Parametric Connectives in Disjunctive Logic Programming

Simona Perri and Nicola Leone
Department of Mathematics,
University of Calabria
I-87030 Rende (CS), Italy
E-mail: {leone,perri}@mat.unical.it

Disjunctive Logic Programming (DLP) is an advanced formalism for Knowledge Representation and Reasoning (KRR). DLP is very expressive in a precise mathematical sense: it allows to express every property of finite structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$). Importantly, the DLP encodings are often simple and natural.

In this paper, we single out some limitations of DLP for KRR, which cannot naturally express problems where the size of the disjunction is not known "a priori" (like N-Coloring), but it is part of the input. To overcome these limitations, we further enhance the knowledge modelling abilities of DLP, by extending this language by *Parametric Connectives (OR and AND)*. These connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property. We formally define the semantics of the new language, named $\mathrm{DLP}^{\bigvee,\bigwedge}$ and we show the usefulness of the new constructs on relevant knowledge-based problems. We address implementation issues and discuss related works.

1. Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [3,14,17,5,10,12,15,2]. The most widely accepted semantics for DLP is the *answer sets semantics* proposed by Gelfond and Lifschitz [10] as an extension of the stable model semantics of normal logic programs [9]. According to this semantics, a disjunctive logic program may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world. Disjunctive logic programs under answer sets semantics are very expressive. It was shown in [6,11] that, under this semantics, disjunctive logic programs capture the complexity class Σ_2^P (i.e., they allow us to express, in a precise mathematical sense, *every* property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP). As Eiter *et al.* [6] showed, the expressiveness of disjunctive logic programming has practical implications, since relevant practical problems can be represented by disjunctive logic programs, while they cannot be expressed by logic programs without disjunctions, given current complexity beliefs. Importantly, even problems of lower complexity can be often expressed more naturally by disjunctive programs than by programs without disjunction.

As an example, consider the well-known problem of 3-coloring, which is the assignment of three colors to the nodes of a graph in such a way that adjacent nodes have different colors. This problem is known to be NP-complete. Suppose that the nodes and the edges are represented by a set F of facts with predicates *node* (unary) and *edge* (binary), respectively. Then, the following DLP program allows us to determine the admissible ways of coloring the given graph.

$r_1: \ col(X,r) \lor col(X,y) \lor col(X,g) \mathrel{:-} node(X).$
$r_2: \ \mathrel{:-} edge(X,Y), col(X,C), col(Y,C).$

Rule r_1 above states that every node of the graph is colored **r**ed or **y**ellow or **g**reen, while r_2 forbids the assignment of the same color to any adjacent nodes. The minimality of answer sets guarantees that every node is assigned only one color. Thus, there is a one-to-one correspondence between the solutions of the 3-coloring problem and the answer sets of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set.

AI Communications
ISSN 0921-7126, IOS Press.

Despite the high expressiveness of DLP, there are several problems which cannot be encoded in DLP in a simple and natural manner. Consider, for instance, the generalization of the 3-coloring problem above, where the number of admissible colors is not known "a priori" but it is part of the input. This problem is called *N-Coloring*: Given a graph G and a set of N colors, find an assignment of the N colors to the nodes of G in such a way that adjacent nodes have different colors.

The most natural encoding for this problem would be obtained by modifying rule r_1 in the above encoding of 3-coloring. The head

$$col(X,r) \,\mathtt{v}\, col(X,y) \,\mathtt{v}\, col(X,g)$$

should be replaced by a disjunction of N atoms representing the N possible ways of coloring the node at hand. This encoding, however, cannot be done in a uniform way, since the number of colors is not known "a priori" but it is part of the input (the program should be changed for each number N of colors; while a uniform encoding requires the program to be fixed, and only the facts encoding the input to be varying).

To overcome these limitations, in this paper we enhance the knowledge modelling abilities of DLP, by extending this language by *Parametric Connectives (OR and AND)*. These connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property. For instance, by using parametric OR we obtain a simple and natural encoding of N-Coloring by modifying the above rule r_1 as follows:

$$\bigvee\{col(X,C) : color(C)\} \text{ :- } node(X).$$

(see Section 4.1 for a full discussion of this example). Intuitively, if the input colors are given by facts $color(c_1), \cdots, color(c_n)$, then the above rule stands for

$$col(X,c_1) \,\mathtt{v}\, \cdots \,\mathtt{v}\, col(X,c_n) \text{ :- } node(X)$$

Shortly, the main contribution of the paper are the following

- We extend Disjunctive Logic Programming by parametric connectives and formally define the semantics of the resulting language, named $\text{DLP}^{\vee,\wedge}$.
- We address knowledge representation issues, showing the impact of the new constructs on relevant KR problems.
- We discuss some implementation issues, providing the design of an extension of the **DLV** system to support $\text{DLP}^{\vee,\wedge}$.

The sequel of the paper is organized as follows. In Section 2, we provide the syntax and the semantics of the $\text{DLP}^{\vee,\wedge}$ language. In Section 3, we illustrate a methodology for declarative programming in standard DLP. In Section 4, we address knowledge representation issues in $\text{DLP}^{\vee,\wedge}$. In Section 5, we describe the implementation of the $\text{DLP}^{\vee,\wedge}$ language in the **DLV** system. In Section 6, we discuss related works. Finally, in Section 7, we draw our conclusions.

