On Structural Analysis of Non-Ground Answer-Set Programs∗

BENJAMIN KIESL,[1] PETER SCHÜLLER,[2] and HANS TOMPITS[1]

[1] *Institut für Informationssysteme, Arbeitsbereich Wissensbasierte Systeme 184/3, Technische Universität Wien*
(*e-mail:* {kiesl,tompits}@kr.tuwien.ac.at)

[2] *Department of Computer Engineering, Faculty of Engineering, Marmara University*
(*e-mail:* peter.schuller@marmara.edu.tr)

submitted 29 April 2015; accepted 5 June 2015

Abstract

The development of answer-set programs often involves domain experts without a background in logic programming. In such situations, it would be beneficial to translate programs into a form which is easier to understand and closer to natural language. Since the structure of a program determines to a great extent how a program should be explained in a clear and comprehensible way, as a first step towards a natural-language representation of answer-set programs, in this paper, we introduce methods for analysing the structure of disjunctive non-ground answer-set programs. In particular, as most programs follow the *generate-define-test paradigm*, we introduce formal definitions to characterise the respective generate, define, and test parts of a program. Thereby, we define the *non-deterministic core* of a program, effectively determining the program's active solution-space generators, following ideas of the weakly perfect model semantics as introduced by Przymusinska and Przymusinski, and we prove that our definitions fulfil desirable properties. Moreover, we also provide an implementation of a tool, using a metaprogramming approach, which classifies the rules of a given program according to our definitions. Finally, we propose an algorithm that, based on our generate-define-test classification, computes the order in which the rules of a program should be explained when translated into natural language.

KEYWORDS: answer-set programming, generate-define-test paradigm, program analysis

1 Introduction

Answer-set programming (ASP) has proven to be a viable tool for knowledge representation, reasoning, and solving complex search problems. In many cases, the process of developing answer-set programs involves domain experts, i.e., persons with a detailed knowledge of some certain field of application but often without

∗ This work has been supported by the Austrian Science Fund (FWF) under project W1255-N23 and by the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 114E777.

a background in ASP. Such experts provide their know-how of a specific topic to software developers who are then concerned with encoding this expertise into answer-set programs. In such a context, arguably the software-engineering process for answer-set programs involving non-ASP experts could be eased if representations of the programs and their output in a form which is closer to natural language would be available.

Since the structure of a program determines to a great extent how a program should be explained in a clear and understandable way, as a first step towards realising natural-language representations of answer-set programs, we introduce in this paper methods for analysing the structure of disjunctive non-ground answer-set programs. We note that work addressing representations in the other direction, i.e., from natural language to ASP, has been discussed in the literature, e.g., by Erdem and Yeniterzi (2009), who formulated queries over biomedical ontologies using a controlled natural language (CNL), which are then translated into ASP for evaluation, and by Schwitter (2012; 2013) who specified various logical puzzles in CNL and provided their solutions via translations into ASP.

A key structural feature of most answer-set programs is that they adhere to the so-called *generate-define-test* paradigm which was informally introduced by Lifschitz (2002) as a refinement of the well-known *guess-and-check* paradigm (Eiter et al. 2000; Leone et al. 2006; Eiter et al. 2009). According to this paradigm, a program can be considered as a combination of three parts, respectively referred to as the *generate*, *define*, and *test part*. The generate part non-deterministically generates a set of candidate solutions from which the test part eliminates those candidates which are not the desired solutions of the specified problem. In order to do so, both the generate and the test part may use concepts which are specified in the define part. Although *generate-define-test* is, as indicated by Denecker et al. (2012), the dominating methodology of ASP, formal criteria to characterise the generate, define, and test parts of an answer-set program have not been introduced so far to the best of our knowledge. We introduce such formal conditions and argue for their adequacy by showing that they possess intuitively desirable properties.

For a natural definition of the non-deterministic generate part of a program, it is necessary to identify cases in which default negation leads to non-deterministic behaviour. From a result of You and Yuan (1994), it immediately follows that a non-disjunctive program has multiple answer sets only if its (ground) dependency graph contains cycles with an even number of negative edges. Furthermore, odd negative cycles can eliminate certain answer-set candidates. But, as demonstrated by Przymusinska and Przymusinski (1990), negative cycles do not always become active which is one reason why the existence of even negative cycles is not sufficient for the creation of multiple answer sets and why odd negative cycles do not necessarily eliminate answer-set candidates.

Based on ideas underlying the procedure for the computation of the weakly-perfect model of a logic program (Przymusinska and Przymusinski 1990), we define the *non-deterministic core* of a program which is obtained by removing certain rules and literals, thereby eliminating negative cycles that cannot become active. Given this, we then define the generate part of a program as those rules which have

instances that are involved in even negative cycles within the dependency graph of the non-deterministic core, together with disjunctive rules. From the remaining rules, those which are constraints or which have instances that are involved within odd negative cycles of the non-deterministic core form the test part of a program, while all other rules comprise the define part.

