Let's Break the Rules: Interactive Procedural-Style Debugging of 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
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. We introduce an interactive procedural-style debugging approach for answer-set programs that avoids the negative aspects of non-declarative debugging of ASP. It is based on an intuitive computation model that allows a user to follow his or her intuition by stepwise determining which applicable rules are considered to be supporting rules. Moreover, we define the notion of a *breakpoint* for answer-set programs that allows to start stepwise debugging of a program Π from an answer set of a subprogram of Π.

1 Introduction

Research on debugging in answer-set programming (ASP) has so far mainly focused on declarative methods [1–7].[1] This has two main reasons. First, it is the aim to have debugging systems that are independent of any particular solving tool or solving method. Second, it is commonly argued that procedural-style debugging would ruin the declarative nature of answer-set programs under which a program is viewed as a set of logic rules describing a problem, where no rule takes precedence and each of the rules is equally important as the others. The main concern in this respect is that, following a procedural debugging approach, the answer-set programs under consideration could be seen as parameters for the solving algorithm. This, in turn, would inveigle the user to a programming style adjusted to the algorithm rather than focussing on clarity of representation. However, on the other side, the goal of a fully declarative view on answer-set programs seems to be idealistic. In fact, even the well-known guess-and-check paradigm imposes a procedural flavour on programs, in the sense that solution candidates are *first* generated and *afterwards* filtered. We also experience that humans who construct an answer set of a program manually will typically start from the facts and proceed in a bottom-up manner, rather than proceeding in a random order. Generally, since writing programs is necessarily an incremental process, already the order in which the rules are written often follows a conceptual structuring the programmer has in mind.

In this paper, we aim for a debugging method that lets a programmer follow his or her intuitions regarding the generation of an answer set and allows to quickly examine

[⋆] This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

[1] The notable exception of this pattern being the work of Wittocx et al. [8] for debugging first-order theories with inductive definitions based on tracing inconsistency proofs.

critical parts of the program. Hence, in a sense, we break with the goal of pure declarativity in debugging while keeping the concerns of the argumentation above addressed.

In procedural programming, stepping through a program is a very common approach. While simple tracers only print information about the internal state of the execution steps for offline analysis, more advanced debuggers allow for modifying variables and the control flow on the fly. In ASP, determining the direction of the search appears to be a natural way of handling the inherent non-determinism of answer-set programs. We introduce a non-deterministic computation model that is in our view easy to understand even for novice programmers, yet able to grasp the answer-set semantics. Like many approaches for bottom-up generation of answer sets [9–12], our computation model is instantiated in the form of a fixed-point operator. It is based on stepwise adding supporting rules of the desired answer set. Thereby, we follow the basic principle that once a rule is applied it has to stay applicable until the end of the computation. This way, every literal and every loop in the evolving answer set remains (externally) supported and answer sets may evolve in a monotonic way. In a debugging session, the programmer serves as an oracle, choosing a rule that he or she considers as a supporting rule. In case that a rule not chosen yet is inconsistent with the last state of an ongoing computation, or if there are only rules applicable that would make a previously chosen rule non-supporting, the computation fails. To reveal the nature of the corresponding "bug", i.e., the mismatch between the actual and the intended semantics, we propose several options for querying the current state of the computation such that the programmer gains insight into why the program does not behave as intended. As a consequence of the design of a computation, at each step the growing interpretation is an answer set of the program part considered so far. This gives rise to a counterpart of *breakpoints* for answer-set programs. Indeed, stepwise debugging of a program Π can be started from an arbitrary answer set I of a subprogram Π' of the grounding of Π, where the supporting rules of Π' with respect to I are also considered to be supporting in the initial computation. This allows for a quick access to those aspects of a program the user is interested in. For example, when a part of a program that was written earlier is already considered correct, the debugging process can be started from there and built on the results of this part. It is also possible to add rules during debugging and immediately see how they influence the generation of the targeted answer set.

