Research Article

Extending Well-Founded Semantics with Clark's Completion for Disjunctive Logic Programs

Juan Carlos Nieves¹ and Mauricio Osorio²

¹Department of Computing Science, Umeå University, 901 87 Umeå, Sweden

²Departamento de Actuaría, Física y Matemáticas, Universidad de las Américas Puebla, Sta. Catarina Mártir, 72820 Cholula, PUE, Mexico

Correspondence should be addressed to Juan Carlos Nieves; jcnieves@cs.umu.se

Received 28 June 2017; Revised 9 October 2017; Accepted 24 October 2017; Published 1 March 2018

Academic Editor: Fabrizio Riguzzi

Copyright © 2018 Juan Carlos Nieves and Mauricio Osorio. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In this paper, we introduce new semantics (that we call D3-WFS-DCOMP) and compare it with the stable semantics (STABLE). For normal programs, this semantics is based on *suitable* integration of the well-founded semantics (WFS) and the Clark's completion. D3-WFS-DCOM has the following appealing properties: First, it agrees with STABLE in the sense that it never defines a nonminimal model or a nonminimal supported model. Second, for normal programs it extends WFS. Third, every stable model of a disjunctive program P is a D3-WFS-DCOM model of P . Fourth, it is constructed using transformation rules accepted by STABLE. We also introduce second semantics that we call D2-WFS-DCOMP. We show that D2-WFS-DCOMP is equivalent to D3-WFS-DCOMP for normal programs but this is not the case for disjunctive programs. We also introduce third new semantics that supports the use of implicit disjunctions. We illustrate how these semantics can be extended to programs including explicit negation, default negation in the head of a clause, and a lub operator, which is a generalization of the aggregation operator *setof* over arbitrary complete lattices.

1. Introduction

One of the most well-known semantics for logic programming is the stable semantics (STABLE) [1, 2]. However, it is well known that very often STABLE is too strong, namely, that there are programs that do not possess stable models [3–5]. Let us consider the following example.

Example 1. Let P be a disjunctive logic program:

$$\begin{aligned} d \vee e &\leftarrow \neg a. \\ c &\leftarrow c. \\ b &\leftarrow a. \\ a &\leftarrow b. \\ a &\leftarrow \neg b, \neg c. \end{aligned}$$

In this program there are not stable models; however if we use our D3-WFS-DCOMP semantics there is a model of the program P , that is, $\{a, b\}$. We can argue that this intended model makes sense as follows: since a tautology such as $c \leftarrow c$

should not matter (in our point of view, what makes the given tautology to be specially harmless is that c does not occur in the head of any other clause), we can delete it getting a semantically equivalent program that we call P_1 . Then, since c does not appear in the head of any clause of P_1 , we can derive $\neg c$ and so we can delete $\neg c$ in the last clause to get a semantically equivalent program P_3 . (Note that what we have done so far in this example is accepted by the STABLE semantics.) Then, since we have that

$$\begin{aligned} a &\leftarrow b \\ a &\leftarrow \neg b \end{aligned}$$

are clauses in P_3 and by reasoning by cases, we derive a . (Reasoning by cases is not accepted in STABLE.) Then, we derive b by simple modus ponens. Finally, d and e can be considered false since a is true.

It has been even argued that STABLE does not define the intended models of a program [6]. It is also possible that there is not best semantics [7]:

“The fact each approach to semantics of negation has its own strength and weakness suggests that there is probably not a *best* semantics for logic programs.”

Nevertheless, STABLE satisfies interesting logic properties as it is shown in [1, 8]. In this paper we study the stable semantics and exhibit some properties that it satisfies. Also we point out some problems with this semantics. With this base, we study some variations to the semantics proposed by [9, 10]. The obtained semantics, that we call D3-WFS-DCOMP, satisfies the following main properties: First, it agrees with STABLE in the sense that it never defines a nonminimal model or a non-minimal supported model. Second, for normal programs it extends the well-founded semantics (WFS) [11]. Third, every stable model of a disjunctive program P is a D3-WFS-DCOMP model of P . Fourth, it is constructed using logic program transformations accepted by STABLE.

We also introduce two more logic programming semantics and prove some properties of them. Moreover, we show that our three proposed semantics are equivalent within the class of normal programs. In the last part of our paper, we highlight that the introduced logic programming semantics are suitable for modeling the *setof* operator of PROLOG using negation as failure.

The rest of the paper is structured as follows: In Section 2, basic definitions on logic programming are presented. In Section 3, we prove some properties of stable model semantics and define our new semantics. In Section 4, we introduce an approach for modeling the *setof* operator of PROLOG with our semantics using negation as failure; moreover, we present how to reduce programs with explicit negation to programs without it. Finally, in Section 5, we outline our conclusions and future work.

2. Background

In order to have a self-contained document, basic concepts on logic programming are introduced in this section. This section does not aim to be a tutorial. The interested reader can see [1] for a deep presentation of the concepts introduced in this section.

A signature $\mathcal{L}$ is a finite set of elements that we call atoms. A literal is an atom or the negation of an atom a that we denote as $\neg a$. Given a set of atoms $\{a_1, \dots, a_n\}$, we write $\neg\{a_1, \dots, a_n\}$ to denote the set $\{\neg a_1, \dots, \neg a_n\}$.

A logic theory is built up using the logical constants $\wedge, \vee, \rightarrow$, and $\neg$. We may denote a (general) clause C as

$$a_1 \vee \dots \vee a_m \leftarrow l_1, \dots, l_n, \quad (1)$$

where $l_1, \dots, l_n$ represents the formula $l_1 \wedge \dots \wedge l_n$, $m > 0$, $n \geq 0$, each a_i is a propositional atom, and each l_i is a propositional literal. When $n = 0$ the clause is considered as $a_1 \vee \dots \vee a_m \leftarrow \text{true}$ (or the simple formula $a_1 \vee \dots \vee a_m$), where *true* is a constant atom with its intended interpretation.

Sometimes, a clause C is denoted by $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$, where $\mathcal{A}$ contains all the head atoms, $\mathcal{B}^+$ contains all the positive body atoms, and $\mathcal{B}^-$ contains all the negative body

atoms. We also use $\text{body}(C)$ to denote $\mathcal{B}^+ \cup \neg \mathcal{B}^-$. When $\mathcal{A}$ is a singleton-set, the clause reduces to a normal clause. A definite clause ([12]) is a normal clause lacking of negative literals; that is, $\mathcal{B}^- = \emptyset$. A *pure* disjunction is a disjunction consisting solely of positive or solely of negative literals. A (general) program is a finite set of clauses. We use $\text{HEAD}(P)$ to denote the set of atoms occurring in the heads of the clauses of P . Given a signature $\mathcal{L}$, we write $\text{Prog}_{\mathcal{L}}$ to denote the set of all programs defined over $\mathcal{L}$.

