Writing Declarative Specifications for Clauses [⋆]

Martin Gebser[1,2], Tomi Janhunen[1], Roland Kaminski[2], Torsten Schaub[2,3⋆⋆], and Shahab Tasharrofi[1]

[1] Helsinki Institute for Information Technology HIIT, Aalto University, FINLAND
[2] Institute for Informatics and Computational Science, University of Potsdam, GERMANY
[3] INRIA Rennes, Bretagne Atlantique Research Centre, FRANCE

Abstract. Modern satisfiability (SAT) solvers provide an efficient implementation of classical propositional logic. Their input language, however, is based on the conjunctive normal form (CNF) of propositional formulas. To use SAT solver technology in practice, a user must create the input clauses in one way or another. A typical approach is to write a procedural program that generates formulas on the basis of some input data relevant for the problem domain and translates them into CNF. In this paper, we propose a declarative approach where the intended clauses are specified in terms of rules in analogy to answer set programming (ASP). This allows the user to write first-order specifications for intended clauses in a schematic way by exploiting term variables. We develop a formal framework required to define the semantics of such specifications. Moreover, we provide an implementation harnessing state-of-the-art ASP grounders to accomplish the grounding step of clauses. As a result, we obtain a general-purpose clause-level grounding approach for SAT solvers. Finally, we illustrate the capabilities of our specification methodology in terms of combinatorial and application problems.

1 Introduction

Satisfiability (SAT) solvers [4] provide an efficient way to implement classical propositional logic. The conjunctive normal form (CNF) of formulas, which is based on disjunctions of literals also known as *clauses*, forms the standard input language supported by solvers. However, writing clauses directly is not very practical from the modeling perspective. This suggests the use of a more expressive language supporting the entire range of logical connectives and allowing for (universally quantified) first-order variables to write formulas in a schematic way. E.g., the following formula aims to deny occurrences of triangles in a directed graph represented by the $\mathsf{edge}/2$ predicate:

$$\mathsf{edge}(X,Y) \land \mathsf{edge}(Y,Z) \land (X \neq Y) \land (X \neq Z) \land (Y \neq Z) \rightarrow \neg\mathsf{edge}(Z,X). \quad (1)$$

On the one hand, variables seem crucial to achieve the flexibility required in modeling but, on the other hand, they lead to the problem of instantiating or *grounding* the variables when actual inference is performed. In the presence of facts $\mathsf{edge}(a,b)$, $\mathsf{edge}(b,c)$, and $\mathsf{edge}(c,a)$, the essential step is to substitute the universally quantified variables X,

[⋆] This work was funded by AoF (251170), DFG (SCHA 550/8 and 550/9), as well as DAAD and AoF (57071677/279121).

[⋆⋆] Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

Y, and Z in (1) by the constants a, b, and c. While $3^3 = 27$ different substitutions are applicable, only one of them is useful for showing unsatisfiability. The theory of grounding goes back to Herbrand's seminal work, and it has been addressed in many contexts, such as first-order model generation and theorem proving (see, e.g., [1, 21]) as well as AI planning (cf. [14]). The substitution of variables by constants or more generally ground terms is subject to combinatorial explosion when the underlying domain grows. To cut down the number of resulting ground instances, a variety of techniques have been proposed, including clause splitting, structural constraints, and contraction techniques to discard or simplify instances [24]. Also, by carefully analyzing variable ranges, it is possible to reduce the number of clauses or formulas generated [21, 29].

The approach proposed in this paper also relies on domain information, but we suggest to use declarative specifications based on *closed world assumption* (CWA) for controlling domains. In case of (1), this means that there is no edge between any given pair of nodes, thus falsifying the implication antecedent, unless specified otherwise. We provide an implementation harnessing state-of-the-art *answer set programming* (ASP) [6] grounders for the computation of domains and variable instantiation, since they offer built-in support for CWA and a rich rule-based language to express domain knowledge.

What remains is choosing the kind of formulas to ground. While free choice among logical connectives seems desirable from the modeling perspective, translation into CNF is necessary to use SAT solvers. The clausification of propositional (ground) formulas often requires the introduction of new variables, e.g., using the Tseitin transformation, to avoid exponential blow-ups, and in some cases the auxiliary variables significantly affect solver performance [2, 3, 15]. The idea of this paper is to write declarative specifications for clauses, thus enabling a user to define the input of a SAT solver directly. Following the traditional *what you see is what you get* principle, clauses in the grounder output can be traced back to the schematic specification. The trade-off is that the user has to decide about potential new variables in a formalization, but specifying such variables at the schematic level also provides more direct access than an implicit clause compilation. In fact, given the expressiveness of modeling languages supported by off-the-shelf ASP grounders [12, 19], we expect that declarative specifications are easier to develop and maintain than their procedural counterparts. For one, it is possible to separate domain descriptions from logical axioms, which enables *uniform* encodings that are independent of particular instance data [23]. For another, the level of abstraction provided by first-order rules makes specifications highly *elaboration tolerant* [20].

The rest of this paper is organized as follows. The syntax and semantics of the clause specification language is defined in Section 2. In Section 3, we illustrate the proposed language on practical modeling scenarios. Section 4 presents a streamlined implementation, interfacing the state-of-the-art ASP grounder GRINGO [11] with SAT or MaxSAT solvers. Finally, we discuss related work and conclude the paper in Section 5.

2 Clause Programs

We begin by presenting the syntax of clause programs and then concentrate on defining their semantics. To specify clause programs in the first-order case with variables, we define *terms* as expressions built from function symbols f, also called constants in case of arity zero, or variable symbols X. The signature for predicate symbols, denoted by $\mathcal{P}$, splits into $\mathcal{P}_{\mathrm{d}}$ and $\mathcal{P}_{\mathrm{v}}$, i.e., *domain* predicates being minimized and those allowed to *vary* as typical in classical logic. A *first-order atom* $\mathsf{p}(t_1, \ldots, t_n)$, or an *atom* for short, consists of an n-ary predicate symbol $\mathsf{p} \in \mathcal{P}$ and terms $t_1, \ldots, t_n$ listed as its arguments. A *literal* is either an atom a or its negation $\neg a$.

