Constraints (2017) 22:307–337
DOI 10.1007/s10601-016-9257-7

SURVEY

What is answer set programming to propositional satisfiability

Yuliya Lierler[1]

Published online: 16 December 2016

Abstract Propositional satisfiability (or satisfiability) and answer set programming are two closely related subareas of Artificial Intelligence that are used to model and solve difficult combinatorial search problems. Satisfiability solvers and answer set solvers are the software systems that find satisfying interpretations and answer sets for given propositional formulas and logic programs, respectively. These systems are closely related in their common design patterns. In satisfiability, a propositional formula is used to encode problem specifications in a way that its satisfying interpretations correspond to the solutions of the problem. To find solutions to a problem it is then sufficient to use a satisfiability solver on a corresponding formula. Niemelä, Marek, and Truszczyński coined answer set programming paradigm in 1999: in this paradigm a logic program encodes problem specifications in a way that the answer sets of a logic program represent the solutions of the problem. As a result, to find solutions to a problem it is sufficient to use an answer set solver on a corresponding program. These parallels that we just draw between paradigms naturally bring up a question: *what is a fundamental difference between the two?* This paper takes a close look at this question.

Keywords Propositional satisfiability · Answer set programming · Computational logic

1 Introduction

Propositional satisfiability (or satisfiability) [6] and answer set programming [7] are two closely related subareas of Artificial Intelligence that are used to model and solve difficult combinatorial search problems. Satisfiability and answer set programming in the past

✉ Yuliya Lierler
ylierler@unomaha.edu

[1] Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182, USA

decade have seen ever faster computational tools, and a growing list of successful applications. Satisfiability solvers are used as general purpose tools in areas such as hardware and software verification [5, 64]; planning [36, 66]; and scheduling [33]. Answer set programming [49, 59] is increasingly leaving its mark in tackling applications in science, humanities, and industry [7]. A space shuttle control system by Nogueira et al. [62] can be regarded as an early success story of this programming paradigm. Recent examples of answer set programming applications include a Linux package configuration tool [25] (that improves the experience of thousands Debian GNU/Linux users working with the operating system); large-scale biological networks repairs [19, 28]; and team building and scheduling [65]. Furthermore, the advances in algorithmic techniques developed in satisfiability are often borrowed and relied upon in other areas of automated reasoning including answer set programming, satisfiability modulo theory, first order model building, and constraint programming. At the same time, answer set programming tools offer remarkably sophisticated techniques for so-called model enumeration problem.

Propositional satisfiability (SAT) is the problem of determining whether a given propositional classical logic formula in conjunctive normal form (CNF formula) has a satisfying interpretation. These satisfying interpretations are called models of a formula. The problem of determining whether there exists a model for a propositional formula is NP-complete. Another relevant task to determining whether a formula is satisfiable is the task of enumerating/generating its models. Here we use the term SAT problem to denote both decision and enumeration problems. To summarize, propositional CNF formulas form the language of SAT, in other words, they form its key syntactic object. Satisfying interpretations of these formulas form the semantic objects of this paradigm. SAT solvers are the software systems that find satisfying interpretations for propositional formulas. To model a problem using SAT, propositional CNF formulas are used to encode problems specifications in a way that its satisfying interpretations correspond to the solutions of the problem. Then, to find solutions to a problem it is sufficient to use a SAT solver on a corresponding formula. Thus, SAT can be seen as a constraint programming paradigm [67].

Answer set programming (ASP) is a programming paradigm, whose key syntactic object is a logic program. A logic program is a collection of rules that in their form are reminiscent of implications in classical logic. Answer sets (stable models) [30] of a logic program form the semantic objects of ASP. These answer sets are reminiscent of satisfying interpretations of formulas in classical logic. The problem of determining whether there exists an answer set for a logic program is NP-complete. Answer set solvers are the software systems that enumerate answer sets for logic programs. Niemelä [59] and Marek and Truszczyński [49] proposed answer set programming as a declarative programming paradigm for solving difficult search problems. In this paradigm a logic program encodes problem specifications in a way that the answer sets of this program represent the solutions of the problem. As a result, to find solutions to a problem it is sufficient to use an answer set solver on a corresponding program. Thus, similarly to SAT, answer set programming can be seen as a constraint programming paradigm.

As mentioned earlier, the problems of determining whether there exists a model for a propositional formula or an answer set of a logic program is NP-complete. Therefore, SAT solvers and answer set solvers are nontrivial software systems tackling computationally difficult tasks. The Davis-Putnam-Logemann-Loveland (DPLL) procedure [11] is a well-known backtrack-search method that explores interpretations to generate models of a propositional formula. The CDCL [50, 51, 81] algorithm is at the heart of most modern SAT solvers. On the one hand, the CDCL algorithm can be seen as an enhancement of DPLL [61].

