Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Interactive Query-Based Debugging of ASP Programs

Kostyantyn Shchekotykhin
Alpen-Adria University, Klagenfurt, Austria
kostyantyn.shchekotykhin@aau.at

Abstract

Broad application of answer set programming (ASP) for declarative problem solving requires the development of tools supporting the coding process. Program debugging is one of the crucial activities within this process. Modern ASP debugging approaches allow efficient computation of possible explanations of a fault. However, even for a small program a debugger might return a large number of possible explanations and selection of the correct one must be done manually. In this paper we present an interactive query-based ASP debugging method which extends previous approaches and finds the preferred explanation by means of observations. The system automatically generates a sequence of queries to a programmer asking whether a set of ground atoms must be true in all (cautiously) or some (bravely) answer sets of the program. Since some queries can be more informative than the others, we discuss query selection strategies which – given user's preferences for an explanation – can find the most informative query reducing the overall number of queries required for the identification of a preferred explanation.

Introduction

Answer set programming is a logic programming paradigm (Baral 2003; Brewka, Eiter, and Truszczynski 2011; Gebser et al. 2012) for declarative problem solving that has become popular during the last decades. The success of ASP is based on its fully declarative semantics (Gelfond and Lifschitz 1991) and availability of efficient solvers, e.g. (Simons, Niemelä, and Soininen 2002; Leone et al. 2006; Gebser et al. 2011). Despite a vast body of the theoretical research on foundations of ASP only recently the attention was drawn to the development of methods and tools supporting ASP programmers. The research in this direction focuses on a number of topics including integrated development environments (Febbraro, Reale, and Ricca 2011; Oetsch, Pührer, and Tompits 2011b; Sureshkumar et al. 2007), visualization (Mikitiuk, Moseley, and Truszczynski 2007; Cliffe et al. 2008), modeling techniques (Oetsch et al. 2011) and, last but not least, debugging of ASP programs (Syrjänen 2006; Brain et al. 2007; Gebser et al. 2008; Pontelli, Son, and El-Khatib 2009; Oetsch, Pührer, and Tompits 2010; Janhunen et al. 2010; Oetsch, Pührer, and Tompits 2011a; Polleres et al. 2013).

Modern ASP debugging approaches apply ASP itself to debug ASP programs. The idea is to transform a faulty program into the special debugging program whose answer sets explain possible causes of a fault. These explanations are given as a set of meta-atoms $\mathcal{D}$, called *diagnosis*, explaining a discrepancy between the set of actual and expected answer sets. In practice considering all possible diagnoses might be inefficient. Therefore, irrelevant diagnoses can be filtered out by specific integrity constraints – *debugging queries*.

However, in real-world scenarios it might be problematic for a programmer to provide a complete debugging query. Namely, in many cases a programmer can easily specify some small number of atoms that must be true in a desired answer set, but not a complete answer set. In this case the debugging system might return many alternative diagnoses. Another diagnosis selection issue is due to inability of a programmer to foresee all consequences of a diagnosis. I.e., in some cases multiple interpretations might have the same explanation for not being answer sets. In this case the modification of a program accordingly to a selected diagnosis might have side-effects in terms of unwanted answer sets. These two problem are addressed by our approach which helps a user to identify the *target diagnosis* $\mathcal{D}_t$. The latter is the preferred explanation for a given set of atoms not being true in an answer set, on the one hand, and is not an explanation for unwanted interpretations, on the other.

In this paper we present an *interactive query-based debugging* method for ASP programs which differentiates between the diagnoses by means of additional observations (de Kleer and Williams 1987; Shchekotykhin et al. 2012). The latter are acquired by automatically generating a sequence of *queries* to an oracle such as a user, a database, etc. Each answer is used to reduce the set of diagnoses until the target diagnosis is found. In order to construct queries our method uses the fact that in most of the cases different diagnoses explain why different sets of interpretations are not answer sets. Consequently, we can differentiate between diagnoses by asking an oracle whether a set of atoms must be true or not in all/some interpretations relevant to the target diagnosis. Each set of atoms which can be used as a query is generated by the debugger automatically using discrepancies in

the sets of interpretations associated with each diagnosis. Note that in the earlier approaches the user has to find all relevant discrepancies and extend a debugging query manually by calling a debugger multiple times and analyzing (parts of) its output.

Given a set of queries our method selects the best one and provides it to an oracle. In this paper we present two query selection strategies which are often used in active learning, namely, myopic and one step look-ahead (Settles 2012). The first strategy implements a greedy "split-in-half" approach that prefers queries allowing to reduce a set of diagnoses by half. The second one uses beliefs/preferences of a user for a cause/explanation of an error expressed in terms of probability to select those queries whose answers provide the most information gain. Given an answer the strategy updates the underlying probabilities, thus, adapting its behavior.

The method presented in this paper suggests an extension of the current debugging techniques by an effective user involvement in the debugging process. To the best of our knowledge there are no approaches to interactive query-based ASP debugging allowing automatic generation and selection of queries.

Preliminaries

Answer set programming A *disjunctive logic program* (DLP) Π is a finite set of rules of the form

$$h_1 \vee \cdots \vee h_l \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$$

where all h_i and b_j are *atoms* and $0 \leq l, m, n$. A *literal* is an atom b or its negation $not\ b$. Each *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are terms. A *term* is either a variable or a constant. A literal, a rule or a program is called *ground*, if they are variable-free. We denote the ground instantiation of a program Π by $Gr(\Pi)$ and by $At(\Pi)$ the set of all ground atoms appearing in $Gr(\Pi)$. The set of atoms $H(r) = \{h_1, \ldots, h_l\}$ is called the *head* of the rule r, whereas the set $B(r) = \{b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n\}$ is the body of r. In addition, it is useful to differentiate between the sets $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ comprising *positive* and *negative* body atoms. A rule $c \in \Pi$ with $H(c) = \emptyset$ is an *integrity constraint* and a rule $f \in \Pi$ with $B(f) = \emptyset$ is a *fact*.