We also present an implementation of a tool, based on a metaprogramming approach, which classifies the rules of a given program according to our definitions. Finally, we propose an algorithm that makes use of our generate-define-test classification to compute a specific order for explaining the rules of a program in natural language.

2 Preliminaries

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

$$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n, \tag{1}$$

where $L_1, \ldots, L_n$ are literals, i.e., atoms from some (implicitly given) function-free first-order language $\mathcal{L}$, possibly preceded by the strong-negation symbol "$\neg$". Literals which are possibly preceded by the default-negation symbol "not" are called *default literals*. Given a rule r as above, we define $head(r) = \{L_1, \ldots, L_k\}$, $body^+(r) = \{L_{k+1}, \ldots, L_m\}$, and $body^-(r) = \{L_{m+1}, \ldots, L_n\}$. The elements of $head(r)$ are referred to as the *head literals* of r, while the elements of $body^+(r)$ and $body^-(r)$ are respectively referred to as the *positive* and *negative body literals* of r. Moreover, we also define $body(r) = body^+(r) \cup body^-(r)$. We call a rule of form (1) *disjunctive* if $k > 1$, a *fact* if $body(r) = \emptyset$, and a *constraint* if $head(r) = \emptyset$. A program Π is *extended* if it contains no disjunctive rules, *normal* if it contains neither disjunctions nor strong negations, and *positive* if for every $r \in \Pi$, $body^-(r) = \emptyset$. Given a rule or program e, we denote by $at(e)$ (resp., $lit(e)$) the set of atoms (resp., literals) in e.

The *grounding* of a logic program Π relative to its Herbrand universe $HU(\Pi)$ is defined as usual and denoted by $grd(\Pi)$. Let $\mathcal{M}$ be a set of ground (variable-free) literals. $\mathcal{M}$ *satisfies* a rule r if $body^+(r) \subseteq \mathcal{M}$ and $body^-(r) \cap \mathcal{M} = \emptyset$ only if $head(r) \cap \mathcal{M} \neq \emptyset$. Furthermore, $\mathcal{M}$ is *closed under* a program Π if $\mathcal{M}$ satisfies all rules in Π, and $\mathcal{M}$ is *logically closed* if it is consistent (i.e., for no literal $L \in \mathcal{M}$ we have $\{L, \neg L\} \subseteq \mathcal{M}$) or contains all literals. The *reduct*, $\Pi^{\mathcal{M}}$, *of* Π *relative to* $\mathcal{M}$ (Gelfond and Lifschitz 1988) is given by $\{head(r) \leftarrow body^+(r) \mid r \in grd(\Pi) \text{ and } body^-(r) \cap \mathcal{M} = \emptyset\}$. $\mathcal{M}$ is an *answer set* of Π if it is a minimal set that is both closed under $\Pi^{\mathcal{M}}$ and logically closed.

The *dependency graph*, $\mathsf{DG}(\Pi)$, of a program Π (Ullman 1988) is the graph which contains as vertices $lit(grd(\Pi))$ and a *positive* (resp., *negative*) edge (B, H) iff there is a rule $r \in grd(\Pi)$ such that $H \in head(r)$ and $B \in body^+(r)$ (resp., $B \in body^-(r)$). A directed path with an even (resp., odd) number of negative edges is called an *even* (resp., *odd*) *negative path*. Accordingly, a path without negative edges is called a *positive path*. A *cycle* is a directed path whose start and end vertex coincide and in which no other vertex is visited more than once. Let ξ be

the function which maps every edge e in $\mathsf{DG}(\Pi)$ to the set of all rules in Π that cause e to be contained in $\mathsf{DG}(\Pi)$. Then, we say that a rule r is *involved* in a path P of $\mathsf{DG}(\Pi)$ if P contains an edge e with $r \in \xi(e)$.

Given a program Π and literals A and B, A *depends positively* (resp., *depends negatively*) on B if $\mathsf{DG}(\Pi)$ contains a positive (resp., negative) path from B to A. Moreover, A *depends* on B if A depends positively or negatively on B.

The *rule graph*, $\mathsf{RG}(\Pi)$, of a program Π (Dimopoulos and Torres 1996) is the graph which contains as vertices the rules of Π and an edge (r_1, r_2) iff there is a predicate p which occurs in the body of r_1 and in the head of r_2.

