Answer Set Programming as SAT modulo Acyclicity[1]

Martin Gebser [2] and **Tomi Janhunen** and **Jussi Rintanen** [3]
Helsinki Institute for Information Technology HIIT
Department of Information and Computer Science, Aalto University, Finland

Abstract. Answer set programming (ASP) is a declarative programming paradigm for solving search problems arising in knowledge-intensive domains. One viable way to implement the computation of answer sets corresponding to problem solutions is to recast a logic program as a Boolean satisfiability (SAT) problem and to use existing SAT solver technology for the actual search. Such mappings can be obtained by augmenting Clark's completion with constraints guaranteeing the strong justifiability of answer sets. To this end, we consider an extension of SAT by graphs subject to an acyclicity constraint, called SAT modulo acyclicity. We devise a linear embedding of logic programs and study the performance of answer set computation with SAT modulo acyclicity solvers.

1 INTRODUCTION

Answer set programming (ASP) [4] is a declarative programming paradigm featuring a rich rule-based syntax for modeling. The paradigm offers an efficient way to solve search problems arising in knowledge-intensive application domains. Typically, a search problem at hand is described in terms of rules in such a way that its solutions tightly correspond to the answer sets of the resulting logic program. Dedicated search engines, also known as *answer set solvers* [15, 24, 33], can be used to compute answer sets for the program. Thereafter solutions can be extracted from the answer sets found.

In addition to the native solvers mentioned above, one viable way to implement the computation of answer sets is to reduce the search problem to a Boolean satisfiability (SAT) problem and to use SAT solvers instead. As regards the computational complexity of the underlying decision problems, i.e., checking the existence of an answer set for a normal logic program or a satisfying assignment for a set of clauses, both are NP-complete. These results imply the existence of polynomial-time computable reductions between the respective decision problems. However, such reductions are *non-modular* by nature [21, 30]. The main complication is related with recursive rules allowed in ASP and, in particular, positively interdependent rules.

Example 1 *Given a normal logic program consisting of two rules $a \leftarrow b$ and $b \leftarrow a$, a reduction towards SAT can be worked out by forming their* completions *[7], i.e., by rewriting the rules as equivalences $a \leftrightarrow b$ and $b \leftrightarrow a$ that serve as the definitions of a and b. The respective set $S = \{a \vee \neg b, \neg a \vee b\}$ of clauses has essentially two models: $M_1 = \{\}$ and $M_2 = \{a, b\}$. The latter is unacceptable as an answer set since a and b support each other in a self-justifying way. An additional clause $\neg a \vee \neg b$ can be used to exclude M_2.*

A number of translations from normal programs into SAT have been developed. The (incremental) approach based on *loop formulas* [26] requires exponential space when applied as a one-shot transformation. There are also polynomial translations exploiting new atoms: a quadratic one [25] in the length n of a program and a translation of the order of $n \log n$ [20, 21]. However, obtaining a transformation linear in n is unlikely and such compactness seems only achievable if an extension of propositional logic, such as *difference logic* [32], is used as target formalism [31].

In any case, Clark's completion [7] forms a basis for practically all transformations. Using new atoms, the completion can be kept linear in n, and it allows one to capture *supported models* [28] of a logic program as classical models of its completion. Given this relationship, further constraints are needed in order to arrive at *stable models*, also known as answer sets [17], in general. In [31], such constraints are called *ranking constraints*, and their main purpose is to guarantee that the rules of a logic program are applied in a non-circular way (recall M_2 from Example 1).

The goal of this paper is to define yet another translation from normal programs into an extension of SAT. This time, we consider an extension based on a *graph* $G = \langle V, E \rangle$ labeled by dedicated Boolean variables $\mathrm{e}_{\langle u,v \rangle}$ corresponding to *directed edges* $\langle u, v \rangle \in E$. Each truth assignment to these variables gives rise to a subgraph G' of G consisting of exactly those edges $\langle u, v \rangle$ of G for which $\mathrm{e}_{\langle u,v \rangle}$ is true. The idea is that G' is constantly subject to an *acyclicity constraint*, i.e., the edges in E' are not allowed to form a cycle. We call this kind of an extension *SAT modulo acyclicity* due to high analogy with the SAT modulo theories (SMT) framework. Indeed, SAT modulo acyclicity is closely related to graph algorithms used in efficient implementations of difference logic [9, 32]. As this logic extends propositional logic by (simple) linear inequalities, it is still conceptually different from SAT modulo acyclicity, where a graph is directly exploited to represent structure relevant to the domain of interest.

In the sequel, we show how the dynamically varying graph component and the acyclicity constraint imposed on it can be exploited to capture the strong justifiability properties of answer sets in analogy to ranking constraints [31]. Interestingly, this translation stays linear in the length n of a program. Moreover, we have implemented the respective translation for normal programs as well as the acyclicity test in the context of the MINISAT (series 2) code base. This enables a performance analysis against other methods available for computing answer sets. It is not necessary to restrict the analysis to normal programs, since extended rule types [33] supported by contemporary ASP solvers can be *normalized*, using existing transformations and

[1] The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.
[2] Also affiliated with the University of Potsdam, Germany.
[3] Also affiliated with Griffith University, Brisbane, Australia.

their implementation in the tool LP2NORMAL2 [2].

The rest of this paper is organized as follows. In Section 2, we review the syntax and semantics of normal logic programs. The main steps involved when translating ASP into propositional logic and its extensions are recalled in Section 3. These steps guide us when devising a new reduction to SAT modulo acyclicity in Section 4. Then, in Section 5, we discuss the constituents of the resulting linear transformation: first adding acyclicity constraints as normal rules, performing completion for the resulting program, and producing the clausal representation. As back-end SAT solvers, we use new variants of the MINISAT and GLUCOSE solvers extended with tests and propagators for acyclicity. Section 6 is devoted to a performance evaluation. A brief summary of related work is given in Section 7, and Section 8 concludes the paper.