An *interpretation* I for Π is a set of ground atoms $I \subseteq At(\Pi)$. A rule $r \in Gr(\Pi)$ is *applicable* under I, if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$, otherwise the rule is *blocked*. We say that r is *unsatisfied* by I, if it is applicable under I and $H(r) \cap I = \emptyset$; otherwise r is *satisfied*. An interpretation I is a *model* of Π, if it satisfies every rule $r \in Gr(\Pi)$. For a ground program $Gr(\Pi)$ and an interpretation I the Gelfond-Lifschitz reduct is defined as $\Pi^I = \{H(r) \leftarrow B^+(r) | r \in Gr(\Pi), I \cap B^-(r) = \emptyset\}$. I is an *answer set* of Π, if I is a minimal model of Π^I (Gelfond and Lifschitz 1991). The program Π is *inconsistent*, if the set of all answer sets $AS(\Pi) = \emptyset$.

ASP debugging Our approach uses a meta-programming technique applied in (Gebser et al. 2008; Oetsch, Pührer, and Tompits 2010) as a "black-box". In this paper we selected the first approach as application scenario, but the same ideas can also be applied to the latter one.

Given an input program Π a debugger generates a *meta-program* $\Delta[\Pi]$ s.t. every answer set of $\Delta[\Pi]$ comprises a set of meta-atoms explaining why an interpretation I is not an answer of Π. The debugger provides explanations of four error types: (1) $unsatisfied(id_r)$ – a rule r is *unsatisfied* by I; (2) $violated(id_r)$ – an integrity constraint r is *applicable* under I; (3) $unsupported(id_a)$ – an atom $a \in At(\Pi)$ is each *unsupported*; and (4) $ufLoop(id_a)$ – an atom $a \in At(\Pi)$ belongs to an *unfounded loop* L. The terms id_r and id_a are unique identifiers of a rule $r \in \Pi$ and an atom $a \in At(\Pi)$, resp. The set $Er(\Delta[\Pi]) \subseteq At(\Delta[\Pi])$ comprises all ground atoms over error-indicating predicates of $\Delta[\Pi]$.

There are seven static modules in the meta-program $\Delta[\Pi]$, see (Gebser et al. 2008). The input module π_{in} comprises two sets of facts about atoms $\{atom(id_a) \leftarrow | a \in At(\Pi)\}$ and rules $\{rule(id_r) \leftarrow | r \in \Pi\}$ of the program Π. Module π_{int} generates an arbitrary interpretation I of a program Π:

$$int(A) \leftarrow atom(A), not\ \overline{int}(A)$$
$$\overline{int}(A) \leftarrow atom(A), not\ int(A)$$

where atom $int(A)$ is complimentary to the atom $\overline{int}(A)$, i.e., no answer set can comprise both atoms. Other modules are responsible for generation of at least one of the four explanations why I is not an answer set of Π listed above.

Fault localization in ASP programs

In this section we show how additional information specified in a *background theory* $\mathcal{B}$ as well as *positive* P and *negative* N test cases can be used to keep the debugger focused only on relevant interpretations and diagnoses.

Our idea of background knowledge is similar to (Brain et al. 2007) and suggests that some set of rules $\mathcal{B} \subseteq \Pi$ must be considered as correct by the debugger. In the meta-programming method the background theory can be accounted by addition of integrity constraints which prune all answer sets of $\Delta[\Pi]$ suggesting that $r \in \mathcal{B}$ is faulty.

Definition 1 *Let $\Delta[\Pi]$ be a meta-program and $\mathcal{B} \subseteq \Pi$ a set of rules considered as correct. Then, a* debugging program *$\Delta[\Pi, \mathcal{B}]$ is defined as an extension of $\Delta[\Pi]$ with the rules:*

$$\{ \leftarrow\ rule(id_r),\ violated(id_r),$$
$$\leftarrow\ rule(id_r),\ unsatisfied(id_r) \mid r \in \mathcal{B}\}$$

In addition to background knowledge, further restrictions on the set of possible explanations of a fault can be made by means of test cases.

Definition 2 *Let $\Delta[\Pi, \mathcal{B}]$ be a debugging program. A* test case *for $\Delta[\Pi, \mathcal{B}]$ is a set $A \subseteq At(\Delta[\Pi, \mathcal{B}])$ of ground atoms over $int/1$ and $\overline{int}/1$ predicates.*

The test cases are either specified by a user before a debugging session or acquired by a system automatically as we show in subsequent sections.

Definition 3 *Let $\Delta[\Pi, \mathcal{B}]$ be a debugging program and $\mathcal{D} \subseteq Er(\Delta[\Pi, \mathcal{B}])$ a set of atoms over error-indicating predicates. Then a* diagnosis program *for $\mathcal{D}$ is defined as follows:*

$$\Delta[\Pi, \mathcal{B}, \mathcal{D}] := \Delta[\Pi, \mathcal{B}] \cup \{\leftarrow d_i \mid d_i \in Er(\Delta[\Pi, \mathcal{B}]) \setminus \mathcal{D}\}$$

Our approach allows four types of test cases corresponding to two ASP reasoning tasks (Leone et al. 2006):

- *Cautious reasoning*: all atoms $a \in A$ are true in *all* answer sets of the diagnosis program. Cautiously true test cases are stored in the set CT^+ whereas cautiously false in the set CT^-.
- *Brave reasoning*: all atoms $a \in A$ are true in *some* answer set of the diagnosis program. The set BT^+ comprises all bravely true test cases and the set BT^- all bravely false test cases.