A normal logic program Π is *stratified* if there exists a mapping f assigning each predicate symbol a natural number such that, for every rule $r \in \Pi$ and every predicate P_1 and P_2, (i) if P_1 occurs in $\mathit{head}(r)$ and P_2 occurs in $\mathit{body}^+(r)$ then $f(P_1) \geq f(P_2)$, and (ii) if P_1 occurs in $\mathit{head}(r)$ and P_2 occurs in $\mathit{body}^-(r)$ then $f(P_1) > f(P_2)$. Let Π' be the program obtained from $\mathit{grd}(\Pi)$ by replacing every atom with a corresponding propositional predicate symbol. Then, Π is *locally stratified* if Π' is stratified (Apt et al. 1988).

3 A Formal Generate-Define-Test Classification

In this section, we introduce formal definitions of the generate, define, and test parts of an answer-set program. Furthermore, we also sketch a metaprogram which classifies the rules of a given answer-set program according to our definitions.

As already noted in Section 1, the generate part should contain those rules which cause non-determinism in the sense that they generate multiple answer-set candidates. Disjunctive rules and negative cycles are obvious sources of such non-determinism, but, although even negative cycles are in the absence of disjunction necessary for the creation of multiple answer sets, they are not sufficient. Consider, for illustration, the program Π_1, consisting of the facts $n(1)$, $n(2)$, and $\mathit{succ}(1, 2)$, together with the following rule which, intuitively, chooses every second element of the set n:

$$\mathit{choose}(Y) \leftarrow n(X), n(Y), \mathit{succ}(X, Y), \text{not } \mathit{choose}(X). \tag{2}$$

The grounding of Π_1 contains, additionally to the facts, the following ground instances of (2):

$$\mathit{choose}(1) \leftarrow n(1), n(1), \mathit{succ}(1, 1), \text{not } \mathit{choose}(1), \tag{3}$$

$$\mathit{choose}(1) \leftarrow n(2), n(1), \mathit{succ}(2, 1), \text{not } \mathit{choose}(2), \tag{4}$$

$$\mathit{choose}(2) \leftarrow n(1), n(2), \mathit{succ}(1, 2), \text{not } \mathit{choose}(1), \tag{5}$$

$$\mathit{choose}(2) \leftarrow n(2), n(2), \mathit{succ}(2, 2), \text{not } \mathit{choose}(2). \tag{6}$$

Rules (4) and (5) are involved in an even negative cycle with literals $\mathit{choose}(1)$ and $\mathit{choose}(2)$. Still, Π_1 has the unique answer set $\{n(1), n(2), \mathit{succ}(1, 2), \mathit{choose}(2)\}$. Intuitively, the negative cycle does not become active in the sense that the body atom $\mathit{succ}(2, 1)$ in Rule (4) cannot be contained in an answer set of Π_1. Modern grounders automatically eliminate a rule like (4) from the grounding of Π_1 but there are more intricate cases where grounders are not able to remove such negative cycles.

The observation that in many cases negative cycles do not become active has already been made by Przymusinska and Przymusinski (1990) when they introduced the weakly-perfect-model semantics and the class of *weakly stratified* programs. The distinguishing feature of weakly stratified programs is that they have unique answer sets even though they are a strict superclass of locally stratified programs.

We next use ideas behind the weakly-perfect-model semantics to define the *non-deterministic core* (short, *nd-core*) of a program. The nd-core is obtained by removing certain rules and literals such that negative cycles which cannot become active are eliminated. We begin with the definition of weakly minimal literals.

Definition 1
A literal $L \in lit(grd(\Pi))$ is *weakly minimal* if it does not depend negatively on other literals and if it does not depend on a literal in the head of a disjunctive rule.

Definition 2
The *deterministic bottom-stratum*, $\mathsf{dbs}(\Pi)$, of Π is the set of all weakly-minimal literals which do not occur in the heads of disjunctive rules. Furthermore, the *deterministic bottom-layer*, $\mathsf{dbl}(\Pi)$, of Π is the set $\{r \in grd(\Pi) \mid head(r) \subseteq \mathsf{dbs}(\Pi)\}$.

The deterministic bottom-layer of a program can be seen as the positive deterministic portion of a program since it is a non-disjunctive positive program which clearly has a unique answer set.

Example 1
Let Π_2 consist of the set of facts $\{n(1), n(2), succ(1,2)\}$ and the following rules:

$$choose(Y) \leftarrow chooseEven, n(X), n(Y), succ(X, Y), \text{not } choose(X), \quad (7)$$
$$choose(X) \leftarrow chooseAll, n(X), \quad (8)$$
$$chooseEven \leftarrow \text{not } chooseAll, \quad (9)$$
$$chooseAll \leftarrow \text{not } chooseEven. \quad (10)$$

