Writing Declarative Specifications for Clauses

Martin Gebser[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 [1] 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:

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

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, Y, and Z in (1) by the constants a, b, and c. While $3^3 = 27$ different substitutions

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

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., [2, 3]) as well as AI planning (cf. [4]). 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 [5]. Also, by carefully analyzing variable ranges, it is possible to reduce the number of clauses or formulas generated [3, 6].

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) [7] 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 [8–10]. 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 [11, 12], 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 [13]. For another, the level of abstraction provided by first-order rules makes specifications highly *elaboration tolerant* [14].

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 [15] with SAT or MaxSAT solvers, and an experimental evaluation where haplotype inference is remodeled using clause programs. 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, including the determination of 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. [16]) that can be calculated by an ASP grounder. We therefore require domain rules (2) of P to be *stratified* (cf. [17]), 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:

$$\mathsf{node}(X) \leftarrow \mathsf{edge}(X, Y). \tag{4}$$
$$\mathsf{node}(Y) \leftarrow \mathsf{edge}(X, Y). \tag{5}$$
$$\mathsf{b}(X) \vee \mathsf{g}(X) \vee \mathsf{r}(X) \leftarrow \mathsf{node}(X). \tag{6}$$
$$\neg\mathsf{b}(X) \vee \neg\mathsf{b}(Y) \leftarrow \mathsf{edge}(X, Y). \tag{7}$$
$$\neg\mathsf{g}(X) \vee \neg\mathsf{g}(Y) \leftarrow \mathsf{edge}(X, Y). \tag{8}$$
$$\neg\mathsf{r}(X) \vee \neg\mathsf{r}(Y) \leftarrow \mathsf{edge}(X, Y). \tag{9}$$

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:

$$\begin{array}{lll}
\mathsf{b}(a) \vee \mathsf{g}(a) \vee \mathsf{r}(a), & \mathsf{b}(b) \vee \mathsf{g}(b) \vee \mathsf{r}(b), & \mathsf{b}(c) \vee \mathsf{g}(c) \vee \mathsf{r}(c), \\
\neg\mathsf{b}(a) \vee \neg\mathsf{b}(b), & \neg\mathsf{b}(b) \vee \neg\mathsf{b}(c), & \neg\mathsf{b}(c) \vee \neg\mathsf{b}(a), \\
\neg\mathsf{g}(a) \vee \neg\mathsf{g}(b), & \neg\mathsf{g}(b) \vee \neg\mathsf{g}(c), & \neg\mathsf{g}(c) \vee \neg\mathsf{g}(a), \\
\neg\mathsf{r}(a) \vee \neg\mathsf{r}(b), & \neg\mathsf{r}(b) \vee \neg\mathsf{r}(c), & \neg\mathsf{r}(c) \vee \neg\mathsf{r}(a).
\end{array}$$

These clauses can be satisfied, e.g., by letting $I_{\mathrm{v}} = \{\mathsf{b}(a), \mathsf{g}(b), \mathsf{r}(c)\}$, 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:

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

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 parameterized by a *conditional literal* $\mathsf{hc}(X, C)$, 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:

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

The clauses resemble those in Example 1, yet using the generic predicate $\mathsf{hc}(X, C)$ for node X *having color* C, rather than dedicated predicates $\mathsf{b}/1$, $\mathsf{g}/1$, and $\mathsf{r}/1$ for blue, green, and red, respectively. Accordingly, an assignment of distinct colors to the three nodes at hand is expressed by a projection like $I_{\mathrm{v}} = \{\mathsf{hc}(a,1), \mathsf{hc}(b,2), \mathsf{hc}(c,3)\}$. ■

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:

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

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

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

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

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

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

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

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

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, obtained by instantiating X, R, and C with 1, 0, and -1 in (15): $\mathsf{target}(1,2,0,-1)$, $\mathsf{target}(1,3,0,-1)$, and $\mathsf{target}(1,4,0,-1)$. These atoms induce nine instances of (16)–(18), whose conjunction is equivalent to formulas

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

Clauses from (19) exclude horizontal attacks: $\neg\mathsf{queen}(1,1) \vee \neg\mathsf{attack}(1,1,0,-1)$, $\neg\mathsf{queen}(1,2) \vee \neg\mathsf{attack}(1,2,0,-1)$, $\neg\mathsf{queen}(1,3) \vee \neg\mathsf{attack}(1,3,0,-1)$. ■

