by inspecting the *inLoop*$(\cdot)$ atoms in the respective answer set, that E_4 contains the loop

$$\{\mathit{conflict\_of\_interest}(m_1, p_1), \mathit{bid}(m_1, p_1, 0)\}$$

which is unfounded by Q_1 with respect to E_4: $\mathit{bid}(m_1, p_1, 0)$ seems to be justified only by the literal $\mathit{conflict\_of\_interest}(m_1, p_1)$ and vice versa. This should not be the case since Q_1 contains the rule

$$\mathit{conflict\_of\_interest}(M, P) \leftarrow \mathit{pc}(M), \mathit{paper}(P), \mathit{author}(M, P)$$

that should support $\mathit{conflict\_of\_interest}(m_1, p_1)$ because all the facts $\mathit{pc}(m_1)$, $\mathit{paper}(p_1)$, and *author*(m_1, p_1) should be contained in Q_1. Now, the error is obvious: Q_4 does not contain the fact *author*(m_1, p_1) but *author*(p_1, m_1)—the order of the arguments was wrong. After Peppermint Patty fixed that bug, her program is correct.

4.2 Some pragmatic issues and future prospects

For a debugging system of practical value, certain pragmatic aspects have to be taken into account which we briefly sketch in what follows. To start with, our encodings can be seen as a "golden design"—tailored towards clarity and readability—which leaves room for optimisations. Related to this issue, solver features like limiting the number of computed answer sets or query answering are needed to avoid unnecessary computation and to limit the amount of information presented to the user.

Our debugging approach requires information about the intended semantics in form of the interpretation representing a desired answer set. Typically, answer sets of programs encoding real-world problems tend to be large which makes it quite cumbersome to manually create interpretations from scratch. It is therefore vital to have convenient means for obtaining an intended answer set in the first place. For this purpose, we envisage a tool-box for managing interpretations that allows for their manipulation and storage. In such a setting, answer sets of previous versions of the debugged program could be a valuable source of interpretations which are then tailored towards an intended answer set of the current version. In addition to manual adaptations, partial evaluation of the program could significantly accelerate the creation of interpretations. We plan to further investigate these issues and aim at incorporating our debugging technique, along with an interpretation management system as outlined, in an integrated development environment (IDE). Here, an important issue is to achieve a suitable user interface for highlighting the identified unsatisfied rules and unfounded loops in the source code and for visualising the involved variable substitutions.

5 Related work

Besides the debugging approach by Gebser et al. (2008), as already discussed earlier, other related approaches on debugging include the work of Pontelli et al. (2009) on *justifications* for non-ground answer-set programs that can be seen as a complementary approach to ours. Their goal is to explain the truth values of literals with respect to a given actual answer set of a program. Explanations are provided in terms of *justifications* which are labelled graphs whose nodes are truth assignments of possibly default-negated ground atoms. The edges represent positive and negative support relations between these truth assignments such that every path ends in an assignment which is either assumed or known to hold. The authors have also introduced justifications for partial answer sets that emerge during the solving process (online justifications), being represented by three-valued interpretations.

The question why atoms are contained or are not contained in an answer set has also been raised by Brain and De Vos (2005) who provide algorithms for recursively computing explanations in terms of satisfied supporting rules. Note that these problems can in principle also be handled by our approach, as illustrated in Section 4.1. Indeed, consider some program P with answer set I and suppose we want to know why a certain set L of literals is contained in I. Using our approach, explanations why $I \setminus L$ is not an answer set of P will reveal rules which are unsatisfied under $I \setminus L$ but which support literals in L under I. Likewise, we can answer the question why expected atoms are missing in an answer set.