Π_2 has two answer sets: one in which $choose(n)$ is true for every number n and one in which $choose(n)$ is only true for even n. The grounding of Π_2 contains the facts together with Rules (9) and (10), and the following instances of (7) and (8):

$$choose(1) \leftarrow chooseEven, n(1), n(1), succ(1,1), \text{not } choose(1), \quad (11)$$
$$choose(1) \leftarrow chooseEven, n(2), n(1), succ(2,1), \text{not } choose(2), \quad (12)$$
$$choose(2) \leftarrow chooseEven, n(1), n(2), succ(1,2), \text{not } choose(1), \quad (13)$$
$$choose(2) \leftarrow chooseEven, n(2), n(2), succ(2,2), \text{not } choose(2), \quad (14)$$
$$choose(1) \leftarrow chooseAll, n(1), \quad (15)$$
$$choose(2) \leftarrow chooseAll, n(2). \quad (16)$$

Then, the deterministic bottom-stratum of $grd(\Pi_2)$ is $\mathsf{dbs}(grd(\Pi_2)) = \{n(1), n(2), succ(1,1), succ(1,2), succ(2,1), succ(2,2)\}$, and thus the deterministic bottom-layer is $\mathsf{dbl}(grd(\Pi_2)) = \{succ(1,2) \leftarrow, n(2) \leftarrow, n(1) \leftarrow\}$. □

For defining the nd-core of a program Π we repeatedly apply the transformation given in Definition 3 until a fixed-point is reached. The intuitive idea behind this transformation is to first evaluate the deterministic positive portion of the program, identified by its deterministic bottom-layer. The deterministic bottom-layer, as well as further rules whose bodies can never become satisfied or which are redundant, are then removed. Additionally, default literals that are satisfied by the unique answer set of the deterministic bottom-layer are removed from the remaining rules. This eliminates certain negative cycles that have a deterministic effect.

Definition 3
Let Π be a ground logic program and $\mathcal{M}$ the unique answer set of $\mathsf{dbl}(\Pi)$. Then, $\nu(\Pi)$ is the program obtained from Π by

(i) removing all rules which contain a negative body literal N such that $N \in \mathcal{M}$, a positive body literal P such that $P \in \mathsf{dbs}(\Pi) \setminus \mathcal{M}$, or a head literal H with $H \in \mathcal{M}$,
(ii) removing those body default literals such that $P \in \mathcal{M}$ for positive body literals P or $N \in \mathsf{dbs}(\Pi) \setminus \mathcal{M}$ for negative body literals N, and
(iii) removing all non-facts whose heads appear as facts in the program.

The removal of non-facts whose heads appear as facts is important for programs like $\Pi_3 = \{a \leftarrow \text{not } b, b \leftarrow \text{not } a, a \leftarrow\}$. Here, it allows to remove the first rule although $\mathsf{dbs}(\Pi_3) = \emptyset$ and so $\nu(\Pi_3)$ has, in contrast to Π_3, no negative cycles.

Based on the operator ν, we can now define the non-deterministic core of a (possibly non-ground) program, intuitively obtained by partially evaluating its deterministic portion (which may involve negative default literals) and transforming the program accordingly.

As a preparatory step, let $\nu^0(\Pi) = \Pi$ and $\nu^{i+1}(\Pi) = \nu(\nu^i(\Pi))$, for $i \in \mathbb{N}^+$. Clearly, the application of ν to a ground logic program Π removes from Π all the literals of $\mathsf{dbs}(\Pi)$ and does not add any literals. Hence, there is an i such that either (a) $\nu^i(\Pi) = \emptyset$ and thus $\nu^{i+1}(\Pi) = \emptyset$, or (b) $\nu^{i+1}(\Pi)$ is non-empty and does not differ from $\nu^i(\Pi)$, as $\mathsf{dbs}(\nu^i(\Pi)) = \emptyset$, and $\nu^i(\Pi)$ does not contain any non-facts whose heads occur as facts. Thus, the following holds:

Theorem 1
For every ground logic program Π, there is a smallest number i_0 such that $\nu^{i_0}(\Pi) = \nu^k(\Pi)$, for each $k \geq i_0$.

Let us write $\nu^\infty(\Pi)$ for $\nu^{i_0}(\Pi)$ with i_0 as in Theorem 1. We then define:

Definition 4
The *non-deterministic core* (or *nd-core*) of a program Π is given by $\nu^\infty(grd(\Pi))$.