Our method has a procedural flavour, but it is independent of a particular solving tool and hence does not interfere with any solver specifics such as internal data structures and performance hacks. Moreover, it does not spoil the declarative nature of ASP as the order in which the rules are considered is not fixed, controlled by the programmer, and may vary for the same program and targeted interpretation in different debugging sessions. In this sense, we see the approach as a reconciliation of a procedural with a declarative view on answer-set programs.

2 Preliminaries

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

$$l_0 \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n, \tag{1}$$

where $n \geq m \geq 0$, "not" denotes *default negation*, and all l_i are literals over a function-free first-order language $\mathcal{L}$. A literal is an atom possibly preceded by the *strong negation* symbol $\neg$. By $\bar{l}$ we denote $\neg a$ for $l = a$ and a for $l = \neg a$. In the sequel, we assume that $\mathcal{L}$ will be implicitly defined by the considered programs. For a rule r of form (1), $\mathrm{B}(r) = \{l_1, \ldots, l_m, \mathrm{not}\ l_{m+1}, \ldots, \mathrm{not}\ l_n\}$ is the *body* of r, $\mathrm{B}^+(r) = \{l_1, \ldots, l_m\}$ is the *positive body* of r, $\mathrm{B}^-(r) = \{l_{m+1}, \ldots, l_n\}$ is the *negative body* of r, and $\mathrm{H}(r) = l_0$ is the *head* of r. Moreover, r is a *fact* if $n = 0$. For facts, we usually omit the symbol "$\leftarrow$", and we identify sets of literals with sets of facts. A literal, rule, or program is *ground* if it contains no variables. The *grounding* of a program $\mathit{\Pi}$ relative to its Herbrand universe, denoted by $gr(\mathit{\Pi})$, is defined as usual.

An *interpretation* I (over some language $\mathcal{L}$) is a finite and consistent set of ground literals (over $\mathcal{L}$). 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 possibly default negated literals, or a program α is defined in the usual manner. We denote the set of supporting rules of a ground program $\mathit{\Pi}$ with respect to an interpretation I as $\mathit{supp}^I(\mathit{\Pi}) = \{r \in \mathit{\Pi} \mid I \models \mathrm{B}(r)\}$. A rule r such that $I \models \mathrm{B}(r)$ is called *applicable* under I. Following Faber, Leone, and Pfeifer [13], we define an *answer set* of a program $\mathit{\Pi}$ as an interpretation I that is a minimal model of $\mathit{supp}^I(gr(\mathit{\Pi}))$. Note that for the programs we consider, this definition is equivalent to the traditional one in terms of the Gelfond-Lifschitz reduct [14].

3 Framework

Next, we introduce the basic computation model that underlies our debugging approach. We are aiming for a scenario in which the programmer has strong control over the direction of the construction of an answer set. The general idea is to first take a part of a program and an answer set of this part. Then, step by step, rules are added such that at every step literals may be added to the interpretation such that it remains to be an answer set of the evolving program part. Hereby, the user only adds rules he or she thinks are applicable in the final answer set. The partial interpretation grows monotonically until it is eventually guaranteed to be an answer set of the overall program, otherwise the programmer is informed why and at which step the computation went wrong.

A main aspect of our method is that the model of computation hides the non-monotonicity of ASP away, evolving towards an intended answer set. As debugging is typically started after a missing expected answer set or a superfluous unintended answer set is detected, one can direct the computation towards this answer set without any backtracking. The individual steps of a computation consists of the growing interpretation and a set of ground rules which are considered as *already applied*.

Definition 1. *A* state *of a program $\mathit{\Pi}$ is a pair $S = \langle I, R\rangle$, where I is an interpretation and $R \subseteq gr(\mathit{\Pi})$ is a set of ground rules.*

Given state $S = \langle I, R\rangle$ of some program, we also write I_S for I and R_S for R. Both components of a state, the interpretation as well as the set of rules, grow monotonically during a computation.