2 NORMAL PROGRAMS

In this section, we review the syntax and semantics of ASP and, in particular, the case of propositional *normal programs*. Such a program P is defined as a set of *rules* r of the form

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \tag{1}$$

where $a, b_1, \ldots, b_n$, and $c_1, \ldots, c_m$ are (propositional) atoms, and $\sim$ stands for *default negation*. The intuition of (1) is that the *head* atom $\mathrm{H}(r) = a$ can be inferred by r if the *positive body* atoms in $\mathrm{B}^+(r) = \{b_1, \ldots, b_n\}$ can be inferred by the other rules of program P, but none of the *negative body* atoms in $\mathrm{B}^-(r) = \{c_1, \ldots, c_m\}$. The entire *body* of r is $\mathrm{B}(r) = \mathrm{B}^+(r) \cup \{\sim c \mid c \in \mathrm{B}^-(r)\}$. The positive part r^+ of a rule r is defined as $\mathrm{H}(r) \leftarrow \mathrm{B}^+(r)$. A normal program P is called *positive*, if we have that $r = r^+$ for every rule $r \in P$.

Next we turn our attention to the semantics of normal programs. The *Herbrand base* $\mathrm{At}(P)$ of a program P is defined as the set of atoms that appear in P. An *interpretation* $I \subseteq \mathrm{At}(P)$ of P specifies which atoms $a \in \mathrm{At}(P)$ are *true* in I ($I \models a$ iff $a \in I$) and which are *false* in I ($I \not\models a$ iff $a \in \mathrm{At}(P) \setminus I$). An entire rule r is satisfied in I, denoted $I \models r$, iff $I \models \mathrm{H}(r)$ is implied by $I \models \mathrm{B}(r)$, where $\sim$ is treated classically, i.e., $I \models \sim c_i$ iff $I \not\models c_i$. A (*classical*) *model* $M \subseteq \mathrm{At}(P)$ of P, denoted $M \models P$, is an interpretation such that $M \models r$ for all $r \in P$. A model $M \models P$ is $\subseteq$*-minimal* iff there is no model $M' \models P$ such that $M' \subset M$. Every *positive* normal program P has a unique $\subseteq$-minimal model, the *least model* $\mathrm{LM}(P)$.

The least model semantics can also cover a normal program P involving default negation if P is first reduced into a positive program $P^I = \{r^+ \mid r \in P, I \cap \mathrm{B}^-(r) = \emptyset\}$ with respect to any interpretation $I \subseteq \mathrm{At}(P)$. Then, an interpretation $M \subseteq \mathrm{At}(P)$ is called a *stable model* of P iff $M = \mathrm{LM}(P^M)$. Stable models are more generally known as *answer sets* [17]. Given that their number can vary, the set of stable models of P is denoted by $\mathrm{SM}(P)$. As shown in [28], stable models form a special case of *supported models* [1]: a model $M \models P$ is supported iff, for every atom $a \in M$, there is a rule $r \in P$ such that $\mathrm{H}(r) = a$ and $M \models \mathrm{B}(r)$.

3 TRANSLATING ASP TOWARDS SAT

In what follows, we present the main ideas needed to translate a normal logic program into propositional logic and its extensions. To this end, we use *difference logic* [32] as the target formalism and essentially present the translation of [31]. For the sake of efficiency, we address the translation at component level. To distinguish the components of a normal logic program P, we define its *positive dependency graph* $\mathrm{DG}^+(P)$ as a pair $\langle \mathrm{At}(P), \leq_+ \rangle$, where $b \leq_+ a$ holds whenever there is a rule $r \in P$ such that $\mathrm{H}(r) = a$ and $b \in \mathrm{B}^+(r)$. A *strongly connected component* (SCC) of $\mathrm{DG}^+(P)$ is a non-empty and maximal subset $C \subseteq \mathrm{At}(P)$ such that $a \leq_+^* b$ and $b \leq_+^* a$ hold for each $a, b \in C$ and the *reflexive* and *transitive* closure $\leq_+^*$ of $\leq_+$. We let $\mathrm{SCC}^+(P)$ stand for the set of SCCs of $\mathrm{DG}^+(P)$. Given an atom $a \in \mathrm{At}(P)$, we denote the SCC $C \in \mathrm{SCC}^+(P)$ such that $a \in C$ by $\mathrm{SCC}(a)$. This allows us to split the *definition* $\mathrm{Def}_P(a) = \{r \in P \mid \mathrm{H}(r) = a\}$ of a into *external* and *internal* parts as follows.

$$\begin{aligned} \mathrm{EDef}_P(a) &= \{r \in \mathrm{Def}_P(a) \mid \mathrm{B}^+(r) \cap \mathrm{SCC}(a) = \emptyset\} \\ \mathrm{IDef}_P(a) &= \{r \in \mathrm{Def}_P(a) \mid \mathrm{B}^+(r) \cap \mathrm{SCC}(a) \neq \emptyset\} \end{aligned}$$

3.1 Program Completion

The *completion* $\mathrm{Comp}(P)$ [7] of a normal program P contains

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \Big(\bigwedge_{b \in \mathrm{B}^+(r)} b \wedge \bigwedge_{c \in \mathrm{B}^-(r)} \neg c \Big) \tag{2}$$

for each atom $a \in \mathrm{At}(P)$. Recall that empty disjunctions and conjunctions correspond to propositional constants $\bot$ and $\top$, respectively. Given a set F of propositional formulas and the set $\mathrm{At}(F)$ of atoms appearing in F, we define interpretations as subsets $I \subseteq \mathrm{At}(F)$ in analogy to Section 2. The satisfaction of propositional formulas is defined in the standard way, and $I \models F$ iff $I \models \phi$ for every formula $\phi \in F$. The set of classical models of F is $\mathrm{CM}(F) = \{M \subseteq \mathrm{At}(F) \mid M \models F\}$. As regards the completion $\mathrm{Comp}(P)$ of a normal program P, it holds that $\mathrm{CM}(\mathrm{Comp}(P))$ coincides with the set of supported models of P [28]. This connection explains why completion is relevant when translating ASP into propositional logic and, indeed, exploited in many translations.