We use $\models$ to denote the consequence relation for classical first-order logic. We will also consider interpretations and models as usual in classical logic.

Given two theories T_1, T_2 we denote $T_1 \equiv_I T_2$ to say that these theories are intuitionistically equivalent; that is, $\forall \alpha \in T_2$, then T_1 proves (using intuitionistic logic) α and vice versa [13].

It will be useful to map a program into a normal program. Given a clause

$$C := \mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^- \quad (2)$$

we write $\text{dis-nor}(C)$ to denote the set of normal clauses:

$$\{a \leftarrow \mathcal{B}^+, \neg \mathcal{B}^- \cup (\mathcal{A} \setminus \{a\}) \mid a \in \mathcal{A}\}. \quad (3)$$

We extend this definition to programs as follows. If P is a program, let $\text{dis-nor}(P)$ denote the normal program: $\bigcup_{C \in P} \text{dis-nor}(C)$. Given a normal program P , we write $\text{Definite}(P)$ to denote the definite program that we obtain from P just by removing every negative literal in P . Given a definite program, by $\text{MM}(P)$ we mean the unique minimal model of P (that always exists for definite programs, see [12]). We assume working with disjunctive programs, unless otherwise stated.

We use an example to illustrate the above definitions. Let P be the program:

$$\begin{aligned} p \vee q &\leftarrow \neg r. \\ p &\leftarrow s, \neg t. \end{aligned}$$

Then $\text{HEAD}(P) = \{p, q\}$, and $\text{dis-nor}(P)$ consists of the clauses:

$$\begin{aligned} p &\leftarrow \neg r, \neg q. \\ q &\leftarrow \neg r, \neg p. \\ p &\leftarrow s, \neg t. \end{aligned}$$

$\text{Definite}(\text{dis-nor}(P))$ consists of the clauses:

$$\begin{aligned} p &\leftarrow \text{true}. \\ q &\leftarrow \text{true}. \\ p &\leftarrow s. \end{aligned}$$

$\text{MM}(\text{Definite}(\text{dis-nor}(P))) = \{p, q\}$.

Since we are interested in disjunctive logic programs, the following question arise:

What are the minimal requirements we want to impose on semantics for disjunctive programs?

Certainly, we want that disjunctive facts, that is, clauses in the program with empty bodies, to be true. Dually, if an atom does not occur in any head, then its negation should be true. These ideas are straightforward generalizations of the case on normal programs. Following the ideas of [14], we propose the following definition.

Definition 2 (semantics). (i) Scenario semantics over a given signature $\mathcal{L}$ is a function such that for every program P over $\mathcal{L}$ it associates a set of models.

(ii) The sceptical semantics induced by scenario semantics is the set of pure disjunctions true in every model of the scenario semantics. (The sceptical semantics derives every literal when its associated scenario semantics has no models.) For any program P we define

$$\text{SEM}_{\min}(P) := \{a \mid a \leftarrow \text{true} \in P\} \cup \{\neg a \mid a \in \mathcal{L}, a \notin \text{HEAD}(P)\}. \quad (4)$$

Given two theories P_1, P_2 , we denote $P_1 \equiv_{\text{SEM}} P_2$ whenever the set of models of P_1 with respect to the scenario semantics SEM is equal to the set of models of P_2 with respect to the scenario semantics SEM.

We introduce some well-accepted logic programming semantics from the state of the art. The first one is the logic programming semantics based on supported models.

Definition 3 (supported model, [14]). A two-valued model I of a (disjunctive) logic program P is supported iff for every ground atom A with $I \models A$ there is a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \neg \mathcal{B}^-$ in P with $A \in \mathcal{A}$, $I \models \mathcal{B}^+ \wedge \neg \mathcal{B}^-$, and $I \not\models \mathcal{A} \setminus \{A\}$.

Supported models are related to the so-called Clark's completion [15]. In order to define the Clark's completion of a logic program, let us introduce the following notation.

Definition 4 (def, sup). Let P be a normal program. Let a be an atom in a given signature $\mathcal{L}$ by the definition of a in P ; we mean the set of clauses $\{a \leftarrow \text{body} \in P\}$ that we denote by $\text{def}(a)$. We define

$$\text{sup}(a) := \begin{cases} \text{false} & \text{if } \text{def}(a) = \emptyset \\ \text{body}_1 \vee \dots \vee \text{body}_n & \text{otherwise,} \end{cases} \quad (5)$$

where $\text{def}(a) = \{a \leftarrow \text{body}_1, \dots, a \leftarrow \text{body}_n\}$.

Considering the previous notation, the Clark's completion with respect to a normal program is presented in the following definition. An extension of the Clark's completion for capturing disjunctive logic programs is also presented in the following definition.

Definition 5 (Clark's completion: $\text{comp}(P)$ ([15]), $\text{dcomp}(P)$). (i) For any normal program P , we define $\text{comp}(P)$ over a given signature $\mathcal{L}$ as the classical theory $\{a \leftrightarrow \text{sup}(a) \mid a \in \mathcal{L}\}$.

(ii) Let P be a disjunctive logic program. Let $\text{dcomp}(P) := \text{comp}(\text{dis-nor}(P))$.

Let us observe that given a normal logic program P , $\text{comp}(P) = \text{dcomp}(P)$. As was mentioned, there is a strong relation between supported models and the Clark's completion. dcomp will play a key role in the definition of the logic programming semantics introduced in this paper. In particular, it will be used for extending the well-founded semantics.

Now the well-accepted STABLE semantics is presented.

Definition 6 (stable, [2]). The Gelfond-Lifschitz transformation (GL-transformation) of a logic program P with respect to an interpretation M is obtained from P by deleting

- (i) each rule that has a negative literal $\neg B$ in its body with $B \in M$,
- (ii) all negative literals in the bodies of the remaining rules.

Clearly the program $\text{GL}(P, M)$ resulting from applying the GL-transformation is negation-free, so $\text{GL}(P, M)$ has at least one model. M is a stable model iff M is a minimal model of $\text{GL}(P, M)$.

It is worth mentioning that Pearce [8] generalized the stable semantics to any propositional theory based on intuitionistic logic. Pearce made the following statement [8]:

“A formula is entailed by a program in the stable model semantics if and only if it belongs to every intuitionistically complete and consistent extension of the program formed by adding only negated atoms.”

