Debugging of Answer Set Programs Using Paracoherent Reasoning

Bernardo Cuteri[1], Carmine Dodaro[2], and Francesco Ricca[1]

[1] University of Calabria, Italy - lastname@mat.unical.it
[2] University of Genoa, Italy - dodaro@dibris.unige.it

Abstract. Answer Set Programming is a well-known declarative programming paradigm proposed in the area of logic programming and non-monotonic reasoning. Although ASP features a simple syntax and an intuitive semantics, errors are common during the development of ASP programs. In this paper we propose a novel debugging approach based on paracoherent reasoning, which allows a user to identify bugs when they are related to wrong constraints. The approach has been implemented in a tool called PARADEBUG, that is made freely available.

Keywords: Answer Set Programming · Debugging Techniques · Paracoherent Reasoning.

1 Introduction

Answer Set Programming (ASP) [21, 42] is a well-established declarative programming paradigm based on the stable model semantics. The simple syntax [22] and the intuitive semantics [42], combined with the availability of robust implementations [4, 11, 12, 24, 36–40, 43, 44, 46, 48, 49], make ASP an ideal candidate for addressing hard combinatorial problems. As matter of fact, ASP has been successfully used in several research areas, including Artificial Intelligence [3, 10, 14, 18, 34], Hydroinformatics [35], Nurse Scheduling [7], Bio-informatics [31, 45], Game Theory [9, 17], and Databases [47, 50]; more recently ASP has been applied to solve industrial applications [1, 27, 30].

Albeit the basic syntax and the semantics of ASP are in general clear also for novice programmers, during the development of ASP encodings, the identification of (trivial) errors can be time consuming. For this reason, during the recent years, several techniques and tools, called *debuggers*, emerged to help the programmer to deal with faults in ASP programs [19, 28, 29, 41, 51], thus making the development process faster and more comfortable.

In this paper we provide a practical contribution in the aforementioned context by considering an important question related to debug, i.e. why an interpretation is not an answer set of a program under consideration. In particular, we report on a preliminary debugging technique based on paracoherent reasoning [8, 15, 16], which allows a user to debug ASP programs when the fault is localized into the constraints of the program. Roughly, the suggested approach

can be described as follows: First, given a non-ground program $\mathcal{P}$, the debugger generates a ground program Π. Next, the constraints of Π are processed and *normalized*, i.e. they are converted into normal rules. Rules created in the normalization step are then rewritten using the paracoherent techniques presented in [11–13] and a paracoherent answer set is computed, which is subsequently used to identify the faulty constraints. This approach has been implemented in a python tool PARADEBUG, that is based on the well-known ASP system DLV2 [5, 2]. The resulting implementation can be used via command-line interface.

2 Preliminaries

2.1 Answer set programming

Syntax. An ASP program $\mathcal{P}$ is a finite set of rules of the form

$$h_1 \vee \ldots \vee h_n \leftarrow \ell_1, \ldots, \ell_m$$

where $n, m \geq 0$, $n + m \neq 0$, $h_1, \ldots, h_n$ are atoms and represent the *head* of the rule, while $\ell_1, \ldots, \ell_m$ are literals and represent the *body* of the rule. In particular, an *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate of ariety k and $t_1, \ldots, t_k$ are *terms*. Terms are alphanumeric strings, and are distinguished in variables and constants. According to the Prolog's convention, only variables start with an uppercase letter. A *literal* is an atom a or its negation *not* a, where *not* denotes the *negation as failure*. For an atom p, $\overline{p} = not\ p$, for a negated atom *not* p, $\overline{not\ p} = p$. A rule is called a *constraint* if $n = 0$, and a *fact* if $n = 1$ and $m = 0$. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_\mathcal{P}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_\mathcal{P}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_\mathcal{P}$. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$. Similarly, given a program $\mathcal{P}$, its *ground instantiation* is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$. Given a ground program Π, let $Rules(\Pi)$, $Constr(\Pi)$, and $At(\Pi)$ denote the set of ground rules, ground constraints and ground atoms occurring in Π.