2. The $\text{DLP}^{\vee,\wedge}$ Language

In this section, we provide a formal definition of the syntax and the semantics of the $\text{DLP}^{\vee,\wedge}$ language.

2.1. Syntax

A variable or a constant is a *term*. A *standard atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *standard literal* is either a *standard positive literal* p or a *standard negative literal* $not\ p$, where p is a standard atom. A *standard conjunction* is $k_1, \cdots, k_n$ where each $k_1, \cdots, k_n$ is a standard literal. A *symbolic literal set* S is $\{L : Conj\}$, where L is a standard literal and $Conj$ is a standard conjunction; L is called the parameter of S and $Conj$ is called the domain of S; if L is a positive standard literal, S is called *positive symbolic literal set.* A *parametric AND literal* is $\bigwedge S$ where S is a symbolic literal set. A *parametric OR literal* is $\bigvee S$ where S is a positive symbolic literal set.

Example 1 $\bigvee\{a(X,Y) : q(X,Y), not\ r(Y)\}$ is a parametric OR literal and $\{a(X,Y) : q(X,Y), not\, r(Y)\}$ is a positive symbolic literal set. Intuitively, the above parametric OR literal stands for the disjunction of all instances of $a(X,Y)$ such that the conjunction $q(X,Y), not\ r(Y)$ is true. △

A *(disjunctive) rule* r is a syntactic of the following form:

$$a_1 \,\mathtt{v}\, \cdots \,\mathtt{v}\, a_n \text{ :- } l_1, \cdots, l_m. \qquad n \geq 0, m \geq 0$$

where $a_1, \cdots, a_n$ are standard positive literals or parametric OR literals and $l_1, \cdots, l_m$ are standard literals or parametric AND literals.
The disjunction $a_1 \,\mathtt{v}\, \cdots \,\mathtt{v}\, a_n$ is the *head* of r, while the conjunction $l_1, \cdots, l_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head literals, and by $B(r)$ the set $\{l_1, ..., l_m\}$ of the body literals. An *(integrity) constraint* is a rule with an empty head.

A DLP$^{\bigvee,\bigwedge}$ *program* $\mathcal{P}$ is a finite set of rules. A $\neg$-free (resp., v-free) program is called *positive* (resp., *normal*). A program where no parametric literals appear is called (*standard*) *DLP program.* A term, an atom, a literal, a rule, or a program are *ground* if no variables appear.

2.2. Syntactic Restrictions and Notation

A variable X appearing solely in a parametric literal of a rule r is a *local variable* of r. The remaining variables of r are called *global variables* of r.

Example 2 Consider the following rule

$p(Y, Z)$:- $\bigwedge\{q(X, Y) : a(X, Z))\}, t(Y), r(Z).$

X is the only local variable, while Y and Z are global variables. △

Safety

A rule r is *safe* if the following conditions hold:

(i) each global variable of r appears in a positive standard literal occurring in the body of r;

(ii) each local variable of r appearing in a symbolic set $\{L : Conj\}$, also appears in a positive literal in $Conj$.

A program is *safe* if all of its rules are safe.

Example 3 Consider the following rules:

$\bigvee\{p(X, Y) : q(Y)\}$:- $r(X)$.
$p(X, Z)$:- $\bigwedge\{q(X, Y) : a(X))\}, s(X, Z)$.
$p(X)$:- $\bigwedge\{q(X, Y) : a(X)\}, t(Y)$.

The first rule is safe, while the second is not, since the local variable Y violates condition (ii). The third rule is not safe either, since the global variable X violates condition (i). △

Stratification

A DLP$^{\bigvee,\bigwedge}$ program $\mathcal{P}$ is *p-stratified* if there exists a function $|| \; ||$, called *level mapping*, from the set of (standard) predicates of $\mathcal{P}$ to ordinals, such that for each pair a and b of (standard) predicates of $\mathcal{P}$, and for each rule $r \in \mathcal{P}$ the following conditions hold:

(i)for each parametric literal γ of r, if a appears in the parameter of γ and b appears in the domain of γ then $||b|| < ||a||$, and
(ii) if a appears in the head of r, and b occurs in a standard atom in the body of r, then $||b|| \leq ||a||$.

Example 4 Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$p(X)$:- $q(X), \bigwedge\{q(Y) : a(X, Y), b(X)\}$.
$q(X)$:- $p(X), b(X)$.

The program is p-stratified, as the level mapping $||a|| = ||b|| = 1 \quad ||p|| = ||q|| = 2$ satisfies the required conditions. If we add the rule $b(X)$:- $p(X)$, then no legal level-mapping exists and the program becomes p-unstratified. △

From now on, throughout this paper, we assume that all rules of a DLP$^{\bigvee,\bigwedge}$ $\mathcal{P}$ are safe and p-stratified.

2.3. Semantics

Program Instantiation. Given a DLP$^{\bigvee,\bigwedge}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$.