From this characterization of the stable semantics, the generalization of STABLE to arbitrary theories is immediate. From now on, we will assume this definition whenever we mention the stable semantics of some given theory.

We introduce a set of transformation rules that will help to define different extension of the well-founded semantics. In particular, the following transformations are defined in [14] and generalize the corresponding definitions for normal programs.

Definition 7 (basic transformation rules). A transformation rule is a binary relation on $\text{Prog}_{\mathcal{L}}$. The following transformation rules are called basic. Let a program $P \in \text{Prog}_{\mathcal{L}}$ be given.

RED⁺: replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ by $\mathcal{A} \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \cap \text{HEAD}(P))$.

RED⁻: delete a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ if there is a clause $\mathcal{A}' \leftarrow \text{true}$ such that $\mathcal{A}' \subseteq \mathcal{B}^-$.

SUB: delete a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ if there is another clause $\mathcal{A}_1 \leftarrow \mathcal{B}_1^+, \neg \mathcal{B}_1^-$ such that $\mathcal{A}_1 \subseteq \mathcal{A}$, $\mathcal{B}_1^+ \subseteq \mathcal{B}^+$, $\mathcal{B}_1^- \subseteq \mathcal{B}^-$.

The transformation rules introduced by Definition 7 are illustrated in the following example:

Example 8 (transformation). Let $\mathcal{L} = \{a, b, c, d, e\}$ and let P be the program:

$a \vee b \leftarrow c, \neg c, \neg d.$
 $a \vee c \leftarrow b.$
 $c \vee d \leftarrow \neg e.$
 $b \leftarrow \neg c, \neg d, \neg e.$

Then $\text{HEAD}(P) = \{a, b, c, d\}$, and $\text{SEM}_{\min}(P) = \{\neg e\}$. We can apply RED^+ to get the program P_1 :

$a \vee b \leftarrow c, \neg c, \neg d.$
 $a \vee c \leftarrow b.$
 $c \vee d \leftarrow \text{true}.$
 $b \leftarrow \neg c, \neg d, \neg e.$

If we apply RED^+ again, we get program P_2 :

$a \vee b \leftarrow c, \neg c, \neg d.$
 $a \vee c \leftarrow b.$
 $c \vee d \leftarrow \text{true}.$
 $b \leftarrow \neg c, \neg d.$

Now, we can apply SUB to get program P_3 :

$a \vee c \leftarrow b.$
 $c \vee d \leftarrow \text{true}.$
 $b \leftarrow \neg c, \neg d.$

A second set of transformations was defined by [14]. These transformations are as follows.

GPPE (Generalized Principle of Partial Evaluation). Suppose P contains $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ and we fix an occurrence of an atom $g \in \mathcal{B}^+$. Then we replace $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ by the n clauses ($i = 1, \dots, n$)

$$\mathcal{A} \cup (\mathcal{A}_i \setminus \{g\}) \leftarrow (\mathcal{B}^+ \setminus \{g\}) \cup B_i^+, \neg \mathcal{B}^- \cup \neg B_i^-, \quad (6)$$

where $\mathcal{A}_i \leftarrow B_i^+, \neg B_i^- \in P$ (for $i = 1, \dots, n$) are all clauses with $g \in \mathcal{A}_i$. If no such clauses exist, we simply delete the former clause.

TAUT (Tautology). Suppose P contains a clause of the form: $\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^-$ and $\mathcal{A} \cap \mathcal{B}^+ \neq \emptyset$, then we delete the given clause.

Let $\mathcal{CS}_1$ be the rewriting system which contains, besides the basic transformation rules, the rules GPPE and TAUT. This system was introduced in [14] and is confluent and terminating as shown in [16]. We write $\text{res}_{\mathcal{CS}_1}(P)$ to denote the residual program of P with respect to the transformations rules $\mathcal{CS}_1$.

Considering $\mathcal{CS}_1$, we define the following logic programming semantics.

Definition 9 (D'-WFS). Given a program P , we define

$$D'\text{-WFS}(P) = \text{SEM}_{\min}(\text{res}_{\mathcal{CS}_1}(P)). \quad (7)$$

We note that our definition of $D'\text{-WFS}(P)$ is very similar to the definition of the D-WFS semantics [17]. The difference is that D-WFS defines pure disjunctions while $D'\text{-WFS}(P)$ defines literals. (The paper could be worked out without defining $D'\text{-WFS}$ but using instead D-WFS.)

Let us note that although the $\mathcal{CS}_1$ system has the nice property of confluence (and termination), its computational properties are not so efficient. In fact, computing the residual form of a program is exponential (even for normal programs, whereas it is known that the WFS ([11]) can be computed in quadratic time).

In order to define a second rewriting system, two more transformation rules are introduced. The first one is Dloop which is defined as follows.

Definition 10 (Dloop [9]). For a program P_1 , let $\text{unf}(P_1) := \mathcal{L} \setminus \text{MM}(\text{Definite}(\text{dis-nor}(P_1)))$. The transformation $\text{Dloop}(\text{dp})$ reduces a program P_1 to $P_2 := \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg \mathcal{B}^- \mid \mathcal{B}^+ \cap \text{unf}(P_1) = \emptyset\}$. We assume that the given transformation takes place only if $P_1 \neq P_2$. We write $P_1 \rightarrow_{\text{Dloop}_A} P_2$ to denote that P_1 transforms to P_2 by a dp transformation, where $A := \text{unf}(P_1)$.

The last transformation, which will be introduced, is Dsuc.

Definition 11 (Dsuc [9]). Suppose that P is a program that includes a fact $a \leftarrow \text{true}$ and a clause $\mathcal{A} \leftarrow \text{Body}$ such that $a \in \text{Body}$. Then we replace this clause by the clause $\mathcal{A} \leftarrow \text{Body} \setminus \{a\}$.

Considering Dloop and Dsuc, the rewriting system $\mathcal{CS}_2$ is introduced.

Definition 12 ($\mathcal{CS}_2$ [9]). Let $\mathcal{CS}_2$ be the rewriting system based on the transformations SUB, RED^+ , RED^- , Dloop, and Dsuc.

We can observe that $\mathcal{CS}_2$ is confluent and terminating.

Lemma 13 (confl. and termination of $\mathcal{CS}_2$). *The system $\mathcal{CS}_2$ is confluent and terminating.*

Proof. (i) $\mathcal{CS}_2$ is terminating since all its transformation rules reduce the size of the given program. The reduction of the size of a given program is done by either the elimination of literals of the body of a clause or the elimination of clauses.