Propositional Logic. Next we illustrate how the satisfiability problem of full propositional logic can be captured in a declarative way. To this end, a meta-representation of a propositional theory is needed, using function symbols (supported by ASP grounders like GRINGO) to represent Boolean connectives. For brevity, we only consider disjunction and negation here, but note that our approach easily extends to other connectives as well. We use constants for atoms, the functions $\mathsf{or}/2$ and $\mathsf{neg}/1$ for disjunction and negation, and the predicate $\mathsf{sentence}/1$ to declare sentences in a theory. Given this, we axiomatize the satisfaction of the theory as follows:

$$\mathsf{subformula}(F) \leftarrow \mathsf{sentence}(F). \tag{22}$$
$$\mathsf{subformula}(F) \leftarrow \mathsf{subformula}(\mathsf{neg}(F)). \tag{23}$$
$$\mathsf{subformula}(F;G) \leftarrow \mathsf{subformula}(\mathsf{or}(F,G)). \tag{24}$$
$$\mathsf{sat}(\mathsf{neg}(F)) \vee \mathsf{sat}(F) \leftarrow \mathsf{subformula}(\mathsf{neg}(F)). \tag{25}$$
$$\mathsf{sat}(\mathsf{or}(F,G)) \vee \neg\mathsf{sat}(F) \leftarrow \mathsf{subformula}(\mathsf{or}(F,G)). \tag{26}$$
$$\mathsf{sat}(\mathsf{or}(F,G)) \vee \neg\mathsf{sat}(G) \leftarrow \mathsf{subformula}(\mathsf{or}(F,G)). \tag{27}$$
$$\neg\mathsf{sat}(\mathsf{neg}(F)) \vee \neg\mathsf{sat}(F) \leftarrow \mathsf{subformula}(\mathsf{neg}(F)). \tag{28}$$
$$\neg\mathsf{sat}(\mathsf{or}(F,G)) \vee \mathsf{sat}(F) \vee \mathsf{sat}(G) \leftarrow \mathsf{subformula}(\mathsf{or}(F,G)). \tag{29}$$
$$\mathsf{sat}(F) \leftarrow \mathsf{sentence}(F). \tag{30}$$

Here, the domain rules (22)–(24) derive the subformulas of the given theory, and the clause rules (25)–(29) evaluate these subformulas according to the interpretation of atoms and the semantics of propositional connectives. Finally, the clause rule (30) asserts that all sentences in the given theory must be satisfied. For instance, the sentence $\neg p \vee \neg q$ is represented by the following clauses:

$$\begin{aligned}&\mathsf{sat}(\mathsf{neg}(p)) \vee \mathsf{sat}(p), && \neg\mathsf{sat}(\mathsf{neg}(p)) \vee \neg\mathsf{sat}(p),\\ &\mathsf{sat}(\mathsf{neg}(q)) \vee \mathsf{sat}(q), && \neg\mathsf{sat}(\mathsf{neg}(q)) \vee \neg\mathsf{sat}(q),\\ &\mathsf{sat}(\mathsf{or}(\mathsf{neg}(p),\mathsf{neg}(q))) \vee \neg\mathsf{sat}(\mathsf{neg}(p)), && \mathsf{sat}(\mathsf{or}(\mathsf{neg}(p),\mathsf{neg}(q))) \vee \neg\mathsf{sat}(\mathsf{neg}(q)),\\ &\neg\mathsf{sat}(\mathsf{or}(\mathsf{neg}(p),\mathsf{neg}(q))) \vee \mathsf{sat}(\mathsf{neg}(p)) \vee \mathsf{sat}(\mathsf{neg}(q)), && \mathsf{sat}(\mathsf{or}(\mathsf{neg}(p),\mathsf{neg}(q))).\end{aligned}$$

In summary, the above use cases illustrate how clause programs can uniformly model non-trivial combinatorial as well as application problems. The presented encodings exploit built-in integer arithmetic, aggregation operations, function symbols, 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 [15]. 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 $a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l$. From the perspective of GRINGO, the semantics of $\mathrm{Gnd}(P)$ is based on consistent sets of classical literals, also known as *answer sets* [18], 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, or optionally into pseudo-Boolean constraints in OPB format. This step is implemented by a tool called SATGRND (v. 1.24), 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] Additionally, the file formats for satisfiability modulo theories (SMT) and mixed integer programming (MIP) are supported.

Beyond this basic transformation, SATGRND can be used to extract graph information from symbolic atom names, as exploited in the SAT modulo graphs approach [20, 21]. Both in plain SAT and SAT modulo graphs, models may be subject to optimization, expressible by optimization statements in the input language of GRINGO, in which case SATGRND generates (weighted partial) MaxSAT problems in DIMACS format, or again optionally OPB format, which supports objective functions. 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]

In order to compare SATGRND's declarative approach with a procedural implementation producing a solver's input, we investigated the optimization problem of haplotype inference [22, 23]. The task is to compute a cardinality-minimal set H of haplotypes that explain a set G of genotypes, as given on the right.