A *substitution* is a mapping from a set of variables to the set $U_{\mathcal{P}}$ of the constants appearing in the program $\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for* r; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{L : Conj\}$, the *instantiation of set* S is the following ground set of pairs $S' = \{\langle\gamma(L) : \gamma(Conj)\rangle \mid \gamma$ is a local substitution for $S\}$[1]; S' is called *ground literal set.*

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation S'. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 5 Consider the following program $\mathcal{P}_1$:

$p(1) \vee q(2, 2)$.
$p(2) \vee q(2, 1)$.
$s(X)$:- $p(X), \bigwedge\{a(Y) : q(X, Y)\}$.

The instantiation $Ground(\mathcal{P}_1)$ is the following:

[1]Given a substitution σ and a DLP$^{\bigvee,\bigwedge}$ object Obj (rule, conjunction, set, etc.), with a little abuse of notation, we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

$p(1) \lor q(2,2).$
$p(2) \lor q(2,1).$
$s(1) \text{ :- } p(1), \bigwedge\{\langle a(1) : q(1,1)\rangle, \langle a(2) : q(1,2)\rangle\}.$
$s(2) \text{ :- } p(2), \bigwedge\{\langle a(1) : q(2,1)\rangle, \langle a(2) : q(2,2)\rangle\}.$

△

Interpretation and models. An *interpretation* for a $\mathrm{DLP}^{\bigvee,\bigwedge}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$.

A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\neg A$ is *true* w.r.t. I if A is false w.r.t. I; otherwise $\neg A$ is false w.r.t. I.

Besides assigning truth values to the standard ground literals, an interpretation provides the meaning also to (ground)literal sets, and to (the instantiation of) parametric literals. Let S be a (ground) literal set. The valuation $I(S)$ of S w.r.t. I is the set

$$\{L \mid (L : conj \in S) \land (conj\ is\ true\ w.r.t\ I)\}.$$

Given a parametric OR literal $\bigvee S$, let S' be the instantiation of S. Then $\bigvee S'$ is true w.r.t I if at least one of the standard literals in $I(S')$ is true w.r.t I. Similarly, given a parametric AND literal $\bigwedge S$, let S' be the instantiation of S. $\bigwedge S'$ is true w.r.t I if all the standard literals in $I(S')$ are true w.r.t I.

Example 6 Let $U_{\mathcal{P}}$ be the set {1,2} and I the interpretation $\{p(1), p(2), a(1,2), a(2,1), b(1), b(2)\}$. Consider the parametric AND literal

$$\bigwedge S = \bigwedge\{p(X) : a(X,Y), b(X)\}$$

Then the instantiation of S is

$$S' = \{\langle p(1) : a(1,1), b(1)\rangle, \langle p(1) : a(1,2), b(1)\rangle, \langle p(2) : a(2,1), b(2)\rangle, \langle p(2) : a(2,2), b(2)\rangle\}$$

and its value w.r.t I is $I(S') = \{p(1), p(2)\}$. $\bigwedge S'$ is true w.r.t. I because both $p(1)$ and $p(2)$ are true w.r.t I. △

Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if at least one literal of $H(r)$ is true w.r.t I. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M.

Answer Sets. First we define the answer sets of standard positive programs (i.e. without parametric literals). Then, we give a reduction from full $\mathrm{DLP}^{\bigvee,\bigwedge}$ programs (i.e. containing negation as failure and parametric literals) to standard positive programs. Such a reduction is used to define answer sets of $\mathrm{DLP}^{\bigvee,\bigwedge}$ programs.

An interpretation $I \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive standard program without parametric literals), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$. An interpretation $I \subseteq B_{\mathcal{P}}$ is an *answer set* for a standard positive program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.[2]

Example 7 The positive program

$a \lor b \lor c.$

has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$. The program

$a \lor b \lor c.$
$\text{:- } a.$

has the answer sets $\{b\}$ and $\{c\}$. Finally, the positive program

$a \lor b \lor c.$
$\text{:- } a.$
$b \text{ :- } c.$
$c \text{ :- } b.$

has the single answer set $\{b, c\}$. △

We next extend the notion of *Gelfond-Lifschitz transformation*[10] to $\mathrm{DLP}^{\bigvee,\bigwedge}$ programs. To this end, we introduce a new transformation δ.

Given a set $F = \{f_1, \cdots, f_n\}$ of ground literals, we define the following transformation δ:

$$\delta(\bigvee F) = f_1 \lor \cdots \lor f_n \quad \delta(\bigwedge F) = f_1, \cdots, f_n$$

The *reduct* or *Gelfond-Lifschitz transform* of a $\mathrm{DLP}^{\bigvee,\bigwedge}$ program $\mathcal{P}$ w.r.t. a set $I \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^I$, obtained from $Ground(\mathcal{P})$ by the following steps:

1. Replace each instance $\bigvee S'$ of a parametric OR literal $\bigvee S$ by $\delta(\bigvee I(S'))$.
2. Replace each instance $\bigwedge S'$ of a parametric AND literal $\bigwedge S$ by $\delta(\bigwedge I(S'))$.

[2] Note that we only consider *consistent answer sets*, while in [10] also the inconsistent set of all possible literals can be a valid answer set.

3. Delete all rules $r \in \mathcal{P}$ for which a negative literal in $B(r)$ is false w.r.t. I.
4. Delete the negative literals from the remaining rules.

