

# Interactive Query-based Debugging of ASP Programs

Kostyantyn Shchekotykhin

Alpen-Adria Universität, Klagenfurt, 9020 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 a 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 best query. That is, the query an answer of which reduces the overall number of queries required for the identification of a preferred explanation.

## 1 Introduction

Answer set programming is a logic programming paradigm [1, 4, 9] for declarative problem solving that has become popular during the last decades. The success of ASP is based on its fully declarative semantics [11] and availability of efficient solvers, e.g. [27, 16, 8]. 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 [7, 21, 29], visualization [5], modeling techniques [19] and, last but not least, debugging of ASP programs.

Modern ASP debuggers are mostly based on declarative strategies. The suggested methods use elegant techniques applying ASP itself to debug ASP programs. The idea is to transform a faulty program in the special debugging program whose answer sets explain possible causes of a fault. These explanations are given by means of meta-atoms. A set of meta-atoms explaining a discrepancy between the set of actual and expected answer sets is called a *diagnosis*. In practice considering all possible diagnoses might be inefficient. Therefore, modern debugging approaches apply built-in minimization techniques of ASP solvers to compute only diagnoses comprising the minimal number of elements. In addition, the number of diagnoses can be reduced by so called *debugging queries*, i.e. sets of integrity constraints filtering out irrelevant diagnoses.

The computation of diagnoses is usually done by considering answer sets of a debugging program. In the approach of [28] a diagnosis corresponds to a set of meta-atoms

indicating that a rule is removed from a program. Brain et al. [3] use the tagging technique [6] to obtain more fine-grained diagnoses. The approach differentiates between three types of problems: unsatisfied rules, unsupported atoms and unfounded loops. Each type is denoted by a special meta-predicate. Extraction of diagnoses can be done by a projection of an answer set of a debugging program to the meta-predicates. More recent techniques [10, 20] apply *meta-programming*, where a program over a meta language is used to manipulate a program over an object language. Answer sets of a debugging meta-program include meta-atoms describing faults of the same nature as of [3].

The main problem of the aforementioned declarative approaches is that 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. Existing debugging systems [2, 10, 20] can be used in an “interactive” mode in which a user specifies only a partial debugging query as an input. Given a set of diagnoses computed by a debugger the user extends the debugging query, thus, filtering out irrelevant answer sets of a meta-program. However, this sort of interactivity still requires a user to select and provide atoms of the debugging query manually. A recent user study in the area of ontology debugging<sup>1</sup> shows that identification of such extensions using simple visualization techniques is a quite tedious process.

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. The simplest example is an integrity constraint which can be violated by multiple interpretations. 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 problems are addressed by our approach which helps a user to identify the *target diagnosis*. 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 [14, 26]. 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. Given a set of queries our method finds the best query according to a query selection strategy chosen by a user.

The suggested debugging approach can use a variety of query selection strategies. In this paper we discuss myopic and one step look-ahead strategies which are commonly

---

<sup>1</sup> Description of the user study and preliminary results can be found at <https://code.google.com/p/rmbd/wiki/UserStudy>

used in active learning [25]. A myopic strategy implements a kind of greedy approach which in our case prefers queries that allow to reduce a set of diagnoses by half, regardless of an oracle's answer. The one step look-ahead strategy uses beliefs/preferences of a user for a cause/explanation of an error represented in terms of probability. Such a strategy selects those queries whose answers provide the most information gain, i.e. in whose answers a strategy is most uncertain about. New information provided by each answer is taken into account using Bayes-update. This allows the strategy to adapt its behavior on the fly.

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

## 2 Preliminaries

A *disjunctive logic program* (DLP)  $\Pi$  is a finite set of rules of the form

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

where all  $h_i$  and  $b_j$  are *atoms* and  $0 \leq l, m, n$ . A *literal* is an atom  $b$  or its negation  $\text{not } b$ . Each *atom* is an expression of the form  $p(t_1, \dots, t_k)$ , where  $p$  is a predicate symbol and  $t_1, \dots, t_k$  are terms. A *term* is either a variable or a constant. The former is denoted by a string starting with an uppercase letter and the latter starting with a lowercase one. A literal, a rule or a program is called *ground*, if they are variable-free. A non-ground program  $\Pi$ , its rules and literals can be grounded by substitution of variables with constants appearing in  $\Pi$ . We denote the grounded instantiation of a program  $\Pi$  by  $Gr(\Pi)$  and by  $At(\Pi)$  the set of all ground atoms appearing in  $Gr(\Pi)$ .