Semantics. Given a program $\mathcal{P}$, its stable models are defined using its ground instantiation Π. Any set $I \subseteq At(\Pi)$ is an interpretation for a program Π. A ground atom p is *true* w.r.t. I if $p \in I$; p is *false* w.r.t. I if $p \notin I$. A ground literal *not* p is *true* w.r.t. I if $p \notin I$; *not* p is *false* w.r.t. I if $p \in I$. An interpretation I is a *model* for Π if, for every $r \in \Pi$, at least one atom in the head of r is true w.r.t. I whenever all literals in the body of r are true w.r.t. I. The *reduct* of a ground program Π w.r.t. a model I is the program Π^I, obtained from Π by (i) deleting all rules $r \in \Pi$ whose negative body is false w.r.t. I and (ii) deleting the negative body from the remaining rules. An interpretation I is an *answer set* (*stable model*) of a program Π if I is a model of Π, and there is no $J \subset I$

such that J is a model of Π^I. Let $AS(\Pi)$ denote the set of all answer sets of Π. A program Π is *coherent* if $AS(\Pi) \neq \emptyset$, *incoherent* otherwise.

Example 1. Consider the following program $\mathcal{P}$:

$$
\begin{array}{l}
node(X) \leftarrow edge(X,Y) \\
node(X) \leftarrow edge(Y,X) \\
col(X,blue) \vee col(X,red) \vee col(X,green) \leftarrow node(X) \\
\leftarrow col(X,C_1), col(Y,C_2), edge(X,Y), C_1 \neq C_2
\end{array}
$$

and the set of facts F:

$$edge(1,2) \leftarrow \qquad edge(2,3) \leftarrow$$

the ground instantiation of $\mathcal{P} \cup F$ is the program Π comprising the following set of rules $Rules(\Pi)$:

$$
\begin{array}{l}
edge(1,2) \leftarrow \qquad\qquad edge(2,3) \leftarrow \\
node(1) \leftarrow \quad node(2) \leftarrow \; node(3) \leftarrow \\
col(1,red) \vee col(1,green) \vee col(1,blue) \leftarrow \\
col(2,red) \vee col(2,green) \vee col(2,blue) \leftarrow \\
col(3,red) \vee col(3,green) \vee col(3,blue) \leftarrow
\end{array}
$$

and the following set of constraints $Constr(\Pi)$:

$$
\begin{array}{ll}
\leftarrow col(2,red), col(1,green) & \leftarrow col(2,red), col(1,blue) \\
\leftarrow col(2,green), col(1,red) & \leftarrow col(2,green), col(1,blue) \\
\leftarrow col(2,blue), col(1,red) & \leftarrow col(2,blue), col(1,green) \\
\leftarrow col(3,red), col(2,green) & \leftarrow col(3,red), col(2,blue) \\
\leftarrow col(3,green), col(2,red) & \leftarrow col(3,green), col(2,blue) \\
\leftarrow col(3,blue), col(2,red) & \leftarrow col(3,blue), col(2,green)\,.
\end{array}
$$

The set of all answer sets of Π is $AS(\Pi) = \{I_1, I_2, I_3\}$, where

$$
\begin{array}{l}
I_1 = A \cup \{col(1,red), col(2,red), col(3,red)\} \\
I_2 = A \cup \{col(1,blue), col(2,blue), col(3,blue)\} \\
I_3 = A \cup \{col(1,green), col(2,green), col(3,green)\}
\end{array}
$$

and $A = \{edge(1,2), edge(2,3), node(1), node(2), node(3)\}$.

2.2 Paracoherent semantics

Let Π be a ground ASP program such that each rule in Π is of the form:

$$a_1 \vee \cdots \vee a_n \leftarrow \; b_1, \cdots, b_k, not\ b_{k+1}, \cdots, not\ b_m \tag{1}$$

where $n \geq 1, m \geq k \geq 0$, and $a_1, \ldots, a_n, b_1, \ldots, b_m$ are ground atoms. Note that Π does not include constraints (since $n \geq 1$).

Externally supported program. The externally supported program is a ground program Π^s obtained as follows. For each rule $r \in \Pi$ of the form (1), Π^s contains the rule:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, not\ b_{k+1}, \cdots, not\ b_m, not\ Kb_{k+1}, \cdots, not\ Kb_m$$

and for each atom $p \in At(\Pi)$, Π^s contains the rules $Kp \vee nKp \leftarrow$ and $gap(Kp) \leftarrow Kp, not\ p$, where Kp, nKp and $gap(Kp)$ are atoms not appearing in Π. Intuitively, an atom Kp can be read as p is believed to hold.

