Answer Set Programming modulo Acyclicity *

Jori Bomanson[1], Martin Gebser[1,2], Tomi Janhunen[1],
Benjamin Kaufmann[2], and Torsten Schaub[2,3]**

[1]Aalto University, HIIT [2]University of Potsdam [3]INRIA Rennes

Abstract. Acyclicity constraints are prevalent in knowledge representation and, in particular, applications where acyclic data structures such as DAGs and trees play a role. Recently, such constraints have been considered in the satisfiability modulo theories (SMT) framework, and in this paper we carry out an analogous extension to the answer set programming (ASP) paradigm. The resulting formalism, ASP modulo acyclicity, offers a rich set of primitives to express constraints related with recursive structures. The implementation, obtained as an extension to the state-of-the-art answer set solver CLASP, provides a unique combination of traditional unfounded set checking with acyclicity propagation.

1 Introduction

Acyclic data structures such as DAGs and trees occur frequently in applications. For instance, Bayesian [1] and Markov [2] network learning as well as Circuit layout [3] are based on respective conditions. When logical formalisms are used for the specification of such structures, dedicated *acyclicity constraints* are called for. Recently, such constraints have been introduced in the *satisfiability modulo theories* (SMT) framework [4] for extending Boolean satisfiability in terms of graph-theoretic properties [5, 6]. The idea of *satisfiability modulo acyclicity* [7] is to view Boolean variables as conditionalized edges of a graph and to require that the graph remains acyclic under variable assignments. Moreover, the respective theory propagators for acyclicity have been implemented in contemporary CDCL-based SAT solvers, MINISAT and GLUCOSE, which offer a promising machinery for solving applications involving acyclicity constraints.

In this paper, we consider acyclicity constraints in the context of *answer set programming* (ASP) [8], featuring a rule-based language for knowledge representation. While SAT solvers with explicit acyclicity constraints offer an alternative mechanism to implement ASP via appropriate translations [7], the goal of this paper is different: the idea is to incorporate acyclicity constraints into ASP, thus accounting for extended rule types as well as reasoning tasks like enumeration and optimization. The resulting formalism, *ASP modulo acyclicity*, offers a rich set of primitives to express constraints related with recursive structures. The implementation, obtained as an extension to the state-of-the-art answer set solver CLASP [9], provides a unique combination of traditional unfounded set [10] checking and acyclicity propagation [5].

* This work was funded by AoF (251170), DFG (SCHA 550/8 and 550/9), as well as DAAD and AoF (57071677/279121). An extended draft with additional elaborations and experiments is available at `http://www.cs.uni-potsdam.de/wv/publications/`.

** Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

2 Background

We consider logic programs built from rules of the following forms:

$$a \leftarrow b_1, \ldots, b_n, \texttt{not}\ c_1, \ldots, \texttt{not}\ c_m. \quad (1)$$

$$\{a\} \leftarrow b_1, \ldots, b_n, \texttt{not}\ c_1, \ldots, \texttt{not}\ c_m. \quad (2)$$

$$a \leftarrow k \leq [b_1 = w_1, \ldots, b_n = w_n, \texttt{not}\ c_1 = w_{n+1}, \ldots, \texttt{not}\ c_m = w_{n+m}]. \quad (3)$$

Symbols $a, b_1, \ldots, b_n, c_1, \ldots, c_m$ stand for (propositional) *atoms*, $k, w_1, \ldots, w_{n+m}$ for non-negative integers, and `not` for (default) *negation*. Atoms like b_i and negated atoms like $\texttt{not}\ c_i$ are called *positive* and *negative literals*, respectively. For a *normal* (1), *choice* (2), or *weight* (3) *rule* r, we denote its *head* atom by $\mathrm{head}(r) = a$ and its *body* by $\mathrm{B}(r)$. By $\mathrm{B}(r)^+ = \{b_1, \ldots, b_n\}$ and $\mathrm{B}(r)^- = \{c_1, \ldots, c_m\}$, we refer to the *positive* and *negative body* atoms of r. When r is a weight rule, the respective sequence of *weighted literals* is denoted by $\mathrm{WL}(r)$, and its restrictions to positive or negative literals by $\mathrm{WL}(r)^+$ and $\mathrm{WL}(r)^-$. A normal rule r such that $\mathrm{head}(r) \in \mathrm{B}(r)^-$ is called an *integrity constraint*, and we below skip $\mathrm{head}(r)$ and $\texttt{not}\ \mathrm{head}(r)$ for brevity, where $\mathrm{head}(r)$ is an arbitrary atom occurring in r only. A *weight constraint program* P, or simply a *program*, is a finite set of rules; P is a *choice program* if it consists of normal and choice rules only, and a *positive program* if it involves neither negation nor choice rules.