The set of atoms  $H(r) = \{h_1, \dots, h_l\}$  is called the *head* of the rule  $r$ , whereas the set  $B(r) = \{b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n\}$  is the *body* of  $r$ . In addition, it is useful to differentiate between the sets  $B^+(r) = \{b_1, \dots, b_m\}$  and  $B^-(r) = \{b_{m+1}, \dots, b_n\}$  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*. A rule  $r$  is *normal*, if  $|H(r)| \leq 1$ . A *normal program* includes only normal rules.

An *interpretation*  $I$  for  $\Pi$  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  $\Pi$ , 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  $\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in Gr(\Pi), I \cap B^-(r) = \emptyset\}$ .  $I$  is an *answer set* of  $\Pi$ , if  $I$  is a minimal model of  $\Pi^I$  [11]. The program  $\Pi$  is *inconsistent*, if the set of all answer sets  $AS(\Pi) = \emptyset$ .

Lee [15] provides another characterization of answer sets of a program  $\Pi$  based on the notion of support. Thus, a rule  $r \in Gr(\Pi)$  is a *support* for  $A \subseteq At(\Pi)$  with respect to  $I$ , if a rule  $r$  is applicable under an interpretation  $I$ ,  $H(r) \cap A \neq \emptyset$  and  $H(r) \cap I \subseteq A$ . A support is *external*, if  $B^+(r) \cap A = \emptyset$ . A set of ground atoms  $A$  is *unsupported* by  $\Pi$  with respect to  $I$ , if no rule in  $Gr(\Pi)$  supports it. A *loop* is a non-empty set

$L \subseteq At(\Pi)$  such that for any two distinct atoms  $a_i, a_j \in L$  there is a path  $P$  in a positive dependency graph  $G = (At(\Pi), \{(h, b) | r \in Gr(\Pi), h \in H(r), b \in B^+(r)\})$ , where  $P \neq \emptyset$  and  $P \subseteq L$ . A loop  $L$  is *unfounded* by  $\Pi$  with respect to  $I$ , if no rule in  $Gr(\Pi)$  supports it externally; otherwise  $L$  is founded. An interpretation  $I$  is an answer set of  $\Pi$ , iff  $I$  is a model of  $\Pi$  such that each atom  $a \in I$  is supported and each loop  $L \subseteq I$  is founded [15].

### 3 Debugging of ASP programs

The approach presented in our paper is based on the meta-programming technique presented in [10]. This debugging method focuses on the identification of *semantical errors* in a disjunctive logic program, i.e. disagreements between the actual answer sets of a program and the expected ones. The main idea is to use a program over a meta-language that manipulates another program over an object-language. The latter is a ground disjunctive program  $\Pi$  and the former is a non-ground normal logic program  $\Delta[\Pi]$ . Each answer set of a meta-program  $\Delta[\Pi]$  comprises a set of atoms specifying an interpretation  $I$  and a number of meta-atoms showing, why  $I$  is not an answer set of a program  $\Pi$ . In addition, the method guarantees that there is at least one answer set of  $\Delta[\Pi]$  for each interpretation  $I$  which is not an answer set of  $\Pi$ .

The debugger provides explanations of four error types denoted by the corresponding *error-indicating* predicates:

1. *Unsatisfied rules*:  $I$  is not a classical model of  $Gr(\Pi)$  because the logical implication expressed by a rule  $r$  is false under  $I$ . Atom *unsatisfied*( $id_r$ ) in an answer set of  $\Delta[\Pi]$  expresses that a rule  $r$  is unsatisfied by  $I$ , where  $id_r$  is a unique identifier of a rule  $r \in \Pi$ .
2. *Violated integrity constraints*:  $I$  cannot be an answer set of  $Gr(\Pi)$ , if a constraint  $r$  is applicable under  $I$ . Atom *violated*( $id_r$ ) indicates that  $r$  is violated under  $I$ .
3. *Unsupported atoms*: there is no rule  $r \in Gr(\Pi)$  which allows derivation of  $\{a\} \subseteq I$  and, therefore,  $I$  is not a minimal model of  $\Pi^I$ . Each unsupported atom  $a$  is indicated by an atom *unsupported*( $id_a$ ) in an answer set of  $\Delta[\Pi]$ , where  $id_a$  is a unique identifier of an atom  $a \in At(\Pi)$ .
4. *Unfounded loops*:  $I$  is not a minimal model of  $\Pi^I$ , if a loop  $L \subseteq I$  is unfounded by a program  $\Pi$  with respect to  $I$ . An atom *ufLoop*( $id_a$ ) expresses that an atom  $a \in At(\Pi)$  belongs to the unfounded loop  $L$ .