Externally extended supported program. The externally extended supported program of Π is the program Π^{es} formed by Π^s and by rule:

$$Ka_1 \vee \cdots \vee Ka_n \vee Kb_{k+1} \vee \cdots \vee Kb_m \leftarrow Kb_1, \cdots, Kb_k, \\ not\ a_1, \cdots, not\ a_n, not\ b_{k+1}, \cdots, not\ b_m$$

for each $r \in \Pi$ of the form (1).

Paracoherent answer sets. For a set of atoms S, let $\mathcal{G}_S$ denote $\{gap(p) \mid gap(p) \in S\}$. For a program Π, $I \in AS(\Pi^{es})$ is an *externally supported model* (resp. *externally extended supported model*) of Π if there is no $M \in AS(\Pi^s)$ (resp. $M \in AS(\Pi^{es})$) such that $\mathcal{G}_M \subset \mathcal{G}_I$. In the following, we refer to externally supported models and externally extended supported models as *paracoherent answer sets*.

3 Debugging approach

In this section, we present our approach to the localization of faults in ground ASP programs. In general, one can differentiate between syntactic and semantic faults. In the first case, syntactic errors are automatically detected by parser of ASP systems, whereas semantic faults can be detected by a user by analyzing the output of the ASP system.

In the latter case, users usually verify the correctness of programs by checking (simple) test instances, as it is common for software development. In particular, the debugging process is carried out by comparing a number of answer sets computed by an ASP system with solutions determined by hand. Therefore, in order to detect a bug, a user must know at least one answer set of the program for the analyzed instance.

Definition 1 (Buggy program). *Let Π^c be a ground intended (correct) program that a user is going to formulate and $AS(\Pi^c)$ be a set of its known answer sets. Then, a ground program Π is said to be buggy with respect to a program Π^c if there exists an answer set $A \in AS(\Pi^c)$ such that $A \notin AS(\Pi)$.*

Note that by this definition our approach deals only with situations in which some answer set of the correct program is missing. The opposite problem – there is an answer set $A \in AS(\Pi)$ such that $A \notin AS(\Pi^c)$ – is not the focus of this paper.

Example 2 (Buggy program). Consider the program $\mathcal{P}$ of Example 1, representing a (buggy) encoding for the graph coloring problem. During the development of the encoding the user might create a simple graph, e.g. considering the sample instance of the two facts in F. For this instance the user expects the assignment of the *blue* color to the nodes 1 and 3 as well as of the *red* color to the node 2 to be among the solutions. However, the corresponding answer set encoding this solution is missing due to a bug in the encoding. In particular, note that the condition $C_1 \neq C_2$ should be replaced by $C_1 = C_2$. $\triangleleft$

The situation in which some solution is missing can be detected by means of testing, which is a common approach in software engineering aiming at identification and localization of faults in programs.

Definition 2 (Test case). *Let Π^c be a ground intended program, and Π be a ground program. A set of atoms $T \subseteq At(\Pi)$ is a* test case *for Π iff there exists an answer set $A \in AS(\Pi^c)$ such that $T \subseteq A$.*

Definition 3 (Test case failure). *Given a ground program Π and a test case T, let $\Pi_T = \{\leftarrow \bar{l} \mid l \in T\}$, we say that T* fails *if $\Pi \cup \Pi_T$ is incoherent.*

Assertions of a test case are modeled by constraints that force the asserted atoms to be in all answer sets. As a result, checking whether a test case T of a program Π passes or not is reduced to checking whether $\Pi \cup \Pi_T$ is coherent, as illustrated in Example 3.

Example 3 (Failing Test Case). Consider the program Π from Example 1 and the test case

$$T = \{col(1, blue), col(2, red), col(3, blue)\}.$$

The program Π_T is composed by the constraints:

$$\leftarrow not\ col(1, blue) \qquad \leftarrow not\ col(2, red) \qquad \leftarrow not\ col(3, blue).$$

Thus, T is failing since $\Pi \cup \Pi_T$ is incoherent. $\triangleleft$

Whenever a test case fails, i.e. the given program Π is buggy, the goal of a debugger is to find an explanation for this observation.

In the following, we assume that given a ground program Π and a test case T, the set $Rules(\Pi) \cup \Pi_T$ is coherent and correct, and the buggy rule is located in $Constr(\Pi)$. Indeed, the goal of our debugger is to find a set of constraints that are causing the incoherence.