An answer set of a program $\mathcal{P}$ is a set $I \subseteq B_{\mathcal{P}}$ such that I is an answer set of $Ground(\mathcal{P})^I$.

Example 8 Consider the following DLP$^{\vee,\wedge}$ program $\mathcal{P}_1$

$p(1).\quad a(1).\quad a(2).$

$q \mathrel{:-} \bigwedge\{not\ p(X) : a(X)\}.$

and $I = \{p(1)\,,\ a(1)\,,\ a(2)\}$. The instantiation of the set $\{not\ p(X) : a(X)\}$ is

$$S' = \{\langle not\ p(1) : a(1)\rangle, \langle not\ p(2) : a(2)\rangle\}.$$

By evaluating S' w.r.t I we obtain

$$I(S') = \{not\ p(1),\ not\ p(2)\}$$

Now, by applying step (2) of the reduct we obtain the program

$p(1).\quad a(1).\quad a(2).$

$q \mathrel{:-} not\ p(1), not\ p(2).$

and then, by applying step (3) we delete the rule, as $not\ p(1)$ is false, obtaining

$$\mathcal{P}_1^I = \{p(1).\,,\ a(1).\,,\ a(2).\}.$$

Obviously, I is an answer set of $\mathcal{P}_1^I$ and then, it is also an answer set for $\mathcal{P}_1$.

Now, consider the program $\mathcal{P}_2$

$p(1).$

$\bigvee\{b(X) : a(X)\}.$

and $J = \{p(1)\}$. We have that the instantiation of $\{b(X) : a(X)\}$ is

$$S' = \{\langle b(1) : a(1)\rangle\}.$$

and $J(S') = \emptyset$. By applying step (1) of the reduct, we obtain an empty disjunction which evaluates false in any interpretation. Then, the reduct $\mathcal{P}_2^J$ has no answer sets and so J it is not an answer set of $\mathcal{P}_2$. Note that $\mathcal{P}_2$ has no answer sets. △

3. Declarative Programming in Standard DLP

3.1. The **GC** *Declarative Programming Methodology*

The standard DLP language can be used to encode problems in a highly declarative fashion, following a "**GC**" (Guess/Check) paradigm. In this section, we will describe this technique and we then illustrate how to apply it on a number of examples. Many problems, also problems of comparatively high computational complexity (that is, even Σ_2^P-complete problems), can be solved in a natural manner with DLP by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are even more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so uniformly over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem **P**, a **GC** program $\mathcal{P}$ for **P** consists of the following two main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, in a way such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program tests whether a solution candidate is in fact an admissible solution, such that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the solutions for the problem instance I.

The two layers above can also use additional auxiliary predicates, which can be seen as a background knowledge.

In general, we may allow both $\mathcal{G}$ and $\mathcal{C}$ to be arbitrary collections of rules in the program, and it may depend on the complexity of the problem which kinds of rules are needed to realize these parts (in particular, the checking part); we defer this discussion to a later point in this chapter.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., all checking is integrated into the guessing part such that solution candidates are always solutions. However, in general the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For many problems, however, a natural **GC** program can be designed, in which the two parts are clearly identifiable and have a simple structure:

- The guessing part $\mathcal{G}$ consists of some disjunctive rules which "guess" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints which check the admissibility of S.

All two layers may also use additional auxiliary predicates, which are defined by normal stratified rules. Such auxiliary predicates may also be associated with the guess for a candidate, and defined in terms of other guessed predicates, leading to a more "educated guess" which reduces blind guessing of auxiliary predicates; this will be seen in some examples below.

Thus, the disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

Remark. The **GC** programming methodology has positive implications also from the Software Engineering viewpoint. Indeed, the modular program structure in **GC** allows us to develop programs incrementally providing support for simpler testing and debugging activities. Indeed, one first writes the Guess module $\mathcal{G}$ and tests that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one deals with the Check module and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ are the admissible problem solutions.

3.2. *Applications of the* **GC** *Programming Technique*

In this section, we illustrate the declarative programming methodology described in Section 3.1 by showing its application on three standard problems from graph theory.

3.2.1. Hamiltonian Path

Consider now a classical NP-complete problem in graph theory, namely *Hamiltonian Path.*

Definition 1 (HAMPATH) *Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?* □

Suppose that the graph G is specified by using predicates *node* (unary) and *arc* (binary), and the starting node is specified by the predicate *start* (unary). Then, the following **GC** program $\mathcal{P}_{hp}$ solves the problem HAMPATH.

$$\left.\begin{array}{l} inPath(X,Y) \vee outPath(X,Y) \\ \quad :\!-\ start(X), arc(X,Y). \\ inPath(X,Y) \vee outPath(X,Y) \\ \quad :\!-\ reached(X), arc(X,Y). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} :\!-\ inPath(X,Y), inPath(X,Y1), Y <> Y1. \\ :\!-\ inPath(X,Y), inPath(X1,Y), X <> X1. \\ :\!-\ node(X), \text{not } reached(X), \text{not } start(X). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} reached(X) :\!-\ inPath(Y,X). \end{array}\right\} \textbf{Auxiliary Predicate}$$

The two disjunctive rules guess a subset S of the given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian Path. Here, an auxiliary predicate *reached* is used, which is associated with the guessed predicate *inPath* using the last rule.