The set  $Er(\Delta[\Pi]) \subseteq At(\Delta[\Pi])$  comprises all ground atoms over error-indicating predicates of the meta-program  $\Delta[\Pi]$ .

There are seven static modules in the meta-program  $\Delta[\Pi]$ , see [10]. The input module  $\pi_{in}$  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  $\Pi$ . Moreover, for each rule  $r \in \Pi$  the module  $\pi_{in}$  defines which atoms are in  $H(r)$ ,  $B^+(r)$  and  $B^-(r)$ . Module  $\pi_{int}$  generates an arbitrary interpretation  $I$  of a program  $\Pi$  as follows:

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

where atom  $int(A)$  is complimentary to the atom  $\overline{int}(A)$ , i.e. no answer set can comprise both atoms. The module  $\pi_{ap}$  checks for every rule, whether it is applicable or blocked under  $I$ . The modules  $\pi_{sat}$ ,  $\pi_{supp}$  and  $\pi_{uflowp}$  are responsible for the computation of at least one of the four explanations why  $I$  is not an answer set of  $\Pi$  listed above. Note,  $\pi_{uflowp}$  searches for unfounded loops only among atoms supported by  $\Pi$  with respect to  $I$ . This method ensures that each of the found loops is critical, i.e. it is a reason for  $I$  not being an answer set of  $\Pi$ . The last module,  $\pi_{noas}$  restricts the answer sets of  $\Delta[\Pi]$  only to those that comprise atoms over the error-indicating predicates.

The fault localization is done manually by means of *debugging queries* which specify an interpretation to be investigated as a set of atoms, e.g.  $I = \{a\}$ . Then  $I$  is transformed into a finite set of constraints, e.g.  $\{\leftarrow \overline{int}(id_a), \leftarrow int(id_b), \dots\}$ , pruning irrelevant answer sets of  $\Delta[\Pi]$ .

## 4 Fault localization in ASP programs

In our work we extend the meta-programming approach by allowing a user to specify *background theory*  $\mathcal{B}$  as well as *positive*  $P$  and *negative*  $N$  test cases. In this section we show how this additional information is used to keep the search focused only on relevant interpretations and diagnoses.

Our idea of background knowledge is similar to [3] 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 to  $\pi_{noas}$  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:

$$\begin{aligned} &\{ \leftarrow rule(id_r), violated(id_r), \\ &\quad \leftarrow rule(id_r), unsatisfied(id_r) \mid r \in \mathcal{B} \} \end{aligned}$$

In addition to background knowledge, further restrictions on the set of possible explanations of a fault can be made by means of test cases. A *test case* for the debugging program  $\Delta[\Pi, \mathcal{B}]$  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 Section 5.

**Definition 2.** 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:

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

In our approach we allow four types of test cases corresponding to the two ASP reasoning tasks [16]:

- *Cautious reasoning*: all atoms  $a \in A$  are true in *all* answer sets of the diagnosis program, resp.  $\Delta[\Pi, \mathcal{B}, \mathcal{D}_t] \models_c A$ , or not, resp.  $\Delta[\Pi, \mathcal{B}, \mathcal{D}_t] \not\models_c A$ . 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, resp.  $\Delta[\Pi, \mathcal{B}, \mathcal{D}_t] \models_b A$ , or not, resp.  $\Delta[\Pi, \mathcal{B}, \mathcal{D}_t] \not\models_b A$ . 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 a projection of an answer set  $as \in AS(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  to the predicates  $int/1$  and  $\overline{int}/1$ . By  $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  we denote a set comprising all sets  $\mathcal{I}_i$  for all  $as_i \in AS(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ . Each set of grounded atoms  $\mathcal{I}$  corresponds to an interpretation  $I$  of the program  $\Pi$  which is not an answer set of  $\Pi$  as explained by  $\mathcal{D}$ . The set  $Int(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  comprises a meta representation of each such interpretation for a diagnosis  $\mathcal{D}$ . Given a set of grounded 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. comprising atoms complimentary to atoms in  $A$ . For the verification whether a program  $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$  fulfills all test cases it is sufficient to check if the following conditions hold:

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

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

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

**Definition 3.** 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, such that 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. That is, a user provides reasonable background knowledge as well as positive and negative test cases that do not interfere with each other.

**Proposition 1.** Given a DPI  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$  a set of atoms  $\mathcal{D} \subseteq Er(\Delta[\Pi, \mathcal{B}])$  is a diagnosis 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.