(ii) Let $\mathcal{CS}_3 = \mathcal{CS}_2 \setminus \{\text{SUB}\}$. Hence, $\mathcal{CS}_3$ is confluent in the class of normal logic programs [18]. (In [18], the rewriting system, which proved to be confluent, includes the so-called failure transformation rule. However, failure is redundant if the given rewriting system includes loop [4]. Loop is the version of Dloop for normal logic programs [4, 9].) Let us prove that $\mathcal{CS}_3$ is confluent in the class of disjunctive logic programs by contradiction. Let us suppose that $\mathcal{CS}_3$ is not confluent. Hence, there exists a disjunctive logic program P such that $\text{res}_{\mathcal{CS}_3}(P) = P_1$, $\text{res}_{\mathcal{CS}_3}(P) = P_2$, and $P_1 \neq P_2$. Hence, $\text{res}_{\mathcal{CS}_3}(\text{dis-nor}(P)) = P_3$, $\text{res}_{\mathcal{CS}_3}(\text{dis-nor}(P)) = P_4$ and $P_3 \neq P_4$, but this is a contradiction since $\mathcal{CS}_3$ is confluent in the class of normal logic programs. Now, we

only need to see that $\mathcal{ES}_2 = \mathcal{ES}_3 \cup \{\text{SUB}\}$ is confluent. The only issue is to see the interaction between SUB and the transformation rules of $\mathcal{ES}_3$. Since SUB only removes clauses that are biggest in size than a given one that is kept by SUB at the given program, SUB does not affect the confluence of $\mathcal{ES}_3$. Hence, $\mathcal{ES}_2$ is confluent. $\square$

It is known that $\mathcal{ES}_2$ characterizes an extension of WFS, which captures the class of disjunctive logic programs. This extension of WFS considering $\mathcal{ES}_2$ is called DI-WFS and is defined as follows.

Definition 14 (DI-WFS). Given a program P , we define

$$\text{DI-WFS}(P) = \text{SEM}_{\min}(\text{res}_{\mathcal{ES}_2}(P)). \quad (8)$$

The relationship between DI-WFS and WFS is stayed by the following theorem.

Theorem 15 (see [9]). *Let P be a normal logic program. The following equivalence holds:*

$$\text{DI-WFS}(P) = \text{WFS}(P). \quad (9)$$

Proof. For the class of normal logic programs, the transformation rules RED^+ , RED^- , SUB, Dsuc, and Dloop are equivalent to N-RED^+ , N-RED^- , N-SUB , Suc, and loop, which have been defined for normal programs [9]. (In [9], the transformation rules N-RED^+ , N-RED^- , N-SUB were named as RED^+ , RED^- , SUB, respectively. In this paper, we added the prefix “N-” in order to denote these transformation rules in the settings of normal logic programs.) Since the rewriting system $\{\text{N-RED}^+, \text{N-RED}^-, \text{N-SUB}, \text{Suc}, \text{loop}\}$ characterizes WFS, $\mathcal{ES}_2$ also characterizes WFS. $\square$

It is worth mentioning that DI-WFS generalizes a system introduced in [18] from normal to disjunctive programs.

Let us consider again Example 8. As we noticed before, program P reduces to P_3 . However, P_3 still reduces (by RED^-) to P_4 , which is as P_3 but the third clause is removed. From P_4 we can apply a Dloop reduction to get P_5 : the single clause $c \vee d \leftarrow \text{true}$. P_5 is the residual form of the $\mathcal{ES}_2$ system.

For this example, it turns out that $\text{D}'\text{-WFS}$ is equivalent to DI-WFS, but this is not true in general. Take, for instance, the following example.

Example 16. Let P be a disjunctive logic program:

$$\begin{array}{ll} x \leftarrow \neg a. & a \leftarrow \neg b. \\ d \leftarrow a. & c \vee b. \\ a \leftarrow d. & b \leftarrow c. \end{array}$$

We can see that $\text{res}_{\mathcal{ES}_1}(P)$ has only two clauses: $x \leftarrow \text{true}$ and $b \leftarrow \text{true}$. This means that the atoms x and b are considered true and the rest of atoms are considered false; hence, $\text{D}'\text{-WFS}(P) = \{x, b, \neg d, \neg c, \neg a\}$. On the other hand, $\text{res}_{\mathcal{ES}_2}(P) = P$. This means that we cannot apply any transformation rule from $\mathcal{ES}_2$ to P . Since all the atoms in the language of P appear in at least one head of the clauses, there are not atoms that can be considered false. Moreover, there are not facts that suggest

that there are atoms that can be considered true. Hence, DI-WFS suggests that all the atoms P are undefined. In other words, $\text{DI-WFS}(P) = \phi$.

From Example 16, we can observe that $\text{D}'\text{-WFS}$ and $\text{D}'\text{-WFS}$ are different in the class of disjunctive logic programs. However for normal programs both semantics are equivalent since they define WFS, but note that the residual programs with respect to $\mathcal{ES}_1$ and $\mathcal{ES}_2$ are not necessarily the same. An advantage of $\mathcal{ES}_2$ over $\mathcal{ES}_1$ is that the residual program with respect to $\mathcal{ES}_2$ is polynomial-time computable.

3. Logic Programming Semantics

This section introduces the main results of this paper. We start by stating and proving some useful properties. Some of these properties can be already proved by other authors or intuitively expected; however, since we did not find explicit references to these properties, the properties are formalized and proved.

We start presenting a relationship between dcomp and supported models.

Lemma 17. *Let P be a logic program. M is a supported model of P iff M is a model of $\text{dcomp}(P)$.*

Proof. M is a model of $\text{dcomp}(P)$ iff M is a model of $\text{comp}(\text{dis-nor}(P))$ (see Definition 5) iff M is a supported model of $\text{dis-nor}(P)$ [1] iff M is a supported model of P (by definition of supported model, see Definition 3). $\square$

Let us highlight that Lemma 17 is introducing an extension of the well-known relationship between the Clark's competition and supported models [15], which was introduced for the class of normal programs, to a relationship between dcomp and supported models of disjunctive logic programs.

In order to avoid misunderstanding, let us now define explicitly the concept of *minimal supported model* of a logic program P .

Definition 18. Let P be a program. M is a minimal supported model of P if M is a model of $\text{dcomp}(P)$ and it is minimal (with respect to set inclusion) among the models of $\text{dcomp}(P)$.

We can observe a relationship between minimal supported model and supported models, which are minimal.