Syrjänen (2006) aims at finding explanations why some propositional program has no answer sets. His approach is based on finding minimal sets of constraints such that their removal yields consistency. Hereby, it is assumed that a program does not involve circular dependencies between literals through an odd number of negations which might also cause inconsistency. Finding reasons for program inconsistency can be handled by our approach when an intended answer set is known, as illustrated by program L_2 in Section 4.1. Otherwise, an interpretation can be chosen from the answer sets resulting from temporarily removing all constraints from the considered program (providing this yields consistency).

Brain et al. (2007) rewrite a program using some additional control atoms, called *tags*, that allow, e.g., for switching individual rules on or off and for analysing the resulting answer sets. Debugging requests in this approach can be posed by adding further rules that can employ tags as well. One such extension allows also for detecting atoms in unfounded loops. However, as opposed to our current approach, the individual loops themselves are not identified.

Caballero et al. (2008) developed a declarative debugging approach for datalog using a classification of error explanations similar to the one by Gebser et al. (2008) and our current work. Their approach is tailored towards query answering and, in contrast to our approach, the language is restricted to stratified datalog. However, Caballero et al. provide an implementation that is based on computing a graph that reflects the execution of a query.

Wittocx et al. (2009) show how a calculus can be used for debugging first-order theories with inductive definitions in the context of model expansion problems, i.e., problems of finding models of a given theory that expand some given interpretation. The idea is to trace the proof of inconsistency of such an unsatisfiable model expansion problem. The authors provide a system that allows for interactively exploring the proof tree.

Besides the mentioned approaches which rely on the semantical behaviour of programs, Mikitiuk et al. (2007) use a translation from logic-program rules to natural language in order to detect program errors more easily. This seems to be a potentially useful feature for an IDE as well, especially for novice and non-expert ASP programmers.

6 Conclusion

Our approach for declaratively debugging non-ground answer-set programs aims at providing intuitive explanations why a given interpretation fails to be an answer set of the program in development. To answer this question, we localise, on the one hand, unsatisfied rules and, on the other hand, loops of the program that are unfounded with respect to the given interpretation. As underlying technique, we use a sophisticated meta-programming method that reflects the complexity of the considered debugging question which resides on the second level of the polynomial hierarchy.

Typical errors in ASP may have quite different reasons and many of them could be avoided rather easily in the first place, e.g., by a compulsory declaration of predicates (Brain and De Vos 2005), forbidding uneven loops through negation (Syrjänen 2006), introducing type checks, or defining program interfaces. We plan to realise these kinds of simple prophylactic techniques for our future IDE for ASP that will incorporate our current debugging approach. In this context, courses on logic programming at our institute shall provide a permanent testbed for our techniques. Moreover, as part of an ongoing research project on methods and methodologies for developing answer-set programs (Oetsch et al. 2010), we want to put research efforts into methodologies that avoid or minimise debugging needs right from the start. As a next direct step regarding our efforts towards debugging, we plan to extend our approach to language features like aggregates, function symbols, and optimisation techniques such as minimise-statements or weak constraints.

References

BRAIN, M. AND DE VOS, M. 2005. Debugging logic programs under the answer-set semantics. In *Proceedings of the 3rd Workshop on Answer Set Programming: Advances in Theory and Implementation* (*ASP'05*)*, Bath, UK, July 27-29, 2005*. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org, Aachen, Germany.

BRAIN, M., GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2007. Debugging ASP programs by means of ASP. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'07*)*, Tempe, AZ, USA, May 15-17, 2007*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, Berlin-Heidelberg, Germany, 31–43.

CABALLERO, R., GARCÍA-RUIZ, Y., AND SÁENZ-PÉREZ, F. 2008. A theoretical framework for the declarative debugging of datalog programs. In *Revised Selected Papers of the 3rd International Workshop on Semantics in Data and Knowledge Bases* (*SDKB'08*)*, Nantes, France, March 29, 2008*, K.-D. Schewe and B. Thalheim, Eds. Lecture Notes in Computer Science, vol. 4925. Springer, Berlin-Heidelberg, Germany, 143–159.