Definition 2. *For two states S and S' of a program $\mathit{\Pi}$, S' is a* successor *of S in $\mathit{\Pi}$, symbolically $S' \succ_{\mathit{\Pi}} S$, if $|R_{S'} \setminus R_S| = 1$ and $I_S \subseteq I_{S'}$.*

A computation *for a program $\mathit{\Pi}$ is a finite sequence $\mathrm{C} = S_0, \ldots, S_n$ of states, for $n \geq 0$, such that for all $0 \leq i < n$ it holds that $S_{i+1} \succ_{\mathit{\Pi}} S_i$.*

Given a computation $\mathrm{C} = S_0, \ldots, S_n$ for a program $\mathit{\Pi}$, we say that S_0 is the *initial state* of $\mathit{\Pi}$ and I_{S_n} is the *result*, $\mathit{res}(\mathrm{C})$, of C.

We next define the semantic properties of *validity* and *stability* for our syntactic notions of states and computations, respectively. Moreover, we introduce the notions of *failure* and *completeness* of a computation.

Definition 3. *A state S is (i)* valid *if $I_S \models \mathrm{B}(r)$ for all $r \in R_S$ and (ii)* stable *if $I_S \in AS(R_S)$.*

A computation $\mathrm{C} = S_0, \ldots, S_n$ is valid *or* stable *if every state S_i, for $0 \leq i \leq n$, is valid or stable, respectively.*

A computation $\mathrm{C} = S_0, \ldots, S_n$ for $\mathit{\Pi}$ has failed at step *i ($0 \leq i \leq n$) if there is no answer set I of $\mathit{\Pi}$ such that $I_{S_i} \subseteq I$ and $R_{S_i} \subseteq \{r \in gr(P) \mid I \models \mathrm{B}(r)\}$. A computation $\mathrm{C} = S_0, \ldots, S_n$ is* complete *if for every rule $r \in gr(\mathit{\Pi})$ such that $I_{S_n} \models \mathrm{B}(r)$ we have $r \in R_{S_n}$.*

In what follows, we instantiate our computation model by means of the non-deterministic operator $\Upsilon_{\mathit{\Pi}}(\cdot)$, transforming a state S for $\mathit{\Pi}$ into another state S' for $\mathit{\Pi}$ such that $S' \succ_{\mathit{\Pi}} S$.

Definition 4. *Let $\mathit{\Pi}$ be a program and S a state of $\mathit{\Pi}$. Then, $\Upsilon_{\mathit{\Pi}}(S) = \langle I, R \rangle$, where $I = I_S \cup \{\mathrm{H}(r)\}$ and $R = R_S \cup \{r\}$ for some rule $r \in gr(\mathit{\Pi}) \setminus R_S$ with $I_S \models \mathrm{B}(r)$ and $\mathrm{H}(r) \notin \bigcup_{l \in I} \bar{l} \cup \bigcup_{r' \in R} \mathrm{B}^-(r')$.*

Theorem 1. *Let $\mathit{\Pi}$ be a program and I an interpretation for $\mathit{\Pi}$. If there is a fixed-point iteration $S_i = \Upsilon_{\mathit{\Pi}}(S_{i-1})$ with $i > 0$ where $S_0 = \langle \emptyset, \emptyset \rangle$ that reaches some fixed point $S_n = \langle I, R \rangle$, then I is an answer set of $\mathit{\Pi}$.*

Theorem 2. *Let $\mathit{\Pi}$ be a program and I an answer set of $\mathit{\Pi}$. Then, there is a fixed-point iteration $S_i = \Upsilon_{\mathit{\Pi}}(S_{i-1})$ with $i > 0$ where $S_0 = \langle \emptyset, \emptyset \rangle$ that reaches the fixed point $S_n = \langle I, \mathit{supp}^I(gr(\mathit{\Pi})) \rangle$.*

Proof (Sketch). By showing that choosing rules from $\mathit{supp}^I(gr(\mathit{\Pi}))$ is possible until S_n is reached. $\square$

It is easy to see that the sequence $S_0, \ldots, S_n$ of Theorem 2 is a valid, stable, and complete computation for $\mathit{\Pi}$.