3.2 Difference Logic

As illustrated by Example 1, extra constraints are needed to close the gap between stable and supported models. To this end, we resort to difference logic, which extends propositional logic with simple linear constraints of the form $x + k \geq y$, where k is an arbitrary integer constant and x and y are integer variables. An interpretation in difference logic consists of a pair $\langle I, v \rangle$, where I is a propositional interpretation and v maps integer variables to their domain so that $\langle I, v \rangle \models x + k \geq y$ iff $v(x) + k \geq v(y)$. Deciding the satisfiability of a formula in difference logic is NP-complete, and efficient decision procedures have been developed in the SMT framework [9, 32].

In what follows, we review the main ideas behind the translation of ASP into difference logic [31]. Given a rule r of the form (1), we introduce a new atom bd_r denoting the satisfaction of $\mathrm{B}(r)$. This is defined by the formula (3) below, and consequently (2) can be rewritten as the formula (4).

$$\mathrm{bd}_r \leftrightarrow \bigwedge_{b \in \mathrm{B}^+(r)} b \wedge \bigwedge_{c \in \mathrm{B}^-(r)} \neg c \tag{3}$$

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \mathrm{bd}_r \tag{4}$$

In addition to program completion, the translation of [31] utilizes *ranking constraints* to capture stable models. The translation is further refined in [23] by distinguishing *external* and *internal support* for atoms $a \in \mathrm{At}(P)$ that belong to *non-trivial* components, so that

$\mathrm{IDef}_P(a) \neq \emptyset$. To formalize this, two new atoms ext_a and int_a are introduced for such atoms a and defined by the formulas below.

$$\mathrm{ext}_a \leftrightarrow \bigvee_{r \in \mathrm{EDef}_P(a)} \mathrm{bd}_r \quad (5)$$

$$\mathrm{int}_a \leftrightarrow \bigvee_{r \in \mathrm{IDef}_P(a)} (\mathrm{bd}_r \wedge \bigwedge_{b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)} (x_a - 1 \geq x_b)) \quad (6)$$

$$a \rightarrow \mathrm{ext}_a \vee \mathrm{int}_a \quad (7)$$

$$\neg\mathrm{ext}_a \vee \neg\mathrm{int}_a \quad (8)$$

$$\neg a \rightarrow (x_a = z) \quad (9)$$

$$\mathrm{ext}_a \rightarrow (x_a = z) \quad (10)$$

In (6), x_a and x_b are integer variables introduced for a as well as other atoms $b \in \mathrm{SCC}(a)$. The intuition behind (6) is that internal support for a requires at least one rule r whose positive body atoms within $\mathrm{SCC}(a)$ must be derived before a in a non-circular way. Yet another special variable z, essentially denoting 0, is used to fix the value of x_a whenever a is false (9) or has external support (10). The translation of a normal program P consisting of the formulas presented above is denoted by $\mathrm{Tr}_{\mathrm{DIFF}}(P)$. It does not yield a bijective correspondence of models, but the following relationship can be established in general. For a tighter relation, the reader is referred to additional formulas based on *strong ranking constraints* [23, 31].

Theorem 1 ([23]) *Let P be a normal logic program and $\mathrm{Tr}_{\mathrm{DIFF}}(P)$ its translation into difference logic.*

1. *If $M \in \mathrm{SM}(P)$, then there is a model $\langle N, v\rangle \models \mathrm{Tr}_{\mathrm{DIFF}}(P)$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $\langle N, v\rangle \models \mathrm{Tr}_{\mathrm{DIFF}}(P)$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

4 TRANSLATION INTO SAT MODULO ACYCLICITY

In this section, the goal is to reformulate the translation $\mathrm{Tr}_{\mathrm{DIFF}}(P)$ for SAT modulo acyclicity. The idea is to extend a set S of clauses by a digraph $G = \langle V, E\rangle$ whose edges $\langle u, v\rangle \in E$ are labeled by propositional atoms $\mathrm{e}_{\langle u,v\rangle}$ present in the set $\mathrm{At}(S)$ of atoms appearing in S. Each interpretation $I \subseteq \mathrm{At}(S)$ selects a subgraph G_I of G based on the edges $\langle u, v\rangle \in E$ such that $I \models \mathrm{e}_{\langle u,v\rangle}$. Thus it is reasonable to assume that this mapping from edges to propositions is injective. An interpretation I is a model of S combined with G iff $I \models S$ and the subgraph G_I is *acyclic*, i.e., there is no sequence $\langle v_0, v_1\rangle, \ldots, \langle v_{n-1}, v_n\rangle$ of edges such that $v_n = v_0$.

The if-direction of (3) is captured by the clause (11) below. On the other hand, the only-if-direction requires a clause (12) for each $b \in \mathrm{B}^+(r)$ and a clause (13) for each $c \in \mathrm{B}^-(r)$.

$$\mathrm{bd}_r \vee \bigvee_{b \in \mathrm{B}^+(r)} \neg b \vee \bigvee_{c \in \mathrm{B}^-(r)} c \quad (11)$$

$$\neg\mathrm{bd}_r \vee b \quad (12)$$

$$\neg\mathrm{bd}_r \vee \neg c \quad (13)$$

Rather than formalizing internal and external support explicitly, we distinguish *well-supporting rules* $r \in \mathrm{IDef}_P(a)$ for an atom a. The purpose of the clauses below is to make ws_r equivalent to the respective conjunction in (6). The clause (14) is responsible for the if-direction whereas (15) and (16), introduced for each atom $b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)$, capture the only-if part.