Given a program P, let $\mathrm{head}(P) = \{\mathrm{head}(r) \mid r \in P\}$ and $\mathrm{At}(P) = \mathrm{head}(P) \cup \bigcup_{r \in P}(\mathrm{B}(r)^+ \cup \mathrm{B}(r)^-)$ denote the sets of head atoms or all atoms, respectively, occurring in P. The *defining rules* of an atom $a \in \mathrm{At}(P)$ are $\mathrm{Def}_P(a) = \{r \in P \mid \mathrm{head}(r) = a\}$. An *interpretation* $I \subseteq \mathrm{At}(P)$ satisfies $\mathrm{B}(r)$ for a normal or choice rule r iff $\mathrm{B}(r)^+ \subseteq I$ and $\mathrm{B}(r)^- \cap I = \emptyset$. The weighted literals of a weight rule r evaluate to $\mathrm{v}_I(\mathrm{WL}(r)) = \sum_{1 \leq i \leq n, b_i \in I} w_i + \sum_{1 \leq i \leq m, c_i \notin I} w_{n+i}$; when r is a weight rule, I satisfies $\mathrm{B}(r)$ iff $k \leq \mathrm{v}_I(\mathrm{WL}(r))$. For any rule r, we write $I \models \mathrm{B}(r)$ iff I satisfies $\mathrm{B}(r)$, and $I \models r$ iff $I \models \mathrm{B}(r)$ implies $\mathrm{head}(r) \in I$. The *supporting rules* of P with respect to I are $\mathrm{SR}_P(I) = \{r \in P \mid \mathrm{head}(r) \in I, I \models \mathrm{B}(r)\}$. Moreover, I is a *model* of P, denoted by $I \models P$, iff $I \models r$ for every $r \in P$ such that r is a normal or weight rule. A model I of P is a *supported model* of P when $\mathrm{head}(\mathrm{SR}_P(I)) = I$. Note that any positive program P possesses a unique *least model*, denoted by $\mathrm{LM}(P)$.

For a normal or choice rule r, $\mathrm{B}(r)^I = \mathrm{B}(r)^+$ denotes the reduct of $\mathrm{B}(r)$ with respect to an interpretation I, and $\mathrm{B}(r)^I = (\max\{0, k - \mathrm{v}_I(\mathrm{WL}(r)^-)\} \leq \mathrm{WL}(r)^+)$ is the reduct of $\mathrm{B}(r)$ for a weight rule r. The *reduct* of a program P with respect to an interpretation I is $P^I = \{\mathrm{head}(r) \leftarrow \mathrm{B}(r)^I \mid r \in \mathrm{SR}_P(I)\}$. Then, I is a *stable model* of P iff $I \models P$ and $\mathrm{LM}(P^I) = I$. While any stable model of P is a supported model of P as well, the converse does not hold in general. However, the following concept provides a tighter notion of support achieving such a correspondence.

Definition 1. *A model I of a program P is* well-supported *by a set $R \subseteq \mathrm{SR}_P(I)$ of rules iff $\mathrm{head}(R) = I$ and there is some ordering $r_1, \ldots, r_n$ of R such that, for each $1 \leq i \leq n$, $\mathrm{head}(\{r_1, \ldots, r_{i-1}\}) \models \mathrm{B}(r_i)^I$.*

In fact, a (supported) model I of a program P is stable iff I is well-supported by some subset of $\mathrm{SR}_P(I)$, and several such subsets may exist. The notion of well-support

counteracts circularity in the *positive dependency graph* $\mathrm{DG}^+(P) = \langle \mathrm{At}(P), \succeq \rangle$ of P, whose edge relation $a \succeq b$ holds for all $a, b \in \mathrm{At}(P)$ such that $\mathrm{head}(r) = a$ and $b \in \mathrm{B}(r)^+$ for some rule $r \in P$. If $a \succeq b$, we also write $\langle a, b \rangle \in \mathrm{DG}^+(P)$.