Lemma 19. *Let P be a program. If M is a minimal model of P and a supported model of P , then M is a minimal supported model of P .*

Proof. Our proof is by contradiction. Let M be a minimal model of P , M be a supported model of P , and M be not a minimal supported model of P . Then, there exists a proper subset M' of M such that M' is a supported model of P . Thus M' is a model of P . But since M' is a proper subset of M , M is not a minimal model of P , obtaining a contradiction. $\square$

Let us observe that the converse of Lemma 19 is false as the following example shows.

Example 20. Let P be the following set of clauses:

$$\begin{aligned} a &\leftarrow \neg a. \\ a &\leftarrow b. \\ b &\leftarrow b. \end{aligned}$$

We can see that $\text{dcomp}(P)$ is

$$\begin{aligned} a &\leftrightarrow \neg a \vee b \\ b &\leftrightarrow b. \end{aligned}$$

Moreover, one can see that $M := \{a, b\}$ is a minimal supported model of $\text{dcomp}(P)$, but, M is not a minimal model of P .

Now let us introduce two properties regarding the transformation rule Dloop.

Lemma 21. *If P_1 is a disjunctive program, $P_1 \rightarrow_{\text{Dloop}_A} P_2$, then $a \in A$ implies that for every clause $\alpha \leftarrow \text{Body} \in P_1$ such that $a \in \alpha$, then $\text{Body} \cap A \neq \emptyset$.*

Proof. It is a straightforward generalization for the case in normal programs [18]. $\square$

Lemma 22. *If $P_1 \rightarrow_{\text{Dloop}_A} P_2$, then $\text{Head}(P_2) \cap A = \emptyset$.*

Proof. By contradiction, suppose $\text{Head}(P_2) \cap A \neq \emptyset$; therefore, there exists $a \in \text{Head}(P_2) \cap A$. Then, there is a rule of the form $a \vee \alpha \leftarrow \beta \in P_2$. Hence, $a \vee \alpha \leftarrow \beta \in P_1$. As $a \in A$, then there exists $b \in \beta \cap A$ (by Lemma 21). Thus $\forall \alpha \leftarrow \beta \notin P_2$, by the construction of P_2 . This leads to a contradiction. $\square$

Considering the results introduced by Pearce [8], we highlight the following properties in the settings of STABLE.

Lemma 23. *Let T_1 and T_2 be two theories; then $T_1 \equiv_I T_2$ implies $T_1 \equiv_{\text{stable}} T_2$.*

Proof. It follows immediately by Theorem 3.4 in [8]. $\square$

Lemma 24. *If P is a theory, I a set of atoms, and M any stable model of P , then $I \cap M = \emptyset \Rightarrow P \equiv_{\text{stable}} P \cup \neg I$.*

Proof. It follows immediately by Theorem 3.4 in [8]. $\square$

The following relationship between stable models and supported models is well known in the state of the art.

Lemma 25 (see [14]). *Let P be program; if M is a stable model of P , then M is a supported model of P .*

We show a property of Dloop regarding minimal models.

Lemma 26. *Let P_1 be a disjunctive program. If M is minimal model of P_1 and $P_1 \rightarrow_{\text{Dloop}_A} P_2$, then $M \cap A = \emptyset$.*

Proof. Suppose $M \cap A \neq \emptyset$; then $b \in M$ and $b \in A$, so b is not in every minimal model of $\text{definite}(\text{dir-nor}(P))$; then $\text{definite}(\text{dis-nor}(P)) \not\models b$, so $\text{definite}(\text{dis-nor}(P)) \not\models b$. Therefore $b \notin M$ for all M minimal model of $\text{dis-nor}(P)$, thereafter $b \notin M$ for all model of P , and then M is not minimal model. $\square$

Osorio et al. in [19] proved that stable semantics is closed under Dloop.

Theorem 27 (Dloop preserve stable, [19]). *Let P_1 be a disjunctive program. If $P_2 = \text{Dloop}(P_1)$, then P_1 and P_2 are equivalent under the stable semantics.*

The following result was presented in [19].

Lemma 28 (STABLE is closed under $\mathcal{CS}_2$ transformations, [19]). *Let P_1 and P_2 be two programs related by any transformation in $\mathcal{CS}_2$. Then P_1 and P_2 have the same STABLE models.*

Considering the style of GL-transformation, the following reduction is introduced.

Definition 29. Let S be a set of literals and P be a program. We define P^S as follows:

$$P^S = \{A \leftarrow \alpha \setminus \alpha \cap S \mid A \leftarrow \alpha \in P \wedge \nexists l \in \alpha \text{ with } l^c \in S\}, \quad (10)$$

where l^c is the complement of the literal l .

In the following lemma, an interesting property of the $\mathcal{CS}_2$ system regarding the reduction introduced by Definition 29 is proved.

Lemma 30. *Let P_1 be a normal program and $P_2 = \text{res}_{\mathcal{CS}_2}(P_1)$. Then $P_2 = P^{\text{SEM}_{\min}(P_2)}$.*

Proof. First note that if P is obtained from P_1 by a single transformation $T \in \mathcal{CS}_2$, then $P^{\text{SEM}_{\min}(P)} = P_1^{\text{SEM}_{\min}(P)}$, so, by a direct induction, we can verify that $P_2^{\text{SEM}_{\min}(P_2)} = P_1^{\text{SEM}_{\min}(P_2)}$. However, $P_2 = P_2^{\text{SEM}_{\min}(P_2)}$, so, by transitivity, we get the desired result. $\square$

$\mathcal{CS}_2$ is an interesting rewriting system in the settings of supported models as shown in the following lemma.

Lemma 31. *If P and P_1 are two programs such that P_1 is obtained from P by any transformation T in $\mathcal{CS}_2$, then M is a supported model of P_1 implies M is supported model of P .*

Proof. Straightforward by checking each case of T . $\square$

So far, we have highlighted relevant properties of the rewriting system $\mathcal{CS}_2$, minimal models and supported models. Now we are ready for introducing our first logic programming semantics, which will be based on the aforementioned terms.

Definition 32 (D3-WFS-DCOMP). Let P be a program. We define a D3-WFS-DCOMP model as a minimal model of P that is also a supported model of $\text{res}_{\mathcal{CS}_2}(P)$.

It is immediate to see that D3-WFS-DCOMP is more powerful than D1-WFS (see Theorem 15). Note that sometimes STABLE is inconsistent, when D3-WFS-DCOMP is not.