G	1 2 3	H	1 2 3
g_1	1 1 0	h_1	1 1 0
g_2	1 2 0	h_2	1 0 0
g_3	2 1 2	h_3	0 1 1

Genotypes g_i are determined by strings of some fixed length l, consisting of the symbols '0', '1', and '2'. Haplotypes h_j are also strings of length l, yet admitting '0' and '1' only. Two (not necessarily distinct) haplotypes h_{j_1}, h_{j_2} explain a genotype g_i if, for each string position $1 \leq k \leq l$, we have that $g_i^k = 2$ implies $h_{j_1}^k \neq h_{j_2}^k$, while

[4] Classical models can be encoded in ASP, e.g., using choice rules and integrity constraints [19].

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

$h^k_{j_1} = h^k_{j_2} = g^k_i$ otherwise. In the above table, g_1 is explained by h_1, h_1, g_2 by h_1, h_2, and g_3 by h_1, h_3. Moreover, one can check that at least three haplotypes are needed to explain g_1, g_2, and g_3, so that $H = \{h_1, h_2, h_3\}$ is an optimal solution.

The tool RPOLY[6] provides a reference implementation of haplotype inference, using a generator to convert instance data into a problem representation in OPB format, which is then passed on to a pseudo-Boolean solver like MINISAT+ [24]. The pseudo-Boolean constraints produced by the generator, described in [22, 23], exploit domain knowledge to achieve a compact representation: duplicated genotypes as well as isomorphic string positions in the input are conflated, and static symmetry breaking is applied to disambiguate pairs of haplotypes used to explain genotypes. For instance, the string positions 1 and 3 are isomorphic for the above genotypes $G = \{g_1, g_2, g_3\}$, given that the other column is reproduced by swapping '0' and '1' in one of the columns. In such a case, either of the isomorphic positions can be reproduced from the other, and only one representative needs to be computed by a solver. Moreover, for each genotype g_i, an arbitrary occurrence of '2' at remaining positions can be picked to statically fix one of the haplotypes explaining g_i to '0', and the other to '1' at this position. In fact, the combination of both techniques directly leads to the above haplotypes $H = \{h_1, h_2, h_3\}$, simply by applying static symmetry breaking to the occurrences of '2' at the second position of g_2 (which gives $h_1 = g_1$ and h_2 to explain g_2) and the first position of g_3, and then using the opposite symbol among '0' and '1' for aligning the '2' at the isomorphic third position of g_3 (thus reproducing h_1 along with its counterpart h_3 to explain g_3).

In general, not all occurrences of '2' can be fixed a priori, and the problem representation generated by RPOLY includes variables t^k_i to indicate whether a remaining occurrence of '2' at the k-th position of g_i is split up similar or opposite to the statically fixed '2' in the two haplotypes explaining g_i. This determines the used haplotypes, and further variables $x^{e_1,e_2}_{i_1,i_2}$ for genotypes g_{i_1}, g_{i_2} such that $i_1 < i_2$ and $e_1, e_2 \in \{0, 1\}$ are implied when any pair of some of the (at most) two haplotypes to explain g_{i_1} or g_{i_2}, respectively, is different. Finally, variables $u^{e_2}_{i_2}$, indicating haplotypes used first in explaining g_{i_2}, i.e., $x^{e_1,e_2}_{i_1,i_2}$ holds for all $i_1 < i_2$ and $e_1 \in \{0, 1\}$, are to be minimized.

We took the ideas implemented by RPOLY as basis for a corresponding encoding of haplotype inference by a clause program,[5] as shown in Figure 1. A problem instance specifies genotypes like the above by facts

$$\mathsf{gene}(1), \mathsf{symb}(1,1,1), \mathsf{symb}(1,2,1), \mathsf{symb}(1,3,0), \mathsf{position}(1),$$
$$\mathsf{gene}(2), \mathsf{symb}(2,1,1), \mathsf{symb}(2,2,2), \mathsf{symb}(2,3,0), \mathsf{position}(2),$$
$$\mathsf{gene}(3), \mathsf{symb}(3,1,2), \mathsf{symb}(3,2,1), \mathsf{symb}(3,3,2), \text{ and } \mathsf{position}(3).$$

Given such facts, the domain rules (31) and (32) take care of filtering duplicates, where the conditional literal $\mathsf{diff}(G_1, G_2)$ in (32) checks whether any genotype G_1 whose identifier is smaller than G_2 differs from G_2 at some string position. If so, an instance of $\mathsf{keep}(G_2)$ indicates that genotype G_2 is not a duplicate and to be explained by haplotypes. Similar to (31), the domain rules (33)–(35) derive instances of $\mathsf{dist}(K_1, K_2)$, expressing that a string position K_2 is not isomorphic to the smaller position K_1. To this end, (33) and (34) apply if '0' and '1' both occur at K_1 and K_2 in some genotype as well as either of them twice in another genotype. Moreover, (35) checks for an occurrence of '2' at either K_1 or K_2 to signal a difference between the two positions. The