Definition 4 (Diagnosis). *Given a ground program Π and a failing test case T, a diagnosis D of the fault is a minimal set of constraints $C \subseteq Constr(\Pi)$ such that $(\Pi \cup \Pi_T) \setminus C$ is coherent.*

Example 4 (Diagnosis). Consider again the program Π from Example 1 and the program Π_T of Example 3. A diagnosis of the fault D is the set of constraints $\{\leftarrow col(2, red), col(1, blue), \leftarrow col(3, blue), col(2, red)\}$. Indeed, $(\Pi \cup \Pi_T) \setminus D$ is coherent and no subset of D is a diagnosis. $\triangleleft$

In our approach, fault identification is done by taking advantages of paracoherent semantics to find buggy constraints. The key idea is as follows. Given a ground program Π and the set of constraints $Constr(\Pi)$, each constraint $c \in Constr(\Pi)$ of the form $\leftarrow l_1, \ldots, l_m$ is *normalized*, i.e. it is rewritten as the following rule:

$$aux_c \leftarrow not\ aux_c, l_1, \ldots, l_m$$

where aux_c is a fresh symbol not appearing elsewhere in the program.

Definition 5 (Normalization). *Given a ground program $\Pi = Rule(\Pi) \cup Constr(\Pi)$, then the normalized program of Π, denoted $Norm(\Pi)$, is composed by $\{aux_c \leftarrow not\ aux_c, \ell_1, \ldots, \ell_m \mid \leftarrow \ell_1, \ldots, \ell_m \in Constr(\Pi)\}$.*

The following example should clarify this aspect.

Example 5 (Normalization). Let Π be the ground program of Example 1. Then, $Norm(\Pi)$ comprises the following set of rules:

$$\begin{aligned}
aux_{c_1} &\leftarrow not\ aux_{c_1}, col(2, red), col(1, green)\\
aux_{c_2} &\leftarrow not\ aux_{c_2}, col(2, red), col(1, blue)\\
aux_{c_3} &\leftarrow not\ aux_{c_3}, col(2, green), col(1, red)\\
aux_{c_4} &\leftarrow not\ aux_{c_4}, col(2, green), col(1, blue)\\
aux_{c_5} &\leftarrow not\ aux_{c_5}, col(2, blue), col(1, red)\\
aux_{c_6} &\leftarrow not\ aux_{c_6}, col(2, blue), col(1, green)\\
aux_{c_7} &\leftarrow not\ aux_{c_7}, col(3, red), col(2, green)\\
aux_{c_8} &\leftarrow not\ aux_{c_8}, col(3, red), col(2, blue)\\
aux_{c_9} &\leftarrow not\ aux_{c_9}, col(3, green), col(2, red)\\
aux_{c_{10}} &\leftarrow not\ aux_{c_{10}}, col(3, green), col(2, blue)\\
aux_{c_{11}} &\leftarrow not\ aux_{c_{11}}, col(3, blue), col(2, red)\\
aux_{c_{12}} &\leftarrow not\ aux_{c_{12}}, col(3, blue), col(2, green)
\end{aligned}$$

Note that the number of symbols added in $Norm(\Pi)$ is bounded by the number of constraints in Π. ◁

Then, paracoherent rewriting techniques described in Section 2.2 are applied to the rules introduced in the normalization step.

Definition 6 (Debugging program). *Let $\Pi = Rules(\Pi) \cup Constr(\Pi)$ be a ground program and a test case T. The debugging program $\Pi^{\mathcal{D}}$ is the program $\Pi_T \cup Rules(\Pi) \cup Norm(\Pi)^{\chi}$, where $\chi \in \{s, es\}$.*

Given a debugging program $\Pi^{\mathcal{D}}$, its paracoherent answer sets can be used to build a diagnosis of the fault. Indeed, atoms of the form aux_c are included in a paracoherent answer set, say A, if and only if the corresponding constraint c cannot be satisfied by any answer set. Therefore, all the constraints associated to atoms of the form aux_c that are true w.r.t. A are part of the diagnosis. Note that, as shown in [15, 16], if the paracoherent reasoning is limited to constraints the two rewriting techniques described in Section 2.2 coincide.