A *clause program* P can have rules of two kinds: *domain rules* of the form (2), also known as normal rules in ASP, as well as *clause rules* of the form (3):

$$a \leftarrow c_1, \ldots, c_m, {\sim}d_1, \ldots, {\sim}d_n. \tag{2}$$

$$a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l \leftarrow c_1, \ldots, c_m, {\sim}d_1, \ldots, {\sim}d_n. \tag{3}$$

In the rules above, a, $c_1, \ldots, c_m$, and $d_1, \ldots, d_n$ are domain atoms expressed in $\mathcal{P}_{\mathrm{d}}$, and the symbol $\sim$ stands for default negation. Domain rules (2) are used to specify appropriate domain relations for variable instantiation. The atoms $a_1, \ldots, a_k$ and $b_1, \ldots, b_l$ in a clause rule (3) are expressed in $\mathcal{P}_{\mathrm{v}}$. The *head* $a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l$ is a schema for propositional clauses where $\vee$ and $\neg$ stand for classical disjunction and negation, respectively. The *body* $c_1, \ldots, c_m, {\sim}d_1, \ldots, {\sim}d_n$ essentially provides the conditions for creating the head clause, which also includes determining variable assignments.

The semantics of clause programs is defined using Herbrand models as follows. Given a clause program P, we define its Herbrand universe $\mathrm{Hu}(P)$ and Herbrand base $\mathrm{Hb}(P)$ in the standard way. The base $\mathrm{Hb}(P)$ is partitioned into $\mathrm{Hb}_{\mathrm{d}}(P)$ and $\mathrm{Hb}_{\mathrm{v}}(P)$ based on the signatures $\mathcal{P}_{\mathrm{d}}$ and $\mathcal{P}_{\mathrm{v}}$, respectively. A (Herbrand) *interpretation* I of P is written as a subset of $\mathrm{Hb}(P)$. Moreover, we distinguish its projections $I_{\mathrm{d}} = I \cap \mathrm{Hb}_{\mathrm{d}}(P)$ and $I_{\mathrm{v}} = I \cap \mathrm{Hb}_{\mathrm{v}}(P)$. Assuming that P is variable-free or *ground*, the body of (2) or (3) is satisfied in I iff $\{c_1, \ldots, c_m\} \subseteq I_{\mathrm{d}}$ and $\{d_1, \ldots, d_n\} \cap I_{\mathrm{d}} = \emptyset$. The head of (2) is satisfied in I iff $a \in I_{\mathrm{d}}$, while the head of (3) is satisfied in I iff $\{b_1, \ldots, b_l\} \subseteq I_{\mathrm{v}}$ implies $\{a_1, \ldots, a_k\} \cap I_{\mathrm{v}} \neq \emptyset$. An interpretation $I \subseteq \mathrm{Hb}(P)$ is a *model* of P iff, for every rule (2) or (3) of P, the satisfaction of the body in I implies the satisfaction of the head in I. To enforce the minimal interpretation of domain predicates, we define the *domain reduct* P^I of P with respect to I to contain a rule $a \leftarrow c_1, \ldots, c_m$ for every domain rule (2) of P such that $\{d_1, \ldots, d_n\} \cap I_{\mathrm{d}} = \emptyset$. The program P^I is a Horn theory and guaranteed to have a unique $\subseteq$-minimal model over $\mathrm{Hb}_{\mathrm{d}}(P)$, the *least model* of P^I.

Definition 1. *Let P be a clause program and* $\mathrm{Gnd}(P)$ *the respective Herbrand instantiation of P over* $\mathrm{Hu}(P)$. *An interpretation $I \subseteq \mathrm{Hb}(P)$ is a* domain stable *model of P iff I is a model of* $\mathrm{Gnd}(P)$ *such that I_{d} is the least model of* $\mathrm{Gnd}(P)^I$.

While the abstract criteria for domain stable models are formulated in terms of the full Herbrand instantiation $\mathrm{Gnd}(P)$, the actual goal is to generate small subsets of $\mathrm{Gnd}(P)$ without affecting domain stable models. The intended way of applying Definition 1 in practice is to let an ASP grounder calculate I_{d}, which also determines the relevant clauses. After that, a SAT solver can be invoked to compute I_{v} such that $I = I_{\mathrm{d}} \cup I_{\mathrm{v}}$ is a model of $\mathrm{Gnd}(P)$. In order to use ASP grounders, we have to restrict variable occurrences in rules. A rule of the form (2) or (3) is called *safe* if all variables occurring in the head also appear in the positive conditions $c_1, \ldots, c_m$ of the body, which thereafter constrain their domains. Moreover, it is reasonable to assume that the domain part of a clause program P has a total well-founded model (cf. [28]) that can be calculated by an ASP grounder. We therefore require domain rules (2) of P to be *stratified* (cf. [26]), which confines recursive dependencies of a predicate in $\mathcal{P}_{\mathrm{d}}$ on itself to be purely based on $c_1, \ldots, c_m$ in the positive body parts of rules. All clause programs considered in the following are safe and their domain rules stratified. This means that rule bodies are fully evaluated during grounding, and the heads of clause rules (3) provide the input of a SAT solver, searching for (classical) models of the propositional clauses.

Example 1. Let us consider the following clause program for graph coloring:

$$\mathsf{node}(X) \leftarrow \mathsf{edge}(X,Y). \tag{4}$$
$$\mathsf{node}(Y) \leftarrow \mathsf{edge}(X,Y). \tag{5}$$
$$\mathsf{black}(X) \vee \mathsf{grey}(X) \vee \mathsf{white}(X) \leftarrow \mathsf{node}(X). \tag{6}$$
$$\neg\mathsf{black}(X) \vee \neg\mathsf{black}(Y) \leftarrow \mathsf{edge}(X,Y). \tag{7}$$
$$\neg\mathsf{grey}(X) \vee \neg\mathsf{grey}(Y) \leftarrow \mathsf{edge}(X,Y). \tag{8}$$
$$\neg\mathsf{white}(X) \vee \neg\mathsf{white}(Y) \leftarrow \mathsf{edge}(X,Y). \tag{9}$$

The idea is that these rules are conjoined with facts representing an input graph. To this end, let us use the three facts from the context of (1). Together with the domain rules (4) and (5), such facts give rise to the following least model I_{d}:

$$\mathsf{edge}(a,b),\ \mathsf{edge}(b,c),\ \mathsf{edge}(c,a),\ \mathsf{node}(a),\ \mathsf{node}(b),\ \text{and}\ \mathsf{node}(c).$$