Corollary 1. *Let $\mathit{\Pi}$ be a program and I an interpretation. Then, there is a complete, valid, and stable computation C for $\mathit{\Pi}$ with initial state $S_0 = \langle \emptyset, \emptyset \rangle$ and $\mathit{res}(\mathrm{C}) = I$ iff I is an answer set of $\mathit{\Pi}$.*

Theorem 2 can be generalised in the sense that we can start to generate a computation for $\mathit{\Pi}$ using $\Upsilon_{\mathit{\Pi}}(\cdot)$ from an arbitrary valid and stable state S_0 of $\mathit{\Pi}$ (not just from $\langle \emptyset, \emptyset \rangle$) and still reach every answer set $I \supseteq I_{S_0}$ of $\mathit{\Pi}$ where $R_{S_0} \subseteq \mathit{supp}^I(gr(\mathit{\Pi}))$.

Algorithm 1 Basic Algorithm

```
Require: $\mathit{\Pi}$ is a program, $S_0$ is a breakpoint of $\mathit{\Pi}$
1: $\mathrm{C} := S_0$
2: $\mathit{forbidden} := \bigcup_{l \in I_{S_0}} \bar{l} \cup \bigcup_{r \in R_{S_0}} \mathrm{B}^-(r)$
3: $i := 0$
4: while $\{r \in \mathit{gr}(\mathit{\Pi}) \setminus R_{S_i} \mid I_{S_i} \models \mathrm{B}(r)\} \neq \emptyset$ do
5:     $\mathit{pool} := \{\langle \mathrm{H}(r), r\rangle \mid r \in \mathit{gr}(\mathit{\Pi}) \setminus R_{S_i}, I_{S_i} \models \mathrm{B}(r), \mathrm{H}(r) \notin \mathrm{B}^-(r), \mathrm{H}(r) \notin \mathit{forbidden}\}$
6:     if $\mathit{pool} = \emptyset$ then
7:         print Computation C is stuck
8:         ANSWER USER QUERIES
9:         return
10:    end if
11:    SHOW DIAGNOSTICS
12:    ANSWER USER QUERIES
13:    USER ASSIGN: $\langle l, r\rangle :=$ an element of $\mathit{pool}$
14:    $S_{i+1} := \langle I_{S_i} \cup \{l\}, R_{S_i} \cup \{r\}\rangle$
15:    $\mathrm{C} := \mathrm{C}, S_{i+1}$
16:    $\mathit{forbidden} := \mathit{forbidden} \cup \mathrm{B}^-(r)$
17:    $\mathit{forbidden} := \mathit{forbidden} \cup \{\bar{l}\}$
18:    $i := i + 1$
19: end while
20: return C
```

Theorem 3. *Let $\mathit{\Pi}'$ and $\mathit{\Pi}$ be programs and consider two interpretations $I' \subseteq I$ such that $I \in AS(\mathit{\Pi})$, $\mathit{\Pi}' \subseteq \mathit{supp}^I(\mathit{gr}(\mathit{\Pi}))$, and $\langle I', \mathit{\Pi}'\rangle$ is a valid and stable state of $\mathit{\Pi}$. Then, there is a fixed-point iteration $S_i = \Upsilon_{\mathit{\Pi}}(S_{i-1})$ with $i > 0$ where $S_0 = \langle I', \mathit{\Pi}'\rangle$ that reaches the fixed point $S_n = \langle I, \mathit{supp}^I(\mathit{gr}(\mathit{\Pi}))\rangle$ such that $\mathrm{C} = S_0, \ldots, S_n$ is a complete and stable computation for $\mathit{\Pi}$.*

This result is very useful for debugging as it allows for taking an arbitrary subset of rules of the program that are considered to be supporting, compute an answer set, and start the search for a bug from there. For example, there will in most cases be no need for stepwise adding the facts of a program. Furthermore, often large parts of a program are trusted and mainly the latest rules added are suspected to be buggy. Moreover, it is convenient that in the course of the development of a program a handful of starting states for debugging sessions can be kept and maintained such that the programmer can quickly initiate debugging from situations he or she is already familiar with.