The predicate *reached* influences through the second rule the guess of *inPath*, which is made somehow inductively: Initially, a guess on an arc leaving the starting node is made by the first rule, and then a guess on an arc leaving from a reached node by the second rule, which is repeated until all reached nodes are treated.

In the Checking Part, the first two constraints check whether the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. This constraint also ensures that this subgraph is connected.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in G that starts at node $v_0 = a$, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is a Hamiltonian Cycle, and by dropping the last arc, we have a Hamiltonian Path.

Thus, given a set of facts $\mathcal{F}$ for *node*, *arc*, and *start*, specifying the problem input, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the input graph has a Hamiltonian Path. Thus, the above program correctly encodes the decision problem of deciding whether a given graph admits an Hamiltonian Path or not.

3.2.2. N-Coloring

Now we consider another classical NP-complete problem from graph theory, namely *N-Coloring.*

Definition 2 (N-COLORING) *Given a graph $G = (V, E)$, a N-Coloring of G is an assignment of one, among N colors, to each vertex in V, in such a way that every pair of vertices joined by an edge in E have different colors.* □

Suppose that the graph G is represented by a set of facts with predicates *vertex* (unary) and *edge* (binary), respectively. Then, the following DLP program $\mathcal{P}_{col}$ determines the admissible ways of coloring the given graph.

$$\left.\begin{array}{l} col(X,I) \vee not\_col(X,I) \text{ :- } \\ \quad vertex(X), color(I). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:- } col(X,I), col(Y,I), edge(X,Y). \\ \text{:- } col(X,I), col(X,J), I <> J. \\ \text{:- } vertex(X), \text{not } colored(X). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} colored(X) \text{ :- } col(X,I). \end{array}\right\} \textbf{Auxiliary Predicate}$$

$col(X, I)$ says that vertex X is assigned to color I and $not\_col(X, I)$ that it is not. The disjunctive rule guesses a graph coloring; the constraints in the checking part verify that the guessed coloring is a legal N-Coloring. In particular the first constraint asserts that two joined vertices cannot have the same color, while the remaining two constraints impose that each vertex is assigned to exactly one color.

The answer sets of $\mathcal{P}_{col}$ are all the possible legal N-Colorings of the graph. That is, there is a one-to-one correspondence between the solutions of the N-Coloring problem and the answer sets of $\mathcal{P}_{col}$. The graph is N-colorable if and only if there exists one of such answer sets.

3.2.3. Maximal Independent Set

Another classical problem in graph theory is the independent set problem.

Definition 3 (Maximal Independent Set) *Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. The set I is* independent *if whenever $i, j \in I$ then there are no edges between i and j in E. An independent set I is* maximal *if no superset of I is an independent set.* □

Suppose that the graph G is represented by a set of facts F with predicates *node* (unary) and *edge* (binary). The following program $\mathcal{P}_{IndSet}$ computes the maximal independent sets of G:

$$\left.\begin{array}{l} r_1: \; in(X) \vee out(X) \text{ :- } node(X). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} c_1: \; \text{:- } in(X), in(Y), edge(X,Y). \\ c_2: \; \text{:- } out(X), not \; toBeExcluded(X). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} r_2: \; toBeExcluded(X) \text{ :- } in(Y), edge(X,Y). \end{array}\right\} \textbf{Auxiliary Predicate}$$

The rule r_1 guesses a set of vertices; $in(X)$ means that node X belongs to the set while $out(X)$ means that it does not. Then, the integrity constraint c_1 verifies that the guessed set is independent. In particular, it says that it is not possible that two nodes joined by an edge belong to the set.

Note that the answer sets of $F \cup \{r_1, c_1\}$ correspond exactly to the independent sets of G.

The maximality of the set is enforced by constraint c_2 using the auxiliary predicate $toBeExcluded$. A node X has to be excluded by the set because a node connected to it is already in the set. Then c_2 says that it is not possible that a node is out of the set if there is no reason to exclude it.

3.3. N-Queens

Next we consider the well-known N-Queens problem.

Definition 4 (N-QUEENS) *Place N queens on a N*N chess board such that the placement of no queen constitutes an attack on any other. A queen attacks another if it is in the same row, in the same column, or on the same diagonal.*

□

Suppose that rows and columns are represented by means of facts $row(1), \cdots, row(N)$ and $column(1), \cdots, column(N)$ respectively. Then, we encode the N-Queens problem as follows.

$$\left.\begin{array}{l} r_1: \; q(X,Y) \vee not\_q(X,Y) \text{ :- } \\ \qquad\qquad row(X), column(Y). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} c_1: \; \text{:- } q(X,Y), q(X,Z), Y \neq Z. \\ c_2: \; \text{:- } q(X,Y), q(Z,Y), X \neq Z. \\ c_3: \; \text{:- } q(X_1,Y_1), q(X_2,Y_2), \\ \qquad\quad X_2 = X_1 + K, Y_2 = Y_1 + K, K > 0. \\ c_4: \; \text{:- } q(X_1,Y_1), q(X_2,Y_2), \\ \qquad\quad X_2 = X_1 + K, Y_1 = Y_2 + K, K > 0. \\ c_5: \; \text{:- } not \; assigned\_row(X), row(X). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} r_2: \; assigned\_row(X) \text{ :- } q(X,\_). \end{array}\right\} \textbf{Auxiliary Predicate}$$