The atoms in I_{d} determine the domains of variables in (6)–(9), resulting in the clauses:

$$\begin{array}{ccc}
 & \mathsf{black}(a) \vee \mathsf{grey}(a) \vee \mathsf{white}(a), & \\
 & \mathsf{black}(b) \vee \mathsf{grey}(b) \vee \mathsf{white}(b), & \\
 & \mathsf{black}(c) \vee \mathsf{grey}(c) \vee \mathsf{white}(c), & \\
\neg\mathsf{black}(a) \vee \neg\mathsf{black}(b), & \neg\mathsf{black}(b) \vee \neg\mathsf{black}(c), & \neg\mathsf{black}(c) \vee \neg\mathsf{black}(a), \\
\neg\mathsf{grey}(a) \vee \neg\mathsf{grey}(b), & \neg\mathsf{grey}(b) \vee \neg\mathsf{grey}(c), & \neg\mathsf{grey}(c) \vee \neg\mathsf{grey}(a), \\
\neg\mathsf{white}(a) \vee \neg\mathsf{white}(b), & \neg\mathsf{white}(b) \vee \neg\mathsf{white}(c), & \neg\mathsf{white}(c) \vee \neg\mathsf{white}(a).
\end{array}$$

These clauses can be satisfied, e.g., by letting $I_{\mathrm{v}} = \{\mathsf{black}(a), \mathsf{grey}(b), \mathsf{white}(c)\}$, which gives rise to a domain stable model $I = I_{\mathrm{d}} \cup I_{\mathrm{v}}$. ■

3 Modeling Methodology and Applications

We have above introduced the paradigm of clause programs in a simple setting where the domain part is written in *normal* ASP-style rules. Using syntactic sugar available in GRINGO, however, the compactness and flexibility of clause programs can be further enhanced. We below illustrate the practice of clause programs on several use cases.

Graph Coloring. To begin with, we generalize the program in Example 1 to n colors:

$$\mathsf{color}(1 \ldots n). \tag{10}$$
$$\mathsf{node}(X;Y) \leftarrow \mathsf{edge}(X,Y). \tag{11}$$
$$\bigvee \mathsf{hascolor}(X,C) : \mathsf{color}(C) \leftarrow \mathsf{node}(X). \tag{12}$$
$$\neg\mathsf{hascolor}(X,C) \vee \neg\mathsf{hascolor}(Y,C) \leftarrow \mathsf{edge}(X,Y),\ \mathsf{color}(C). \tag{13}$$

By setting the constant n to some integer, say 3, it defines a range of colors by (10): $\mathsf{color}(1)$, $\mathsf{color}(2)$, and $\mathsf{color}(3)$. The separator ";" in the second domain rule (11) is used to specify alternative terms for which the head atom is instantiated, so that (11) amalgamates (4) and (5). Unlike (6), the clause rule (12), applying to each term X from $\mathsf{node}(X)$, is parametrized by a *conditional literal* $\mathsf{hascolor}(X,C)$, where instances over all terms C from $\mathsf{color}(C)$ are included in a disjunction. This enables the specification of clauses whose length depends *dynamically* on a problem instance, such as the number of colors in this case. Finally, the clause rule (13) generalizes (7)–(9).

Example 2. Based on the least model I_{d} from Example 1, augmented with $\mathsf{color}(1)$, $\mathsf{color}(2)$, and $\mathsf{color}(3)$, the clauses obtained from (12) and (13) are as follows:

$$\begin{array}{ll}
\multicolumn{2}{c}{\mathsf{hascolor}(a,1) \vee \mathsf{hascolor}(a,2) \vee \mathsf{hascolor}(a,3),}\\
\multicolumn{2}{c}{\mathsf{hascolor}(b,1) \vee \mathsf{hascolor}(b,2) \vee \mathsf{hascolor}(b,3),}\\
\multicolumn{2}{c}{\mathsf{hascolor}(c,1) \vee \mathsf{hascolor}(c,2) \vee \mathsf{hascolor}(c,3),}\\
\neg\mathsf{hascolor}(a,1) \vee \neg\mathsf{hascolor}(b,1), & \neg\mathsf{hascolor}(b,1) \vee \neg\mathsf{hascolor}(c,1),\\
\neg\mathsf{hascolor}(a,2) \vee \neg\mathsf{hascolor}(b,2), & \neg\mathsf{hascolor}(b,2) \vee \neg\mathsf{hascolor}(c,2),\\
\neg\mathsf{hascolor}(a,3) \vee \neg\mathsf{hascolor}(b,3), & \neg\mathsf{hascolor}(b,3) \vee \neg\mathsf{hascolor}(c,3),\\
\neg\mathsf{hascolor}(c,1) \vee \neg\mathsf{hascolor}(a,1), & \\
\neg\mathsf{hascolor}(c,2) \vee \neg\mathsf{hascolor}(a,2), & \neg\mathsf{hascolor}(c,3) \vee \neg\mathsf{hascolor}(a,3).
\end{array}$$

The clauses resemble those in Example 1, yet using the generic predicate $\mathsf{hascolor}/2$, including colors as arguments, rather than $\mathsf{black}/1$, $\mathsf{grey}/1$, and $\mathsf{white}/1$. Accordingly, an assignment of distinct colors to the three nodes at hand is expressed by a projection like $I_{\mathrm{v}} = \{\mathsf{hascolor}(a,1), \mathsf{hascolor}(b,2), \mathsf{hascolor}(c,3)\}$. ■

n-Queens. The next clause program, encoding the well-known n-queens problem, illustrates the use of built-in integer arithmetic supported by ASP grounders like GRINGO:

$$\mathsf{coord}(1\dots n). \quad \mathsf{dir}(0,-1). \quad \mathsf{dir}(-1,0). \quad \mathsf{dir}(-1,-1). \quad \mathsf{dir}(-1,1). \tag{14}$$

$$\mathsf{target}(X,Y,R,C) \leftarrow \mathsf{coord}(X;Y;X{+}R;Y{+}C), \mathsf{dir}(R,C). \tag{15}$$

$$\mathsf{attack}(X{+}R,Y{+}C,R,C) \vee \neg\mathsf{queen}(X,Y) \leftarrow \mathsf{target}(X,Y,R,C). \tag{16}$$

$$\begin{aligned}&\mathsf{attack}(X{+}R,Y{+}C,R,C) \vee \neg\mathsf{attack}(X,Y,R,C)\\ &\qquad\qquad \leftarrow \mathsf{target}(X,Y,R,C), \mathsf{target}(X{-}R,Y{-}C,R,C).\end{aligned} \tag{17}$$

$$\begin{aligned}&\neg\mathsf{attack}(X{+}R,Y{+}C,R,C) \vee \mathsf{queen}(X,Y) \vee\\ &\quad \bigvee \mathsf{attack}(X,Y,R,C) : \mathsf{target}(X{-}R,Y{-}C,R,C) \leftarrow \mathsf{target}(X,Y,R,C).\end{aligned} \tag{18}$$

$$\neg\mathsf{queen}(X{+}R,Y{+}C) \vee \neg\mathsf{attack}(X{+}R,Y{+}C,R,C) \leftarrow \mathsf{target}(X,Y,R,C). \tag{19}$$

$$\mathsf{queen}(X,1) \vee \bigvee \mathsf{attack}(X,1,0,-1) : \mathsf{target}(X,2,0,-1) \leftarrow \mathsf{coord}(X). \tag{20}$$

$$\mathsf{queen}(1,Y) \vee \bigvee \mathsf{attack}(1,Y,-1,0) : \mathsf{target}(2,Y,-1,0) \leftarrow \mathsf{coord}(Y). \tag{21}$$