In the meta-programming approach we handle the test cases as follows: Let $\mathcal{I}$ be a set of ground atoms resulting from the projection of an answer set $as \in AS(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ to the predicates $int/1$ and $\overline{int}/1$. Thus, each set $\mathcal{I}$ is a meta representation of an interpretation I of the program Π which is not an answer set of Π as explained by $\mathcal{D}$. By $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ we denote the set comprising all sets $\mathcal{I}_i$ for all $as_i \in AS(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$. Given a set of ground atoms A, we say that $\mathcal{I}$ satisfies A (denoted $\mathcal{I} \models A$), if $A \subseteq \mathcal{I}$. $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ satisfies A (denoted $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models A$), if $\mathcal{I} \models A$ for every $\mathcal{I} \in Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$. Analogously, we say that a set $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ is consistent with A, if there exists $\mathcal{I} \in Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ which satisfies A.

Let A be a test case, then $\overline{A}$ denotes a complementary test case, i.e., $\overline{A} = \{\overline{int}(a) \mid int(a) \in A\} \cup \{int(a) \mid \overline{int}(a) \in A\}$. For the verification whether a diagnosis program $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$ fulfills all test cases it is sufficient to check if the following conditions hold:

- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models ct^+ \quad \forall ct^+ \in CT^+$
- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models \overline{bt^-} \quad \forall bt^- \in BT^-$
- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \cup \overline{ct^-}$ is consistent $\quad \forall ct^- \in CT^-$
- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \cup bt^+$ is consistent $\quad \forall bt^+ \in BT^+$

As we can see, a diagnosis program has the same verification procedure w.r.t. both cautiously true CT^+ and bravely false BT^- test cases. The same holds for the cautiously false CT^- and bravely true BT^+ test cases. Therefore, in the following we can consider only the set P of *positive* and the set N of *negative* test cases that are defined as:

$$P := CT^+ \cup \left\{\overline{bt^-} \mid bt^- \in BT^-\right\}$$
$$N := BT^+ \cup \left\{\overline{ct^-} \mid ct^- \in CT^-\right\}$$

Definition 4 *Let $\Delta[\Pi, \mathcal{B}]$ be a debugging program, P be a set of positive test cases, N be a set of negative test cases and $Er(\Delta[\Pi, \mathcal{B}])$ denote a set of all ground atoms over error-indicating predicates of $\Delta[\Pi, \mathcal{B}]$. A* diagnosis problem *is to find such set of atoms $\mathcal{D} \subseteq Er(\Delta[\Pi, \mathcal{B}])$, called* diagnosis*, s.t. the following requirements hold:*

- *the diagnosis program $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$ is consistent*
- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models p, \quad \forall p \in P$
- $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ *is consistent with* $n, \quad \forall n \in N$.

A tuple $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ is a diagnosis problem instance *(DPI).*

In the following we assume that the background theory $\mathcal{B}$ together with the sets of test cases P and N always allow computation of the target diagnosis $\mathcal{D}_t$. I.e., a user does not insert faulty rules into the background knowledge or defines positive/negative test cases that interfere with each other.

Proposition 1 *A diagnosis $\mathcal{D}$ for a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ does not exists if either (i) $\Delta' := \Delta[\Pi, \mathcal{B}] \cup \{a_i \leftarrow \mid a_i \in p, \forall p \in P\}$ is inconsistent or (ii) $\exists n \in N$ s.t. the program $\Delta' \cup \{a_i \leftarrow \mid a_i \in n\}$ is inconsistent.*

Verification whether a set of atoms over error-indicating predicates is a diagnosis w.r.t. Definition 4 can be done according to the following proposition.

Proposition 2 *Let $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ be a DPI. Then, a set of atoms $\mathcal{D} \subseteq Er(\Delta[\Pi, \mathcal{B}])$ is a diagnosis for $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ iff $\Delta' := \Delta[\Pi, \mathcal{B}, \mathcal{D}] \cup \bigcup_{p \in P} \{a_i \leftarrow \mid a_i \in p\}$ is consistent and $\forall n \in N \ : \ \Delta' \cup \{a_i \leftarrow \mid a_i \in n\}$ is consistent.*

Definition 5 *A diagnosis $\mathcal{D}$ for a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ is* minimal *iff there is no diagnosis $\mathcal{D}'$ s.t. $|\mathcal{D}'| < |\mathcal{D}|$.*

In our approach we consider only minimal diagnoses of a DPI since they might require less changes to the program than non-minimal ones and, thus, are usually preferred by users. However, this does not mean that our debugging approach is limited to minimal diagnoses of an initial DPI. As we will show in the subsequent sections the interactive debugger acquires test cases and updates the DPI automatically s.t. all possible diagnoses of the initial DPI are investigated. Computation of minimal diagnoses can be done by extension of the debugging program with such optimization criteria that only answer sets including minimal number of atoms over error-indicating predicates are returned by a solver. Also, in practice a set of all minimal diagnoses is often approximated by a set of n diagnoses in order to improve the response time of a debugging system.