[6] http://sat.inesc-id.pt/software/rpoly/

$$\mathsf{diff}(G_1, G_2) \leftarrow \mathsf{symb}(G_1, K, X),\ \mathsf{gene}(G_2),\ G_1 < G_2,\ {\sim}\mathsf{symb}(G_2, K, X). \tag{31}$$

$$\mathsf{keep}(G_2) \leftarrow \mathsf{gene}(G_2),\ \mathsf{diff}(G_1, G_2) : (\mathsf{gene}(G_1),\ G_1 < G_2). \tag{32}$$

$$\mathsf{flip}(K_1, K_2) \leftarrow \mathsf{symb}(G, K_1, X),\ \mathsf{symb}(G, K_2, 1 - X),\ K_1 < K_2,\ X < 2. \tag{33}$$

$$\mathsf{dist}(K_1, K_2) \leftarrow \mathsf{symb}(G, K_1, X),\ \mathsf{symb}(G, K_2, X),\ \mathsf{flip}(K_1, K_2),\ X < 2. \tag{34}$$

$$\mathsf{dist}(K_1, K_2) \leftarrow \mathsf{symb}(G, K_1, X_1),\ \mathsf{symb}(G, K_2, X_2),\ K_1 < K_2,\ X_1/2 + X_2/2 = 1. \tag{35}$$

$$\mathsf{pick}(K_2) \leftarrow \mathsf{position}(K_2),\ \mathsf{dist}(K_1, K_2) : (\mathsf{position}(K_1),\ K_1 < K_2). \tag{36}$$

$$\mathsf{twos}(K, S) \leftarrow \mathsf{pick}(K),\ S = |\{G : (\mathsf{keep}(G),\ \mathsf{symb}(G, K, 2))\}|. \tag{37}$$

$$\mathsf{vary}(G, K) \leftarrow \mathsf{keep}(G),\ (S, K) = \min\{(T, L) : (\mathsf{twos}(L, T),\ \mathsf{symb}(G, L, 2))\}. \tag{38}$$

$$\neg\mathsf{same}(G_1, 0 \ldots 1, G_2, 0 \ldots 1) \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ X < 2,\ \mathsf{symb}(G_1, K, X),\ \mathsf{symb}(G_2, K, 1 - X). \tag{39}$$

$$\neg\mathsf{same}(G_1, 0 \ldots 1, G_2, E) \vee \bigvee \mathsf{vary}(G_2, K) : E = X \vee \bigvee \neg\mathsf{vary}(G_2, K) : E \neq X \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ X < 2,\ \mathsf{symb}(G_1, K, X),\ \mathsf{symb}(G_2, K, 2),\ E = 0 \ldots 1. \tag{40}$$

$$\neg\mathsf{same}(G_1, E, G_2, 0 \ldots 1) \vee \bigvee \mathsf{vary}(G_1, K) : E = X \vee \bigvee \neg\mathsf{vary}(G_1, K) : E \neq X \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ X < 2,\ \mathsf{symb}(G_1, K, 2),\ \mathsf{symb}(G_2, K, X),\ E = 0 \ldots 1. \tag{41}$$

$$\neg\mathsf{same}(G_1, E, G_2, E) \vee \neg\mathsf{vary}(G_1, K) \vee \mathsf{vary}(G_2, K) \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ \mathsf{symb}(G_1, K, 2),\ \mathsf{symb}(G_2, K, 2),\ E = 0 \ldots 1. \tag{42}$$

$$\neg\mathsf{same}(G_1, E, G_2, E) \vee \mathsf{vary}(G_1, K) \vee \neg\mathsf{vary}(G_2, K) \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ \mathsf{symb}(G_1, K, 2),\ \mathsf{symb}(G_2, K, 2),\ E = 0 \ldots 1. \tag{43}$$

$$\neg\mathsf{same}(G_1, E, G_2, 1 - E) \vee \neg\mathsf{vary}(G_1, K) \vee \neg\mathsf{vary}(G_2, K) \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ G_1 < G_2,\ \mathsf{symb}(G_1, K, 2),\ \mathsf{symb}(G_2, K, 2),\ E = 0 \ldots 1. \tag{44}$$

$$\neg\mathsf{same}(G_1, E, G_2, 1 - E) \vee \bigvee \mathsf{vary}(G_1, K) : G_1 < G_2 \vee \bigvee \mathsf{vary}(G_2, K) : G_1 < G_2 \leftarrow \mathsf{keep}(G_1),\ \mathsf{keep}(G_2),\ \mathsf{pick}(K),\ (G_1, E) < (G_2, 1),\ \mathsf{symb}(G_1, K, 2),\ \mathsf{symb}(G_2, K, 2),\ E = 0 \ldots 1. \tag{45}$$