On the other hand, it has been shown that there are problems for which CDCL provides exponential speed up [4, 63]. Extensions of CDCL also form the basis of state-of-the art answer set solvers [2, 27, 31, 42]. Thus, the solving technology underlying the search for solutions is closely related in SAT and ASP.

Observed parallels between answer set programming and satisfiability naturally bring up a question: *what is a fundamental difference between the two paradigms?* This is the question that we address in this paper. A short answer that summarizes our discussion follows

> Answer set programming provides a declarative constraint *programming language*, while SAT does not.

Typically, to use a SAT solver to compute solutions of a search program, a specialized program implemented in an imperative programming language is designed to generate specific propositional formulas that encode an instance of a problem. Propositional formula is an artifact in spirit of declarative programming: it provides specifications or, in other words, constraints of a problem. In contrast to imperative programming, where we provide instructions on steps to be executed in order to achieve a solution. Yet, propositional language of SAT solvers itself is incapable to serve the role of a declarative programming language. Encoding practical problems often requires thousands to millions of propositional clauses.[1] Indeed, propositional clauses are constructed by means of propositional (Boolean) atoms so that knowledge must be encoded in these basic binary terms. As a result, one not only has to design a SAT encoding of a problem, but also find means to generate this encoding.

Answer set programming relies on dialects of a logic programming language in spirit of Prolog. Thus, its language allows first-order atoms with variables. Simple ASP programs look like Prolog programs, while more complex programs may contain specialized expressions that are convenient for modeling different kinds of constraints including aggregates. To use an answer set solver to compute solutions of a search program, a logic program that captures specifications of a problem is designed. This way no imperative programming is required as a step in utilizing answer set programming technology. To draw a parallel to terminology used in programming languages, propositional CNF formulas of SAT can be viewed as a low-level programming language, whereas logic programs of ASP form a high-level programming language. The concept of elaboration tolerance was first mentioned in [52] as the ability of a computer program's representation of a problem to accept changes in problem specifications without need to rewrite an entire program. Answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. In contrast, SAT lacks this property.

Answer set programming can be seen as a convenient declarative programming front-end for SAT-like technology. It is used to declaratively model problems' specifications. An answer set *solver* is only one of the two main building blocks of a typical ASP system. A *grounder* is another block. A grounder is a software system that takes logic programs with variables as its input and produces propositional programs (programs whose logic rules may only contain propositional atoms) as its output. Propositional programs are crucial in devising efficient solving procedures, yet it is the logic programming language with variables

[1] A clause is the main building block of a CNF formula. A clause is a disjunction of atoms or their negations, whereas a CNF formula is a conjunction of clauses.

that facilitates modeling and effective problem solving by means of answer set programming. Modern languages supported by ASP technology also support additional constructs, e.g., aggregates [20, 22], to facilitate concise modeling of problems. First-order logic is a counterpart of propositional logic that allows modeling with variables. In this paper we illustrate how a built-in closed world assumption (presumption that what is not currently known to be *true* is *false*) as well as an ability to express transitive closure makes logic programs a *convenient* modeling language for describing search problems that SAT and ASP commonly deal with. Classical first order formulas lack the closed world assumption and, more generally, are monotonic. Possibly this explains the fact that it is logic programs under answer set semantics – a nonmonotonic formalism – which built the basis for the declarative programming language utilizing SAT-like technology.

Before proceeding to the technical content of the paper we illustrate the effect of the built-in closed world assumption and the benefits of ASP as a nonmonotonic formalism on a toy *bird-penguin* domain. Consider the following knowledge:

birds normally fly
penguins do not fly

The first statement is an example of *default*—a statement that includes a word such as *normally* or *usually*. We can use this default to draw conclusions as long as they do not contradict our knowledge. The second statement presents an exception to the default *birds normally fly*: although penguins are birds they do not fly. ASP provides means to encode defaults and exceptions. For example, the following logic programming rules capture the two statements above:

$$\begin{array}{l} fly(X) \leftarrow bird(X),\ not\ abnormal\_fly(X). \\ abnormal\_fly(X) \leftarrow penguin(X). \end{array} \tag{1}$$

The former rule states the default *birds normally fly* by saying that if an entity is a bird and not known to be abnormal with respect to the flying property, then this entity flies. The later rule states that if an entity is a penguin, then it is abnormal with respect to flying property. Adding

$$bird(tweety). \tag{2}$$

as a fact to program (1) allows us to conclude that a bird named *tweety* flies, because the resulting program has a unique answer set consisting of atoms

$$\{bird(tweety), fly(tweety)\}.$$

Indeed, there is no evidence for bird *tweety* to be abnormal with respect to the flying property. This illustrates the built-in closed world assumption in ASP.