Computation of n diagnoses for the debugging program $\Delta[\Pi, \mathcal{B}]$ of a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ is done as shown in Algorithm 1. The algorithm calls an ASP solver to compute one answer set as of the debugging program (line 3). In case $\Delta[\Pi, \mathcal{B}]$ has an answer set the algorithm obtains a set $\mathcal{D}$ (line 5) and generates a diagnosis program $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$ (line 6). The latter, together with the sets of positive and negative test cases is used to verify whether $\mathcal{D}$ is a diagnosis or

Algorithm 1: COMPUTEDIAGNOSES($\langle \Delta[\Pi, \mathcal{B}], P, N \rangle, n$)

Input: DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$, maximum number of minimal diagnoses n
Output: a set of diagnoses $\mathbf{D}$

```
1 D ← ∅;
2 while |D| < n do
3     as ← GETANSWERSET(Δ[Π, B]);
4     if as = ∅ then exit loop;
5     D ← as ∩ Er(Δ[Π, B]));
6     Δ[Π, B, D] ← DIAGNOSISPROGRAM(Δ[Π, B], D);
7     if VERIFY(Δ[Π, B, D], P, N) then D ← D ∪ {D};
8     Δ[Π, B] ← EXCLUDE(Δ[Π, B], D);
9 return D;
```

not (line 7). All diagnoses are stored in the set $\mathbf{D}$. In order to exclude the answer set as from $AS(\Delta[\Pi, \mathcal{B}])$ the algorithm calls the EXCLUDE function (line 8) which extends the debugging program with the following integrity constraint, where atoms $d_1, \ldots, d_n \in \mathcal{D}$ and $d_{n+1}, \ldots, d_m \in Er(\Delta[\Pi, \mathcal{B}]) \setminus \mathcal{D}$:

$$\leftarrow d_1, \ldots, d_n, not\ d_{n+1}, \ldots, not\ d_m$$

Note, similarly to the model-based diagnosis (Reiter 1987; de Kleer and Williams 1987) we assume that each error-indicating atom $er \in \mathcal{D}$ is relevant to an explanation of a fault, whereas all other atoms $Er(\Delta[\Pi]) \setminus \mathcal{D}$ are not.

Example Let us exemplify our debugging approach on the following program Π_e:

$$\mathrm{r}_1 : a \leftarrow not\ d \qquad \mathrm{r}_2 : b \leftarrow a \qquad \mathrm{r}_3 : c \leftarrow b$$
$$\mathrm{r}_4 : d \leftarrow c \qquad \mathrm{r}_5 :\leftarrow d$$

Assume also that the *background theory* $\mathcal{B} = \{\leftarrow d\}$ and, therefore, the debugging program $\Delta[\Pi_e, \mathcal{B}]$ comprises two integrity constraints:

$$\leftarrow\ rule(id_{r_5}),\ violated(id_{r_5})$$
$$\leftarrow\ rule(id_{r_5}),\ unsatisfied(id_{r_5})$$

Since the program Π_e is inconsistent, a user runs the debugger to clarify the reason. In fact, the inconsistency is caused by an odd loop. I.e., if d is set to false, then the body of the rule r_1 is satisfied and a is derived. However, given a and the remaining rules d must be set to true. In case when d is true, a is not derived and, consequently, there is no justification for d. The debugging program $\Delta[\Pi_e, \mathcal{B}]$ of a $DPI_1 := \langle \Delta[\Pi_e, \mathcal{B}], \emptyset, \emptyset \rangle$ has 16 answer sets. The addition of optimization criteria allows to reduce the number of answer sets to 4 comprising only the minimal number of atoms over the error-indicating predicates. Since both sets of test cases are empty, a projection of these answer sets to the error-indicating predicates results in the diagnoses:

$$\mathcal{D}_1 : \{unsatisfied(id_{\mathrm{r}_1})\} \quad \mathcal{D}_2 : \{unsatisfied(id_{\mathrm{r}_2})\}$$
$$\mathcal{D}_3 : \{unsatisfied(id_{\mathrm{r}_3})\} \quad \mathcal{D}_4 : \{unsatisfied(id_{\mathrm{r}_4})\}$$

Definition 4 allows to identify the target (preferred) diagnosis $\mathcal{D}_t$ for the program Π_e by providing sufficient information in the sets $\mathcal{B}$, P and N. Assume that DPI_1 is updated with two test cases: one positive $\{int(a)\}$ and one negative $\{\overline{int}(b)\}$. The debugger generates $DPI_2 := \langle \Delta[\Pi_e, \mathcal{B}], \{\{int(a)\}\}, \{\{\overline{int}(b)\}\} \rangle$. These test cases require $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_t]) \models \{int(a)\}$ and $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_t])$ to be consistent with $\{\overline{int}(b)\}$ correspondingly. Given this information the debugger will return only one diagnosis in our example, namely $\mathcal{D}_2$, since $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_2]) \models \{int(a)\}$ and $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_2])$ is consistent with $\{\overline{int}(b)\}$. Indeed, a simple correction of Π_e by a user removing the rule r_2 results in a consistent program Π_2 s.t. all new answer sets of Π_2 fulfill all given test cases. The other sets of atoms $\mathcal{D}_1, \mathcal{D}_3, \mathcal{D}_4$ are not diagnoses of DPI_2 because they violate the requirements. Thus, $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_1]) \not\models \{int(a)\}$ and $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_i])$ is not consistent with $\{\overline{int}(b)\}$ for $\mathcal{D}_i \in \{\mathcal{D}_3, \mathcal{D}_4\}$. Consequently, $\mathcal{D}_2$ is the only possible diagnosis and it is accepted by a user as the target diagnosis $\mathcal{D}_t$.