$$\mathsf{used}(G_2, E_2) \vee \bigvee \mathsf{same}(G_1, E_1, G_2, E_2) : (\mathsf{keep}(G_1),\ E_1 = 0 \ldots 1,\ (G_1, E_1) < (G_2, E_2)) \leftarrow \mathsf{keep}(G_2),\ E_2 = 0 \ldots 1. \tag{46}$$

$$\mathrm{minimize}\ |\{(G, E) : \mathsf{used}(G, E)\}|. \tag{47}$$

Fig. 1. A clause program encoding haplotype inference

domain rule (36) then derives $\mathsf{pick}(K_2)$ if the conditional literal $\mathsf{dist}(K_1, K_2)$ yields that no smaller position K_1 is isomorphic to K_2. For the given problem instance, we obtain $\mathsf{keep}(1)$, $\mathsf{keep}(2)$, and $\mathsf{keep}(3)$, as none of the three genotypes is a duplicate, along with $\mathsf{keep}(1)$ and $\mathsf{keep}(2)$, since the third string position is isomorphic to the first.

The final domain rule (37) determines the number of occurrences of '2' at non-isomorphic string positions in order to perform static symmetry breaking by means of the clause rule (38). In the latter rule, we use the $\min$ aggregation operation of GRINGO to pick an occurrence of '2' (if there is any) in a genotype g_i such that the overall number of '2's at the respective position k is minimal. This in turn maximizes the number of '0's and '1's at position k, following the rationale that fixing such a position to '0' or '1' directly discards plenty options of sharing one of the two haplotypes used to explain g_i. The atom $\mathsf{vary}(g_i, k)$ in a unit clause expressed by (38) stands for the variable t_i^k, whose truth signals that the first haplotype explaining g_i contains '0' at position k and the second '1', while '0' and '1' are swapped when t_i^k or $\mathsf{vary}(g_i, k)$, respectively, is false.

The purpose of the clause rules (39)–(45) is to assert a literal $\neg\mathsf{same}(g_{i_1}, e_1, g_{i_2}, e_2)$ for genotypes g_{i_1}, g_{i_2} such that $i_1 \leq i_2$ and $e_1, e_2 \in \{0, 1\}$ when two of the haplotypes explaining g_{i_1} and g_{i_2} are different, so that the literals correspond to the aforementioned variables $x_{i_1,i_2}^{e_1,e_2}$. In a nutshell, rule (39) applies to haplotypes whose genotypes differ on '0' and '1' at a position, (40) and (41) align a '0' or '1' in either g_{i_1} or g_{i_2} with the interpretation of $t_{i_2}^k$ or $t_{i_1}^k$, respectively, and (42)–(45) compare $t_{i_1}^k$ and $t_{i_2}^k$ in case both g_{i_1} and g_{i_2} contain '2' at position k. Note that (45) yields $\neg\mathsf{same}(g_i, 0, g_i, 1)$ for genotypes g_i with some occurrence of '2', and all clauses have in common that they imply $\neg\mathsf{same}(g_{i_1}, e_1, g_{i_2}, e_2)$ for pairs of haplotypes that differ at some position. For instance, the facts given above along with the domain rules (31)–(37) lead to the clauses:

$$
\begin{array}{lll}
\neg\mathsf{same}(1,0,2,0) \vee \neg\mathsf{vary}(2,2), & \neg\mathsf{same}(1,1,2,0) \vee \neg\mathsf{vary}(2,2), & \\
\neg\mathsf{same}(2,0,3,0) \vee \neg\mathsf{vary}(2,2), & \neg\mathsf{same}(2,0,3,1) \vee \neg\mathsf{vary}(2,2), & \\
\neg\mathsf{same}(1,0,2,1) \vee \mathsf{vary}(2,2), & \neg\mathsf{same}(1,1,2,1) \vee \mathsf{vary}(2,2), & \neg\mathsf{same}(2,0,2,1), \\
\neg\mathsf{same}(2,1,3,0) \vee \mathsf{vary}(2,2), & \neg\mathsf{same}(2,1,3,1) \vee \mathsf{vary}(2,2), & \\
\neg\mathsf{same}(1,0,3,0) \vee \neg\mathsf{vary}(3,1), & \neg\mathsf{same}(1,1,3,0) \vee \neg\mathsf{vary}(3,1), & \\
\neg\mathsf{same}(2,0,3,0) \vee \neg\mathsf{vary}(3,1), & \neg\mathsf{same}(2,1,3,0) \vee \neg\mathsf{vary}(3,1), & \\
\neg\mathsf{same}(1,0,3,1) \vee \mathsf{vary}(3,1), & \neg\mathsf{same}(1,1,3,1) \vee \mathsf{vary}(3,1), & \\
\neg\mathsf{same}(2,0,3,1) \vee \mathsf{vary}(3,1), & \neg\mathsf{same}(2,1,3,1) \vee \mathsf{vary}(3,1), & \neg\mathsf{same}(3,0,3,1).
\end{array}
$$