3 Acyclicity Constraints

In [5], the SAT problem has been extended by explicit acyclicity constraints. The basic idea is to label edges of a directed graph with dedicated Boolean variables. While satisfying the clauses of a SAT instance referring to these labeling variables, also the directed graph consisting of edges whose labeling variables are true must be kept acyclic. Thus, the graph behind the labeling variables imposes an additional constraint on satisfying assignments. In what follows, we propose a similar extension of logic programs subject to stable model semantics.

Definition 2. *The acyclicity extension of a logic program P is a pair $\langle V, e \rangle$, where*

1. *V is a set of nodes and*
2. *$e : \mathrm{At}(P) \to V \times V$ is a partial injection that maps atoms of P to edges.*

In the sequel, a program P is called an *acyclicity program* if it has an acyclicity extension $\langle V, e \rangle$. To define the semantics of acyclicity programs, we identify the graph of the acyclicity check as follows. Given an interpretation $I \subseteq \mathrm{At}(P)$, we write $e(I)$ for the set of edges $e(a)$ induced by atoms $a \in I$ for which $e(a)$ is *defined*. For a given acyclicity extension $\langle V, e \rangle$, the graph $e(\mathrm{At}(P))$ is the maximal one that can be obtained under any interpretation and is likely to contain cycles. If not, then the extension can be neglected altogether as no cycles can arise. To be precise about the acyclicity condition being imposed, we recall that a graph $\langle V, E \rangle$ with the set $E \subseteq V^2$ of edges has a *cycle* iff there is a non-trivial directed path from any node $v \in V$ back to itself via the edges in E. An *acyclic* graph $\langle V, E \rangle$ has no cycles of this kind.

Definition 3. *Let P be an acyclicity program with an acyclicity extension $\langle V, e \rangle$. An interpretation $M \subseteq \mathrm{At}(P)$ is a stable (or supported) model of P subject to $\langle V, e \rangle$ iff M is a stable (or supported) model of P such that the graph $\langle V, e(M) \rangle$ is acyclic.*

Example 1. Consider a directed graph $\langle V, E \rangle$ and the task to find a Hamiltonian cycle through the graph, i.e., a cycle that visits each node of the graph exactly once. Let us encode the graph by introducing the fact $\mathtt{node}(v)$ for each $v \in V$ and the fact $\mathtt{edge}(v, u)$ for each $\langle v, u \rangle \in E$. Then, it is sufficient (i) to pick beforehand an arbitrary initial node, say v_0, for the cycle, (ii) to select for each node exactly one outgoing and one incoming edge to be on the cycle, and (iii) to check that the cycle is not completed before the path spanning along the selected edges returns to v_0. Assuming that a predicate $\mathtt{hc}$ is used to represent selected edges, the following (first-order) rules express (ii):

$$1\{\mathtt{hc}(v, u) : \mathtt{edge}(v, u)\}1 \leftarrow \mathtt{node}(v).$$
$$1\{\mathtt{hc}(v, u) : \mathtt{edge}(v, u)\}1 \leftarrow \mathtt{node}(u).$$

To enforce (iii), we introduce an acyclicity extension $\langle V, e \rangle$, where e maps an atom $\mathtt{hc}(v, u)$ to an edge $\langle v, u \rangle$ whenever v and u are different from v_0. ■

Our next objective is to relate acyclicity programs with ordinary logic programs in terms of translations. It is well-known that logic programs subject to stable model semantics can express reachability in graphs, which implies that also acyclicity is expressible. To this end, we present a translation based on *elimination orderings* [11].

Definition 4. *Let P be an acyclicity program with an acyclicity extension $\langle V, e\rangle$. The translation $\mathrm{Tr}_{\mathrm{EL}}(P, V, e)$ extends P as follows.*

1. *For each atom $a \in \mathrm{At}(P)$ such that $e(a) = \langle v, u\rangle$, the rules:*

$$\mathtt{el}(v, u) \leftarrow \mathtt{not}\ a. \tag{4}$$
$$\mathtt{el}(v, u) \leftarrow \mathtt{el}(u). \tag{5}$$

2. *For each node $v \in V$ such that $\langle v, u_1\rangle, \ldots, \langle v, u_k\rangle$ are the edges in $e(\mathrm{At}(P))$ starting from v:*

$$\mathtt{el}(v) \leftarrow \mathtt{el}(v, u_1),\ \ldots,\ \mathtt{el}(v, u_k). \tag{6}$$
$$\leftarrow \mathtt{not}\ \mathtt{el}(v). \tag{7}$$