4 Tool description and usage example

In this section, we describe a practical implementation of the debugging technique for the computation of a diagnosis described in Section 3. The resulting tool, namely PARADEBUG, is freely available here: `https://www.mat.unical.it/~dodaro/paradebug`. Moreover, we also present a usage example of PARADEBUG to debug a program.

PARADEBUG is implemented in *Python* and it takes advantage of the ASP system DLV2 [5], which uses I-DLV [23] and WASP [6] as grounder and solver, respectively. As far as we know, DLV2 is the only ASP system that is able to compute paracoherent answer sets.

PARADEBUG takes as input a ground program Π and a test case T. Then, Π is normalized and the debugging program $\Pi^{\mathcal{D}}$ is built starting from Π and T. Subsequently, $\Pi^{\mathcal{D}}$ is provided as input to the internal solver WASP, which returns as output a paracoherent answer set of $\Pi^{\mathcal{D}}$, say M. M is then processed by PARADEBUG and the set of constraints representing the diagnosis are returned.

Usage example. Consider the following buggy ASP program representing an encoding of the Graph Coloring problem:

```
node(X) :- edge(X,_).
node(X) :- edge(_,X).

col(X,blue) | col(X,red) | col(X,green) :- node(X).
:- col(X1,C1), col(X2,C2), edge(X1,X2), C1 != C2.

edge(1,2).
edge(2,3).
```

Note that it represents the program $\mathcal{P}$ of Example 1. In this case, the buggy rule is

```
:- col(X1,C1), col(X2,C2), edge(X1,X2), C1 != C2.
```

since the condition `C1 != C2` should be `C1 = C2`.

Test cases can be specified according to the directives `assertTrue`, that specifies atoms that must be in an answer set, and `assertFalse`, that specifies atoms that must not be in an answer set. In particular, a test case can be the following:

```
assertTrue: col(1,blue).
assertTrue: col(2,red).
assertTrue: col(3,blue).
```

which basically represent a scenario where the atoms `col(1,blue)`, `col(2,red)`, and `col(3,blue)` are all true. However, this answer set cannot be obtained due to the buggy rule. PARADEBUG can thus be used to find what is the buggy rule by using the following command:

```
dlv2 --mode=idlv test.asp | ./paradebug.py testcase
```

The output of PARADEBUG is the set of buggy constraints as follows:

```
Set of constraints causing the incoherence:
:- col(1,blue),  col(2,red).
:- col(2,red),  col(3,blue).
```

which indeed represents a correct diagnosis as shown in Example 4.

5 Related work

There are multiple approaches to ASP debugging suggested in the literature including algorithmic [19, 54] and meta-programming [20, 41, 51–53] methods, see [32] for a comprehensive survey on the topic. The algorithmic approaches include IDEAS [19], a state-of-the-art tool, that aims at identifying why a set of atoms is an answer set or why a set of atoms is not in any answer set. Moreover, IDEAS implements a query-based interaction with users in order to find an explanation of an observed fault.

Concerning meta-programming debuggers, they are usually based on a general ASP encoding modeling all possible reasons of why some interpretation of the faulty program is not an answer set. Among the tool based on meta-programming, two tools emerged, namely SPOCK [41] and OUROBOROS [51, 52]. The first can be applied only to ground programs, whereas OUROBOROS can tackle non-grounded programs as well.

Our approach is also related to the one implemented in the debugging tool DWASP, that is also built on top of the ASP solver WASP. The approach of DWASP relies on iterative calls to the WASP and on the computation of the so-called *unsatisfiable cores*. Our approach is instead based on only one call to the underlying ASP system. Moreover, DWASP features a query-based approach to help the user to identify the buggy rules. Such a feature is not yet implemented in PARADEBUG, but it can be integrated as well.

6 Conclusion and future work

In this paper we presented a debugging approach for ASP programs based on paracoherent reasoning. This technique has been implemented in a tool called PARADEBUG, based on the ASP system DLV2. As future work, we plan to extend the technique presented in this paper to normal and disjunctive rules that are not supported at the moment. Moreover, we plan to implement the query-based approach presented in [28, 29] and to integrate the resulting tool into the IDE ASPIDE [33]. Furthermore, we also plan to investigate whether the techniques presented in this paper can be extended to other paracoherent semantics, e.g. [25, 26].