*Proof.* (sketch) ( $\Rightarrow$ ) Let  $\mathcal{D}$  be a diagnosis for  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ . Since  $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$  is consistent and  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models p$  for all  $p \in P$  it follows that  $\Delta[\Pi, \mathcal{B}, \mathcal{D}] \cup \bigcup_{p \in P} \{a_i \leftarrow \mid a_i \in p\}$  is consistent. The latter program has answer sets because every  $p \in P$  is a subset of every  $\mathcal{I} \in \text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$ . In addition, since the set of meta-interpretations  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  is consistent with every  $n \in N$  there exists such set  $\mathcal{I} \in \text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  that  $n \subseteq \mathcal{I}$ . Therefore the program  $\Delta[\Pi, \mathcal{B}, \mathcal{D}] \cup \{a_i \leftarrow \mid a_i \in n\}$  has at least one answer set. Taking into account that  $\Delta'$  is consistent we can conclude that  $\Delta' \cup \{a_i \leftarrow \mid a_i \in n\}$  is consistent as well.

( $\Leftarrow$ ) Let  $\mathcal{D} \subseteq \text{Er}(\Delta[\Pi, \mathcal{B}])$  and  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$  be a DPI. Since  $\Delta'$  is consistent the diagnosis program  $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$  is also consistent. Moreover, for all  $p \in P$   $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}]) \models p$  because  $\{a_i \leftarrow \mid a_i \in p\} \subseteq \Delta'$ . Finally, for every  $n \in N$  consistency of  $\Delta' \cup \{a_i \leftarrow \mid a_i \in n\}$  implies that there must exist  $\mathcal{I} \in \text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}])$  satisfying  $n$ .  $\square$

**Definition 4.** A diagnosis  $\mathcal{D}$  for a DPI  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$  is a minimal diagnosis iff there is no diagnosis  $\mathcal{D}'$  such that  $|\mathcal{D}'| < |\mathcal{D}|$ .

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 such that 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 problem instance  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$  is done as shown in Algorithm 1. The algorithm calls an ASP solver to compute one answer set  $as$  of the debugging program. In case  $\Delta[\Pi, \mathcal{B}]$  has an answer set the algorithm obtains a set  $\mathcal{D}$  and generates a diagnosis program  $\Delta[\Pi, \mathcal{B}, \mathcal{D}]$ . The latter, together with the sets of positive and negative test cases is used to verify whether  $\mathcal{D}$  is a diagnosis or not. 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 extends the debugging program with the following integrity constraint, where atoms  $d_1, \dots, d_n \in \mathcal{D}$  and  $d_{n+1}, \dots, d_m \in \text{Er}(\Delta[\Pi, \mathcal{B}]) \setminus \mathcal{D}$ :

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

Note, similarly to the model-based diagnosis [24, 14] our approach assumes that each error-indicating atom  $er \in \mathcal{D}$  is relevant to an explanation of a fault, whereas all other atoms  $\text{Er}(\Delta[\Pi]) \setminus \mathcal{D}$  are not. That is, some interpretations are not an answer sets of a program only because of reasons suggested by a diagnosis. Consequently, if a user selects a diagnosis  $\mathcal{D}$  resulting in the debugging process, i.e. declares  $\mathcal{D}$  as a correct explanation of a fault, then all other diagnoses are incorrect explanations.

*Example* Let us exemplify our debugging approach on the following program  $\Pi_e$ :

$$r_1 : a \leftarrow \text{not } d \quad r_2 : b \leftarrow a \quad r_3 : c \leftarrow b \quad r_4 : d \leftarrow c \quad r_5 : \leftarrow d$$