Example 33. For instance, let us consider again the logic program P that was introduced by Example 1:

$d \vee e \leftarrow \neg a.$
 $c \leftarrow c.$
 $b \leftarrow a.$
 $a \leftarrow b.$
 $a \leftarrow \neg b, \neg c.$

We know that STABLE is inconsistent. In order to infer a D3-WFS-DCOMP model, we need to infer $\text{res}_{\mathcal{ES}_2}(P)$. One can see that Dloop can be applied to P . Dloop will remove the clause $c \leftarrow c$. Hence, $P \rightarrow_{\text{Dloop}_A} P3$ such that $P3 := P \setminus \{c \leftarrow c\}$. In $P3$, we can apply RED^+ . After applying RED^+ , no transformation rule of $\mathcal{ES}_2$ can be applied. Hence, $\text{res}_{\mathcal{ES}_2}(P)$ is

$d \vee e \leftarrow \neg a.$
 $b \leftarrow a.$
 $a \leftarrow b.$
 $a \leftarrow \neg b.$

Let us observe that $\{a, b\}$ is a minimal model of P ; moreover, $\{a, b\}$ is a supported model of $\text{res}_{\mathcal{ES}_2}(P)$. Hence, $\{a, b\}$ is a D3-WFS-DCOMP model.

We can also observe that every model inferred by D3-WFS-DCOMP is a supported model.

Lemma 34. *Let P be a program. If M is D3-WFS-DCOMP model of P , then M is a supported model of P .*

Proof. Let M be a D3-WFS-DCOMP model of P , so, M is a supported model of $\text{res}_{\mathcal{ES}_2}(P)$. Thus, by Lemma 31 and a direct induction, M is a supported model of P . $\square$

Due to its construction, we see that D3-WFS-DCOMP is similar to STABLE. However, STABLE is inconsistent more often than D3-WFS-DCOMP. This comment is formalized in the following theorem, which is one of the main results of this paper.

Theorem 35. *Let P be a normal program.*

- (1) *If M is a D3-WFS-DCOMP model of P , then M extends the WFS semantics (i.e., M agrees in the true/false assignments with WFS).*
- (2) *If M is a D3-WFS-DCOMP model of P , then M is a minimal model of P as well as a minimal model of $\text{comp}(P)$.*
- (3) *M is a STABLE model of P implies M is a D3-WFS-DCOMP model of P .*

Proof. (1) It follows by construction and Theorem 15. (2) It follows by construction and Lemmas 19 and 34. (3) Let M be a stable model of P . Hence (by Lemma 28) M is a stable model of $\text{res}_{\mathcal{ES}_2}(P)$. Hence by [14], M is a supported model of $\text{res}_{\mathcal{ES}_2}(P)$. On the other hand, since M is stable model of

P , it is well known that M is a minimal model of P . Hence, M is a minimal model of P as well as a supported model of $\text{res}_{\mathcal{ES}_2}(P)$. Finally (by definition) M is a D3-WFS-DCOMP model of P . $\square$

D3-WFS-DCOMP is the first logic programming semantics that we aim to introduce in this paper. D3-WFS-DCOMP is based on minimal models and supported models; moreover, it considers the normal form (residual program) of the given logic program *with respect to* $\mathcal{ES}_2$. However, if we consider the reduction introduced by Definition 29 and D'-WFS instead of the normal form of the given logic program *with respect to* $\mathcal{ES}_2$, we get a new logic programming semantics. Formally, this new logic programming semantics is defined as follows.

Definition 36. Let P be a program. We define a D2-WFS-DCOMP model as a minimal model of P that is also a supported model of $P^{\text{D'-WFS}(P)}$.

Although both D3-WFS-DCOMP and D2-WFS-DCOMP are based on minimal models and supported models, these logic programming semantics are different.

Lemma 37. *The semantics D3-WFS-DCOMP is not equivalent to D2-WFS-DCOMP.*

Proof. Consider again Example 16. With D3-WFS-DCOMP, we obtain two models: $\{a, b, d\}$ and $\{b, x\}$. However, with D2-WFS-DCOMP we only obtain the model $\{b, x\}$. $\square$

We conjecture the following: for every program P , every D2-WFS-DCOMP model of P is a D3-WFS-DCOMP model of P .

As part of our study of logic programming semantics for disjunctive logic programs, we also suggest studying a logic programming semantics that enforces the interpretation of $\vee$ as *inclusive disjunction*. The interpretation of $\vee$ as inclusive disjunction has been explored by other authors. For instance, in [20], the authors also define semantics based on this notion.

In order to interpret $\vee$ as inclusive disjunction, the concept of *w-supported model* is defined as follows.

Definition 38. Let P be a program, a *w-supported model* M of P is a model of P such that for every $a \in M$, there is a clause $a \leftarrow \text{body}$ such that $a \in M$ and the body is true in M .

Unlike supported models (see Definition 3), which allow semantics to infer models with only one true atom in each head of each disjunctive clause, *w-supported models* allow semantics to infer models with more than one true atom in each head of each disjunctive clause. This subtle difference between supported and *w-supported models* allows us to define different behaviours of the semantics for disjunctive logic programs.

Considering similar constructions to D3-WFS-DCOMP and D2-WFS-DCOMP, the logic programming semantics D3-WFS1-DCOMP is defined as follows.

Definition 39. Let P be a program. We define a D3-WFS1-DCOMP model as a minimal model of P that is also a w -supported model of $P^{D1-WFS(P)}$.

An interesting property, which shares the three logic programming semantics introduced in this paper, is that these semantics coincide in the class of normal logic programs.

Theorem 40. Let P be any normal program; then $D3-WFS-DCOMP(P) = D2-WFS-DCOMP(P) = D3-WFS1-DCOMP(P)$.

Proof. Since $D1-WFS(P) = D'-WFS(P)$ (for normal programs) and by Lemma 30, then it is immediate that $D3-WFS-DCOMP(P) = D2-WFS-DCOMP(P)$. Since supported models are the same as w -supported models (for normal programs), then $D3-WFS-DCOMP(P) = D3-WFS1-DCOMP(P)$. $\square$

Note that, however, for disjunctive programs D3-WFS-DCOMP is (in general) different from D1-WFS1-DCOMP. Take for instance, Example 2 in [20].

4. Applications

In this section, we present an application of the logic programming semantics introduced in Section 3. In particular, we sketch how to extend our semantics to consider programs with clauses allowing an empty head, explicit negation, datalog programs, and finally a lub operator.

- (i) With respect to datalog programs, we first obtain the ground instantiation of the program and then we proceed as with propositional programs. This is a well-known approach and we do not discuss it any more.
- (ii) With respect to programs with explicit negation, it is possible to reduce programs with explicit negation to programs without it. The idea is originally considered in [21].
- (iii) With respect to programs that include clauses with empty head we proceed as follows: Say that we have the clause
 $\leftarrow \alpha$, then simply translate it as $a \leftarrow \alpha, \neg a$,
 where a is a new atom. Clearly, α is false in every D3-WFS-DCOMP model of the program.