Diagnosis	Interpretations
$\mathcal{D}_1 : unsatisfied(id_{r_1})$	$\{\{\overline{int}(a), \overline{int}(b), \overline{int}(c), \overline{int}(d)\}\}$
$\mathcal{D}_2 : unsatisfied(id_{r_2})$	$\{\{int(a), \overline{int}(b), \overline{int}(c), \overline{int}(d)\}\}$
$\mathcal{D}_3 : unsatisfied(id_{r_3})$	$\{\{int(a), int(b), \overline{int}(c), \overline{int}(d)\}\}$
$\mathcal{D}_4 : unsatisfied(id_{r_4})$	$\{\{int(a), int(b), int(c), \overline{int}(d)\}\}$

Table 1: Interpretations $Int(\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_i])$ for each of the diagnoses $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_4\}$.

Query-based diagnosis discrimination

The debugging system might generate a set of diagnoses for a given DPI. In our example, the debugger returns four minimal diagnoses $\{\mathcal{D}_1, \ldots, \mathcal{D}_4\}$ for DPI_1. As it is shown in the previous section, additional information, provided in the background theory and test cases of a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ can be used by the debugging system to reduce the set of diagnoses. However, in a general case the user does not know which sets of test cases should be provided to the debugger s.t. the target diagnosis can be identified. I.e., in many cases it might be difficult to provide a complete specification of a debugging query localizing a fault[1]. Therefore, the debugging method should be able to: (a) find an appropriate set of atoms $A \subseteq At(\Pi)$; and (b) query the user or some other oracle, whether the atoms A are cautiously/bravely true/false in the interpretations associated with the target diagnosis. To generate a query for a set of diagnoses $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ the debugging system can use the diagnosis programs $\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]$, where $\mathcal{D}_i \in \mathbf{D}$.

Since often different diagnoses explain why different sets of interpretations of a program Π are not answer sets of Π, we can use discrepancies between the sets of interpretations to discriminate between corresponding diagnoses. In our example, for each diagnosis program $\Delta[\Pi_e, \mathcal{B}, \mathcal{D}_i]$ an ASP solver returns a set of answer sets encoding an interpretation which is not an answer set of Π_e and a diagnosis, see Table 1. Without any additional information the debugger cannot decide which of these atoms must be true in the missing answer sets of Π_e. To get this information the debugging algorithm should be able to access some oracle which can answer a number of queries.

Definition 6 *Let $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ be a DPI, then a* query *is set of ground atoms $Q \subseteq At(\Pi)$.*

Each answer of an oracle provides additional information. Therefore, we update the actual DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ to $\langle \Delta[\Pi, \mathcal{B}], P', N' \rangle$ as follows: if an oracle answers

- *cautiously true*, then $P' \leftarrow P \cup \{int(a) \mid a \in Q\}$;
- *cautiously false*, then $N' \leftarrow N \cup \{\overline{int}(a) \mid a \in Q\}$;
- *bravely true*, then $N' \leftarrow N \cup \{int(a) \mid a \in Q\}$;
- *bravely false*, then $P' \leftarrow P \cup \{\overline{int}(a) \mid a \in Q\}$.

The goal of asking a query is to obtain new information characterizing the target diagnosis. For instance, the debugger asks a user about classification of the set of atoms

[1] A recent user study https://code.google.com/p/rmbd/wiki/UserStudy confirms this observation for the ontology debugging.

$\{c\}$. If the answer is *cautiously true*, the new $DPI_3 = \langle \Delta[\Pi_e, \mathcal{B}], \{\{int(c)\}\}, \emptyset \rangle$ has only one diagnosis $\mathcal{D}_4$ which is the target diagnosis w.r.t. a user answer. All other minimal sets of atoms over error-indicating predicates are not diagnoses because they do not fulfill the necessary requirements of Definition 4. If the answer is *bravely false*, then the set $\{\overline{int}(c)\}$ is added to P and $\mathcal{D}_4$ is rejected. Consequently, we have to ask an oracle another question in order to discriminate between the remaining diagnoses. Since there are many subsets of $At(\Pi)$ which can be queried, the debugger has to generate and ask only those queries which allow to discriminate between the diagnoses of the current DPI.

Definition 7 *Each diagnosis $\mathcal{D}_i \in \mathbf{D}$ for a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ can be assigned to one of the three sets $\mathbf{D^P}$, $\mathbf{D^N}$ or $\mathbf{D}^\emptyset$ depending on the query Q where:*

- $\mathcal{D}_i \in \mathbf{D^P}$ *if* $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]) \models \{int(a) \mid a \in Q\}$
- $\mathcal{D}_i \in \mathbf{D^N}$ *if* $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]) \models \{\overline{int}(a) \mid a \in Q\}$
- $\mathcal{D}_i \in \mathbf{D}^\emptyset$ *if* $\mathcal{D}_i \notin (\mathbf{D^P} \cup \mathbf{D^N})$

A partition *of the set of diagnoses $\mathbf{D}$ w.r.t. a query Q is denoted by a tuple $\langle Q, \mathbf{D_i^P}, \mathbf{D_i^N}, \mathbf{D_i^\emptyset} \rangle$.*

Given a DPI we say that the diagnoses in $\mathbf{D^P}$ predict a positive answer (*yes*) as a result of the query Q, diagnoses in $\mathbf{D^N}$ predict a negative answer (*no*), and diagnoses in $\mathbf{D}^\emptyset$ do not make any predictions. Note, the answer *yes* corresponds to classification of the query into the set of positive test cases P, whereas the answer *no* is a result of a classification of the query into the set of negative test cases N. Therefore, without limiting the generality, in the following we consider only these two answers.