$$\mathrm{ws}_r \vee \neg\mathrm{bd}_r \vee \bigvee_{b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)} \neg\mathrm{e}_{\langle a,b\rangle} \quad (14)$$

$$\neg\mathrm{ws}_r \vee \mathrm{bd}_r \quad (15)$$

$$\neg\mathrm{ws}_r \vee \mathrm{e}_{\langle a,b\rangle} \quad (16)$$

It remains to clausify (4), but taking the external and internal support of a properly into account. We introduce (17) to make a true whenever it is supported by some rule $r \in \mathrm{Def}_P(a)$. On the other hand, (18) falsifies a when it lacks both external and internal support.

$$a \vee \neg\mathrm{bd}_r \quad (17)$$

$$\neg a \vee \bigvee_{r \in \mathrm{EDef}_P(a)} \mathrm{bd}_r \vee \bigvee_{r \in \mathrm{IDef}_P(a)} \mathrm{ws}_r \quad (18)$$

The translation $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ of a normal program P has the clauses defined above, and the resulting graph $G = \langle V, E\rangle$ consists of $V = \{a \in \mathrm{At}(P) \mid \mathrm{IDef}_P(a) \neq \emptyset\}$ and $E = \{\langle a, b\rangle \mid a \in \mathrm{At}(P), r \in \mathrm{IDef}_P(a), b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)\}$. The correctness of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ can be justified on the basis of Theorem 1 as follows.

Theorem 2 *Let P be a normal logic program and $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ its translation into SAT modulo acyclicity.*

1. *If $M \in \mathrm{SM}(P)$, then there is a model $N \models \mathrm{Tr}_{\mathrm{ACYC}}(P)$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \models \mathrm{Tr}_{\mathrm{ACYC}}(P)$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

Proof sketch. Let G be the graph associated with $\mathrm{Tr}_{\mathrm{ACYC}}(P)$. For the first item, it is sufficient to show that, if $\langle N, v\rangle \models \mathrm{Tr}_{\mathrm{DIFF}}(P)$, then there is an interpretation I satisfying the clauses of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ such that G_I is acyclic and $I \cap \mathrm{At}(P) = N \cap \mathrm{At}(P)$. For the second item, it needs to be established that, if N satisfies the clauses of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ and G_N is acyclic, then $\langle I, v\rangle \models \mathrm{Tr}_{\mathrm{DIFF}}(P)$ such that $I \cap \mathrm{At}(P) = N \cap \mathrm{At}(P)$ and v is obtained from G_N by setting $v(x_a)$ to be the maximum distance from a to a leaf node in G_N. □

The translation introduced above does not yet include any clauses corresponding to formulas (9) and (10), which reset the integer variable x_a associated with a when the value of this variable is irrelevant. For the translation into SAT modulo acyclicity, the respective idea is to explicitly *disable* edges that are clearly irrelevant for checking non-circular support through rules. The clause (19), i.e., the analog of (9), is introduced for any pair a and b of atoms such that there is some $r \in \mathrm{IDef}_P(a)$ with $b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)$. To cover (10), we need a similar clause (20) conditioned by external support provided by $r \in \mathrm{EDef}_P(a)$.

$$a \vee \neg\mathrm{e}_{\langle a,b\rangle} \quad (19)$$

$$\neg\mathrm{bd}_r \vee \neg\mathrm{e}_{\langle a,b\rangle} \quad (20)$$

$$\neg\mathrm{ws}_r \vee \neg\mathrm{e}_{\langle a,b\rangle} \quad (21)$$

Actually, it is possible to generalize this principle for internally and, more precisely, well-supporting rules. The clause (21) can be incorporated for any pair a and b of atoms such that $\{r, r'\} \subseteq \mathrm{IDef}_P(a)$ and $b \in (\mathrm{B}^+(r') \setminus \mathrm{B}^+(r)) \cap \mathrm{SCC}(a)$. The last condition is essential: note that ws_r being true presumes that each $\mathrm{e}_{\langle a,b\rangle}$ such that $b \in \mathrm{B}^+(r) \cap \mathrm{SCC}(a)$ is true. The intuition is that r alone is sufficient to provide the internal support for a and no other $r' \in \mathrm{IDef}_P(a)$ is necessary in this respect. Thus the check for non-circular support is feasible with (potentially) fewer edges present in the graph. The respective extension of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ by the clauses of forms (19)–(21) above is denoted by $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$.

Proposition 1 *Let P be a normal logic program and $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$ its extended translation into SAT modulo acyclicity.*

1. *If $M \in \mathrm{SM}(P)$, then there is a model $N \models \mathrm{Tr}^+_{\mathrm{ACYC}}(P)$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \models \mathrm{Tr}^+_{\mathrm{ACYC}}(P)$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

The main observation behind Proposition 1 is that any subgraph of an acyclic graph is also acyclic. This is why the additional clauses do not interfere with satisfiability but, on the other hand, can favor computational performance. Finally, it is worth pointing out that the translation of [23] does not have any corresponding formula, and hence the extension based on (21) is a novel contribution.

Example 2 *To illustrate $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$, consider a logic program P as follows.*

$$\begin{array}{llll} r_1 : a \leftarrow b & r_3 : b \leftarrow a & r_5 : c \leftarrow a, b & \\ r_2 : a \leftarrow c & r_4 : b \leftarrow c, {\sim}d & r_6 : c \leftarrow {\sim}d & r_7 : d \leftarrow {\sim}c \end{array}$$

We have $\mathrm{SCC}^+(P) = \{\{a, b, c\}, \{d\}\}$, so that $\mathrm{EDef}_P(a) = \mathrm{EDef}_P(b) = \emptyset$, $\mathrm{EDef}_P(c) = \{r_6\}$, and $\mathrm{EDef}_P(d) = \{r_7\}$. The following definitions are captured by the clauses of forms (11)–(13).

$$\begin{array}{llll} \mathrm{bd}_{r_1} \leftrightarrow b & \mathrm{bd}_{r_3} \leftrightarrow a & \mathrm{bd}_{r_5} \leftrightarrow a \wedge b & \\ \mathrm{bd}_{r_2} \leftrightarrow c & \mathrm{bd}_{r_4} \leftrightarrow c \wedge \neg d & \mathrm{bd}_{r_6} \leftrightarrow \neg d & \mathrm{bd}_{r_7} \leftrightarrow \neg c \end{array}$$