---

**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  $\mathbf{D} \leftarrow \emptyset$ ;
2 while  $|\mathbf{D}| < n$  do
3    $as \leftarrow \text{GETANSWERSET}(\Delta[\Pi, \mathcal{B}]);$ 
4   if  $as = \emptyset$  then exit loop;
5    $\mathcal{D} \leftarrow as \cap Er(\Delta[\Pi, \mathcal{B}]);$ 
6    $\Delta[\Pi, \mathcal{B}, \mathcal{D}] \leftarrow \text{DIAGNOSISPROGRAM}(\Delta[\Pi, \mathcal{B}], \mathcal{D});$ 
7   if  $\text{VERIFY}(\Delta[\Pi, \mathcal{B}, \mathcal{D}], P, N)$  then  $\mathbf{D} \leftarrow \mathbf{D} \cup \{\mathcal{D}\};$ 
8    $\Delta[\Pi, \mathcal{B}] \leftarrow \text{EXCLUDE}(\Delta[\Pi, \mathcal{B}], \mathcal{D});$ 
9 return  $\mathbf{D}$ ;
```

---

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

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

Since the program  $\Pi_e$  is inconsistent, a user runs the debugger to clarify the reason. In fact, the inconsistency is caused by an odd loop. That is, 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 following diagnoses:

$$\begin{aligned} \mathcal{D}_1 : \{unsatisfied(id_{r_1})\} \quad \mathcal{D}_2 : \{unsatisfied(id_{r_2})\} \\ \mathcal{D}_3 : \{unsatisfied(id_{r_3})\} \quad \mathcal{D}_4 : \{unsatisfied(id_{r_4})\} \end{aligned}$$

Definition 3 allows to identify the target (preferred) diagnosis  $\mathcal{D}_t$  for the program  $\Pi_e$  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  $\{\overline{int(b)}\}$  – and 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  $\Pi_e$  by a user removing the rule  $r_2$  results in a consistent program  $\Pi_2$  such that all new answer sets of  $\Pi_2$  fulfill all given test cases. All 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                                                                      |
|------------------------------------------------|--------------------------------------------------------------------------------------|
| $\mathcal{D}_1 : \text{unsatisfied}(id_{r_1})$ | $\{\{\overline{int}(a), \overline{int}(b), \overline{int}(c), \overline{int}(d)\}\}$ |
| $\mathcal{D}_2 : \text{unsatisfied}(id_{r_2})$ | $\{\{int(a), \overline{int}(b), \overline{int}(c), \overline{int}(d)\}\}$            |
| $\mathcal{D}_3 : \text{unsatisfied}(id_{r_3})$ | $\{\{int(a), int(b), \overline{int}(c), \overline{int}(d)\}\}$                       |
| $\mathcal{D}_4 : \text{unsatisfied}(id_{r_4})$ | $\{\{int(a), int(b), int(c), \overline{int}(d)\}\}$                                  |

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

## 5 Query-based diagnosis discrimination

The debugging system might generate a set of diagnoses for a given DPI. In our example for simple  $DPI_1$  the debugger returns four minimal diagnoses  $\{\mathcal{D}_1, \dots, \mathcal{D}_4\}$ . 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. That is, in many cases it might be difficult to provide a complete specification of a debugging query localizing a fault. Therefore, the debugging method should be able to find an appropriate set of atoms  $A \subseteq At(\Pi)$  on its own and only query the user or some other oracle, whether these atoms are cautiously/bravely true/false in the interpretations associated with the target diagnosis. To generate a query for a set of diagnoses  $\mathbf{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_n\}$  the debugging system can use the diagnosis programs  $\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]$ , where  $\mathcal{D}_i \in \mathbf{D}$ .

Since in many cases different diagnoses explain why different sets of interpretations of a program  $\Pi$  are not its answer sets, we can use discrepancies between the sets of interpretations to discriminate between the 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  $\Pi_e$  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  $\Pi_e$ . To get this information the debugging algorithm should be able to access some oracle which can answer a number of queries.

**Definition 5.** 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 which is used to update the actual DPI  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle$ . Thus, if an oracle answers

- *cautiously true*, the set  $\{int(a) \mid a \in Q\}$  is added to  $P$ ;
- *cautiously false*, the set  $\{\overline{int}(a) \mid a \in Q\}$  is added to  $N$ ;
- *bravely true*, the set  $\{int(a) \mid a \in Q\}$  is added to  $N$ ;
- *bravely false*, the set  $\{\overline{int}(a) \mid a \in Q\}$  is added to  $P$ .

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  $\{c\}$ . 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

---

**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    $E_i \leftarrow \text{COMMONATOMS}(\mathbf{D}_i^P)$ ;
4    $Q_i \leftarrow \{a \mid \text{int}(a) \in E_i\}$ ;
5   if  $Q_i \neq \emptyset$  then
6      $\langle Q_i, \mathbf{D}_i^P, \mathbf{D}_i^N, \mathbf{D}_i^\emptyset \rangle \leftarrow \text{GENERATEPARTITION}(Q_i, \mathbf{D}, \mathbf{D}_i^P)$ ;
7     if  $\mathbf{D}_i^N \neq \emptyset$  then  $\mathbf{PR} \leftarrow \mathbf{PR} \cup \{\langle Q_i, \mathbf{D}_i^P, \mathbf{D}_i^N, \mathbf{D}_i^\emptyset \rangle\}$ ;
8 return  $\mathbf{PR}$ ;
```

---

sets of atoms over error-indicating predicates are not diagnoses because they do not fulfill the necessary requirements of Definition 3. If the answer is *bravely false*, then the set  $\{\overline{\text{int}(c)}\}$  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  $\text{At}(\Pi)$  which can be queried, the debugger has to ask only those queries which allow to discriminate between the diagnoses of the current DPI.