We represent queens by atoms of the form $q(X, Y)$. Atom $q(X, Y)$ is true if a queen is placed in the chess board at row X and column Y. The disjunctive rule guesses the position of the queens; then constraints c_1 and c_2 assert that two queens cannot be on the same row or on the same column, respectively. Constraints c_3 and c_4 assert that two queens cannot be on the same diagonal (from top left to bottom right (constraint c_3) and from top right to bottom left (constraint c_4)). The last constraint enforces that all N queens have been placed on the board (as each row has a queen on it).

4. Knowledge Representation by DLP$^{\vee,\wedge}$

In this section, we show how DLP extended by parametric connectives can be used to encode relevant problems in a natural and elegant way. To this end, we encode in DLP$^{\vee,\wedge}$ the problems previously represented in standard DLP in section 3 by exploiting parametric connectives. [3]

4.1. N-Coloring

In the previous section we showed an encoding for the N-Coloring problem, following the **GC** paradigm. Now, we show how the extension of DLP with parametric connectives allows us to represent the N-Coloring problem in a more intuitive way by simply modifying the elegant encoding of 3-colorability described in the Introduction.

Suppose again that the graph in input is represented by predicates *vertex* (unary) and *edge* (binary) and the set of N admissible colors is provided by a set of facts $color(c_1), \cdots, color(c_N)$. Then, the following DLP$^{\vee,\wedge}$ program computes the N-Colorings of the graph.

$$
\begin{array}{ll}
r: & \bigvee\{col(X,C) : color(C)\} \text{ :- } vertex(X). \\
c: & \text{ :- } \; col(X,C), col(Y,C), edge(X,Y), X \neq Y.
\end{array}
$$

Rule r guesses all possible N-Colorings. It contains in the head a parametric literal representing the disjunction of all the atoms $col(X, c_1), \cdots, col(X, c_N)$, where $c_1, \cdots, c_N$ are the N colors (i.e. the disjunction of all the atoms representing the possible ways to color X). For each vertex v, the following ground rule belongs to the instantiation of the program:

$$
\bigvee\{\langle col(v,c_1) : color(c_1)\rangle, \cdots, \langle col(v,c_N) : color(c_N)\rangle\} \\
\text{ :- } vertex(v).
$$

[3] We omit Hamiltonian Path since parametric connectives are not useful for its representation.

Since vertex v and $color(c_1), \cdots, color(c_N)$ are always true, the above rule stands for the following disjunction

$$col(v, c_1) \vee \cdots \vee col(v, c_N)$$

The integrity constraint c simply checks that the N-Coloring is correct, that is, adjacent nodes must always have different colors.

4.2. Maximal Independent Set

Another problem which can be easily encoded in a more intuitive way by DLP$^{\vee,\wedge}$ is Maximal Independent Set, shown in section 3.2.3. Indeed, this problem can be represented by the following *one-rule* encoding.

$$in(X) \text{ :- } node(X), \bigwedge\{\text{not } in(Y) : arc(X,Y)\}.$$

As usual, the graph in input is encoded by predicates *node* and *arc* and the atom $in(X)$ means that node X belongs to the set. Intuitively, such rule says that node X belongs to the independent set if, for each node Y which is connected to it, Y does not belong to the set. In particular, the parametric AND literal $\bigwedge\{not\ in(Y) : arc(X,Y)\}$ is the conjunction of all the literals $not\ in(Y)$ such that there exists an edge between X and Y.

Note that, differently from the **GC** encoding shown in the previous section this formulation does not need the predicate $out(X)$ and the auxiliary predicate $toBeExcluded(X)$ used to mark the nodes that have to be excluded by the set.

It is worthwhile noting that we do not need further rules to express maximality property, which, indeed, comes for free.

4.3. N-Queens

We next provide a DLP$^{\vee,\wedge}$ encoding of the N-Queens problem.

$$
\begin{array}{ll}
r: & \bigvee\{q(X,Y) : column(Y)\} \text{ :- } row(X). \\
c_1: & \text{ :- } q(X,Y), q(Z,Y), X \neq Z. \\
c_2: & \text{ :- } q(X_1,Y_1), q(X_2,Y_2), \\
 & \quad X_2 = X_1 + K, Y_2 = Y_1 + K, K > 0. \\
c_3: & \text{ :- } q(X_1,Y_1), q(X_2,Y_2), \\
 & \quad X_2 = X_1 + K, Y_1 = Y_2 + K, K > 0.
\end{array}
$$

As in the DLP encoding, queens are represented by atoms of the form $q(X,Y)$ and rows and columns are represented by means of facts $row(1), \cdots, row(N)$ and $column(1), \cdots, column(N)$.