Moreover, the clauses of forms (14)–(16) define well-support through $r_1, \ldots, r_5$ as follows.

$$\begin{array}{ll} \mathrm{ws}_{r_1} \leftrightarrow \mathrm{bd}_{r_1} \wedge \mathrm{e}_{\langle a,b\rangle} & \mathrm{ws}_{r_3} \leftrightarrow \mathrm{bd}_{r_3} \wedge \mathrm{e}_{\langle b,a\rangle} \\ \mathrm{ws}_{r_2} \leftrightarrow \mathrm{bd}_{r_2} \wedge \mathrm{e}_{\langle a,c\rangle} & \mathrm{ws}_{r_4} \leftrightarrow \mathrm{bd}_{r_4} \wedge \mathrm{e}_{\langle b,c\rangle} \\ & \mathrm{ws}_{r_5} \leftrightarrow \mathrm{bd}_{r_5} \wedge \mathrm{e}_{\langle c,a\rangle} \wedge \mathrm{e}_{\langle c,b\rangle} \end{array}$$

The introduced atoms are used by the clauses of forms (17) and (18), expressing that any supported atom must be true but also requires some well-supporting rule if it has no external support.

$$\begin{array}{lll} a \vee \neg\mathrm{bd}_{r_1} & a \vee \neg\mathrm{bd}_{r_2} & \neg a \vee \mathrm{ws}_{r_1} \vee \mathrm{ws}_{r_2} \\ b \vee \neg\mathrm{bd}_{r_3} & b \vee \neg\mathrm{bd}_{r_4} & \neg b \vee \mathrm{ws}_{r_3} \vee \mathrm{ws}_{r_4} \\ c \vee \neg\mathrm{bd}_{r_5} & c \vee \neg\mathrm{bd}_{r_6} & \neg c \vee \mathrm{bd}_{r_6} \vee \mathrm{ws}_{r_5} \\ d \vee \neg\mathrm{bd}_{r_7} & & \neg d \vee \mathrm{bd}_{r_7} \end{array}$$

The above formulas correspond to the set $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ of clauses. While P has two stable models, $\{a, b, c\}$ and $\{d\}$, there are 30 (acyclic) models of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$. The reason for this sharp increase is that edges may be freely added to the ones from well-supporting rules as long as the respective subgraph remains acyclic. In order to tighten the selection of edges, $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$ further contains the following clauses of forms (19)–(21).

$$\begin{array}{rrr} a \vee \neg\mathrm{e}_{\langle a,b\rangle} & b \vee \neg\mathrm{e}_{\langle b,a\rangle} & c \vee \neg\mathrm{e}_{\langle c,a\rangle} \\ a \vee \neg\mathrm{e}_{\langle a,c\rangle} & b \vee \neg\mathrm{e}_{\langle b,c\rangle} & c \vee \neg\mathrm{e}_{\langle c,b\rangle} \\ \neg\mathrm{ws}_{r_2} \vee \neg\mathrm{e}_{\langle a,b\rangle} & \neg\mathrm{ws}_{r_4} \vee \neg\mathrm{e}_{\langle b,a\rangle} & \neg\mathrm{bd}_{r_6} \vee \neg\mathrm{e}_{\langle c,a\rangle} \\ \neg\mathrm{ws}_{r_1} \vee \neg\mathrm{e}_{\langle a,c\rangle} & \neg\mathrm{ws}_{r_3} \vee \neg\mathrm{e}_{\langle b,c\rangle} & \neg\mathrm{bd}_{r_6} \vee \neg\mathrm{e}_{\langle c,b\rangle} \end{array}$$

The addition of these clauses reduces the number of models to 4. Since edges from false atoms are suppressed, the model corresponding to $\{d\}$ yields a subgraph without any edge. The three remaining models augment $\{a, b, c, \mathrm{bd}_{r_1}, \ldots, \mathrm{bd}_{r_6}\}$ with either $\{\mathrm{e}_{\langle a,b\rangle}, \mathrm{e}_{\langle b,c\rangle}, \mathrm{ws}_{r_1}, \mathrm{ws}_{r_4}\}$, $\{\mathrm{e}_{\langle a,c\rangle}, \mathrm{e}_{\langle b,c\rangle}, \mathrm{ws}_{r_2}, \mathrm{ws}_{r_4}\}$, or $\{\mathrm{e}_{\langle a,c\rangle}, \mathrm{e}_{\langle b,a\rangle}, \mathrm{ws}_{r_2}, \mathrm{ws}_{r_3}\}$, representing distinct derivations of a and b by means of internal support. Notably, the viable derivations do not contain one another, given that $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$ prohibits redundant edges.

5 IMPLEMENTATION

In this section, we describe our translation-based implementation of ASP using the reduction presented in Section 4. First of all, we assume that GRINGO is run to instantiate ASP programs, typically containing term variables in first-order style. The outcome is a ground logic program, which is subsequently processed as follows.

Normalization. Besides normal rules (1), contemporary ASP solvers support a number of extensions such as *choice rules*, *cardinality rules*, *weight rules*, and *optimization statements* [33]. We use the existing tool LP2NORMAL2 (v. 1.10) [2] to *normalize* ground programs involving extended rules of the first three types. For the moment, we do not support optimization statements, mainly because the current back-end solvers are lacking optimization capabilities.