In analogy to debugging in procedural programs, we identify potential starting points—valid and stable states of a program—as *breakpoints* that enable a programmer to directly jump to an interesting debugging situation.

Definition 5. *A valid and stable state of a program $\mathit{\Pi}$ is called a* breakpoint *of $\mathit{\Pi}$.*

4 Debugging Strategy

In this section, we want to sketch an interactive debugging algorithm that implements the operator $\Upsilon_{\mathit{\Pi}}(\cdot)$ for generating a computation for a program $\mathit{\Pi}$ and that allows the user to control and analyse the progress of this process. We first discuss the basic structure of the proposed debugging process, realised by Algorithm 1, and afterwards discuss how an interface between the programmer and the system should be designed. Without its interactive parts, the procedure works similar to a non-deterministic algorithm due to Iwayama and Satoh [11], except that rules deriving a literal that is considered to be false are filtered out prior to selection in Algorithm 1.

Algorithm. The input of the algorithm is given by a program Π to be debugged and a breakpoint S_0 of Π. In Line 1, the variable C for the computed computation is initialised with the breakpoint. Then, the literals considered to be false, according to the rules in R_{S_0} chosen to be supporting, are computed and stored in the $\mathit{forbidden}$ variable. The loop entered in Line 4 iterates as long as there are further applicable rules that are not contained in the rules R_{S_i} of the currently considered state S_i. At the beginning of every iteration, the set *pool* is computed that consists of all pairs $\langle l, r\rangle$ of a literal l that might be added to the growing interpretation, being the head of an applicable unconsidered rule r. The elements of *pool* reflect the non-deterministic results of $\Upsilon_{\Pi}(S_i)$ in the sense that $\langle l, r\rangle \in \mathit{pool}$ iff there is some pair $\langle I, R\rangle$ with $\Upsilon_{\Pi}(S_i) = \langle I, R\rangle$ where $I = I_{S_i} \cup \{l\}$ and $R = R_{S_i} \cup \{r\}$. If *pool* is empty (Line 6), the non-deterministic branch considered in the computation did not lead to an answer set of Π as there are unconsidered rules that derive a literal which is considered to be forbidden. In this case, before aborting, the algorithm enters a phase of user interaction (Line 8) that allows the programmer to explore the reasons for failure—this is further outlined below.

If *pool* is non-empty, the user is first informed about some key properties of the current status of the computation. Also this interface-related step in the algorithm, along with the subsequent interaction phase in Line 12, are described below in more detail. After the user decides to proceed, he or she chooses an element of *pool* for updating the current interpretation and the set of rules considered applied in Line 13. Consequently, the next state is computed and the variable $\mathit{forbidden}$ is updated.

Interaction and Interface Aspects. In Line 11, our algorithm is designed to present a report about the current status of the computation. In a concrete realisation, this should include information whether the computation has failed, i.e., whether there is no answer set I of Π such that both $I_{S_i} \subseteq I$ and $R_{S_i} \subseteq \mathit{supp}^I(\mathit{gr}(\Pi))$. Moreover, if it has failed, the user should be informed if there is an answer set I of Π such that $I_{S_i} \subseteq I$ but $R_{S_i} \not\subseteq \mathit{supp}^I(\mathit{gr}(\Pi))$. In the latter case, the user might target a computation that has an actual answer set I of Π as its result but considers a rule as a supporting rule which is not supporting with respect to I. For example, for the program consisting of the rule $b \leftarrow \mathrm{not}\ a$ and the facts a and b, if a computation arrives at state $\langle\{a, b\}, \{a, b \leftarrow \mathrm{not}\ a\}\rangle$, it has failed although $\{a, b\}$ is an answer set because the rule $b \leftarrow \mathrm{not}\ a$ has assumed to be supporting. Another information to be given as instructed at Line 11 is whether there is a rule $r \in \mathit{gr}(\Pi) \setminus R_{S_i}$ not considered so far that cannot become satisfied at any latter iteration because $\mathrm{B}^+(r) \subseteq I_{S_i}$, $\mathrm{B}^-(r) \subseteq \mathit{forbidden}$, and also $\mathrm{H}(r) \in \mathit{forbidden}$. In such a case, the computation will always fail.