CHEN, Y., LIN, F., WANG, Y., AND ZHANG, M. 2006. First-order loop formulas for normal logic programs. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning* (*KR'06*)*, Lake District, UK, June 2-5, 2006*, P. Doherty, J. Mylopoulos, and C. A. Welty, Eds. AAAI Press, Menlo Park, CA, USA, 298–307.

DENECKER, M., VENNEKENS, J., BOND, S., GEBSER, M., AND TRUSZCZYNSKI, M. 2009. The second answer set programming competition. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR'09*)*, Potsdam, Germany, September 14-18, 2009*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer, Berlin-Heidelberg, Germany, 637–654.

EITER, T., FABER, W., FINK, M., PFEIFER, G., AND WOLTRAN, S. 2004. Complexity of model checking and bounded predicate arities for non-ground answer set programming. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning* (*KR'04*)*, Whistler, Canada, June 2-5, 2004*, D. Dubois, C. A. Welty, and M.-A. Williams, Eds. AAAI Press, Menlo Park, CA, USA, 377–387.

EITER, T., GOTTLOB, G., AND MANNILA, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems 22,* 3 (Sept.), 364–418.

GEBSER, M., PÜHRER, J., SCHAUB, T., AND TOMPITS, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (*AAAI'08*)*, Chicago, IL, USA, July 13-17, 2008*, D. Fox and C. P. Gomes, Eds. AAAI Press, Menlo Park, CA, USA, 448–453.

GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2009. spock: A debugging support tool for logic programs under the answer-set semantics. In *Revised Selected Papers of the 17th International Conference on Applications of Declarative Programming and Knowledge Management and the 21st Workshop on Logic Programming* (*INAP'07/WLP'07*)*, Würzburg, Ger-*

many, October 4-6, 2007, D. Seipel, M. Hanus, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 5437. Springer, Berlin-Heidelberg, Germany, 247–252.

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 *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (*IJCAI'05*)*, Edinburgh, Scotland, UK, July 30-August 5, 2005*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, Denver, CO, USA, 503–508.

LEE, J. AND MENG, Y. 2008. On loop formulas with variables. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning* (*KR'08*)*, Sydney, Australia, September 16-19, 2008*, G. Brewka and J. Lang, Eds. AAAI Press, Menlo Park, CA, USA, 444–453.

LEONE, N., RULLO, P., AND SCARCELLO, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation 135,* 2, 69–112.

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing answer sets of a logic program with SAT solvers. *Artificial Intelligence 157,* 1–2, 115–137.

MIKITIUK, A., MOSELEY, E., AND TRUSZCZYNSKI, M. 2007. Towards debugging of answer-set programs in the language PSpb. In *Proceedings of the 2007 International Conference on Artificial Intelligence* (*ICAI'07*)*, Volume II, Las Vegas, NV, USA, June 25-28, 2007*, H. R. Arabnia, M. Q. Yang, and J. Y. Yang, Eds. CSREA Press, Bogart, GA, USA, 635–640.

OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2010. Methods and methodologies for developing answer-set programs—Project description. In *Technical Communications of the 26th International Conference on Logic Programming* (*ICLP'10*), M. Hermenegildo and T. Schaub, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 7. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

PONTELLI, E., SON, T. C., AND EL-KHATIB, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming 9,* 1, 1–56.

SYRJÄNEN, T. 2006. Debugging inconsistent answer set programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning* (*NMR'06*)*, Lake District, UK, May 30-June 1, 2006*, J. Dix and A. Hunter, Eds. Institut für Informatik, Technische Universität Clausthal, Technical Report, Clausthal, Germany, 77–83.

WITTOCX, J., VLAEMINCK, H., AND DENECKER, M. 2009. Debugging for model expansion. In *Proceedings of the 25th International Conference on Logic Programming* (*ICLP'09*)*, Pasadena, CA, USA, July 14-17, 2009*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, Berlin-Heidelberg, Germany, 296–311.