4.1. Semantics of lub-Programs. A very interesting issue consists in modeling the *setof* operator of PROLOG. In a meeting, whose aim was to get a picture of the activities in logic programming and its role in the coming years, Gelfond pointed out the following:

“We need to find elegant ways for accomplishing simple tasks, e.g. counting the numbers of elements in the database satisfying some property P . At the moment we have neither clear semantics nor efficient implementation of *setof* of other aggregates used for this type of problems” [22].

Some time has passed since Gelfond has pointed the aforementioned quote; however, his views still hold these days.

Let us consider the following example.

$$s(S) \leftarrow \text{setof}(X, p(X), S),$$

where the intended meaning is as follows: $s(S)$ is true when $S = \{X : p(X)\}$.

Several authors have suggested translating the above expression using negation as failure more or less as follows:

$$\begin{aligned} s(S) &\leftarrow \text{setof-}s(S), \\ \text{setof-}s(S) &\leftarrow \text{try-}s(S), \neg \text{bad-}s(S), \\ \text{try-}s(S) &\leftarrow p(X), \text{singleton-set}(X, S), \\ \text{try-}s(S) &\leftarrow \text{try-}s(S1), \text{try-}s(S2), \text{union}(S1, S2, S), \\ \text{try-}s(X) &\leftarrow \text{emptyset}(X), \\ \text{bad-}s(S) &\leftarrow p(X), \neg \text{member}(X, S), \end{aligned}$$

where *union*, *member*, and *singleton-set* have their intended meaning. For instance, *singleton-set*(X, S) is true if $S = \{X\}$. Given any normal program extended with *setof*, if the translation is a stratified program, then the stratified model of the program captures the “intended” semantics of the original program [23]. However, when the program is not stratified, then we can consider using stable models. Several times the stable models still captures the intended meaning, but sometimes it becomes undefined.

Let us consider the following simple, but yet interesting, example about *setof*. Consider the definite program:

$$\begin{aligned} a &\leftarrow \text{true} \\ b &\leftarrow a, c \end{aligned}$$

represented as:

$$\begin{aligned} \text{rule}([a]) \\ \text{rule}([b, a, c]). \end{aligned}$$

That is, $\text{rule}([X|Y])$ represents the clause $X \leftarrow Y$. Think that we want to write a program that computes the minimal model. Bearing in mind the fix-point semantics, we could write

$$\begin{aligned} \text{mm}(S) &\leftarrow \text{setof}(X, \text{t-p}(X), S). \\ \text{t-p}(X) &\leftarrow \text{rule}(X, Y), \text{mm}(S), \text{subset}(Y, S). \end{aligned}$$

That is, $\text{mm}(S)$ is true (in the intended semantics) iff S is the minimal model of the program represented by the EDB (extensional database) *rule*. (It is however questionable if the above program should be considered legal.) Note that mm and t-p are mutually dependent. If we use the translation of *setof* to NF given above, then STABLE has no models at all, while any of our proposed semantics, in Section 3, defines the intended model.

Following the approach from [23], but adapted to the more familiar environment of PROLOG, a LUB operator is a generalization of *setof* over any complete lattice. The intuitive reading is that $\text{LUB}_L(X, p(X), S)$ is true if S is the least upper bound of each X such that $p(X)$.

Let P be a logic program:

$$h(X, S) \leftarrow \text{LUB}_L(C, n(X, C), S).$$

Now translate the program above as follows:

$$h(X, S) \leftarrow f_=(X, S).$$

$$f_=(X, S) \leftarrow n(X, S).$$

In addition, we need to add axioms that relate the predicate symbols $f_=(X, S)$ with $f_>(X, S)$ for each functional symbol f . Let us consider again our example. The axioms for f in this case are as follows:

- (1) $f_=(Z, S) \leftarrow f_>(Z, S), \neg f_>(Z, S).$
- (2) $f_>(Z, S) \leftarrow f_>(Z, S1), S1 >_L S.$
- (3) $f_>(Z, S) \leftarrow f_>(Z, S1), S1 >_L S.$
- (4) $f_>(Z, \perp).$
- (5) $f_>(Z, C) \leftarrow f_>(Z, C_1), f_>(Z, C_2), \text{lub}_L(C_1, C_2, C).$

We understand that $S1 >_L S$ means that $S1 \geq_L S$ and $S1 \neq S$. And $\text{lub}_L(C_1, C_2, C)$ interprets that C is the least upper bound of C_1 and C_2 . The first two clauses are the same (modulo notation) as in Definition 4.2 in [24]. Clause (5) is not useful for total-order domains.

Now we need to instantiate the translated program. In this paper we restrict our attention to finite ground programs. By adding simple type declarations, we can ensure to get a finite ground program. In this case, we borrow the declaration style of Relationlog; see [23]. Considering any semantics introduced by Section 3, the intended model of LUB_L is inferred.

5. Conclusion and Future Work

The well-founded semantics (WFS) has been regarded as an approximation of the stable semantics STABLE. Moreover, supported model in the settings of the Clark's complication is other well-established semantics for data management. In this paper, we explored the integration of WFS and the Clark's complication for defining three logic programming semantics: D3-WFS-DCOMP, D2-WFS-DCOMP, D3-WFS1-DCOMP. Each of these semantics suggests different interpretations of disjunctive logic programs; however, they suggest a common interpretation in the class of normal logic programs (Theorem 40).

Considering the properties of D3-WFS-DCOMP, D2-WFS-DCOMP, D3-WFS1-DCOMP, these semantics can be regarded as alternative options for STABLE. The aim of our proposals was to explore a misbehavior of disjunctive (and normal) stable semantics that becomes undefined (inconsistent) very often.

From our previous research, we have observed that STABLE has problems for capturing basic aggregation operators such as *setof*. We have sketched some applications of the introduced logic programming semantics for capturing a lub operator, a general form of *setof*. We noticed that STABLE could not define the intended meaning of a simple program that uses our lub operator.

It is well-accepted that STABLE is the state of the art in the nonmonotonic reasoning community. Indeed, there is a solid platform of STABLE solvers that has supported the use of STABLE in real applications [1, 25]. Nevertheless, different extensions of STABLE have been suggested in order to deal with some misbehaviors and extensions of STABLE [3, 26–29]. We consider that the particular semantics introduced by this paper suggest a feasible option for logic based specification languages that aim at semantics close to STABLE, WFS, and the Clark's complication.