Example 2
Let $\Pi = grd(\Pi_3)$, where Π_3 is the program from the above. In Example 1, we already identified that $\mathsf{dbl}(\Pi) = \{n(1) \leftarrow, n(2) \leftarrow, succ(1,2) \leftarrow\}$. Its unique answer set is $\mathcal{M} = \{n(1), n(2), succ(1,2)\}$. Now, when computing $\nu(\Pi)$, Rules (11), (12), and (14) are removed since their body literals $succ(1,1)$, $succ(2,1)$, and $succ(2,2)$

are contained in $\mathsf{dbs}(\Pi) \setminus \mathcal{M}$. As well, the facts $succ(1,2) \leftarrow$, $n(1) \leftarrow$, and $n(2) \leftarrow$ are removed since their heads are contained in $\mathcal{M}$. Finally, the body literals $n(1)$, $n(2)$, and $succ(1,2)$ are removed from the remaining rules since they are contained in $\mathcal{M}$. We thus obtain $\nu^1(\Pi) = \nu(\Pi)$ as follows:

$$choose(2) \leftarrow chooseEven, \text{not } choose(1), \tag{17}$$
$$choose(1) \leftarrow chooseAll, \tag{18}$$
$$choose(2) \leftarrow chooseAll, \tag{19}$$
$$chooseEven \leftarrow \text{not } chooseAll, \tag{20}$$
$$chooseAll \leftarrow \text{not } chooseEven. \tag{21}$$

Since $\nu^1(\Pi)$ does not contain any weakly-minimal literals, we have that $\mathsf{dbs}(\nu^1(\Pi))= \emptyset$. Therefore, $\nu^2(\Pi) = \nu^1(\Pi)$, and so $\nu^1(\Pi)$ is already the nd-core of Π_3. □

In this example, only one application of ν was necessary for computing the nd-core of Π_3. But this is not always the case. In fact, the number of computation steps cannot be bounded by a constant, as stated by the following theorem:

Theorem 2
For every $n \in \mathbb{N}$, there is a ground program Π^n such that $\nu^\infty(\Pi^n) \neq \nu^{n-1}(\Pi^n)$.

Having the nd-core of a program at our disposal, we can now define the generate, define, and test parts. In the following, an *instance* of a rule r refers to a rule which was obtained from r by grounding it and which has possibly been modified by ν.

Definition 5
Let Π be a logic program. Then,

(i) the *generate part* of Π consists of all rules which are disjunctive or which have instances that are involved in even negative cycles within the dependency graph of the nd-core of Π,
(ii) the *test part* of Π consists of all rules which are not in the generate part and which are either constraints or which have instances that are involved in (odd) negative cycles within the dependency graph of the nd-core of Π, and
(iii) the *define part* of Π consists of all rules which are neither in the generate part nor in the test part.

Rules of the generate, define, and test part are referred to as *generating*, *defining*, and *testing rules*, respectively.

Note that we are explicitly speaking of instances of (non-ground) rules which are involved in negative cycles and not of rules whose head literals or predicates are contained in a negative cycle. Note also that facts are contained in the define part but it would be straightforward to define a separate part for them.

Also, often in non-ground ASP, one deals with *uniform problem encodings*, in which a program Π is detached from the set of facts providing the *input* of the program. From a database point of view, the program is the *intensional database* (IDB) whilst the facts constitute the *extensional database* (EDB). When trying to analyse the structure of such a uniform encoding Π, the analysis of Π can be done

by computing the nd-core of $\Pi \cup \Pi_f$, where Π_f consists of input facts comprising instances of atoms of all extensional predicates of Π.

Example 3
For the program Π_2, the generate part consists of the rules

$$\mathit{chooseEven} \leftarrow \text{not } \mathit{chooseAll} \quad \text{and} \quad \mathit{chooseAll} \leftarrow \text{not } \mathit{chooseEven},$$

while the define part contains the facts together with the following rules:

$$\mathit{choose}(Y) \leftarrow \mathit{chooseEven}, n(X), n(Y), \mathit{succ}(X, Y), \text{not } \mathit{choose}(X), \tag{22}$$
$$\mathit{choose}(X) \leftarrow \mathit{chooseAll}, n(X). \tag{23}$$

The test part of Π_2 is empty. □

In the dependency graph of Π_2, instances of (22) are involved in negative cycles with an even number of negative edges as well as in negative cycles with an odd number of negative edges. But, contrary to common intuition, these cycles neither cause the creation of multiple (candidate) answer sets nor do they eliminate them. Hence, (22) is a defining rule and not a generating or a testing rule. In fact, (22) would even be a defining rule if predicate *succ* would not be defined by facts but by an equivalent set of rules including default negation.

Next, we discuss some properties of the above definitions.

Lemma 1
The nd-core of every weakly stratified program, and thus of every (locally) stratified program, is empty.

Consequently, the following holds:

Theorem 3
The generate part of every weakly stratified program, and thus of every (locally) stratified program, is empty.

The following theorem expresses that a non-deterministic behaviour of a program always has its origin in the generate part.

Theorem 4
A program with an empty generate part has at most one answer set.

Note that the converse of this statement does not hold since the generate part can create multiple answer-set candidates which are all eliminated by the test part.