The intuitive reading of the new atom $\mathtt{el}(v, u)$ is that the edge $\langle v, u\rangle \in e(\mathrm{At}(P))$ has been eliminated, meaning that it cannot belong to any cycle. Analogously, the atom $\mathtt{el}(v)$ denotes the elimination of a node $v \in V$. By the rule (4), an edge $\langle v, u\rangle$ is eliminated when the atom a such that $e(a) = \langle v, u\rangle$ is false, while the rule (5) is applicable once the end node u is eliminated. Then, the node v gets eliminated by the rule (6) if all edges starting from it are eliminated. Finally, the constraint (7) ensures that all nodes are eliminated. That is, the success of the acyclicity test presumes that $\mathtt{el}(v, u)$ or $\mathtt{el}(v)$, respectively, is derivable for each edge $\langle v, u\rangle \in e(\mathrm{At}(P))$ and each node $v \in V$.

Theorem 1. *Let P be an acyclicity program with an acyclicity extension $\langle V, e\rangle$ and $\mathrm{Tr}_{\mathrm{EL}}(P, V, e)$ its translation into an ordinary logic program.*

1. *If M is a stable model of P subject to $\langle V, e\rangle$, then $M' = M \cup \{\mathtt{el}(v, u) \mid \langle v, u\rangle \in e(\mathrm{At}(P))\} \cup \{\mathtt{el}(v) \mid v \in V\}$ is a stable model of $\mathrm{Tr}_{\mathrm{EL}}(P, V, e)$.*
2. *If M' is a stable model of $\mathrm{Tr}_{\mathrm{EL}}(P, V, e)$, then $M = M' \cap \mathrm{At}(P)$ is a stable model of P subject to $\langle V, e\rangle$.*

Transformations in the other direction are of interest as well, i.e., the goal is to capture stable models by exploiting the acyclicity constraint. While the existing translation from ASP into SAT modulo acyclicity [7] provides a starting point for such a transformation, the target syntax is given by rules rather than clauses.

Definition 5. *Let P be a weight constraint program. The acyclicity translation of P consists of $\mathrm{Tr}_{\mathrm{ACYC}}(P) = \bigcup_{a \in \mathrm{At}(P)} \mathrm{Tr}_{\mathrm{ACYC}}(P, a)$ with an acyclicity extension $\langle \mathrm{At}(P), e\rangle$ such that $e(\mathtt{dep}(a, b)) = \langle a, b\rangle$ for each edge $\langle a, b\rangle \in \mathrm{DG}^+(P)$, where $\mathrm{Tr}_{\mathrm{ACYC}}(P, a)$ extends $\mathrm{Def}_P(a)$ for each atom $a \in \mathrm{At}(P)$ as follows.*

1. *For each edge $\langle a, b\rangle \in \mathrm{DG}^+(P)$, the choice rule:*

$$\{\mathtt{dep}(a, b)\} \leftarrow b. \tag{8}$$

2. *For each defining rule (1) or (2) of a, the rule:*

$$\mathtt{ws}(r) \leftarrow \mathtt{dep}(a, b_1), \ldots, \mathtt{dep}(a, b_n), \mathtt{not}\ c_1, \ldots, \mathtt{not}\ c_m. \tag{9}$$

3. *For each defining rule (3) of a, the rule:*

$$\mathtt{ws}(r) \leftarrow k \leq [\mathtt{dep}(a, b_1) = w_1, \ldots, \mathtt{dep}(a, b_n) = w_n, \mathtt{not}\ c_1 = w_{n+1}, \ldots, \mathtt{not}\ c_m = w_{n+m}]. \tag{10}$$

4. *For $\mathrm{Def}_P(a) = \{r_1, \ldots, r_k\}$, the constraint:*

$$\leftarrow a, \mathtt{not}\ \mathtt{ws}(r_1), \ldots, \mathtt{not}\ \mathtt{ws}(r_k). \tag{11}$$

The rules (9) and (10) specify when r provides well-support for a, i.e., the head atom a non-circularly depends on $\mathrm{B}(r)^+ = \{b_1, \ldots, b_n\}$. The constraint (11) expresses that $a \in \mathrm{At}(P)$ must have a well-supporting rule $r \in \mathrm{Def}_P(a)$ whenever a is true. To this end, respective dependencies have to be established in terms of the choice rules (8).