The interaction phases referred to at Lines 8 and 12 should allow the user to pose more detailed queries about the evolving computation to gain further insights in the program's behaviour and potential bugs. Once the computation is stuck (Line 8), the programmer might be interested in why a computation has failed. Here, the system should be able to state why applicable rules in $gr(\Pi) \setminus R_{S_i}$ cannot be added, i.e., at which former state their head literals were added to $\mathit{forbidden}$. Moreover, for rules $r \in gr(\Pi)$ such that $R_{S_i} \not\models \mathrm{B}(r)$, the system should highlight the rule's false body literals. In the interaction phase of Line 12, the user should additionally be allowed to inspect which answer sets of the final program Π could still be reached by the current computation, i.e., answer sets $I \in AS(\Pi)$ such that $I_{S_i} \subseteq I$ and $R_{S_i} \subseteq \mathit{supp}^I(gr(\Pi))$.

We next discuss considerations concerning the interface for realising our approach. For dealing with rules of the grounding of the program to debug, e.g., when querying which literals of a rule instantiation are false, it is essential to have a human-computer-interface that allows for quickly and conveniently referencing a certain ground instance of a non-ground rule $r \in \Pi$. One option here is to specify partial variable assignments using auto-complete mechanisms for filtering the ground instances of r to display.

Another aspect is that in a realisation of the debugging method it will be convenient to select multiple ground rule instances at the same time and hence determine several subsequent states of the computation at once. In particular, an option could be to select all instances of a non-ground rule in the set pool at once. One aspect here is that when multiple rules are selected, checks are required whether these rules can all jointly be supporting. Moreover, for selecting different instances of a non-ground rule, again the graphical interface should be designed in such a way that a programmer is able to select intended instances fast without being visually distracted by too many of them.

In our approach, an evolving interpretation is not automatically joined with its immediate consequences to give the programmer maximal freedom to guide the computation. However, the user should have the ability to let non-conflicting elements $\langle l, r \rangle$ of pool, where $\mathrm{B}^-(r) \subseteq \mathit{forbidden}$, be applied on demand.

A further useful feature is navigation in and reuse of an ongoing computation. The programmer should at any point in the algorithm be able to backtrack one or more states and continue from an earlier state to revise a decision on which element of pool to choose. Moreover, a feature for loading and saving a computation into the variable C should be provided such that previous debugging sessions can be resumed if necessary.

5 An Application Scenario

The following typical examples of errors occurring in ASP were adopted from previous work [7] illustrating another debugging approach (cf. also the discussion in Section 6). We now show how these debugging problems can be handled with the current method.

Assume that students have the task to encode the problem of assigning papers to members of the program committee (PC) of a conference for reviewing, based on some bidding information in terms of ASP. We consider two cases illustrating different debugging problems. In the first case, multiple answer sets are expected but the program yields only one of the expected answer sets. In the other case, it is expected that a program is inconsistent, but it actually yields some answer set. We illustrate that, in both cases, our approach gives valuable hints how to debug the program in an iterative way.

In what follows, an atom $pc(X)$ means that X is a PC member, $\mathit{paper}(X)$ 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 between 0 and 3, expressing a degree of preference for the paper.