The notion of a partition has an important property. Namely, each partition $\langle Q, \mathbf{D_i^P}, \mathbf{D_i^N}, \mathbf{D_i^\emptyset} \rangle$ indicates the changes in the set of diagnoses after the sets of test cases of an actual DPI are updated w.r.t. the answer of an oracle.

Property 1 *Let $\mathbf{D}$ be a set of diagnoses for a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$, Q be a query, $\langle Q, \mathbf{D_i^P}, \mathbf{D_i^N}, \mathbf{D_i^\emptyset} \rangle$ be a partition of $\mathbf{D}$ w.r.t. Q and $v \in \{yes, no\}$ be an answer of an oracle to a query Q.*

- *if $v = yes$, then the set of diagnoses $\mathbf{D}'$ for the updated DPI $\langle \Delta[\Pi, \mathcal{B}], P', N \rangle$ does not comprise any elements of $\mathbf{D^N}$, i.e., $\mathbf{D}' \cap \mathbf{D^N} = \emptyset$ and $(\mathbf{D^P} \cup \mathbf{D}^\emptyset) \subseteq \mathbf{D}'$.*
- *if $v = no$, then for the set of diagnoses $\mathbf{D}'$ of the updated DPI $\langle \Delta[\Pi, \mathcal{B}], P, N' \rangle$ it holds that $\mathbf{D}' \cap \mathbf{D^P} = \emptyset$ and $(\mathbf{D^N} \cup \mathbf{D}^\emptyset) \subseteq \mathbf{D}'$.*

Consequently, depending on the answer of an oracle to a query Q, the set of diagnoses of an updated DPI comprises either $\mathbf{D^P} \cup \mathbf{D}^\emptyset$ or $\mathbf{D^N} \cup \mathbf{D}^\emptyset$.

In order to generate queries, we have to investigate for which sets $\mathbf{D^P}, \mathbf{D^N} \subseteq \mathbf{D}$ a query exists that can be used to differentiate between them. A straight forward approach to query generation is to generate and verify all possible subsets of $\mathbf{D}$. This is feasible if we limit the number n of minimal diagnoses to be considered during the query generation. E.g., given $n = 9$ the algorithm has to verify 512 partitions in the worst case. In general, the number of diagnoses n must be selected by a user. The larger is the value of n the more time it takes to find a query, but an answer to this query will provide more information to a debugger.

Algorithm 2: FINDPARTITIONS($\langle \Delta[\Pi, \mathcal{B}], P, N \rangle, \mathbf{D}$)

Input: DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$, a set of diagnoses $\mathbf{D}$
Output: a set of partitions $\mathbf{PR}$

1 $\mathbf{PR} \leftarrow \emptyset$;
2 **foreach** $\mathbf{D_i^P} \in \mathcal{P}(\mathbf{D})$ **do**
3 $E_i \leftarrow$ COMMONATOMS($\mathbf{D_i^P}$);
4 $Q_i \leftarrow \{a \mid int(a) \in E_i\}$;
5 **if** $Q_i \neq \emptyset$ **then**
6 $\left\langle Q_i, \mathbf{D_i^P}, \mathbf{D_i^N}, \mathbf{D_i^\emptyset} \right\rangle \leftarrow$ GENPART($Q_i, \mathbf{D}, \mathbf{D_i^P}$);
7 **if** $\mathbf{D_i^N} \neq \emptyset$ **then**
 $\mathbf{PR} \leftarrow \mathbf{PR} \cup \{\left\langle Q_i, \mathbf{D_i^P}, \mathbf{D_i^N}, \mathbf{D_i^\emptyset} \right\rangle\}$;
8 **return** $\mathbf{PR}$;

Given a set of diagnoses $\mathbf{D}$ for a DPI $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ Algorithm 2 computes a set of partitions $\mathbf{PR}$ comprising all queries discriminating between the diagnoses in $\mathbf{D}$. For each element $\mathbf{D_i^P}$ of the power set $\mathcal{P}(\mathbf{D})$, the algorithm checks whether there is a set of atoms common to all interpretations of all diagnoses in $\mathbf{D_i^P}$. The function COMMONATOMS (line 3) returns an intersection of all sets $\mathcal{I} \in Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_j])$ for all $\mathcal{D}_j \in \mathbf{D_i^P}$. Given a non-empty query, the function GENPART (line 6) uses Definition 7 to obtain a partition by classifying each diagnosis $\mathcal{D}_k \in \mathbf{D} \setminus \mathbf{D_i^P}$ into one of the sets $\mathbf{D_i^P}$, $\mathbf{D_i^N}$ or $\mathbf{D_i^\emptyset}$. Finally, all partitions allowing to discriminate between the diagnoses, i.e., comprising non-empty sets $\mathbf{D_i^P}$ and $\mathbf{D_i^N}$, are added to the set $\mathbf{PR}$.