That is, the two haplotypes explaining g_2 or g_3, respectively, are inherently different from one another, and the unit clauses $\neg\mathsf{same}(2,0,2,1)$ and $\neg\mathsf{same}(3,0,3,1)$ represent the truth of $x_{2,2}^{0,1}$ and $x_{3,3}^{0,1}$. Moreover, the clauses including, e.g., $\neg\mathsf{same}(1,e,3,0)$ and $\neg\mathsf{same}(1,e,3,1)$ with $e \in \{0,1\}$ stand for $x_{1,3}^{e,0} \vee \neg t_3^1$ and $x_{1,3}^{e,1} \vee t_3^1$, thus reflecting differences between the haplotypes for g_1 that contain '1' at position 1 and either of the two haplotypes for g_3. Also note that none of the clauses refers to '2' at the third position of g_3 or t_3^3, respectively, since the third string position is isomorphic to the first.

The last clause rule (46) implies $\mathsf{used}(g_{i_2}, e_2)$, corresponding to variables $u_{i_2}^{e_2}$ for genotypes g_{i_2} and $e_2 \in \{0,1\}$, to indicate first uses of haplotypes. Such atoms are subject to minimization in view of the $\mathrm{minimize}$ statement in (47), instantiated as follows:

$$
\begin{array}{l}
\mathsf{used}(1,0), \\
\mathsf{used}(1,1) \vee \mathsf{same}(1,0,1,1), \\
\mathsf{used}(2,0) \vee \mathsf{same}(1,0,2,0) \vee \mathsf{same}(1,1,2,0), \\
\mathsf{used}(2,1) \vee \mathsf{same}(1,0,2,1) \vee \mathsf{same}(1,1,2,1) \vee \mathsf{same}(2,0,2,1), \\
\mathsf{used}(3,0) \vee \mathsf{same}(1,0,3,0) \vee \mathsf{same}(1,1,3,0) \vee \mathsf{same}(2,0,3,0) \vee \mathsf{same}(2,1,3,0),
\end{array}
$$

$$\mathsf{used}(3,1) \vee \mathsf{same}(1,0,3,1) \vee \mathsf{same}(1,1,3,1) \vee \mathsf{same}(2,0,3,1) \vee \mathsf{same}(2,1,3,1) \vee \mathsf{same}(3,0,3,1),$$
$$\text{minimize } |\{(1,0) : \mathsf{used}(1,0), (1,1) : \mathsf{used}(1,1), (2,0) : \mathsf{used}(2,0), (2,1) : \mathsf{used}(2,1), (3,0) : \mathsf{used}(3,0), (3,1) : \mathsf{used}(3,1)\}|.$$

One can check that all clauses are satisfied by a domain stable model I such that

$$I_{\mathrm{v}} = \{\mathsf{vary}(2,2), \mathsf{vary}(3,1), \mathsf{same}(1,0,1,1), \mathsf{same}(1,0,2,1), \mathsf{same}(1,0,3,1), \mathsf{used}(1,0), \mathsf{used}(2,0), \mathsf{used}(3,0)\}.$$

The number of distinct haplotypes, given by the predicate $\mathsf{used}/2$, is minimal, and in total there are 21 optimal models comprising the above haplotypes $H = \{h_1, h_2, h_3\}$.

In the pseudo-Boolean constraints of RPOLY as well as the encoding in Figure 1, the variables $x_{i_1,i_2}^{e_1,e_2}$ and $u_{i_2}^{e_2}$, signaling differences between haplotypes and those to count in the objective function, are handled by implications forcing them to true. In the following, we refer to this encoding approach by "Implication". We also implemented an encoding variant, indicated by "Equivalence", where such derived variables are matched to the conditions they express and cannot vary in case a condition does not apply. The stronger assertions of "Equivalence" thus reduce combinatorics to the prize of an increased number of clauses. Moreover, [22] mentions a condition under which at least three of four haplotypes explaining two genotypes g_{i_1}, g_{i_2} must be different, which can be expressed by clauses of the form $\neg x_{i_1,i_2}^{e_1,e_2} \vee \neg x_{i_1,i_2}^{e_3,e_4}$ for $e_1, e_2, e_3, e_4 \in \{0,1\}$ such that $e_1 \neq e_3$ or $e_2 \neq e_4$. Interestingly, respective pseudo-Boolean constraints are not generated by RPOLY, while the encoding variants denoted by "Implication-LB" and "Equivalence-LB" include such clauses. The four available encoding variants can be activated easily via command-line switches of GRINGO, and the encoding extensions for enabling flexibility amount to another ten selectively used schematic clause rules.[5]