References

1. Abseher, M., Gebser, M., Musliu, N., Schaub, T., Woltran, S.: Shift design with answer set programming. Fundam. Inform. **147**(1), 1–25 (2016)
2. Adrian, W.T., Alviano, M., Calimeri, F., Cuteri, B., Dodaro, C., Faber, W., Fuscà, D., Leone, N., Manna, M., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV: advancements and applications. KI **32**(2-3), 177–179 (2018)
3. Adrian, W.T., Manna, M., Leone, N., Amendola, G., Adrian, M.: Entity set expansion from the web via ASP. In: ICLP-TC. OASICS, vol. 58, pp. 1:1–1:5 (2017)
4. Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M., Ricca, F.: Evaluation of disjunctive programs in WASP. In: LPNMR. LNCS, vol. 11481, pp. 241–255. Springer (2019)
5. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: LPNMR. Lecture Notes in Computer Science, vol. 10377, pp. 215–221. Springer (2017)
6. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 40–54. Springer (2015)
7. Alviano, M., Dodaro, C., Maratea, M.: An advanced answer set programming encoding for nurse scheduling. In: AI*IA. Lecture Notes in Computer Science, vol. 10640, pp. 468–482. Springer (2017)
8. Amendola, G.: Dealing with incoherence in ASP: split semi-equilibrium semantics. In: DWAI@AI*IA. CEUR Workshop Proceedings, vol. 1334, pp. 23–32. CEUR-WS.org (2014)
9. Amendola, G.: Preliminary results on modeling interdependent scheduling games via answer set programming. In: RiCeRcA@AI*IA. CEUR Workshop Proceedings, vol. 2272 (2018)
10. Amendola, G.: Solving the stable roommates problem using incoherent answer set programs. In: RiCeRcA@AI*IA. CEUR Workshop Proceedings, vol. 2272 (2018)
11. Amendola, G., Dodaro, C., Faber, W., Leone, N., Ricca, F.: On the computation of paracoherent answer sets. In: AAAI. pp. 1034–1040. AAAI Press (2017)
12. Amendola, G., Dodaro, C., Faber, W., Pulina, L., Ricca, F.: Algorithm selection for paracoherent answer set computation. In: JELIA. LNCS, vol. 11468, pp. 479–489. Springer (2019)
13. Amendola, G., Dodaro, C., Faber, W., Ricca, F.: Externally supported models for efficient computation of paracoherent answer sets. In: AAAI. pp. 1720–1727. AAAI Press (2018)
14. Amendola, G., Dodaro, C., Leone, N., Ricca, F.: On the application of answer set programming to the conference paper assignment problem. In: AI*IA. LNCS, vol. 10037, pp. 164–178. Springer (2016)
15. Amendola, G., Eiter, T., Fink, M., Leone, N., Moura, J.: Semi-equilibrium models for paracoherent answer set programs. Artif. Intell. **234**, 219–271 (2016)
16. Amendola, G., Eiter, T., Leone, N.: Modular paracoherent answer sets. In: JELIA. Lecture Notes in Computer Science, vol. 8761, pp. 457–471. Springer (2014)
17. Amendola, G., Greco, G., Leone, N., Veltri, P.: Modeling and reasoning about NTU games via answer set programming. In: IJCAI'16. pp. 38–45 (2016)
18. Balduccini, M., Gelfond, M., Watson, R., Nogueira, M.: The usa-advisor: A case study in answer set planning. In: LPNMR. Lecture Notes in Computer Science, vol. 2173, pp. 439–442. Springer (2001)
19. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In: Answer Set Programming. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org (2005)