Example (cont.) Reconsider the set of diagnoses $\mathbf{D} = \{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}$ for the DPI $\langle \Delta[\Pi_e, \{\leftarrow d\}], \emptyset, \emptyset \rangle$. The power set $\mathcal{P}(\mathbf{D}) = \{\{\mathcal{D}_1\}, \{\mathcal{D}_2\}, \ldots, \{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}\}$ comprises 15 elements. We exclude the element corresponding to $\emptyset$ since it does not allow to compute a query. In each iteration an element of $\mathcal{P}(\mathbf{D})$ is assigned to the set $\mathbf{D_i^P}$. E.g., if the algorithm assigned $\mathbf{D_0^P} = \{\mathcal{D}_1, \mathcal{D}_2\}$, then the set Q_0 is empty since the set $E_0 = \{\overline{int}(b), \overline{int}(c), \overline{int}(d)\}$ (see Table 1). Therefore, the set $\{\mathcal{D}_1, \mathcal{D}_2\}$ is rejected and removed from $\mathcal{P}(\mathbf{D})$. Assume that in the next iteration the algorithm selected $\mathbf{D_1^P} = \{\mathcal{D}_2, \mathcal{D}_3\}$, for which the set of common atoms $E_1 = \{int(a), \overline{int}(c), \overline{int}(d)\}$ and, thus, $Q_1 = \{a\}$. The remaining diagnoses $\mathcal{D}_1$ and $\mathcal{D}_4$ are classified according to Definition 7. I.e., the algorithm selects the first diagnosis $\mathcal{D}_1$ and verifies whether $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_1]) \models \{int(a)\}$. Given the negative answer, the algorithm checks if $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_1]) \models \{\overline{int}(a)\}$. Since the condition is satisfied the diagnosis $\mathcal{D}_1$ is added to the set $\mathbf{D_1^N}$. The second diagnosis $\mathcal{D}_4$ is added to the set $\mathbf{D_1^P}$ as it satisfies the first requirement $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}_4]) \models \{int(a)\}$. The resulting partition $\langle \{a\}, \{\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}, \{\mathcal{D}_1\}, \emptyset \rangle$ is added to the set $\mathbf{PR}$.

In general, Algorithm 2 returns a large number of possible partitions and the debugger has to select the best one. In this paper, we discuss two query selection strategies.

Myopic query strategies determine the best query using

only the set of partitions $\mathbf{PR}$. A popular "Split-in-half" strategy prefers those queries which allow to remove a half of the diagnoses from the set $\mathbf{D}$, regardless of the answer of an oracle. That is, "Split-in-half" selects a query

$$Q_s = \arg\min_{Q_i} \left| |\mathbf{D_i^P}| - |\mathbf{D_i^N}| \right| + |\mathbf{D_i^\emptyset}|$$

In our example, $\langle \{b\}, \{\mathcal{D}_3, \mathcal{D}_4\}, \{\mathcal{D}_1, \mathcal{D}_2\}, \emptyset \rangle$ is the preferred partition, since the set of all diagnoses of an updated DPI will comprise only two elements regardless of the answer of an oracle.

One step look-ahead strategies, such as prior entropy or information gain (Settles 2012), allow to find the target diagnosis using less queries by incorporating heuristics (user preferences). These heuristics assess the prior probability $p(\mathcal{D}_i)$ of each diagnosis $\mathcal{D}_i \in \mathbf{D}$ to be the target one (de Kleer and Williams 1987; Shchekotykhin et al. 2012). E.g., such heuristic can assign higher probabilities to diagnoses comprising atoms over $unsatisfiable/1$ predicate if a user expects this type of error. The widely used one step look-ahead strategy selects a query which, given the answer of an oracle, minimizes the expected entropy of the set of diagnoses. Let $p(Q_i = v)$ denote the probability that an oracle gives an answer $v \in \{yes, no\}$ to a query Q_i and $p(\mathcal{D}_j | Q_i = v)$ be the probability of diagnosis $\mathcal{D}_j$ given an oracle's answer. The expected entropy after querying Q_i is computed as (see (Shchekotykhin et al. 2012) for details):

$$H_e(Q_i) = \sum_{v \in \{yes,no\}} p(Q_i = v) \times \\ - \sum_{\mathcal{D}_j \in \mathbf{D}} p(\mathcal{D}_j | Q_i = v) \log_2 p(\mathcal{D}_j | Q_i = v)$$

After a query Q_s is selected by a strategy

$$Q_s = \arg\min_{Q_i} H_e(Q_i)$$

the system asks an oracle to provide its classification. Given the answer v of an oracle, i.e. $Q_s = v$, we have to update the probabilities of the diagnoses to take the new information into account. The update is performed by the Bayes rule

$$p(\mathcal{D}_j | Q_i = v) = \frac{p(Q_i = v | \mathcal{D}_j) p(\mathcal{D}_j)}{p(Q_i = v)}$$

In order to reduce the number of queries a user can specify a threshold, e.g., $\sigma = 0.95$. If the difference in probabilities between two most probable diagnoses is greater than σ, the algorithm returns the most probable diagnosis.