Theorem 2. *Let P be a weight constraint program and $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ its translation into an acyclicity program with an acyclicity extension $\langle \mathrm{At}(P), e\rangle$.*

1. *If M is a stable model of P, then there is an ordering $r_1, \ldots, r_n$ of some $R \subseteq \mathrm{SR}_P(M)$ such that $M' = M \cup \{\mathtt{ws}(r) \mid r \in R\} \cup \{\mathtt{dep}(\mathrm{head}(r_i), b) \mid 1 \leq i \leq n, b \in B_i\}$, where $B_i \subseteq \mathrm{B}(r_i)^+ \cap \mathrm{head}(\{r_1, \ldots, r_{i-1}\})$ for each $1 \leq i \leq n$, is a supported model of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ subject to $\langle \mathrm{At}(P), e\rangle$.*
2. *If M' is a supported model of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ subject to $\langle \mathrm{At}(P), e\rangle$, then $M = M' \cap \mathrm{At}(P)$ is a stable model of P and M is well-supported by $R = \{r \mid \mathtt{ws}(r) \in M'\}$.*

It is well-known that supported and stable models coincide for *tight* logic programs [12, 13]. The following theorem shows that translations produced by $\mathrm{Tr}_{\mathrm{ACYC}}$ possess an analogous property subject to the acyclicity extension $\langle \mathrm{At}(P), e\rangle$. This opens up an interesting avenue for investigating the efficiency of stable model computation—either using unfounded set checking or the acyclicity constraint, or both.

Theorem 3. *Let P be a weight constraint program and $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ its translation into an acyclicity program with an acyclicity extension $\langle \mathrm{At}(P), e\rangle$. Then, M is a supported model of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ subject to $\langle \mathrm{At}(P), e\rangle$ iff M is a stable model of $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ subject to $\langle \mathrm{At}(P), e\rangle$.*

As witnessed by Theorems 2 and 3, the translation $\mathrm{Tr}_{\mathrm{ACYC}}$ provides means to capture stability in terms of the acyclicity constraint. However, the computational efficiency of the translation can be improved when additional constraints governing $\mathtt{dep}(v, u)$ atoms are introduced. The purpose of these constraints is to falsify dependencies in settings where they are not truly needed. We first concentrate on choice programs and will then extend the consideration to weight rules below. The following definition adopts the cases from [7] but reformulates them in terms of rules rather than clauses.

Definition 6. *Let P be a choice program. The strong acyclicity translation of P, denoted by $\mathrm{Tr}_{\mathrm{ACYC+}}(P)$, extends $\mathrm{Tr}_{\mathrm{ACYC}}(P)$ as follows.*

1. *For each* $\langle a, b\rangle \in \mathrm{DG}^+(P)$*, the constraint:*

$$\leftarrow \mathtt{dep}(a, b),\ \mathtt{not}\ a. \tag{12}$$

2. *For each* $\langle a, b\rangle \in \mathrm{DG}^+(P)$ *and* $r \in \mathrm{Def}_P(a)$ *such that* $b \notin \mathrm{B}(r)^+$*, the constraint:*

$$\leftarrow \mathtt{dep}(a, b),\ \mathtt{ws}(r). \tag{13}$$

Intuitively, dependencies from a are not needed if a is false (12). Quite similarly, a particular dependency may be safely preempted (13) if the well-support for a is provided by a rule r not involving this dependency.

The strong acyclicity translation for weight rules includes additional subprograms.

Definition 7. *Let* P *be a weight constraint program and* $r \in P$ *a weight rule of the form (3), where* $\mathrm{head}(r) = a$*,* $|\{b_1, \ldots, b_n\}| = n$*, and* $w_1, \ldots, w_n$ *are ordered such that* $w_{i-1} \leq w_i$ *for each* $1 < i \leq n$*. The strong acyclicity translation* $\mathrm{Tr}_{\mathrm{ACYC}+}(P)$ *of* P *is fortified as follows.*

1. *For* $1 < i \leq n$*, the rules:*

$$\mathtt{nxt}(r, i) \leftarrow \mathtt{dep}(a, b_{i-1}). \tag{14}$$
$$\mathtt{nxt}(r, i) \leftarrow \mathtt{nxt}(r, i-1). \tag{15}$$
$$\mathtt{chk}(r, i) \leftarrow \mathtt{nxt}(r, i),\ \mathtt{dep}(a, b_i). \tag{16}$$