20. Brain, M., Gebser, M., Schaub, T., Tompits, H., Woltran, S.: "That is Illogical Captain !" – The Debugging Support Tool spock for Answer-Set Programs : System Description. In: SEA. pp. 71–85 (2007)
21. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
22. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: ASP-Core-2 Input Language Format (2013), `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf`
23. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I-DLV: the new intelligent grounder of DLV. Intelligenza Artificiale **11**(1), 5–20 (2017)
24. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. **231**, 151–181 (2016)
25. Costantini, S., Formisano, A.: Negation as a resource: a novel view on answer set semantics. Fundam. Inform. **140**(3-4), 279–305 (2015)
26. Costantini, S., Formisano, A.: Query answering in resource-based answer set semantics. TPLP **16**(5-6), 619–635 (2016)
27. Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., Schekotihin, K.: Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). TPLP **16**(5-6), 653–669 (2016)
28. Dodaro, C., Gasteiger, P., Musitsch, B., Ricca, F., Shchekotykhin, K.M.: Interactive debugging of non-ground ASP programs. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 279–293. Springer (2015)
29. Dodaro, C., Gasteiger, P., Reale, K., Ricca, F., Schekotihin, K.: Debugging non-ground ASP programs: Technique and graphical tools. TPLP **19**(2), 290–316 (2019)
30. Dodaro, C., Leone, N., Nardi, B., Ricca, F.: Allotment problem in travel industry: A solution based on ASP. In: RR. Lecture Notes in Computer Science, vol. 9209, pp. 77–92. Springer (2015)
31. Erdem, E., Öztok, U.: Generating explanations for biomedical queries. TPLP **15**(1), 35–78 (2015)
32. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming - A survey of explanation approaches. TPLP **19**(2), 114–203 (2019)
33. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: integrated development environment for answer set programming. In: LPNMR. Lecture Notes in Computer Science, vol. 6645, pp. 317–330. Springer (2011)
34. Gaggl, S.A., Manthey, N., Ronca, A., Wallner, J.P., Woltran, S.: Improved answer-set programming encodings for abstract argumentation. TPLP **15**(4-5), 434–448 (2015)
35. Gavanelli, M., Nonato, M., Peano, A.: An ASP approach for the valves positioning optimization in a water distribution system. J. Log. Comput. **25**(6), 1351–1369 (2015)
36. Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., Schaub, T.: Evaluation techniques and systems for answer set programming: a survey. In: IJCAI'18. pp. 5450–5456 (2018)
37. Gebser, M., Maratea, M., Ricca, F.: The design of the sixth answer set programming competition - - report -. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 531–544. Springer (2015)
38. Gebser, M., Maratea, M., Ricca, F.: What's hot in the answer set programming competition. In: AAAI. pp. 4327–4329. AAAI Press (2016)
39. Gebser, M., Maratea, M., Ricca, F.: The design of the seventh answer set programming competition. In: Balduccini, M., Janhunen, T. (eds.) LPNMR. LNCS, vol. 10377, pp. 3–9. Springer (2017)

40. Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition. Journal of Artif. Intell. Res. **60**, 41–95 (2017)
41. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: AAAI. pp. 448–453. AAAI Press (2008)
42. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4), 365–386 (1991)
43. Giunchiglia, E., Leone, N., Maratea, M.: On the relation among answer set solvers. Ann. Math. Artif. Intell. **53**(1-4), 169–204 (2008)
44. Giunchiglia, E., Maratea, M.: On the Relation Between Answer Set and SAT Procedures (or, Between cmodels and smodels). In: ICLP. LNCS, vol. 3668, pp. 37–51 (2005)
45. Koponen, L., Oikarinen, E., Janhunen, T., Säilä, L.: Optimizing phylogenetic supertrees using answer set programming. TPLP **15**(4-5), 604–619 (2015)
46. Lierler, Y., Maratea, M., Ricca, F.: Systems, engineering environments, and competitions. AI Magazine **37**(3), 45–52 (2016)
47. Manna, M., Ricca, F., Terracina, G.: Taming primary key violations to query large inconsistent data via ASP. TPLP **15**(4-5), 696–710 (2015)
48. Maratea, M., Pulina, L., Ricca, F.: A multi-engine approach to answer-set programming. TPLP **14**(6), 841–868 (2014)
49. Maratea, M., Ricca, F., Faber, W., Leone, N.: Look-back techniques and heuristics in DLV: implementation, evaluation, and comparison to QBF solvers. J. Algorithms **63**(1-3), 70–89 (2008)
50. Marileo, M.C., Bertossi, L.E.: The consistency extractor system: Answer set programs for consistent query answering in databases. Data Knowl. Eng. **69**(6), 545–572 (2010)
51. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: On debugging non-ground answer-set programs. TPLP **10**(4-6), 513–529 (2010)
52. Polleres, A., Frühstück, M., Schenner, G., Friedrich, G.: Debugging non-ground ASP programs with choice rules, cardinality and weight constraints. In: LPNMR. Lecture Notes in Computer Science, vol. 8148, pp. 452–464. Springer (2013)
53. Shchekotykhin, K.M.: Interactive query-based debugging of ASP programs. In: AAAI. pp. 1597–1603. AAAI Press (2015)
54. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: NMR. pp. 77–84 (2006)