In [30], we use stable models semantics for capturing Dung's argumentation approach [31]. Moreover, we have shown that any logic programming semantics can induce an argumentation semantics in the style of Dung's argumentation approach [32]. Hence, in our future work, we will explore the application of the suggested semantics, in this paper, in abstract argumentation theory. For instance, in argumentation theory, there are a few works *with respect to* aggregation of arguments, and we believe that the axiomatization presented in Section 4.1 could be useful for handling aggregation between arguments and/or set of arguments.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

- [1] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, Cambridge, Mass, USA, 2003.
- [2] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," in *Proceedings of the 5th Conference on Logic Programming*, R. Kowalski and K. Bowen, Eds., pp. 1070–1080, MIT Press, 1988.
- [3] J. Dix, "A classification theory of semantics of normal logic programs. II. Weak properties," *Fundamenta Informaticae*, vol. 22, no. 3, pp. 257–288, 1995.
- [4] J. Dix, M. Osorio, and C. Zepeda, "A general theory of confluent rewriting systems for logic programming and its applications," *Annals of Pure and Applied Logic*, vol. 108, no. 1-3, pp. 153–188, 2001.
- [5] J.-H. You and L. Y. Yuan, "A three-valued semantics for deductive databases and logic programs," *Journal of Computer and System Sciences*, vol. 49, no. 2, pp. 334–361, 1994.
- [6] T. C. Przymusiński, "Well-founded completions of logic programs," in *Proceedings of the 8th International Conference Logic Programming*, pp. 726–741, June 1991.
- [7] P. M. Dung, "On the relations between stable and well-founded semantics of logic programs," *Theoretical Computer Science*, vol. 105, no. 1, pp. 7–25, 1992.
- [8] D. Pearce, "Stable inference as intuitionistic validity," *Journal of Logic Programming*, vol. 38, no. 1, pp. 79–91, 1999.
- [9] J. Arrazola, J. Dix, and M. Osorio, "Confluent rewriting systems in non-monotonic reasoning," *Computación y Sistemas*, vol. 2, pp. 104–123, 1999.
- [10] M. Osorio and F. Zacarias, "High-Level Logic Programming," in *Foundations of Information and Knowledge Systems*, Lecture

- Notes in Computer Science, pp. 226–240, Springer Berlin Heidelberg, Berlin, Germany, 2000.
- [11] A. V. Gelder, K. A. Ross, and J. S. Schlipf, “The well-founded semantics for general logic programs,” *Journal of the ACM*, vol. 38, no. 3, pp. 620–650, 1991.
 - [12] J. W. Lloyd, *Foundations of Logic Programming*, Springer, Berlin, Germany, 1987.
 - [13] D. van Dalen, *Logic and Structure*, Springer-Verlag, Berlin, Germany, 3rd edition, 1994.
 - [14] S. Brass and J. Dix, “Characterizations of the disjunctive stable semantics by partial evaluation,” *Journal of Logic Programming*, vol. 32, no. 3, pp. 207–228, 1997.
 - [15] K. L. Clark, “Logic and Databases,” in *chapter Negation as Failure*, pp. 293–322, Plenum Press, 1978.
 - [16] S. Brass and J. Dix, “Characterizations of the disjunctive well-founded semantics: confluent calculi and iterated gcwa,” *Journal of Automated Reasoning*, vol. 20, no. 1, pp. 143–165, 1998.
 - [17] G. Brewka, J. Dix, and K. Konolige, *Nonmonotonic Reasoning: An overview*, CSLI Lectures Notes 73, CSLI Publications, Stanford, CA, USA, 1997.
 - [18] S. Brass, U. Zukowski, and B. Freitag, “Transformation-based bottom-up computation of the well-founded model,” in *Non-Monotonic Extensions of Logic Programming, NMELP ’96*, T. C. P. Jürgen Dix and L. M. Pereira, Eds., vol. 1216 of *Lecture Notes in Computer Science*, pp. 171–201, Springer, Berlin, Germany, 1997.
 - [19] M. Osorio, J. A. Navarro, and J. Arrazola, “Equivalence in answer set programming,” in *Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001*, pp. 57–75, Springer, Berlin, Germany, 2001.
 - [20] S. Greco, “Minimal founded semantics for disjunctive logic programming,” *LPNMR*, pp. 221–235, 1999.
 - [21] C. Baral and M. Gelfond, “Logic programming and knowledge representation,” *The Journal of Logic Programming*, vol. 19–20, no. 1, pp. 73–148, 1994.
 - [22] J. Dix, “The logic programming paradigm,” *AI Communications*, vol. 11, no. 2, pp. 123–131, 1998.
 - [23] M. Osorio and B. Jayaraman, “Aggregation and negation-as-failure,” *New Generation Computing*, vol. 17, no. 3, pp. 255–284, 1999.
 - [24] A. V. Gelder, “The well-founded semantics of aggregation,” *PODS*, pp. 127–138, 1992.
 - [25] M. Osorio, J. Diaz, and A. Santoyo, “0-1 integer programming for computing semi-stable semantics of argumentation frameworks,” *Computación y Sistemas*, vol. 21, no. 3, pp. 457–471, 2017.
 - [26] J. Dix, “A classification theory of semantics of normal logic programs. I. Strong properties,” *Fundamenta Informaticae*, vol. 22, no. 3, pp. 227–255, 1995.
 - [27] M. Osorio, J. A. Navarro, J. R. Arrazola, and V. Borja, “Logics with Common Weak Completions,” *Journal of Logic and Computation*, vol. 16, no. 6, pp. 867–890, 2006.
 - [28] M. Osorio, J. C. Nieves, and C. Giannella, “Useful Transformations in Answer Set Programming,” in *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning (AAAI Spring 2001 Symposium)*, A. Proveti and S. T. Cao, Eds., Stanford, USA, March 2001.
 - [29] J. C. Nieves, M. Osorio, and U. Cortés, “Semantics for possibilistic disjunctive programs,” *Theory and Practice of Logic Programming*, vol. 13, no. 1, pp. 33–70, 2013.
 - [30] J. C. Nieves, U. Cortés, and M. Osorio, “Preferred extensions as stable models,” *Theory and Practice of Logic Programming*, vol. 8, no. 4, pp. 527–543, July 2008.
 - [31] P. M. Dung, “On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games,” *Artificial Intelligence*, vol. 77, no. 2, pp. 321–357, 1995.
 - [32] J. C. Nieves, M. Osorio, and C. Zepeda, “A schema for generating relevant logic programming semantics and its applications in argumentation theory,” *Fundamenta Informaticae*, vol. 106, no. 2–4, pp. 295–319, 2011.