Translation into Extended CNF and SMT. The actual transformation from normal rules into clauses takes place in two steps. Each rule $r \in P$ of an input program P is first rewritten by using the new atoms bd_r and ws_r involved in the translation $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ and by adding *normal rules* defining these new atoms. In addition, atomic propositions $\mathrm{e}_{\langle a,b\rangle}$ corresponding to the edges of the graph are introduced, and the further constraints of $\mathrm{Tr}^+_{\mathrm{ACYC}}(P)$ are optionally incorporated. This first step is implemented by a translator called LP2ACYC (v. 1.13) [13]. The second step of the transformation concerns the completion of the program as well as producing the clausal representation in an extended DIMACS format. The output produced by the tool ACYC2SAT (v. 1.24) has a dedicated section for expressing the graph for acyclicity checking. Support for difference logic in the SMT LIB 2.0 format is obtained similarly by using the translator ACYC2SOLVER (v. 1.7), which produces the required formula syntax (command line option `--diff`). The graph is here represented by implications $\mathrm{e}_{\langle a,b\rangle} \rightarrow (x_a > x_b)$, where x_a and x_b are integer variables associated with atoms a and b involved in the same SCC.

Back-End Solvers. To implement the search for satisfying assignments corresponding to answer sets, we use high-performance extensions of SAT solvers by acyclicity constraints as presented in [13, 14]. These solvers are based on the publicly available MINISAT solver, and they take as input a set of clauses, a graph, and a mapping from the edges of the graph to propositional variables. The solvers' search algorithms work exactly like the underlying MINISAT solver, except that when assigning an edge variable to *true*, corresponding to the addition of an edge to the graph, a propagator for the acyclicity constraint checks whether the graph contains a cycle. If so, a conflict is reported to the solver. Moreover, the propagator checks whether the graph now contains any *path* leading from some node u to another node v such that $\langle v, u\rangle$ is a potential edge. In that case, the solver infers $\neg\mathrm{e}_{\langle v,u\rangle}$ for the the propositional variable $\mathrm{e}_{\langle v,u\rangle}$ representing the presence of the edge $\langle v, u\rangle$ in the graph. The solvers presented in [14] include ACYCGLUCOSE and ACYCMINISAT, which are analogous extensions of the GLUCOSE and MINISAT solvers for plain SAT. We below compare our SAT modulo acyclicity solvers to the Z3 SMT solver, winner of the difference logic category (QF_IDL) in the 2011 SMT solver competition.

6 EXPERIMENTS

We empirically evaluate the introduced translations into SAT modulo acyclicity on the Hamiltonian cycle problem as well as the tasks of finding a directed acyclic graph, forest, or tree subject to XOR-constraints over edges [13]. The formulation of the Hamiltonian cycle problem as a logic program relies on positively recursive rules to keep track of the reachability of nodes. Likewise, the aforementioned

Problem	Hamilton		Acyclic				Forest				Tree			
Size	100	150	25	50	75	100	25	50	75	100	25	50	75	100
CLASP	0.95	20.16	0.12	0.76	282.01	831.33	4.09	1039.26	1501.76	1632.94	4.37	1193.09	1495.32	1995.19
ACYCGLUCOSE	0.07	0.15	0.05	**0.28**	8.09	964.28	**0.64**	304.44	1006.73	1418.25	**0.74**	315.83	999.07	1414.68
ACYCMINISAT	**0.04**	**0.12**	**0.03**	0.29	4.76	647.13	0.72	498.00	920.43	1269.47	0.83	544.43	1025.02	1224.28
Z3	2.45	50.64	0.29	7.61	167.74	2278.63	4.54	1205.63	1755.28	2690.67	4.75	1208.36	1726.56	2538.20
ACYCGLUCOSE-$\mathrm{Tr}_{\mathrm{ACYC}}$	0.93	13.75	0.09	0.34	5.47	1165.24	1.28	328.93	1012.53	1447.94	1.40	271.93	973.22	1388.82
ACYCMINISAT-$\mathrm{Tr}_{\mathrm{ACYC}}$	0.76	7.28	0.09	0.57	3.54	404.14	0.86	505.18	894.59	1123.87	0.80	484.92	879.18	1030.79
Z3-$\mathrm{Tr}_{\mathrm{ACYC}}$	35.80	331.11	24.47	7.21	907.72	2335.39	6.78	1156.39	2211.60	2585.17	6.30	1178.44	2266.66	2714.01
ACYCGLUCOSE-$\mathrm{Tr}^{+}_{\mathrm{ACYC}}$	**0.04**	0.18	0.14	0.33	5.91	1215.41	1.05	**294.43**	1044.64	1471.99	1.09	**264.28**	931.28	1379.15
ACYCMINISAT-$\mathrm{Tr}^{+}_{\mathrm{ACYC}}$	0.08	0.32	0.09	0.58	**3.25**	**258.47**	0.80	495.43	**887.67**	**1040.15**	0.77	473.64	**852.78**	**1016.50**
Z3-$\mathrm{Tr}^{+}_{\mathrm{ACYC}}$	27.72	239.83	20.32	6.47	952.43	2240.60	8.99	1111.26	2101.47	2524.52	7.03	1230.51	1976.20	2562.70

Table 1. Empirical comparison between direct encodings and SAT modulo acyclicity translations of Hamiltonian cycle and directed acyclic graph problems

acyclicity properties can be expressed in terms of recursive specifications based on elimination orders, and respective ASP encodings are developed in [12]. Direct SAT modulo acyclicity or difference logic encodings, on the other hand, focus on the absence of cycles (by disregarding the incoming edges of a fixed starting node in case of the Hamiltonian cycle problem). Acyclic graph structures, in general, are central to numerous application problems, e.g., [3, 5, 8, 10, 19].

Our evaluation includes the ASP solver CLASP (v. 3.0.4), the SAT modulo acyclicity solvers ACYCGLUCOSE (based on GLUCOSE v. 3.0) and ACYCMINISAT (based on MINISAT v. 2.2.0), and the SMT solver Z3 (v. 4.3.1) supporting difference logic. While CLASP is run on logic programs P encoding the investigated problems, the other three solvers are applied to corresponding direct problem formulations as well as the translations $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ and $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$, as indicated by the suffix $\mathrm{Tr}_{\mathrm{ACYC}}$ or $\mathrm{Tr}^{+}_{\mathrm{ACYC}}$, respectively, in Table 1, which provides average runtimes over 100 (randomly generated) instances per problem and graph size in terms of nodes. The instances consist of planar directed graphs in case of the Hamiltonian cycle problem or, otherwise, XOR-constraints over edges to be fulfilled by a directed acyclic graph, forest, or tree, respectively. All experiments were run on a cluster of Linux machines with a timeout of 3600 seconds per instance, taking aborts as 3600 seconds within averages. Minimum average runtimes per column are highlighted in boldface.