The interactive debugging system (Algorithm 3) takes a ground program or a ground instantiation of a non-ground program as well as a query selection strategy as an input. Optionally, a user can provide background knowledge, relevant test cases as well as a set of heuristics assessing probabilities of diagnoses. If the first three sets are not specified, then the corresponding arguments are initialized with $\emptyset$. In case a user specified no heuristics, we add a simple function that assigns a small probability value to every diagnosis. The

Algorithm 3: INTERACTIVEDEBUG$(\Pi, S, \mathcal{B}, P, N, H, n, \sigma)$

Input: ground disjunctive program Π, query selection strategy S, background knowledge $\mathcal{B}$, sets of positive P and negative N test cases, set of heuristics H, maximum number minimal diagnoses n, acceptance threshold σ
Output: a diagnosis $\mathcal{D}$

```
⟨Δ[Π, B], P, N⟩ ← GENERATEDPI(Π, B); D ← ∅;
repeat
    D ← D ∪ COMPUTEDIAGS(⟨Δ[Π, B], P, N⟩, n − |D|);
    PR ← FINDPARTITIONS(⟨Δ[Π, B], P, N⟩, D);
    Q ← SELECTQUERY(PR, H, S);
    if Q = ∅ then exit loop;
    A ← GETANSWER(Q);
    ⟨Δ[Π, B], P, N⟩ ← UPDATEDPI(A, ⟨Δ[Π, B], P, N⟩);
    D ← UPDATEDIAGNOSES(A, Q, PR, H);
until ABOVETHRESHOLD(D, H, σ) ∨ |D| ≤ 1;
return MOSTPROBABLEDIAGNOSIS(D, S, H);
```

algorithm starts with the initialization of a DPI. The debugging program $\Delta[\Pi, \mathcal{B}]$ is generated by `spock`[2], which implements the meta-programming approach of (Gebser et al. 2008). First, the main loop of Algorithm 3 computes the required number of diagnoses s.t. $|\mathbf{D}| = n$. Next, we find a set of partitions for the given diagnoses and select a query according to a query strategy S chosen by a user. If the myopic strategy is used, then SELECTQUERY ignores probabilities of diagnoses. The oracle is asked to classify the query and its answer is used to update the DPI as well as the set $\mathbf{D}$. From the latter we remove all elements that are not diagnoses of the updated DPI. The main loop of the algorithm exits if either there is a diagnosis which probability satisfies the threshold σ or only one diagnosis remains. Finally, the most probable diagnosis (the first diagnosis in case of a myopic strategy) is returned to a user. Algorithm 3 was prototypically implemented as a part of a general diagnosis framework[3]. A plug-in for an IDE providing a user-friendly interface for the interactive debugging is in development.

Conclusions

In this paper we presented an approach to the interactive query-based debugging of disjunctive logic programs. The differentiation between the diagnoses is done by means of queries which are automatically generated from answer sets of the debugging meta-program. Each query partitions a set of diagnoses into subsets that make different predictions for an answer of an oracle. Depending on the availability of heuristics assessing the probability of a diagnosis to be the target one, the debugger can use different query selection strategies to find the most informative query allowing efficient identification of the target diagnosis.

Acknowledgments

We would like to thank Gerhard Friedrich and Patrick Rodler for the discussions regarding query selection strategies and anonymous reviewers for their helpful comments.

[2] www.kr.tuwien.ac.at/research/debug

[3] https://code.google.com/p/rmbd/wiki/AspDebugging

References

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

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*, 31–43.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Cliffe, O.; Vos, M.; Brain, M.; and Padget, J. 2008. Aspviz: Declarative visualisation and animation using answer set programming. In Garcia de la Banda, M., and Pontelli, E., eds., *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, 724–728. Springer Berlin Heidelberg.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

Febbraro, O.; Reale, K.; and Ricca, F. 2011. ASPIDE: Integrated development environment for answer set programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning*, 317–330. Springer.

Gebser, M.; Pührer, J.; Schaub, T.; and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of 23rd AAAI Conference on Artificial Intelligence (AAAI'08)*, 448–453.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications* 24(2):107–124.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publischers.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New generation computing* 9(3-4):365–386.

Janhunen, T.; Niemelä, I.; Oetsch, J.; Pührer, J.; and Tompits, H. 2010. On Testing Answer-Set Programs. In *19th European Conference on Artificial Intelligence (ECAI-2010)*, 951–956.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)* 7(3):499–562.

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*, 635–640.

Oetsch, J.; Pührer, J.; Seidl, M.; Tompits, H.; and Zwickl, P. 2011. VIDEAS : Supporting Answer-Set Program Development using Model-Driven Engineering Techniques. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning*, 382–387.

Oetsch, J.; Pührer, J.; and Tompits, H. 2010. Catching the Ouroboros: On Debugging Non-ground Answer-Set Programs. *Theory and Practice of Logic Programming* 10(4-6):2010.

Oetsch, J.; Pührer, J.; and Tompits, H. 2011a. Stepping through an Answer-Set Program. In *Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning*, volume 231875, 134–147.

Oetsch, J.; Pührer, J.; and Tompits, H. 2011b. The SeaLion has Landed: An IDE for Answer-Set Programming – Preliminary Report. *CoRR* abs/1109.3989.

Polleres, A.; Frühstück, M.; Schenner, G.; and Friedrich, G. 2013. Debugging Non-ground ASP Programs with Choice Rules, Cardinality and Weight Constraints. In Cabalar, P., and Son, T., eds., *Logic Programming and Nonmonotonic Reasoning*, volume 8148 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 452–464.

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(01):1.

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

Settles, B. 2012. *Active Learning*, volume 6 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publischers.

Shchekotykhin, K.; Friedrich, G.; Fleiss, P.; and Rodler, P. 2012. Interactive ontology debugging: Two query strategies for efficient fault localization. *Web Semantics: Science, Services and Agents on the World Wide Web* 12-13(0):88–103.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Sureshkumar, A.; Vos, M. D.; Brain, M.; and Fitch, J. 2007. APE: An AnsProlog* Environment. In *Software Engineering for Answer Set Programming*, 101–115.

Syrjänen, T. 2006. Debugging Inconsistent Answer Set Programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning*, 77–84.