The disjunctive rule guesses the position of the queens; in particular, for each row X, we guess the column where the queen has to be placed. Constraint c_1 asserts that two queens cannot be on the same column, while the fact that two queens cannot be on the same row is ensured by the guess. Furthermore, constraints c_2 and c_3 assert that two queens cannot be placed on the same diagonal, as in the standard DLP encoding. It is worthwhile noting that, the standard DLP encoding needs a further constraint ensuring that all the queens have been placed on the chess board while, by using the $\text{DLP}^{\vee,\wedge}$ encoding, this constraint can be omitted.

5. Implementation Issues

In this section we illustrate the design of the implementation of the parametric connectives in the **DLV** system. We first briefly describe the architecture of **DLV** and we then discuss the impact of the implementation of parametric connectives in **DLV**.

5.1. **DLV** *Architecture*

An outline of the general architecture of the **DLV** system is depicted in Figure 1. The general flow in this picture is top-down. The principal User Interface is command-line oriented, but also a Graphical User Interface (GUI) for the core systems and most front-ends is available. Subsequently, front-end transformations might be performed. Input data can be supplied by regular files, and also by relational databases. The **DLV** core then produces answer sets one at a time, and each time an answer set is found, the "Filtering" module is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The **DLV** core consists of three major components: the "Intelligent Grounding", the "Model Generator", and the "Model Checker" modules that share a principal data structure, the "Ground Program". The "Ground Program" is created by the "Intelligent Grounding" using differential (and other advanced) database techniques together with suitable data structures, and used by the "Model Generator" and the "Model Checker". The Ground Program is guaranteed to have exactly the same answer sets as the original program.

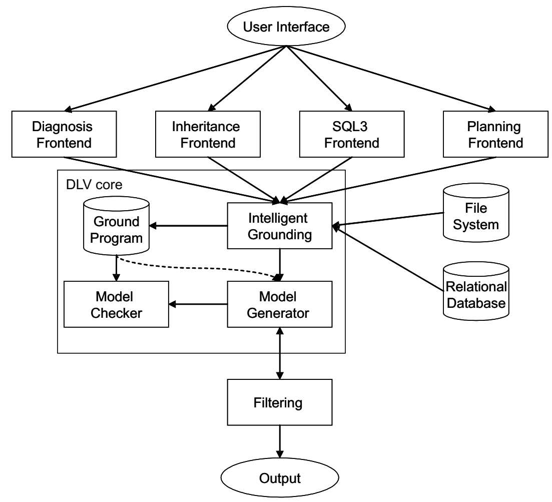

Fig. 1. The System Architecture of **DLV**

For some syntactically restricted classes of programs (e.g. stratified programs), the "Intelligent Grounding" module already computes the corresponding answer sets.

For harder problems, most of the computation is performed by the "Model Generator" and the "Model Checker". Roughly, the former produces some candidate answer sets (models) [7,8], the stability and minimality of which are subsequently verified by the latter.

The "Model Checker" (MC) verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is known to be co-NP-complete. However, MC exploits the fact that minimal model checking — the hardest part — can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs (see Appendix A).

5.2. *Efficient Implementation of Parametric Connectives in* **DLV**

Implementing the full $\text{DLP}^{\vee,\wedge}$ language in the **DLV** system, would have a strong impact on **DLV** requiring many changes to all modules of the **DLV** core, including the "Model Generator" (MG) and the "Model Checker" (MC). Making such changes would increase the complexity of the code and it could lead to an efficiency loss, because, besides the standard literals, a new kind of literals, should be manipulated. In order to obtain an efficient implementation, we impose a syntactic restriction on the domain predicates (i.e. on the predicates appearing in the conjunction on the right side of symbolic sets) that allows us to translate parametric literals into standard conjunctions and disjunctions during the instantiation. In this way, the grounding produces standard DLP programs and no changes to "Model Generator" and "Model Checker" are necessary.

In particular, we impose that such predicates are normal (disjunction-free) and stratified [1]. For each symbolic set $S = \{L : Conj\}$, all domain literals of S in $Conj$ are instantiated before than dealing with the parameter L. Thus, when the symbolic set S has to be grounded all the domain predicates of S are fully instantiated and ready to be used. Thanks to the imposed restrictions on the domain predicates (which are normal, stratified predicates), their truth values are fully decided, that is they are either true or false. Consequently, we can limit the instantiation of S only to the "useful" atoms, that is, the instances of L such that the corresponding instances of $Conj$ are true.

Example 9 Consider the program

$a(1).\ \ a(2).\ \ a(3).\ \ a(4).\ \ c(1).$
$b(X) :\!-\, a(X),\ not\ c(X).$
$\bigvee\{p(X) : b(X)\}.$

The grounding procedure first instantiates the rule $b(X) :\!-\, a(X),\ not\ c(X)$, and generates the instances $b(2),\ b(3),\ b(4)$ for the domain predicate b. Next, it considers $\bigvee\{p(X) : b(X)\}$ generating the standard disjunction $p(2) \,\mathrm{v}\, p(3) \,\mathrm{v}\, p(4)$. △

6. Related Work

We are not aware of other proposals for extending DLP by parametric connectives. However, our work has some similarity with other extensions of logic programming by other forms of nested operators like for instance the nested expressions defined in [13].

Our parametric disjunction has some similarity also with weight constraints of Smodels [16].