Among direct encodings of the Hamiltonian cycle problem in the upper left part of Table 1, the SAT modulo acyclicity solvers ACYCGLUCOSE and ACYCMINISAT have an edge over the SMT solver Z3 and CLASP running on logic programs P. While the translations $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ and $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$ do not yield the same performance of Z3 as direct problem formulation, the average runtimes of ACYCGLUCOSE and ACYCMINISAT with the translation $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$ come close to direct encoding. This indicates the adequacy of the translation from logic programs into SAT modulo acyclicity.

For finding directed acyclic graphs, forests, or trees fulfilling XOR-constraints, the performance of ACYCMINISAT is even better with the translation $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$ than direct formulation on graphs with 75 or 100 nodes. The search statistics of ACYCMINISAT revealed a reduction of the number of decisions and conflicts by about one order of magnitude on average, recompensating the size overhead (roughly factor 5) of the translation. This observation suggests that translation, in particular, $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$, exposes problem structure that is not immediately accessible to search with the direct SAT modulo acyclicity encoding. The effect of translations on search performance is nevertheless solver-specific, as ACYCGLUCOSE and Z3 cannot take similar advantage of them as ACYCMINISAT. Unlike with the Hamiltonian cycle problem, ACYCGLUCOSE does also not benefit significantly from using $\mathrm{Tr}^{+}_{\mathrm{ACYC}}(P)$ and in some cases, e.g., finding directed acyclic graphs or forests with 100 nodes, performs even better with $\mathrm{Tr}_{\mathrm{ACYC}}(P)$. Despite of this, running ACYCGLUCOSE and ACYCMINISAT with translations from logic programs into SAT modulo acyclicity turns out to be competitive to direct encoding. Both solvers are further able to achieve performance improvements compared to the ASP solver CLASP on the considered problems. The disadvantages of Z3 are presumably owed to the fact that difference logic is more expressive than what is needed for acyclicity checking.

7 RELATED WORK

Native ASP solvers, such as SMODELS [33], DLV [24], and CLASP [15], use search techniques analogous to those for SAT, yet tailored to the needs of logic programs. Albeit rules and clauses are different base primitives, it is interesting to compare the *unfounded set* [34] falsification of ASP solvers to the propagation principles of SAT modulo acyclicity solvers [14]. For instance, with reachability-based encodings of the Hamiltonian cycle problem (cf. [13]), ASP solvers are able to detect inconsistency of a partial assignment such that false edge variables yield a partition of the given graph. However, true edge variables need not necessarily form a cycle yet, so that SAT modulo acyclicity solvers are not guaranteed to detect such an inherent inconsistency. On the other hand, the falsification of edge variables that would close a cycle has no counterpart in ASP solvers, where proposals to converse unfounded set propagation [6, 11, 16] did not lead to practical success. Even if such mechanisms were available, with reachability-based encodings of the Hamiltonian cycle problem, they would aim at identifying edges necessary to avoid the partitioning of a graph, which is orthogonal to preventing (sub)cycles. With our translation technique into SAT modulo acyclicity, true edge variables represent some derivation for the atoms in a stable model, which can be viewed as making *source pointers* [33], originally introduced as a data structure for unfounded set checking, explicit. The inclusion of source pointers or respective edge variables is the reason why one stable model may correspond to several (classical) models in SAT modulo acyclicity.

Several systems translate a logic program given as input into a set of clauses and use a SAT solver for the actual search. The early ASSAT system [26] exploits loop formulas to exclude non-stable supported models, and the same idea is adopted in the design of the CMODELS system [18]. The LP2SAT system [21] and its derivatives

[22] are based on a subquadratic one-shot transformation, hence improving the quadratic encoding described in [25]. However, the representation of well-supporting rules, possibly augmented with the completing clauses in (19)–(21), in SAT modulo acyclicity is linear and thus more compact than translations into plain SAT.

Logic programs can also be translated into SMT. In this respect, difference logic [23] and the logic of bit vectors [29] are covered by linear translations, such as the one detailed in Section 3. Yet another translation into mixed integer programming (MIP) is presented in [27]. Due to the use of numeric variables, this translation is also linear, and it enables the application of MIP solvers like CPLEX to compute answer sets. The respective translations into SMT and MIP are based on similar principles as the translation into SAT modulo acyclicity developed in this paper.

8 CONCLUSION

We have presented novel mappings of logic programs under stable model semantics to SAT modulo acyclicity. Similar to previous SMT and MIP translations, our embeddings in SAT modulo acyclicity are linear, yet without relying on numeric variables utilized in SMT and MIP formulations. Although our translations into SAT modulo acyclicity yield, in general, a one-to-many correspondence between stable and classical models, our experiments indicate that solvers for SAT modulo acyclicity can be highly effective. The translators LP2NORMAL2, LP2ACYC, and ACYC2SAT, together with our SAT modulo acyclicity solvers ACYCGLUCOSE and ACYCMINISAT [14], form a new translation-based implementation [13] of ASP.

REFERENCES

[1] K. Apt, H. Blair, and A. Walker, 'Towards a theory of declarative knowledge', in *Foundations of Deductive Databases and Logic Programming*, ed., J. Minker, pp. 89–148. Morgan Kaufmann, (1988).

[2] J. Bomanson and T. Janhunen, 'Normalizing cardinality rules using merging and sorting constructions', in *Proc. International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pp. 187–199. Springer, (2013).

[3] M. Bonet and K. John, 'Efficiently calculating evolutionary tree measures using SAT', in *Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pp. 4–17. Springer, (2009).