Student Linus tried to formalise that each paper is non-deterministically assigned to at least one member of the PC. His program looks as follows:[2]

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

Linus expects that the two rules realise the non-deterministic guess, and then the constraint prunes away all answer set candidates where a paper is not assigned to some PC member. Linus is desperate since the non-deterministic guess seems not to work correctly; the only answer set of Q is

$$\begin{aligned} S_Q = F_Q \cup \{ & assigned(p_1, m_1), assigned(p_1, m_2), \\ & assigned(p_2, m_1), assigned(p_2, m_2)\}, \end{aligned}$$

where F_Q is the set of facts in Q, although Linus expected one answer set for each possible assignment. In particular, he expected

$$E_Q = (S_Q \cup \{\neg assigned(p_1, m_2)\}) \setminus \{assigned(p_1, m_2)\}$$

to be an answer set. Linus starts the debugging algorithm at breakpoint $\langle F_Q, F_Q \rangle$. Among the available applicable ground rules, he chooses

$$r_1 = \neg assigned(p_1, m_2) \leftarrow paper(p_1), pc(m_2), \text{not } assigned(p_1, m_2)$$

which derives $\neg assigned(p_1, m_2)$ as intended. The debugger warns that the constraint

$$c_1 = \leftarrow paper(p_1), pc(m_2), \text{not } assigned(p_1, m_2)$$

can now not be satisfied in any further state, however, as $assigned(p_1, m_2)$ has been considered a forbidden literal when adding r_1.

Linus already suspects the cause for the issue, but to be sure he continues the debugging process. As every paper needs to be assigned, and in the answer set Linus is aiming for PC member m_2 not being assigned to paper p_1, he next chooses

$$assigned(p_1, m_1) \leftarrow paper(p_1), pc(m_1), \text{not } \neg assigned(p_1, m_1)$$

to be added as supporting rule. As already predicted by the debugging algorithm, m_1 being assigned to p_1 does not change constraint c_1 from being unsatisfied. Linus suspicion has been confirmed, the constraint

$$\leftarrow paper(P), pc(M), \text{not } assigned(P, M)$$

requires that each paper is assigned to *all* PC members. The original intention was that it only requires a paper to be assigned to a single PC member. Hence, he replaces the constraint by the two rules

[2] Note that an expression r of form $\leftarrow \mathrm{B}^+(r), \mathrm{B}^-(r)$ is a *constraint* standing for a rule $a \leftarrow \mathrm{B}^+(r), \mathrm{B}^-(r) \cup \{a\}$, where a is a globally new atom.

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

The resulting program yields the nine expected answer sets as expected.

In the meantime, Peppermint Patty is writing a program addressing the following issue: If a PC member M bids 0 on some paper P, then 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 Patty's solution:

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

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

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

What Patty finds puzzling is that S_R does not contain any atoms signalling a conflict of interest. Similar to Linus, she starts out with breakpoint $\langle F_R, F_R \rangle$, where F_R is the set of facts in R. To her surprise, the rule

$$r_1 = \mathit{conflict}(m_1, p_1) \leftarrow \mathit{pc}(m_1), \mathit{paper}(p_1), \mathit{author}(m_1, p_1)$$

is not applicable with respect to F_R. She queries the debugging system why the body of r_1 is not satisfied with respect to F_R and gets the answer that $\mathit{author}(m_1, p_1) \notin F_R$. Now Patty realises that she used the fact $\mathit{author}(p_1, m_1)$ instead of $\mathit{author}(m_1, p_1)$, accidentally swapping the predicate's arguments. After this bug is fixed the program is correct.

6 Related Work

Pontelli, Son, and El-Khatib [6] introduced *justifications* for ASP. Roughly speaking, a justification is a labelled directed graph that explains the truth value of a literal with respect to an answer-set in terms of dependency on the truth values of fellow literals. Contrary to our approach, justifications pull out focused information concerning a single artifact—the literal under consideration—rather than giving a holistic view on the computation. Interesting with respect to our technique is the notion of an *online justification* that explains truth values with respect to partial answer sets emerging during the solving process. As our approach is compatible with the model of computation for online justifications, they can be used in a combined debugging approach. While interactively stepping through a computation allows for following individual intuitions concerning rule applications, justifications like graphs could keep track of the chosen support for individual literals of interest. A potential shortcoming concerning the intuition of justifications is the absence of program rules, constituting the actual source code artifacts in the graphs.