To compare solving performance relative to input generated by RPOLY or by using clause programs and SATGRND, we ran the pseudo-Boolean solvers MINISAT+ (v. 1.0) and CLASP (v. 3.1.4), the latter performing unsatisfiability-based optimization (cf. [25]), sequentially on a Linux machine equipped with Intel Xeon E5-4650 2.70GHz processors. Instance data, out of which we selected the 63 instances such that some of the two solvers took more than ten seconds in a preliminary screening phase, was kindly provided by the authors of RPOLY. All solver runs were completed with an optimal solution, i.e., no effective time or memory limit was enforced. Table 1 provides averages over the 63 selected instances in terms of runtime and numbers of conflicts as well as constraints, the latter as reported by CLASP and MINISAT+, relative to input generated by RPOLY or SATGRND with the four encoding variants outline above. The conversion of instance data to a problem representation in OPB format, using RPOLY or SATGRND, was done offline and does thus not contribute to measured runtimes. Clearly, the procedural implementation by RPOLY is noticeably quicker than the grounding step of SATGRND, as the latter is geared for modeling flexibility rather than low-level performance.

Considering the average runtimes of both solvers, the best highlighted in boldface, CLASP is an order of magnitude faster on input provided by SATGRND, while the opposite effect applies to MINISAT+ on input generated by RPOLY. We attribute such inverse behavior to different selections of string positions for static symmetry breaking. In our encoding in Figure 1, we use a greedy approach aiming to reduce the resulting number

Table 1. Experiments with CLASP and MINISAT+ on haplotype inference benchmarks

	RPOLY	Implication	Implication-LB	Equivalence	Equivalence-LB	
Runtime	182.3	**3.3**	3.5	4.7	5.5	CLASP
Conflicts	466,933	47,262	52,420	57,789	67,178	CLASP
Constraints	36,299	28,318	28,454	49,054	49,192	CLASP
Runtime	**133.6**	1789.8	1402.7	2639.1	2467.4	MINISAT+
Conflicts	863,514	6,779,058	6,441,567	6,769,964	5,866,433	MINISAT+
Constraints	36,859	28,500	28,638	51,003	51,142	MINISAT+

of clauses: for each genotype, pick some occurrence of '2' that maximizes the number of '0's and '1's at this position. The strategy applied by RPOLY is, to our knowledge, not documented in the literature, and the apparent difference to ours can be observed on the numbers of constraints reported by CLASP and MINISAT+ in the first two columns of Table 1. In fact, there is a lot of room for different strategies, and declarative specifications by clause programs offer means for the rapid prototyping of alternative approaches.

Regarding the runtime differences between CLASP and MINISAT+, we want to stress that CLASP is a recent system, whereas MINISAT+ is not actively maintained. Hence, rather than further comparing the solvers to each other, it is more meaningful to concentrate on the effect of the encoding variants in the last three columns of Table 1, whose clauses differ from the pseudo-Boolean constraints of RPOLY. Here, we observe an expected rough doubling of size, witnessed by numbers of constraints, for the two "Equivalence" approaches. Since the size increase deteriorates runtimes and does not reduce conflicts significantly, the more relaxed approach taken by RPOLY and the "Implication" encoding is clearly the right choice. Moreover, the addition of clauses asserting necessary differences between haplotypes explaining different genotypes in "Implication-LB" (and "Equivalence-LB") modestly improves the runtime of MINISAT+ and its reported conflicts, yet not by a substantial amount. There are, however, no gains for CLASP, which again confirms the choice of RPOLY not to generate such constraints as appropriate. In summary, our practical case study demonstrates the utility of clause programs to implement a problem encoding, investigate the effect of alternative formulations, and identify parts that are critical for solving performance.

5 Discussion of Related Work and Conclusion

In this paper, we promote declarative domain specifications in contrast to 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 [26] 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 [2, 27, 28, 6] 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 [3, 5], although domain

constraints are imposed. The IDP3 system [29] exploits PROLOG-style rules to express domain information, but it processes them through query answering rather than bottom-up evaluation. In [4], 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 [30, 31] can also be translated into CNF (see, e.g., [10]), yet expressing recursive domain specifications remains difficult.

Since its initial conception [32], SATGRND has been used in several lines of work: firstly as a grounder to support high-level declarative specifications for the SAT-TO-SAT solver [33], and also as a tool to convert meta-representations of quantified Boolean formulas to layers of CNFs [34]. Secondly, SATGRND has been used in [35] to support declarative solver development for knowledge representation languages. Specifically, we took advantage of SATGRND to specify and implement a solver for combined logic programs [36].

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 [37]. The combination of GRINGO and SATGRND forms a general-purpose grounding tool not confined to a particular application domain. Due to the 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 beneficial for SAT application developers in order to rapidly devise and experiment with encodings directly at clause level.