2. *The weight rule:*

$$\mathtt{red}(r) \leftarrow k \leq [\mathtt{chk}(r, 2) = w_2, \ldots, \mathtt{chk}(r, n) = w_n, \\ \mathtt{not}\ c_1 = w_{n+1}, \ldots, \mathtt{not}\ c_m = w_{n+m}]. \tag{17}$$

3. *For each* $\langle a, b\rangle \in \mathrm{DG}^+(P)$ *such that* $b \in \mathrm{B}(r)^+$*, the constraint:*

$$\leftarrow \mathtt{dep}(a, b),\ \mathtt{red}(r). \tag{18}$$

The idea is to cancel dependencies $\langle a, b\rangle \in \mathrm{DG}^+(P)$ by the constraint (18) when the well-support obtained though r can be deemed redundant by the rule (17). To this end, the rules of the forms (14) and (15) identify an atom among $b_1, \ldots, b_n$ of smallest weight having an active dependency from a, i.e., $\mathtt{dep}(a, b_i)$ is true, provided such an i exists. By the rules of the form (16), any further dependencies are extracted, and (17) checks whether the remaining literals are sufficient to reach the bound k. If so, all dependencies from a are viewed as redundant. This check covers also cases where, e.g., negative literals suffice to satisfy the body and positive dependencies play no role.

4 Discussion

In this paper, we propose a novel SMT-style extension of ASP by explicit acyclicity constraints in analogy to [5]. These kinds of constraints have not been directly addressed in previous SMT-style extensions of ASP [14–16]. The new extension, herein coined ASP

modulo acyclicity, offers a unique set of primitives for applications involving DAGs or tree structures. One interesting application is the embedding of ASP itself, given that unfounded set checking can be captured (Theorem 2). The utilized notion of well-supporting rules resembles *source pointers* [17], used in native answer set solvers to record rules justifying true atoms. In fact, a major contribution of this work is the implementation of new translations and principles in tools. For instance, CLASP [9] features enumeration and optimization, which are not supported by ACYCMINISAT and ACYCGLUCOSE [5]. Thereby, a replication of supported (and stable) models under translations can be avoided by using the projection capabilities of CLASP [18]. Last but not least, acyclicity programs enrich the variety of modeling primitives available to users.

References

1. Cussens, J.: Bayesian network learning with cutting planes. In: Proc. UAI'11. AUAI Press (2011) 153–160
2. Corander, J., Janhunen, T., Rintanen, J., Nyman, H., Pensar, J.: Learning chordal Markov networks by constraint satisfaction. In: Proc. NIPS'13. NIPS Foundation (2013) 1349–1357
3. Erdem, E., Lifschitz, V., Wong, M.: Wire routing and satisfiability planning. In: Proc. CL'00. Springer (2000) 822–836
4. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. IOS Press (2009) 825–885
5. Gebser, M., Janhunen, T., Rintanen, J.: SAT modulo graphs: Acyclicity. In: Proc. JELIA'14. Springer (2014) 137–151
6. Bayless, S., Bayless, N., Hoos, H., Hu, A.: SAT modulo monotonic theories. In: Proc. AAAI'15. AAAI Press (2015) 3702–3709
7. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In: Proc. ECAI'14. IOS Press (2014) 351–356
8. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
9. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187-188** (2012) 52–89
10. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
11. Gebser, M., Janhunen, T., Rintanen, J.: ASP encodings of acyclicity properties. In: Proc. KR'14. AAAI Press (2014)
12. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
13. Erdem, E., Lifschitz, V.: Tight logic programs. Theory and Practice of Logic Programming **3**(4-5) (2003) 499–518
14. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proc. ICLP'09. Springer (2009) 235–249
15. Liu, G., Janhunen, T., Niemelä, I.: Answer set programming via mixed integer programming. In: Proc. KR'12. AAAI Press (2012) 32–42
16. Lee, J., Meng, Y.: Answer set programming modulo theories and reasoning about continuous changes. In: Proc. IJCAI'13. IJCAI/AAAI Press (2013) 990–996
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
18. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected Boolean search problems. In: Proc. CPAIOR'09. Springer (2009) 71–86