The facts in (14) provide row and column coordinates, ranging from 1 to some integer value for n, as well as the differences between the coordinates of adjacent cells in horizontal, vertical, and diagonal directions. Particular adjacent cells are indicated by the domain rule (15), where an instance of $\mathsf{target}(X,Y,R,C)$ expresses that the cells at coordinates (X,Y) and $(X{+}R,Y{+}C)$ are adjacent. Given this, the clause rules

(16)–(18) specify conditions enforcing that $\mathsf{attack}(X{+}R, Y{+}C, R, C)$ is true iff some cell with coordinates $(X{-}k{*}R, Y{-}k{*}C)$ for $k \geq 0$ hosts a queen, represented by a corresponding instance of $\mathsf{queen}(X,Y)$. The clauses specified by (19) then forbid a queen at $(X{+}R, Y{+}C)$ if the cell is horizontally, vertically, or diagonally attacked. Finally, the clause rules (20) and (21) express that any row or column must contain some queen, which can be checked at the first row or column position, respectively.

Example 3. For $n = 4$, the least model I_{d} includes the following atoms indicating horizontal attacks along the first row, corresponding to substitutions that instantiate X, R, and C with 1, 0, and -1 in (15):

$$\mathsf{target}(1,2,0,-1), \mathsf{target}(1,3,0,-1), \text{and } \mathsf{target}(1,4,0,-1).$$

These atoms induce nine instances of the clause rules (16)–(18), whose conjunction is equivalent to the following formulas:

$$\begin{aligned}
&\mathsf{attack}(1,1,0,-1) \leftrightarrow \mathsf{queen}(1,2) \vee \mathsf{attack}(1,2,0,-1),\\
&\mathsf{attack}(1,2,0,-1) \leftrightarrow \mathsf{queen}(1,3) \vee \mathsf{attack}(1,3,0,-1),\\
&\mathsf{attack}(1,3,0,-1) \leftrightarrow \mathsf{queen}(1,4).
\end{aligned}$$

Respective clauses obtained from (19) exclude queens at horizontally attacked cells:

$$\begin{aligned}
&\neg\mathsf{queen}(1,1) \vee \neg\mathsf{attack}(1,1,0,-1),\\
&\neg\mathsf{queen}(1,2) \vee \neg\mathsf{attack}(1,2,0,-1),\\
&\neg\mathsf{queen}(1,3) \vee \neg\mathsf{attack}(1,3,0,-1).
\end{aligned}$$

Note that corresponding formulas (defining instances of $\mathsf{attack}(X{+}R, Y{+}C, R, C)$) and clauses forbid that queens attack one another along other rows, columns, or diagonals. Finally, clauses like $\mathsf{queen}(1,1) \vee \mathsf{attack}(1,1,0,-1)$ from (20) and (21) require some queen in each row and column, so that instances of $\mathsf{queen}(X,Y)$ in a projection I_{v} provide solutions to the n-queens problem. In fact, the two solutions for $n = 4$ are characterized by domain stable models including $\mathsf{queen}(1,2)$, $\mathsf{queen}(2,4)$, $\mathsf{queen}(3,1)$, and $\mathsf{queen}(4,3)$ or $\mathsf{queen}(1,3)$, $\mathsf{queen}(2,1)$, $\mathsf{queen}(3,4)$, and $\mathsf{queen}(4,2)$, respectively. ■

Markov Network Structure Learning. After considering combinatorial problems, we now turn to encodings inspired by practical application scenarios. To begin with, the Markov network structure learning problem in [7] is based on undirected (acyclic) forests, and the following clause program implements the respective SAT encoding part, described (in natural language) in [7]:

$$\mathsf{node}(1 \ldots n). \tag{22}$$

$$\mathsf{level}(0 \ldots (n-1)/2). \tag{23}$$

$$\mathsf{pair}(X,Y) \leftarrow \mathsf{node}(X;Y),\ X < Y. \tag{24}$$

$$\mathsf{maps}(X,Y,X,Y) \leftarrow \mathsf{pair}(X,Y). \tag{25}$$

$$\mathsf{maps}(Y,X,X,Y) \leftarrow \mathsf{pair}(X,Y). \tag{26}$$

$$\begin{aligned}\mathsf{del}(X,L) \vee \bigvee \mathsf{edge}(X_1,Y_1,L) : \mathsf{maps}(X,Y,X_1,Y_1) : Y \neq Z&\\ \leftarrow \mathsf{node}(X;Z), \mathsf{level}(L), X \neq Z.&\end{aligned} \tag{27}$$

$$\begin{aligned}\neg\mathsf{del}(X,L) \vee \neg\mathsf{edge}(X_1,Y_1,L) \vee \neg\mathsf{edge}(X_2,Y_2,L)&\\ \leftarrow \mathsf{maps}(X,Y,X_1,Y_1),\ \mathsf{maps}(X,Z,X_2,Y_2),\ \mathsf{level}(L),\ Y < Z.&\end{aligned} \tag{28}$$

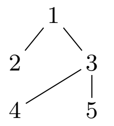

Fig. 1. An undirected forest with $n = 5$ nodes

$$\mathsf{edge}(X, Y, L) \lor \neg\mathsf{edge}(X, Y, L-1) \lor \mathsf{del}(X, L-1) \lor \mathsf{del}(Y, L-1) \leftarrow \mathsf{pair}(X, Y),\ \mathsf{level}(L),\ 0 < L. \quad (29)$$

$$\neg\mathsf{edge}(X, Y, L) \lor \mathsf{edge}(X, Y, L-1) \leftarrow \mathsf{pair}(X, Y),\ \mathsf{level}(L),\ 0 < L. \quad (30)$$

$$\neg\mathsf{edge}(X, Y, L) \lor \neg\mathsf{del}(X_1, L-1) \leftarrow \mathsf{maps}(X_1, Y_1, X, Y),\ \mathsf{level}(L),\ 0 < L. \quad (31)$$

$$\mathsf{del}(X, (n-1)/2) \leftarrow \mathsf{node}(X). \quad (32)$$

$$\mathsf{edge}(X, Y) \lor \neg\mathsf{edge}(X, Y, 0) \leftarrow \mathsf{pair}(X, Y). \quad (33)$$

$$\neg\mathsf{edge}(X, Y) \lor \mathsf{edge}(X, Y, 0) \leftarrow \mathsf{pair}(X, Y). \quad (34)$$