**Definition 6.** Each diagnosis  $\mathcal{D}_i \in \mathbf{D}$  of 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 it holds that:  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]) \models \{\text{int}(a) \mid a \in Q\}$
- $\mathcal{D}_i \in \mathbf{D}^N$  if it holds that:  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_i]) \models \{\overline{\text{int}(a)} \mid a \in Q\}$
- $\mathcal{D}_i \in \mathbf{D}^\emptyset$  if  $\mathcal{D}_i \notin (\mathbf{D}^P \cup \mathbf{D}^N)$

A partition of the set of diagnoses  $\mathbf{D}$  with respect to 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 to the set of positive test cases  $P$ , whereas the answer *no* is a result of a classification of the query to the set of negative test cases  $N$ . Therefore, without limiting the generality, in the following we consider only these two answers.

Each partition  $\langle Q, \mathbf{D}_i^P, \mathbf{D}_i^N, \mathbf{D}_i^\emptyset \rangle$  allows to discriminate between the diagnoses. That is, if an oracle gives the answer *yes* to a query  $Q$ , then the set of diagnoses  $\mathbf{D}'$  of the updated DPI will not comprise the diagnoses  $\mathbf{D}^N$ . Otherwise, given the answer *no*, the diagnoses  $\mathbf{D}^P$  will not appear in  $\mathbf{D}'$ .

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 and selection. For instance, given  $n = 9$  the algorithm has to verify 512 partitions in the worst case.

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 that can be used to discriminate 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` returns an intersection of all sets  $\mathcal{I} \in \text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_j])$  for all  $\mathcal{D}_j \in \mathbf{D}_i^P$ . Given a non-empty query the function `GENERATEPARTITION` uses Definition 6 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 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  $\mathcal{P}(\mathbf{D}) = \{\{\mathcal{D}_1\}, \{\mathcal{D}_2\}, \dots, \{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4\}\}$  comprises 15 elements, assuming we omit 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  $\mathbf{D}_i^P$ . For instance, the algorithm assigned  $\mathbf{D}_0^P = \{\mathcal{D}_1, \mathcal{D}_2\}$ . In this case the set  $Q_0$  is empty since the set  $E_0 = \{\overline{\text{int}}(b), \overline{\text{int}}(c), \overline{\text{int}}(d)\}$  (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  $E_1 = \{\text{int}(a), \overline{\text{int}}(c), \overline{\text{int}}(d)\}$  and, thus,  $Q_1 = \{a\}$ . The remaining diagnoses  $\mathcal{D}_1$  and  $\mathcal{D}_4$  are classified according to Definition 6. That is, the algorithm selects the first diagnosis  $\mathcal{D}_1$  and verifies whether  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_1]) \models \{\text{int}(a)\}$ . Given the negative answer, the algorithm checks if  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_1]) \models \{\overline{\text{int}}(a)\}$ . 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  $\text{Int}(\Delta[\Pi, \mathcal{B}, \mathcal{D}_4]) \models \{\text{int}(a)\}$ . 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. A random selection might not be a good strategy as it can overload an oracle with unnecessary questions (see [26] for an evaluation of a random strategy). Therefore, the debugger has to decide query of which partition should be asked first in order to minimize the total number of queries to be answered. Query selection is the central topic of *active learning* [25] which is an area of machine learning developing methods that are allowed to query an oracle for labels of unlabeled data instances. Most of the query selection measures used in active learning can be applied within our approach. In this paper, we discuss two query selection strategies, namely, myopic and one step look-ahead.

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 partition  $\langle Q_i, \mathbf{D}_i^P, \mathbf{D}_i^N, \mathbf{D}_i^\emptyset \rangle$  such that  $|\mathbf{D}_i^P| = |\mathbf{D}_i^N|$  and  $\mathbf{D}_i^\emptyset = \emptyset$ . 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 [25], allow to find the target diagnosis using less queries by incorporating heuristics assessing the prior probability  $p(\mathcal{D}_i)$  of each diagnosis  $\mathcal{D}_i \in \mathbf{D}$  to be the target one [14, 26]. Such heuristics can express different preferences/expectations of a user for a fault ex-