In concluding this section, we remark that we implemented a tool which classifies logic programs into its generate, define, and test parts according to our definitions. The tool is implemented in Java and performs the computation of the nd-core and the identification of rules which are involved within its negative cycles by means of an ASP metaprogramming approach, using an answer-set program consisting of the following modules:

(i) Π_{reify}, representing the input program as a set of facts,

(ii) $\Pi_{dependency}$, defining the dependency notions,
(iii) $\Pi_{bottomlayer}$, identifying the deterministic bottom-stratum and the corresponding deterministic bottom-layer,
(iv) $\Pi_{nd\text{-}core}$, encoding the operator ν and computing the nd-core, and
(v) Π_{cycle}, identifying rules which are involved in even and odd negative cycles within the dependency graph.

Since the whole metaprogram consists of more than 100 rules we do not list it here. It is available at `www.kr.tuwien.ac.at/staff/kiesl/gdt_classification.lp`.

4 An Explanation Order on Rules

For obtaining an understandable explanation of an answer-set program, the order in which the rules are explained is crucial. One way to obtain arguably useful explanations is to start with generating rules and then continue by explaining how the predicates of the generate part are constrained by rules from the test part, thereby also taking the involved defining rules into account. In what follows, we discuss a method for assigning an explanation order on rules which also copes in an intuitive way with testing rules that constrain several generating rules.

Example 4
Assume that we want to assign colours to the vertices of a graph such that every vertex is either red, blue, or has no colour at all. Furthermore, some of the vertices are not allowed to be either red or blue, represented by predicates $must\_not\_be\_red$ and $must\_not\_be\_blue$. A possible encoding is the following program Π_4:

$$red(X) \vee \neg red(X) \leftarrow vertex(X), \tag{24}$$
$$blue(X) \vee \neg blue(X) \leftarrow vertex(X), \tag{25}$$
$$\leftarrow vertex(X), red(X), must\_not\_be\_red(X), \tag{26}$$
$$\leftarrow vertex(X), blue(X), must\_not\_be\_blue(X), \tag{27}$$
$$\leftarrow vertex(X), red(X), blue(X). \tag{28}$$

There are two generating rules, (24) and (25). Among the three constraints, (26) is only related to literals generated by (24), (27) is only related to those generated by (25), and (28) is related to literals that are generated by both generating rules.

We believe that a useful explanation of the whole program is as follows: we start by explaining the generating rule (24) together with its corresponding constraint (26); after that we explain (25) together with (27), and at the end we mention (28) which constrains both generating rules. A natural-language explanation realising this order could be as follows:

1. *For any vertex, choose whether it is red or not* [Rule (24)] *such that no vertex which must not be red is red* [Rule (26)].
2. *For any vertex, choose whether it is blue or not* [Rule (25)] *such that no vertex which must not be blue is blue* [Rule (27)].
3. *Additionally, it must not be the case that there is a vertex which is both red and blue* [Rule (28)]. □

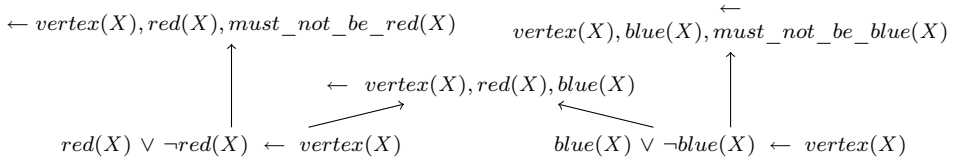

Fig. 1. The rule graph of Π_4.

To formalise the intuition behind this order, we make use of the strongly-connected components (SCCs) of the rule graph $\mathsf{RG}(\Pi)$ of a program Π. Given a rule $r \in \Pi$ we denote by $\mathsf{SCC}(r)$ the SCC of $\mathsf{RG}(\Pi)$ containing r. We call the graph obtained by collapsing SCCs of $\mathsf{RG}(\Pi)$ the *reduction graph of* $\mathsf{RG}(\Pi)$. An SCC of $\mathsf{RG}(\Pi)$ is *maximal* if it does not have an outgoing edge within the reduction graph of $\mathsf{RG}(\Pi)$.

Definition 6
A rule of a program Π is *maximal* if it is contained in a maximal SCC of $\mathsf{RG}(\Pi)$.

Definition 7
A generating rule r_G of a program Π is *maximal* if in $\mathsf{RG}(\Pi)$ (i) no generating rule in $\mathsf{SCC}(r_G)$ has more outgoing edges than r_G and (ii) there is no directed path from r_G to a generating rule in a different SCC of $\mathsf{RG}(\Pi)$.

The intuition behind this definition is to give priority to explaining generating rules that are most influential in the program (according to outgoing paths) and that are nearest to testing rules and definitions.

We next formalise that testing rules can be associated with certain rules.