The domain rules (22)–(26) provide the n nodes of a graph, ordered pairs of them as edge candidates, $\lceil \frac{n}{2} \rceil$ levels for a fixpoint construction to check acyclicity, and instances of $\mathsf{maps}(X, Y, X_1, Y_1)$ to access the ordered edge representation (X_1, Y_1) for unordered distinct nodes X and Y. Intuitively, the clause rules (27) and (28) express that a node X can be *deleted* at a level L iff it participates in at most one undeleted edge at L. While non-deletion of a candidate pair (X, Y) at level 0 indicates that (X, Y) contributes an edge, (X, Y) is undeleted at a greater level L iff (X, Y) as well as its incident nodes X and Y are undeleted at $L - 1$, where the latter is specified by the clause rules (29)–(31). The unit clauses posted by (32) require any node to be deleted at the maximum level, thus enforcing acyclicity. Finally, the equivalences between ternary $\mathsf{edge}(X, Y, 0)$ and binary $\mathsf{edge}(X, Y)$ instances asserted by (33) and (34) serve merely for symbolic output representation.

Example 4. Figure 1 displays an undirected graph with $n = 5$ nodes, so that the levels 0, 1, and 2 are available for verifying acyclicity. The undeleted edges at level 0, those shown in Figure 1, are reflected by true instances of $\mathsf{edge}(X, Y, 0)$, i.e., $\mathsf{edge}(1, 2, 0)$, $\mathsf{edge}(1, 3, 0)$, $\mathsf{edge}(3, 4, 0)$, and $\mathsf{edge}(3, 5, 0)$. Observe that the nodes 2, 4, and 5 have just one incident edge each, and the following clauses from (27), whose contained instances of $\mathsf{edge}(X, Y, 0)$ are false, assert their deletion at level 0:

$$\begin{gathered}\mathsf{del}(2,0) \lor \mathsf{edge}(2,3,0) \lor \mathsf{edge}(2,4,0) \lor \mathsf{edge}(2,5,0),\\ \mathsf{del}(4,0) \lor \mathsf{edge}(1,4,0) \lor \mathsf{edge}(2,4,0) \lor \mathsf{edge}(4,5,0),\\ \mathsf{del}(5,0) \lor \mathsf{edge}(1,5,0) \lor \mathsf{edge}(2,5,0) \lor \mathsf{edge}(4,5,0).\end{gathered}$$

On the other hand, the nodes 1 and 3 cannot be deleted at level 0, as expressed by instances of (28) such that $\mathsf{edge}(X, Y, 0)$ holds for literals of the form $\neg\mathsf{edge}(X, Y, 0)$:

$$\begin{gathered}\neg\mathsf{del}(1,0) \lor \neg\mathsf{edge}(1,2,0) \lor \neg\mathsf{edge}(1,3,0),\\ \neg\mathsf{del}(3,0) \lor \neg\mathsf{edge}(1,3,0) \lor \neg\mathsf{edge}(3,4,0),\\ \neg\mathsf{del}(3,0) \lor \neg\mathsf{edge}(1,3,0) \lor \neg\mathsf{edge}(3,5,0),\\ \neg\mathsf{del}(3,0) \lor \neg\mathsf{edge}(3,4,0) \lor \neg\mathsf{edge}(3,5,0).\end{gathered}$$

Note that either of the three clauses including $\neg\mathsf{del}(3, 0)$ is sufficient to falsify $\mathsf{del}(3, 0)$. However, the deletion of the nodes 2, 4, and 5, indicated by $\mathsf{del}(2, 0)$, $\mathsf{del}(4, 0)$, $\mathsf{del}(5, 0)$, leads to deletion of (further) edges at level 1 via corresponding clauses from (31):

$$\neg\mathsf{edge}(1,2,1) \vee \neg\mathsf{del}(2,0),\ \ \neg\mathsf{edge}(3,4,1) \vee \neg\mathsf{del}(4,0),\ \ \neg\mathsf{edge}(3,5,1) \vee \neg\mathsf{del}(5,0).$$

Since false instances of $\mathsf{edge}(X,Y,0)$ are propagated to level 1 by clauses from (30), e.g., $\neg\mathsf{edge}(1,4,1) \vee \mathsf{edge}(1,4,0)$, the only atom of the form $\mathsf{edge}(X,Y,1)$ that holds, asserted by the instance $\mathsf{edge}(1,3,1) \vee \neg\mathsf{edge}(1,3,0) \vee \mathsf{del}(1,0) \vee \mathsf{del}(3,0)$ of (29), is $\mathsf{edge}(1,3,1)$. Nevertheless, the nodes 1 and 3 are in turn deleted at level 1 in view of the following clauses from (27):

$$\begin{array}{l}\mathsf{del}(1,1) \vee \mathsf{edge}(1,2,1) \vee \mathsf{edge}(1,4,1) \vee \mathsf{edge}(1,5,1),\\ \mathsf{del}(3,1) \vee \mathsf{edge}(2,3,1) \vee \mathsf{edge}(3,4,1) \vee \mathsf{edge}(3,5,1).\end{array}$$

Given that instances of (30) carry once deleted edges forward to greater levels, in view of (27), also deleted nodes are propagated on. As a consequence, $\mathsf{del}(1,2)$, $\mathsf{del}(2,2)$, $\mathsf{del}(3,2)$, $\mathsf{del}(4,2)$, and $\mathsf{del}(5,2)$ hold, as required by (32) for the maximum level 2. That is, the undirected forest in Figure 1 passes the acyclicity check, whereas any node X involved in a cycle could never be deleted and would yield a contradiction with (32). ■