---

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

---

**Input:** ground disjunctive program  $\Pi$ , 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  $\sigma$

**Output:** a diagnosis  $\mathcal{D}$

```

1  $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle \leftarrow \text{GENERATEDPI}(\Pi, \mathcal{B}); \mathbf{D} \leftarrow \emptyset;$ 
2 while BELOWTHRESHOLD( $\mathbf{D}, H, \sigma$ )  $\wedge |\mathbf{D}| > 1$  do
3    $\mathbf{D} \leftarrow \mathbf{D} \cup \text{COMPUTEDIAGNOSES}(\langle \Delta[\Pi, \mathcal{B}], P, N \rangle, n - |\mathbf{D}|);$ 
4    $\mathbf{PR} \leftarrow \text{FINDPARTITIONS}(\langle \Delta[\Pi, \mathcal{B}], P, N \rangle, \mathbf{D});$ 
5    $Q \leftarrow \text{SELECTQUERY}(\mathbf{PR}, H, S);$ 
6   if  $Q = \emptyset$  then exit loop;
7    $A \leftarrow \text{GETANSWER}(Q);$ 
8    $\langle \Delta[\Pi, \mathcal{B}], P, N \rangle \leftarrow \text{UPDATEDPI}(A, \langle \Delta[\Pi, \mathcal{B}], P, N \rangle);$ 
9    $\mathbf{D} \leftarrow \text{UPDATEDIAGNOSES}(A, Q, \mathbf{PR}, H);$ 
10 return MOSTPROBABLEDIAGNOSIS( $\mathbf{D}, S, H$ );
```

---

planation. For instance, one heuristic can state that rules including many literals are more likely to be faulty. Another heuristics can assign higher probabilities to diagnoses comprising atoms over *unsatisfiable*/1 predicate if a user expects this type of error. In addition, personalized heuristics can be learned by analyzing the debugging actions of a user in, e.g., ASPIDE [7] or SeaLion [21].

A widely used one step look-ahead strategy [14] suggests that the best query is the one 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 let  $p(\mathcal{D}_j | Q_i = v)$  be the probability of diagnosis  $\mathcal{D}_j$  given an oracle's answer. The expected entropy after querying  $Q_i$  is computed as (see [26] for details):

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

Given the answer  $v$  of an oracle to the selected query  $Q_s$ , i.e.  $Q_s = v$ , we have to update the probabilities of the diagnoses to take the new information into account.

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

In order to reduce the number of queries a user can specify a threshold, e.g.  $\sigma = 0.95$ . If the absolute difference in probabilities between two most probable diagnoses is greater than this threshold, the query process stops and returns the most probable diagnosis. The evaluation results presented in [26] show that this criterion often allows a significant reduction in the number of required queries.

The interactive debugging system (Algorithm 3) takes a ground program or a ground instantiation of 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 starts with the initialization of a DPI. The debugging program  $\Delta[\Pi, \mathcal{B}]$  is generated by `spock`<sup>2</sup>, which implements the meta-programming approach of [10]. First, the main loop of Algorithm 3 computes the required number of diagnoses such that  $|\mathbf{D}| = n$ . Next, we find a set of partitions for the given diagnoses and select a query according to a query strategy  $S$  selected by a user. If the user selected the myopic strategy then probabilities of diagnoses are ignored by `SELECTQUERY`. The oracle is asked to classify the query and the answer is used to update the DPI as well as the set  $\mathbf{D}$  from which 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  $\sigma$  or only one diagnosis remains. Finally, the most probable diagnosis or, in case of a myopic strategy, the first diagnosis is returned to a user. Algorithm 3 was prototypically implemented as a part of a general diagnosis framework<sup>3</sup>. A plug-in for SeaLion providing a user-friendly interface for our method is currently in development.

## 6 Summary and future work

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.

In the future work we are going to investigate the applicability of our approach to the method of [20] since (a) this method can be applied to non-grounded programs and (b) it was recently extended to programs with choice rules, cardinality and weight constraints [22]. In addition, there is a number of other debugging methods for ASP that might be integrated with the suggested query selection approach. For instance, the method of [17] can be used to translate the program and queries into a natural language representation, thus, simplifying the query classification problem. Another technique that can be used to simplify the query answering is presented in [23] where the authors suggest a graph-based justification technique for truth values with respect to an answer set. Moreover, we would like to research whether query generation and selection ideas can be applied in the debugging method of [18]. This interactive framework allows a programmer to step through an answer set program by iteratively extending a state of a program (partial reduct) with new rules. The authors suggest a filtering approach that helps a user to find such rules and variable assignments that can be added to a state. We want to verify whether the filtering can be extended by querying about disagreements between the next states, such as “if user adds a rule  $r_1$  then  $r_2$  cannot be added”.