Acknowledgments. This work was funded by the Academy of Finland (251170), DFG (SCHA 550/9), as well as DAAD and the Academy of Finland (57071677 and 279121). We are grateful to João Marques-Silva and Inês Lynce for kindly providing us with the benchmark instances used in Section 4.

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
2. Aavani, A., Wu, X., Tasharrofi, S., Ternovska, E., Mitchell, D.: Enfragmo: A system for modelling and solving search problems with logic. In: Proceedings of LPAR'12, Springer (2012) 15–22
3. Navarro-Pérez, J., Voronkov, A.: Proof systems for effectively propositional logic. In: Proceedings of IJCAR'08, Springer (2008) 426–440
4. Helmert, M.: Concise finite-domain representations for PDDL planning tasks. Artificial Intelligence **173**(5-6) (2009) 503–535
5. Schulz, S.: A comparison of different techniques for grounding near-propositional CNF formulae. In: Proceedings of FLAIRS'02, AAAI Press (2002) 72–76

6. Wittocx, J., Mariën, M., Denecker, M.: Grounding FO and FO(ID) with bounds. Journal of Artificial Intelligence Research **38** (2010) 223–269
7. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12) (2011) 92–103
8. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: A theoretical and empirical study. Constraints **16**(2) (2011) 195–221
9. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Proceedings of AAAI'10, AAAI Press (2010) 15–20
10. Huang, J.: Universal Booleanization of constraint models. In: Proceedings of CP'08, Springer (2008) 144–158
11. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder gringo. In: Proceedings of LPNMR'09, Springer (2011) 502–508
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
13. Schlipf, J.: The expressive powers of the logic programming semantics. Journal of Computer and System Sciences **51** (1995) 64–86
14. McCarthy, J.: Elaboration tolerance. `http://www-formal.stanford.edu/jmc/elaboration.ps` (2003)
15. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: Proceedings of LPNMR'11, Springer (2011) 345–351
16. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3) (1991) 620–650
17. Ullman, J.: Principles of Database and Knowledge-Base Systems. CS Press (1988)
18. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3-4) (1991) 365–386
19. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
20. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In: Proceedings of ECAI'14, IOS Press (2014) 351–356
21. Gebser, M., Janhunen, T., Rintanen, J.: SAT modulo graphs: Acyclicity. In: Proceedings of JELIA'14, Springer (2014) 137–151
22. Graça, A., Marques-Silva, J., Lynce, I., Oliveira, A.: Efficient haplotype inference with combined CP and OR techniques. In: Proceedings of CPAIOR'08, Springer (2008) 308–312
23. Graça, A., Marques-Silva, J., Lynce, I., Oliveira, A.: Efficient haplotype inference with pseudo-Boolean optimization. In: Proceedings of AB'07, Springer (2007) 125–139
24. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 1–26
25. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: Technical Communications of ICLP'12, LIPIcs (2012) 212–221
26. East, D., Iakhiaev, M., Mikitiuk, A., Truszczyński, M.: Tools for modeling and solving search problems. AI Communications **19**(4) (2006) 301–312
27. Blockeel, H., Bogaerts, B., Bruynooghe, M., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., Verwer, S.: Modeling machine learning and data mining problems with FO(.). In: Technical Communications of ICLP'12, LIPIcs (2012) 14–25
28. Jansen, J., Dasseville, I., Devriendt, J., Janssens, G.: Experimental evaluation of a state-of-the-art grounder. In: Proceedings of PPDP'14, ACM Press (2014) 249–258
29. Jansen, J., Jorissen, A., Janssens, G.: Compiling input* FO(.) inductive definitions into tabled Prolog rules for IDP3. Theory and Practice of Logic Programming **13**(4-5) (2013) 691–704
30. Cadoli, M., Schaerf, A.: Compiling problem specifications into SAT. Artificial Intelligence **162**(1-2) (2005) 89–120

31. Stuckey, P., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine **35**(2) (2014) 55–60
32. Gebser, M., Janhunen, T., Kaminski, R., Schaub, T., Tasharrofi, S.: Writing declarative specifications for clauses. In: Proceedings of GTTV'15. (2015)
33. Janhunen, T., Tasharrofi, S., Ternovska, E.: SAT-to-SAT: Declarative extension of SAT solvers with new propagators. In: Proceedings of AAAI'16, AAAI Press (2016) 978–984
34. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Solving QBF instances with nested SAT solvers. In: Proceedings of Beyond NP. (2016)
35. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Declarative solver development: Case studies. In: Proceedings of KR'16, AAAI Press (2016) 74–83
36. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Stable-unstable semantics: Beyond NP with normal logic programs. Theory and Practice of Logic Programming (2016) to appear
37. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86