Instruction Scheduling. As a second application problem, we consider instruction scheduling [27] as performed by compilers. The input can be viewed as a directed acyclic graph with weighted edges, representing dependencies between instructions along with latencies. The task consists of mapping instructions to execution slots such that the distances between dependent instructions' slots respect latencies and the execution finishes as early as possible. In order to represent this problem in propositional logic, the number of execution slots must be finite, and the following domain rules can be used for extracting bounds:

$$\mathsf{lower}(I_1,1) \leftarrow \mathsf{instruction}(I_1). \tag{35}$$

$$\mathsf{lower}(I_2,L+G) \leftarrow \mathsf{lower}(I_1,L),\ \mathsf{gap}(I_1,I_2,G). \tag{36}$$

$$\mathsf{reach}(I_1,I_2) \leftarrow \mathsf{gap}(I_1,I_2,G). \tag{37}$$

$$\mathsf{reach}(I_1,I_3) \leftarrow \mathsf{gap}(I_1,I_2,G),\ \mathsf{reach}(I_2,I_3). \tag{38}$$

$$\mathsf{total}(T) \leftarrow T = |\{I : \mathsf{instruction}(I)\}|. \tag{39}$$

$$\begin{aligned}\mathsf{indep}(I,T-D) \leftarrow\ & \mathsf{instruction}(I),\ \mathsf{total}(T),\\ & D = |\{I_1 : \mathsf{reach}(I_1,I)\}| + |\{I_2 : \mathsf{reach}(I,I_2)\}|.\end{aligned} \tag{40}$$

$$\mathsf{upper}(I_1,D) \leftarrow \mathsf{indep}(I_1,D). \tag{41}$$

$$\mathsf{upper}(I_2,U+G+D-1) \leftarrow \mathsf{upper}(I_1,U),\ \mathsf{gap}(I_1,I_2,G),\ \mathsf{indep}(I_2,D). \tag{42}$$

$$\begin{aligned}\mathsf{range}(I,L,U) \leftarrow\ & \mathsf{instruction}(I),\ L = \max\{M : \mathsf{lower}(I,M)\},\\ & U = \max\{N : \mathsf{upper}(I,N)\}.\end{aligned} \tag{43}$$

Omitting the details, let us highlight that the domain rules (35)–(43) feature again integer arithmetic but also aggregation operations provided by GRINGO, that is, term counting in (39) and (40) as well as maximum term calculation in (43). Moreover, the rules (36), (38), and (42) are recursive, in the sense that the head atom of one instance may in turn be used as a positive body element in another rule instance. Given that the arithmetic expressions in (36) and (42) can produce new terms, there are no a priori restrictions on ground terms. In fact, the finiteness of instantiations relies on acyclic

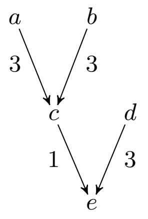

Fig. 2. A directed acyclic graph representing instruction dependencies and latencies

graphs given as inputs, while ASP grounders supporting arithmetic (as well as uninterpreted functions) are fully capable of dealing with semi-decidable problems (cf. [11]).

Example 5. The directed acyclic graph in Figure 2 is represented by facts as follows:

$\mathsf{instruction}(a;b;c;d;e)$, $\mathsf{gap}(a,c,3)$, $\mathsf{gap}(b,c,3)$, $\mathsf{gap}(c,e,1)$, and $\mathsf{gap}(d,e,3)$.

The domain rule (35) provides the trivial lower bound 1 for each of the instructions a, b, c, d, and e. In view of the dependencies of c on a and b along with the latencies 3, (36) further yields $\mathsf{lower}(c,4)$, and the dependencies of e on c and d allow for deriving $\mathsf{lower}(e,2)$, $\mathsf{lower}(e,4)$, and $\mathsf{lower}(e,5)$. The effective lower bounds for executing instructions, in (43) extracted from maximum values M in atoms of the form $\mathsf{lower}(I,M)$, are 1 for the instructions a, b, and d, 4 for c, and 5 for e. For each instruction I, the domain rules (37)–(40) determine the number of independent instructions that neither reach I nor are reached from I, also counting I itself in order to guarantee positive values. In fact, $\mathsf{indep}(e,1)$ indicates that e has dependencies to all other instructions, $\mathsf{indep}(c,2)$ reflects that c is independent of d, $\mathsf{indep}(a,3)$ and $\mathsf{indep}(b,3)$ further take the mutual independency of a and b into account, and $\mathsf{indep}(d,4)$ expresses that e is the only instruction depending on d. The numbers of independent instructions, whose execution can be freely interleaved, are taken as trivial upper bounds in (41). Such upper bounds are further propagated by the domain rule (42), where latencies as well as independent instructions are considered to make sure that some slot remains for the execution of a dependent instruction. In this way, $\mathsf{upper}(c,7)$, $\mathsf{upper}(e,3)$, $\mathsf{upper}(e,7)$, and $\mathsf{upper}(e,8)$ are derived as additional atoms of the form $\mathsf{upper}(I,N)$. The maximum value N for an instruction I is in (43) taken as the effective upper bound for the execution of I. As a consequence, the interval $[1,3]$ is determined as range for executing a and b, $[4,7]$ for c, $[1,4]$ for d, and $[5,8]$ for e. ■

Given the execution ranges determined by means of the domain rules (35)–(43), the following encoding part expresses optimal instruction scheduling in terms of MaxSAT:

$$\mathsf{range}(I,L\ldots U) \leftarrow \mathsf{range}(I,L,U). \tag{44}$$

$$\begin{aligned}\mathsf{opt}(L+1\ldots U) \leftarrow {} & L=\max\{M:\mathsf{range}(I,M,N)\},\\ & U=\max\{N:\mathsf{range}(I,M,N)\}.\end{aligned} \tag{45}$$

$$\begin{aligned}\mathsf{later}(I_1,I_2,G,L_2+1\ldots U_1+G) \leftarrow {} & \mathsf{gap}(I_1,I_2,G),\ \mathsf{range}(I_1,L_1,U_1),\\ & \mathsf{range}(I_2,L_2,U_2).\end{aligned} \tag{46}$$

$$\begin{aligned}\mathsf{block}(I_1,I_2,S+1) \leftarrow {} & \mathsf{range}(I_1;I_2,S),\ \mathsf{range}(I_2,S+1),\ I_1\neq I_2,\\ & \sim\!\mathsf{reach}(I_1,I_2),\ \sim\!\mathsf{reach}(I_2,I_1).\end{aligned} \tag{47}$$

$$\begin{aligned}&\mathsf{value}(I,S) \vee \textstyle\bigvee \neg\mathsf{delay}(I,S) : \mathsf{range}(I,S-1) \vee \\ &\qquad \textstyle\bigvee \mathsf{delay}(I,S+1) : \mathsf{range}(I,S+1) \leftarrow \mathsf{range}(I,S).\end{aligned} \tag{48}$$