Another graph-based approach that aims at visualising answer-set computation is realised in the `noMoRe`-system that utilises rule dependency graphs (RDGs) which are directed labelled graphs where the nodes are the rules of a given program [15]. Answer sets can be computed by stepwise colouring the nodes of the RDG of a ground program either green or red, reflecting whether a rule is considered applicable or not. At special steps, an answer set is formed by the heads of the green coloured rules.

Other previous work on declarative debugging centred on the question why a given interpretation is not an answer set of a program [7]. The answers are given in terms of rule instances that are unsatisfied, or loops that are unfounded. The reasons for an interpretation not being an answer set might be easier to understand in the procedural approach as the user is not confronted with several unsatisfied rules of the whole intended answer set at once and unfounded loops never appear in a computation of our current approach.

In contrast to fixed-point definitions of answer sets (as, e.g., described in the works of Saccà and Zaniolo [9] and of Leone, Rullo, and Scarcello [16]) that aim at a quick automatic evaluation, we do not always apply all knowledge that is available and focus on the programmer's intuition instead, e.g., immediate consequences of a current state of a computation are not added necessarily in case they are not of interest to the user.

While the algorithm we use for our computation focuses on readability and works on the rule level, the algorithms used for computing answer sets [17, 16, 18] are often quite involved and hard to grasp for an ordinary programmer. Moreover, following a concrete execution of a solver will not be very helpful when the user is not able to focus on parts of the execution that are of interest. Indeed, a tracing system developed for DLV [19] is intended for debugging the solver itself rather than the answer-set programs.

7 Conclusion

We presented a debugging approach for ASP with a procedural flavour that allows the programmer to follow his or her own intuitions on which rules to apply. It is based on an intuitive and simple computation model in which at each state a rule that is considered a supporting rule is added. In future work, we want to extend the approach to disjunctive programs and implement a prototype as part of an integrated development environment for ASP. Besides making debugging easy, using a debugging system as envisaged could improve the inexperienced programmer's understanding of the answer-set semantics.

References

1. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proc. ASP'05, `CEUR-WS.org` (2005)
2. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proc. NMR'06. (2006) 77–83

3. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proc. LPNMR'07. Volume 4483 of LNCS, Springer (2007) 31–43
4. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A theoretical framework for the declarative debugging of datalog programs. In: Proc. SDKB'08. Volume 4925 of LNCS, Springer (2008) 143–159
5. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proc. AAAI'08, AAAI Press (2008) 448–453
6. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1) (2009) 1–56
7. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4–6) (2010) 513–529
8. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Proc. ICLP'09. Volume 5649 of LNCS, Springer (2009) 296–311
9. Saccà, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Proc. PODS'90, ACM Press (1990) 205–217
10. Fages, F.: A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. In: Proc. ICLP'90. (1990) 441–458
11. Iwayama, N., Satoh, K.: Computing abduction by using TMS with top-down expectation. Journal of Logic Programming **44**(1-3) (2000) 179 – 206
12. Seipel, D., Minker, J., Ruiz, C.: Model generation and state generation for disjunctive logic programs. Journal of Logic Programming **32**(1) (1997) 49–69
13. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Proc. JELIA'04. Volume 3229 of LNCS, Springer (2004) 200–212
14. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3-4) (1991) 365–386
15. Bösel, A., Linke, T., Schaub, T.: Profiling answer set programming: The visualization component of the nomore system. In: Proc. JELIA'04. Volume 3229 of LNCS, Springer (2004) 702–705
16. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation **135**(2) (1997) 69–112
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
18. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In: Proc. KR'08, AAAI Press (2008) 422–432
19. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for dlv. In: Proc. SEA'09. (2009)