Definition 8
Given a logic program Π, let $r_T \in \Pi$ be a testing rule and $r \in \Pi$ a rule. Then, r *is constrained by* r_T if there is a directed path from r to r_T in $\mathsf{RG}(\Pi)$, otherwise r is *unconstrained by* r_T.

Note that testing rules are trivially constrained by themselves.

Example 5 (*continued*)
Figure 1 depicts the rule graph of Π_4. According to our definitions, (26) constrains only (24), (27) constrains only (25), and (28) constrains both (24) and (25). All three constraints induce maximal components and thus they are maximal rules. Both generating rules (24) and (25) are maximal ones. □

Algorithm 1 computes an order for explaining rules of a given program. Intuitively, rules are explained in the order they are visited during a depth-first traversal of the rule graph, starting from maximal rules and maximal generating rules. During the traversal, rules which are in the same SCC are preferred. Additionally, as soon as all rules from which a generating rule r_G is reachable have been visited, testing rules which constrain r_G are explained. Step (a) first explains unconstrained maximal rules, then maximal generating rules, and finally arbitrary maximal rules.

Theorem 5
Algorithm 1 always terminates and assigns a unique number to each rule.

```
input  : rule graph G
output: order in which the rules are explained, defined by order
let sameSCC and otherSCC be stacks
let i := 1
while G contains unlabelled rules do
    if otherSCC is empty then
(a)     let r be an unlabelled rule according to the following priority:
        - first choose unconstrained maximal rules,
        - then choose maximal generating rules,
        - then choose maximal rules.
        push(sameSCC, r)
    while sameSCC or otherSCC is not empty do
        if sameSCC is not empty then
            let r := pop(sameSCC)
        else
            let r := pop(otherSCC)
        if r is not labelled then
            let order(r) := i      // now we consider r to be labelled
            let i := i + 1
            // iterate in the same order as the r' occur in the body of r
            for all edges (r', r) in G do
                if r' is in the same SCC as r then
                    push(sameSCC, r')
                else
                    push(otherSCC, r')
    let l be the list of unlabelled testing rules
    order l by the number of generating rules they constrain, descending
    for c in l do
        if all generating rules constrained by c are labelled then
(b)         push(otherSCC, c)
```

Algorithm 1: Computes the order in which rules are explained.

Example 6 (*continued*)
Π_4 does not contain unconstrained rules, so in Step (a) of Algorithm 1 we choose one of the generating rules, say (24). It has no incoming edges but after labelling it with 1, all rules constrained by (26) are labelled, hence (26) is pushed in Step (b) and labelled with 2. Subsequently, the next unlabelled maximal generating rule is chosen in (a), viz. Rule (25), and labelled with 3. Then, (27) is pushed in (b) and labelled with 4 since all rules which are constrained by (27) were labelled. Finally, (28) is pushed in (b) since now both rules which it constrains have been labelled. Note that (28) is visited after (27) because (28) constrains more generating rules than (27). We end up with the following order:

$$
\begin{array}{lrcl}
1: & red(X) \vee \neg red(X) & \leftarrow & vertex(X), \\
2: & & \leftarrow & vertex(X),\ red(X),\ must\_not\_be\_red(X), \\
3: & blue(X) \vee \neg blue(X) & \leftarrow & vertex(X),
\end{array}
$$

4: $\leftarrow\ vertex(X),\ blue(X),\ must\_not\_be\_blue(X),$

5: $\leftarrow\ vertex(X),\ red(X),\ blue(X),$

which is exactly the order we wanted at the beginning of this section. □

5 Conclusion

We introduced methods for structural analysis of non-ground disjunctive answer-set programs. In particular, we introduced formal definitions for the generate, define, and test parts of a program. In order to arrive at natural definitions, we first defined the non-deterministic core of a program, which is obtained by eliminating deterministically determined rules and literals. This has the potential to eliminate those negative cycles that cannot become active and thus should not be considered in the generate or test part. We then defined generating rules to be disjunctive rules as well as rules having instances that are involved in the non-deterministic core's even negative cycles. From the remaining rules, we defined the test part to contain constraints and rules which have instances that are involved in the core's odd negative cycles, and all remaining rules are then considered to represent definitions.

We implemented a tool based on metaprogramming which classifies rules in an answer-set program accordingly. The aim of this tool is to apply it for translating programs into (controlled) natural language, however by itself this tool can be useful for developers to quicker gain an understanding of a given program. Based on our definitions of generate-define-test, we introduced an algorithm which computes an order for translating rules of a program into CNL. We believe we found criteria that yield a clear and understandable explanation of the overall program.