$$\neg\mathsf{value}(I,S) \vee \mathsf{delay}(I,S) \leftarrow \mathsf{range}(I,S;S-1). \tag{49}$$

$$\neg\mathsf{value}(I,S) \vee \neg\mathsf{delay}(I,S+1) \leftarrow \mathsf{range}(I,S;S+1). \tag{50}$$

$$\mathsf{delay}(I_2,S) \vee \neg\mathsf{delay}(I_1,S-G) \leftarrow \mathsf{later}(I_1,I_2,G,S). \tag{51}$$

$$\begin{aligned}&\mathsf{delay}(I_2,S) \vee \neg\mathsf{value}(I_1,S-1) \vee \\ &\qquad \textstyle\bigvee \neg\mathsf{delay}(I_2,S-1) : \mathsf{range}(I_2,S-2) \leftarrow \mathsf{block}(I_1,I_2,S).\end{aligned} \tag{52}$$

$$\begin{aligned}&\neg\mathsf{delay}(I_2,S) \vee \textstyle\bigvee \mathsf{delay}(I_1,S-G) : \mathsf{later}(I_1,I_2,G,S) \vee \\ &\qquad \textstyle\bigvee \mathsf{value}(I_1,S-1) : \mathsf{block}(I_1,I_2,S) \leftarrow \mathsf{range}(I_2,S;S-1).\end{aligned} \tag{53}$$

$$\begin{aligned}&\neg\mathsf{delay}(I_2,S) \vee \textstyle\bigvee \mathsf{delay}(I_1,S-G) : \mathsf{later}(I_1,I_2,G,S) \vee \\ &\qquad \mathsf{delay}(I_2,S-1) \leftarrow \mathsf{range}(I_2,S;S-2).\end{aligned} \tag{54}$$

$$\mathsf{penalty}(S) \vee \neg\mathsf{delay}(I,S) \leftarrow \mathsf{range}(I,S),\ \mathsf{opt}(S). \tag{55}$$

$$\neg\mathsf{penalty}(S) \vee \bigvee \mathsf{delay}(I,S) : \mathsf{range}(I,S) \leftarrow \mathsf{opt}(S). \tag{56}$$

$$\mathrm{minimize}\ |\{S : \mathsf{penalty}(S)\}|. \tag{57}$$

The additional domain rules (44)–(47) define auxiliary predicates utilized in the clause rules below. In particular, (44) and (45) provide potential execution slots for instructions as well as slots lying beyond the lower bound of any instruction. Only the latter are subject to minimization because the execution of all instructions cannot be finished before. More importantly, (46) and (47) indicate instructions I_1 whose execution can be responsible for delaying I_2, in the sense that I_2 may have to be scheduled later than its lower bound. On the one hand, such instructions I_1 include those that I_2 depends on, so that the execution slots of I_1 and I_2 must be separated by at least the latency G included in atoms of the form $\mathsf{later}(I_1,I_2,G,S)$, where S refers to a next such slot for I_2. On the other hand, the execution of an independent task I_1 at a slot S may delay I_2 to the next slot, as expressed by atoms of the form $\mathsf{block}(I_1,I_2,S+1)$. For an instruction I_2 with an associated interval $[L,U]$ of potential execution slots, the clause rules (51)–(54) investigate possible delays to establish that instances of $\mathsf{delay}(I_2,S)$ are true iff I_2 cannot be executed before $S \in (L,U]$, i.e., other tasks rule all slots in $[L,S)$ out. Given this, actual execution slots are extracted by means of the clause rules (48)–(50) and indicated by instances of $\mathsf{value}(I,S)$ in a domain stable model. Finally, schedules are optimized by penalizing atoms of the form $\mathsf{penalty}(S)$, which in view of (55) and (56) signal that some instruction is executed at S or a later slot. Note that the $\mathrm{minimize}$ statement in (57) weighs instances of $\mathsf{penalty}(S)$, whose complements correspond to soft unit clauses, equally by 1, while arbitrary integer weights can be used in general.

Example 6. For the instructions displayed in Figure 2, there are two optimal schedules such that a and b are executed at slots 1 and 2 in either order, d is executed at slot 3, c at slot 5, and eventually e at slot 6. The corresponding domain stable models thus agree on the atoms $\mathsf{value}(d,3)$, $\mathsf{value}(c,5)$, and $\mathsf{value}(e,6)$. In view of the execution ranges given in Example 5, the atoms $\mathsf{delay}(d,2)$, $\mathsf{delay}(d,3)$, $\mathsf{delay}(c,5)$, $\mathsf{delay}(e,6)$, and $\mathsf{penalty}(6)$ hold in addition, among which the latter is subject to minimization. Note that the execution of d is delayed by the instructions a and b occupying the first two slots, while dependencies of c on a and b or of e on c and d, respectively, are responsible for delaying the execution of c and e beyond their (imperfect) lower bounds. The two optimal domain stable models differ as follows: one includes $\mathsf{value}(a, 1)$, $\mathsf{value}(b, 2)$, and $\mathsf{delay}(b, 2)$, while the other contains $\mathsf{value}(b, 1)$, $\mathsf{value}(a, 2)$, and $\mathsf{delay}(a, 2)$. In fact, the topology in Figure 2 shows that a and b are symmetric and can be freely permuted. In order to cut redundant solutions obtained by swapping the slots of a and b, the presented encoding could be augmented with domain rules identifying such symmetries along with clause rules implementing lexicographic symmetry breaking (cf. [22]). ■

In summary, the above use cases illustrate how clause programs can be harnessed to model non-trivial combinatorial as well as application problems in a uniform fashion. The presented encodings exploit built-in integer arithmetic, aggregation operations, and the closed world assumption of ASP in concise first-order specifications of schematic clauses. In particular, fixpoint constructions enable deriving (implicit) domains of variables from instance data, thus reducing the need for involved procedural computations.