<sup>2</sup> [www.kr.tuwien.ac.at/research/debug](http://www.kr.tuwien.ac.at/research/debug)

<sup>3</sup> <https://code.google.com/p/rmbd/wiki/AspDebugging>

One more interesting source of heuristics, that we also going to investigate, can be obtained during testing of ASP programs [13]. The idea comes from spectrum-based fault localization (SFL) [12], which is widely applied to software debugging. Given a set of test cases specifying inputs and outputs of a program SFL generates an observation matrix  $A$  which comprises information about: (i) parts of a program executed for a test case and (ii) an error vector  $E$  comprising results of tests executions. Formally, given a program with  $n$  software components  $C := \{c_1, \dots, c_n\}$  and a set of test cases  $T := \{t_1, \dots, t_m\}$  a hit spectra is a pair  $(A, E)$ .  $A$  is a  $n \times m$  matrix where each  $a_{ij} = 1$  if  $c_j$  was involved in execution of the test case  $t_i$  and  $a_{ij} = 0$  otherwise. Similarly for each  $e_i \in E$ ,  $e_i = 1$  if the test case  $t_i$  failed and  $e_i = 0$  in case of a success. Obviously, statistics collected by the hit spectra after execution of all tests allows to determine the components that were involved in execution of failed test cases. Consequently, we can obtain a set of fault probabilities for the components  $C$ . The same methodology can be applied to debugging and testing of ASP programs. For each test case  $t_i$  we have to keep a record which sets of ground rules (Gelfond-Lifschitz reducts) were used to obtain answer sets that violate/satisfy  $t_i$ . Next, we can use the obtained statistics to derive fault probabilities for ground rules of an ASP program being debugged. The probabilities of diagnoses can then be computed from the probabilities of rules as it is shown in [26].

## 7 Acknowledgments

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

## References

1. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
2. Brain, M., Vos, M.D.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings of the 3rd International Workshop on Answer Set Programming. pp. 141–152 (2005)
3. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 31–43 (2007)
4. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM 54(12), 92–103 (2011)
5. Cliffe, O., Vos, M., Brain, M., Padget, J.: Aspviz: Declarative visualisation and animation using answer set programming. Logic Programming. pp. 724–728 (2008)
6. Delgrande, J.P., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming 3(02), 129–187 (2003)
7. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Logic Programming and Nonmonotonic Reasoning. pp. 317–330 (2011)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Communications 24(2), 107–124 (2011)

9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan & Claypool Publishers (2012)
10. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proceedings of 23rd AAAI Conference on Artificial Intelligence. pp. 448–453 (2008)
11. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. *New generation computing* 9(3-4), 365–386 (1991)
12. Harrold, M.J., Rothermel, G., Wu, R., Yi, L.: An empirical investigation of program spectra. *ACM SIGPLAN Notices* 33(7), 83–90 (1998)
13. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On Testing Answer-Set Programs. In: 19th European Conference on Artificial Intelligence. pp. 951–956 (2010)
14. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. *Artificial Intelligence* 32(1), 97–130 (Apr 1987)
15. Lee, J.: A Model-theoretic Counterpart of Loop Formulas. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence. pp. 503–508 (2005)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)* 7(3), 499–562 (2006)
17. Mikitiuk, A., Moseley, E., Truszczynski, M.: Towards Debugging of Answer-Set Programs in the Language PSpb. In: Proceedings of the 2007 International Conference on Artificial Intelligence. pp. 635–640 (2007)
18. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an Answer-Set Program. In: Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning. pp. 134–147 (2011)
19. Oetsch, J., Pührer, J., Seidl, M., Tompits, H., Zwickl, P.: VIDEAS : Supporting Answer-Set Program Development using Model-Driven Engineering Techniques. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 382–387 (2011)
20. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: On Debugging Non-ground Answer-Set Programs. *Theory and Practice of Logic Programming* 10(4-6), (2010)
21. Oetsch, J., Pührer, J., Tompits, H.: The SeaLion has Landed: An IDE for Answer-Set Programming – Preliminary Report. CoRR abs/1109.3989 (2011)
22. Polleres, A., Frühstück, M., Schenner, G., Friedrich, G.: Debugging Non-ground ASP Programs with Choice Rules, Cardinality and Weight Constraints. In: *Logic Programming and Nonmonotonic Reasoning*. pp. 452–464 (2013)
23. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming* 9(1), 1 (2009)
24. Reiter, R.: A Theory of Diagnosis from First Principles. *Artificial Intelligence* 32(1), 57–95 (1987)
25. Settles, B.: Active Learning, Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2012)
26. Shchekotykhin, K., Friedrich, G., Fleiss, P., Rodler, P.: Interactive ontology debugging: Two query strategies for efficient fault localization. *Web Semantics: Science, Services and Agents on the World Wide Web* 12-13, 88 – 103 (2012)
27. Simons, P., Niemelä, I., Soeninen, T.: Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2), 181–234 (2002)
28. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning. pp. 77–84 (2006)
29. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog\* Environment. In: *Software Engineering for Answer Set Programming*. pp. 101–115 (2007)