A weight constraint is an expression of the form $l\{L : D\}u$. The integer numbers l and u represent the lower and the upper bound of the constraint, respectively. $L : D$ is called conditional literal, L is a standard literal and the conditional part D is a domain predicate which is required to be normal and stratified. Intuitively, the instantiation of $\{L : D\}$ is the multiset S of all instances of L such that the corresponding instances of D are true. Given an interpretation I, $l\{L : D\}u$ is true w.r.t I if the number of atoms in S which are true w.r.t. I is in the range $[l..u]$. Thus, the parametric OR literal

$\bigvee\{col(X, C) : color(C)\}$

is similar to the Smodels weight constraint

$1\{col(X, C) : color(C)\}1$

However, it is worthwhile noting that the above Smodels construct derives exactly one atom while the semantics of $\mathrm{DLP}^{\vee,\wedge}$ follows the standard interpretation of disjunction (at least one atom is derived). For instance, the $\mathrm{DLP}^{\vee,\wedge}$ program P_1

$c(1).\ \ c(2).$
$\bigvee\{a(X) : c(X)\}.$
$a(1) :\!-\, a(2).$
$a(2) :\!-\, a(1).$

has the single answer set $\{a(1),\ a(2),\ c(1),\ c(2)\}$. On the contrary, the Smodels program P_1'

$c(1).\ \ c(2).$
$1\{a(X) : c(X)\}1.$
$a(1) :\!-\, a(2).$
$a(2) :\!-\, a(1).$

has no answer sets. We could simulate the above $\mathrm{DLP}^{\vee,\wedge}$ behavior in Smodels by rewriting the weight constraint into $1\{a(X) : c(X)\}2$. However, this solution is specific to this particular example, and it does not work in general. For instance, consider the following programs P_2 and P_2'

$P_2:\ c(1).\ \ c(2).$ $\quad\bigvee\{a(X) : c(X)\}.$

$P_2':\ c(1).\ \ c(2).$ $\quad 1\{a(X) : c(X)\}2.$

P_2 has two answer sets: $\{a(1),\ c(1),\ c(2)\}$ and $\{a(2),\ c(1),\ c(2)\}$; while P_2' has three answer sets: $\{a(1),\ c(1),\ c(2)\}$, $\{a(2),\ c(1),\ c(2)\}$ and $\{a(1),\ a(2),\ c(1),\ c(2)\}$.

7. Conclusions

We have proposed $\mathrm{DLP}^{\vee,\wedge}$, an extension of DLP by parametric connectives. These connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property enhancing the knowledge modelling abilities of DLP.

We have formally defined the semantics of the new language, and we have shown the usefulness of $\mathrm{DLP}^{\vee,\wedge}$ on relevant knowledge-based problems.

It is worthwhile noting that the expressive power of the language (intended as the set of problems which can be encoded in the language) does not change, because its data complexity does not increase. However, a number of problems can be represented in a more natural way by using the parametric connectives in $\text{DLP}^{\vee,\wedge}$.

Ongoing work concerns the implementation of parametric literals in the **DLV** system following the design presented in section 5. Further work concerns an experimentation activity devoted to the evaluation of the impact of parametric connectives on system efficiency. We believe that the conciseness of the encoding obtained through parametric literals in some cases, like for instance N-Coloring and N-Queens, brings a positive gain on the efficiency of the evaluation.

Furthermore, it would be interesting to investigate whether there exists a uniform translation from $\text{DLP}^{\vee,\wedge}$ to standard DLP or not.

Acknowledgments

This work was supported by the European Commission under projects INFOMIX, IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and IST-2001-37004 WASP.

References

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[2] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2002.

[3] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[4] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[5] T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[6] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[7] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

[8] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

[9] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[10] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[11] G. Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.

[12] V. Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[13] V. Lifschitz, L. R. Tang, and H. Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):369–389, 1999.

[14] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* The MIT Press, Cambridge, Massachusetts, 1992.

[15] J. Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[16] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[17] B. Wolfinger, editor. Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Berlin, August 1994. German Society for Computer Science (GI), Springer. 13^{th} IFIP World Computer Congress, Hamburg, Germany.

Appendix

A. Head Cycle Free Programs

An interesting subclass of disjunctive logic programs is given by head-cycle free programs.

Definition 5 A program $\mathcal{P}$ is called *head-cycle free (HCF)* [4], if there is a level mapping $||\ ||_h$ of $\mathcal{P}$ such that, for every rule r of $\mathcal{P}$,

1. For any positive literal l in the body of r, and for any l' in the head of r , $||l||_h \leq ||l'||_h$;
2. For any pair l, l' in the head of r, $||l||_h \neq ||l'||_h$.

□

Example 10 Consider the following program P_1.

$P_1 : a \vee b.$
$\quad a \text{ :- } b.$

It is easy to see that P_1 is head-cycle free; an admissible level mapping for P_1 is given by $||a||_h = 2$ and $||b||_h = 1$. Consider now the program

$$P_2 = P_1 \cup \{b \text{ :- } a.\}$$

Program P_2 is not head-cycle free, since a and b should belong to the same level by Condition (1) of Definition 5 while they cannot by Condition (2) of that definition. Note, however, that P_2 is stratified. △

Intuitively, head cycle freeness forbids recursion through disjunction.