4 Implementation

To implement the grounding of clause programs, we utilize the state-of-the-art ASP grounder GRINGO [11]. This is feasible because GRINGO (from version 2 on) supports classical literals and disjunctive rule heads as in (3). By hiding and hence omitting the domain part of a clause program P, the ground program $\mathrm{Gnd}(P)$ is essentially a set of ground disjunctions $a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l$. For GRINGO, the semantics of $\mathrm{Gnd}(P)$ is based on consistent sets of classical literals, also known as *answer sets* [13], which can be viewed as minimal *hitting sets* for the disjunctions in $\mathrm{Gnd}(P)$. For the purposes of this work, however, we re-establish the semantic connection between an atom a and its classical negation $\neg a$ by transforming disjunctions into a set $\mathrm{Cl}(\mathrm{Gnd}(P))$ of clauses in DIMACS format, serving as input of SAT solvers. This step is implemented by a tool called SATGRND (v. 1.21), which passes the symbolic names of atoms on as comments in its output. The transformation preserves classical models and satisfiability, so that satisfying assignments of $\mathrm{Cl}(\mathrm{Gnd}(P))$ correspond to domain stable models of P.[4]

Beyond this basic transformation, SATGRND can be used to extract graph information from symbolic atom names, as exploited in the SAT modulo graphs approach [9, 10]. Both in plain SAT and SAT modulo graphs, models may be subject to optimization, expressible by $\mathrm{minimize}$ statements in the input language of GRINGO, in which case SATGRND generates (weighted partial) MaxSAT problems in DIMACS format. Moreover, SATGRND permits the computation of classical models for (disjunctive) logic programs in general and is provided along with sample encodings for the use cases in the previous section.[5]

[4] Classical models can also be expressed within ASP, e.g., in terms of choice rules and integrity constraints [25].

[5] `http://research.ics.aalto.fi/software/asp/satgrnd/`

5 Discussion of Related Work and Conclusion

In this paper, we promote declarative domain specifications in contrast with procedural ones that are typical when solvers are interfaced with a programming library (see, e.g., the Python interface of Microsoft's Z3). Naturally, other declarative approaches exist. In the context of pseudo-Boolean solvers, the system PSGRND [8] can be used to ground clauses and their extensions. The domain information, however, is given by type declarations for predicates, and it is not possible to define types in terms of others. The first-order approaches of [1, 5, 17, 29] also aim to restrict variable domains recursively over the structure of first-order formulas, where the CWA is limited to predicates that are defined (inductively) in terms of those allowed to vary. The same can be stated about the methods proposed for *effectively* propositional logic [21, 24], although domain constraints are imposed. The IDP3 system [18] exploits PROLOG-style rules to express domain information, but it processes them through query answering rather than bottom-up evaluation. In [14], the grounding problem is addressed in the context of *planning domain definition language* (PDDL) descriptions over finite domains. While this approach explores a Datalog representation and grounding techniques similar to ASP, it is specialized to planning tasks. The interface provided by GRINGO is more general, in particular, given that domains need not be finitely bounded a priori. Last but not least, note that traditional constraint models can also be translated into CNF (see, e.g., [15]), yet expressing recursive domain specifications remains difficult.

In conclusion, we suggest to utilize ASP grounders for instantiating first-order clauses involving term variables. This provides us with means to control the resulting propositional clauses in a declarative way and to avoid the implicit introduction of new Boolean variables, which is practically necessary otherwise, e.g., when translating logic programs into SAT [16]. The combination of GRINGO and SATGRND forms a general-purpose grounding tool not confined to a particular application domain. Due to the highly versatile and eventually Turing-complete input language of GRINGO, complex domain specifications can be written to support fine-grained instantiation of term variables. The uniform rule-based syntax makes specifications highly elaboration tolerant and independent of particular instance data. We expect that the grounding methodology introduced in this paper can be highly useful for SAT application developers in order to devise compact encodings directly at clause level.

References

1. A. Aavani, X. Wu, E. Ternovska, and D. Mitchell. Grounding formulas with complex terms. In *Proceedings of Canadian AI'11*, pages 13–25. Springer, 2011.
2. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: A theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
3. G. Audemard, G. Katsirelos, and L. Simon. A restriction of extended resolution for clause learning SAT solvers. In *Proceedings of AAAI'10*, pages 15–20. AAAI Press, 2010.
4. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
5. H. Blockeel, B. Bogaerts, M. Bruynooghe, B. De Cat, S. De Pooter, M. Denecker, A. Labarre, J. Ramon, and S. Verwer. Modeling machine learning and data mining problems with FO(.). In *Technical Communications of ICLP'12*, pages 14–25. Schloss Dagstuhl, 2012.

6. G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
7. J. Corander, T. Janhunen, J. Rintanen, H. Nyman, and J. Pensar. Learning chordal Markov networks by constraint satisfaction. In *Proceedings of NIPS'13*, pages 1349–1357. NIPS Foundation, 2013.
8. D. East, M. Iakhiaev, A. Mikitiuk, and M. Truszczyński. Tools for modeling and solving search problems. *AI Communications*, 19(4):301–312, 2006.
9. M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *Proceedings of ECAI'14*, pages 351–356. IOS Press, 2014.
10. M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: Acyclicity. In *Proceedings of JELIA'14*, pages 137–151. Springer, 2014.
11. M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. In *Proceedings of LPNMR'11*, pages 345–351. Springer, 2011.
12. M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele. On the input language of ASP grounder gringo. In *Proceedings of LPNMR'09*, pages 502–508. Springer, 2011.
13. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
14. M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.
15. J. Huang. Universal Booleanization of constraint models. In *Proceedings of CP'08*, pages 144–158. Springer, 2008.
16. 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.
17. J. Jansen, I. Dasseville, J. Devriendt, and G. Janssens. Experimental evaluation of a state-of-the-art grounder. In *Proceedings of PPDP'14*, pages 249–258. ACM Press, 2014.
18. J. Jansen, A. Jorissen, and G. Janssens. Compiling input* FO(.) inductive definitions into tabled Prolog rules for IDP3. *Theory and Practice of Logic Programming*, 13(4-5):691–704, 2013.
19. 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.
20. J. McCarthy. Elaboration tolerance, 2003. `http://www-formal.stanford.edu/jmc/elaboration.ps`.
21. J. Navarro-Pérez and A. Voronkov. Proof systems for effectively propositional logic. In *Proceedings of IJCAR'08*, pages 426–440. Springer, 2008.
22. K. Sakallah. Symmetry and satisfiability. In Biere et al. [4], chapter 10, pages 289–338.
23. J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51:64–86, 1995.
24. S. Schulz. A comparison of different techniques for grounding near-propositional CNF formulae. In *Proceedings of FLAIRS'02*, pages 72–76. AAAI Press, 2002.
25. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
26. J. Ullman. *Principles of Database and Knowledge-Base Systems*. CS Press, 1988.
27. P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of CP'01*, pages 625–639. Springer, 2001.
28. 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.
29. J. Wittocx, M. Mariën, and M. Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.