[4] G. Brewka, T. Eiter, and M. Truszczyński, 'Answer set programming at a glance', *Communications of the ACM*, **54**(12), 92–103, (2011).

[5] D. Brooks, E. Erdem, S. Erdogan, J. Minett, and D. Ringe, 'Inferring phylogenetic trees using answer set programming', *Journal of Automated Reasoning*, **39**(4), 471–511, (2007).

[6] X. Chen, J. Ji, and F. Lin, 'Computing loops with at most one external support rule', in *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pp. 401–410. AAAI Press, (2008).

[7] K. Clark, 'Negation as failure', in *Logic and Data Bases*, eds., H. Gallaire and J. Minker, pp. 293–322. Plenum Press, (1978).

[8] J. Corander, T. Janhunen, J. Rintanen, H. Nyman, and J. Pensar, 'Learning chordal Markov networks by constraint satisfaction', in *Proc. Annual Conference on Neural Information Processing Systems (NIPS'13)*, pp. 1349–1357. Volume 26 of *Advances in Neural Information Processing Systems*, (2013).

[9] S. Cotton and O. Maler, 'Fast and flexible difference constraint propagation for DPLL(T)', in *Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pp. 170–183. Springer, (2006).

[10] J. Cussens, 'Bayesian network learning with cutting planes', in *Proc. International Conference on Uncertainty in Artificial Intelligence (UAI'11)*, pp. 153–160. AUAI Press, (2011).

[11] C. Drescher and T. Walsh, 'Efficient approximation of well-founded justification and well-founded domination', in *Proc. International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pp. 277–289. Springer, (2013).

[12] M. Gebser, T. Janhunen, and J. Rintanen, 'ASP encodings of acyclicity properties', in *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*. AAAI Press, (2014).

[13] M. Gebser, T. Janhunen, and J. Rintanen, 'SAT modulo acyclicity tools'. `http://research.ics.aalto.fi/software/asp/lp2acyc/`, (2014).

[14] M. Gebser, T. Janhunen, and J. Rintanen, 'SAT modulo graphs: Acyclicity'. Submitted, (2014).

[15] M. Gebser, B. Kaufmann, and T. Schaub, 'Conflict-driven answer set solving: From theory to practice', *Artificial Intelligence*, **187**, 52–89, (2012).

[16] M. Gebser and T. Schaub, 'Tableau calculi for logic programs under answer set semantics', *ACM Transactions on Computational Logic*, **14**(2), 15:1–15:40, (2013).

[17] M. Gelfond and V. Lifschitz, 'Classical negation in logic programs and disjunctive databases', *New Generation Computing*, **9**, 365–385, (1991).

[18] E. Giunchiglia, Y. Lierler, and M. Maratea, 'Answer set programming based on propositional satisfiability', *Journal of Automated Reasoning*, **36**(4), 345–377, (2006).

[19] T. Jaakkola, D. Sontag, A. Globerson, and M. Meila, 'Learning Bayesian network structure using LP relaxations', in *Proc. International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, pp. 358–365. Volume 9 of *JMLR Proceedings*, (2010).

[20] T. Janhunen, 'Representing normal programs with clauses', in *Proc. European Conference on Artificial Intelligence (ECAI'04)*, pp. 358–362. IOS Press, (2004).

[21] T. Janhunen, 'Some (in)translatability results for normal logic programs and propositional theories', *Journal of Applied Non-Classical Logics*, **16**(1-2), 35–86, (2006).

[22] T. Janhunen and I. Niemelä, 'Compact translations of non-disjunctive answer set programs to propositional clauses', in *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, eds., M. Balduccini and T. Son, pp. 111–130. Springer, (2011).

[23] T. Janhunen, I. Niemelä, and M. Sevalnev, 'Computing stable models via reductions to difference logic', in *Proc. International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pp. 142–154. Springer, (2009).

[24] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, 'The DLV system for knowledge representation and reasoning', *ACM Transactions on Computational Logic*, **7**(3), 499–562, (2006).

[25] F. Lin and J. Zhao, 'On tight logic programs and yet another translation from normal logic programs to propositional logic', in *Proc. International Joint Conference on Artificial Intelligence (IJCAI'03)*, pp. 853–858. Morgan Kaufmann, (2003).

[26] F. Lin and Y. Zhao, 'ASSAT: Computing answer sets of a logic program by SAT solvers', *Artificial Intelligence*, **157**(1), 115–137, (2004).

[27] G. Liu, T. Janhunen, and I. Niemelä, 'Answer set programming via mixed integer programming', in *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pp. 32–42. AAAI Press, (2012).

[28] V. Marek and V. Subrahmanian, 'The relationship between stable, supported, default and autoepistemic semantics for general logic programs', *Theoretical Computer Science*, **103**(2), 365–386, (1992).

[29] M. Nguyen, T. Janhunen, and I. Niemelä, 'Translating answer-set programs into bit-vector logic', in *Proc. International Conference on Applications of Declarative Programming and Knowledge Management (INAP'11)*, pp. 95–113. Springer, (2013).

[30] I. Niemelä, 'Logic programs with stable model semantics as a constraint programming paradigm', *Annals of Mathematics and Artificial Intelligence*, **25**(3-4), 241–273, (1999).

[31] I. Niemelä, 'Stable models and difference logic', *Annals of Mathematics and Artificial Intelligence*, **53**(1-4), 313–329, (2008).

[32] R. Nieuwenhuis and A. Oliveras, 'DPLL(T) with exhaustive theory propagation and its application to difference logic', in *Proc. International Conference on Computer Aided Verification (CAV'05)*, pp. 321–334. Springer, (2005).

[33] P. Simons, I. Niemelä, and T. Soininen, 'Extending and implementing the stable model semantics', *Artificial Intelligence*, **138**(1-2), 181–234, (2002).

[34] A. Van Gelder, K. Ross, and J. Schlipf, 'The well-founded semantics for general logic programs', *Journal of the ACM*, **38**(3), 620–650, (1991).