Our approach differs from justification-based methods (Pontelli et al. 2009; Damásio et al. 2013) which aim at explaining the truth of certain atoms rather than explaining the whole program itself and which in turn are closely related to the debugging of logic programs (Pereira et al. 1993; Brain and De Vos 2005; Brain et al. 2007; Gebser et al. 2008; Oetsch et al. 2010; Oetsch et al. 2011; Shchekotykhin 2015).

As future work, we plan to automatically realise a translation of answer-set programs into controlled natural language. Moreover, we want to extend our methods to programs of broader syntactic classes, e.g., classes with language features like choice constructs or aggregate relations.

We also plan to investigate how our definition of the non-deterministic core relates to program normal forms as introduced by Brass et al. (1996; 2001) for the computation of the well-founded semantics. Note that the normal forms introduced by Costantini and Provetti (2005) might differ quite drastically from the original program, so they do not seem promising for our purposes of rule classification.

We think that optimisation of non-ground logic programs can profit from our work, e.g., Baral (2003) demonstrated that for many programs the efficiency can be improved by modifying the generate part such that conditions from the test part are already taken into account when generating candidate solutions, thereby shrinking the solver search space. Arguably, such optimisations can be performed easier and possibly automated if corresponding program parts can be identified automatically.

References

Apt, K. R., Blair, H. A., and Walker, A. 1988. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Chapter 2, 89–148.

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

Brain, M. and De Vos, M. 2005. Debugging logic programs under the answer set semantics. In *Proceedings of the 3rd International ASP Workshop* (*ASP 2005*). CEUR Workshop Proceedings. 142–152.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. 2007. Debugging ASP programs by means of ASP. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 31–43.

Brass, S. and Dix, J. 1996. Characterizing D-WFS: Confluence and iterated GCWA. In *Proceedings of the 4th European Workshop on Logics in Artificial Intelligence* (*JELIA 1996*), J. J. Alferes, L. M. Pereira, and E. Orlowska, Eds. Lecture Notes in Computer Science, vol. 1126. Springer, 268–283.

Brass, S., Dix, J., Freitag, B., and Zukowski, U. 2001. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming 1*, 497–538.

Costantini, S. and Provetti, A. 2005. Normal forms for answer sets programming. *Theory and Practice of Logic Programming 5*, 747–760.

Damásio, C. V., Analyti, A., and Antoniou, G. 2013. Justifications for logic programming. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2013*), P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 530–542.

Denecker, M., Lierler, Y., Truszczynski, M., and Vennekens, J. 2012. A Tarskian informal semantics for answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming* (*ICLP 2012*), A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 277–289.

Dimopoulos, Y. and Torres, A. 1996. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science 170,* 1, 209–244.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. 2000. Declarative problem-solving using the DLV system. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, 79–103.

Eiter, T., Ianni, G., and Krennwallner, T. 2009. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009*, S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 5689. Springer, 40–110.

Erdem, E. and Yeniterzi, R. 2009. Transforming controlled natural language biomedical queries into answer set programs. In *Proceedings of the Workshop on Current Trends in Biomedical Natural Language Processing* (*BioNLP 2009*). Association for Computational Linguistics, 117–124.

Gebser, M., Pührer, J., Schaub, T., and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (*AAAI 2008*). Vol. 8. 448–453.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic*

Programming (*ICLP/SLP 1988*), R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

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 Computional Logic 7,* 3, 499–562.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1–2, 39 – 54.

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*, 513–529.

Oetsch, J., Pührer, J., and Tompits, H. 2011. Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 134–147.

Pereira, L. M., Damásio, C. V., and Alferes, J. J. 1993. Diagnosis and debugging as contradiction removal. In *Proceedings of the 2nd International Workshop on Logic Programming and Non-monotonic Reasoning* (*LPNMR 1993*), L. M. Pereira and A. Nerode, Eds. The MIT Press, 316–330.

Pontelli, E., Son, T. C., and Elkhatib, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming 9*, 1–56.

Przymusinska, H. and Przymusinski, T. C. 1990. Weakly stratified logic programs. *Fundamenta Informaticae 13,* 1, 51–65.

Schwitter, R. 2012. Answer set programming via controlled natural language processing. In *Proceedings of the 3rd International Workshop on Controlled Natural Language* (*CNL 2012*). Lecture Notes in Computer Science, vol. 7427. Springer, 26–43.

Schwitter, R. 2013. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming 13,* 4-5, 487–501.

Shchekotykhin, K. M. 2015. Interactive query-based debugging of ASP programs. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence* (*AAAI 2015*), B. Bonet and S. Koenig, Eds. AAAI Press, 1597–1603.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems.* Vol. 1. Computer Science Press.

You, J.-H. and Yuan, L. Y. 1994. A three-valued semantics for deductive databases and logic programs. *Journal of Computer and System Sciences 49,* 2, 334–361.