On the other hand, consider treating these rules in (1) and (2) as classical logic formulas that seemingly encode the same knowledge:

$$\begin{array}{l} bird(X) \wedge \neg abnormal\_fly(X) \rightarrow fly(X) \\ penguin(X) \rightarrow abnormal\_fly(X) \\ bird(tweety) \end{array}$$

This set of formulas has several models: one suggesting that the bird named *tweety* flies; and another one suggesting that the bird *tweety* does not fly as it is abnormal with respect to the flying property.

Now let us add the fact $penguin(tweety)$ to the program constructed from the rules in (1) and (2). The new program will correctly identify *tweety* as abnormal with respect to the

flying property and withdraw the earlier conclusion that *tweety* flies. Indeed, the only answer set of the resulting program is

$$\{penguin(tweety), bird(tweety), abnormal\_fly(tweety)\}.$$

This illustrates the benefits of ASP as a nonmonotonic formalism.

To demonstrate the elaboration tolerance of ASP, consider the case of learning more information about specific species of birds in *bird-penguin* domain. For example, in addition to the fact that *penguins do not fly* we learn that *ostriches do not fly*. We can extend program (1) with a new exception by adding the rule

$$abnormal\_fly(X) \leftarrow ostrich(X).$$

to capture new knowledge.

To compare and contrast the SAT and ASP paradigms, we start the paper by introducing propositional satisfiability (Section 2) and propositional logic programs (Section 3). Section 4 introduces notion of completion. In Section 5, logic programs with variables are presented. We also discuss common modeling patterns used in SAT and answer set programming in Sections 2, 3, 6, and 7. We use graph coloring and Hamiltonian cycle search problems as runnings examples in this paper. Section 8 presents details on SAT and answer set solvers in the form that makes it easy to draw parallels between these systems. So-called effectively propositional logic is discussed in Section 9 as a counterpart to programs with variables of ASP for SAT formalism. Section 10 elaborates on related work and provides more links to literature.

2 Satisfiability and test modeling methodology

We consider atoms as customary in predicate logic. A *literal* is an atom or a negated atom. *Ground* literals and atoms are literals and, respectively, atoms that contain no variables. A *clause* is a non-empty disjunction of literals. We sometimes identify a clause with the set of its literals. We say that a clause is *ground* if it consists of ground literals. A formula is said to be in *conjunctive normal form (CNF)* if it is a conjunction of ground clauses (possibly the empty conjunction $\top$). A set M of literals is called *consistent* if there is no atom a such that M contains a and its complement $\neg a$. A *signature* is a set of ground atoms. A set M of literals over signature σ is *complete* if for every atom a in σ either a or $\neg a$ (possibly both) occurs in M. An *interpretation* over signature σ is a complete and consistent set of literals over σ. For a formula F, by σ_F we denote the signature composed of atoms that occur in F. We also call σ_F the *signature of* F. We say that a clause C is *satisfied* by a consistent set M of literals (over σ), written $M \models C$, when $M \cap C \neq \emptyset$. Let F be a CNF formula. Formula F is *satisfied* by an interpretation M over σ_F, written $M \models F$, if M satisfies each conjunct (a clause) in F. We call such interpretations *models*. Formula F is *satisfiable* if it has models.

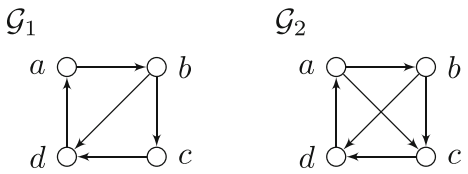
Consider a *graph coloring* problem *GC* :

> A *3-coloring* of a directed graph is a labeling of its vertexes with at most 3 colors such that no two vertexes sharing an edge have the same color.

For instance, see two specific graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Fig. 1. It is easy to see that there are six distinct 3-colorings for graph $\mathcal{G}_1$ including the following:

> assigning color 1 to vertexes a and c, color 2 to vertex b, and color 3 to vertex d forms a 3-coloring of $\mathcal{G}_1$.

Fig. 1 Sample graphs $\mathcal{G}_1$ and $\mathcal{G}_2$



We denote this 3-coloring of graph $\mathcal{G}_1$ by $\mathcal{S}_{\mathcal{G}_1}$. Graph $\mathcal{G}_2$ has no 3-colorings.

We now illustrate how the problem of finding a 3-coloring of a graph (V, E) can be encoded as a SAT problem. For every vertex $v \in V$ and every color $i \in \{1, 2, 3\}$, we introduce an atom c_{vi} that intuitively says that vertex v is assigned color i. The models of the formula

$$\begin{array}{ll} \bigvee_{1 \le i \le 3} c_{vi} & (v \in V), \\ \neg c_{vi} \vee \neg c_{vj} & (v \in V,\ 1 \le i < j \le 3), \\ \neg c_{vi} \vee \neg c_{wi} & (\{v, w\} \in E,\ 1 \le i \le 3). \end{array} \tag{3}$$

are in one to one correspondence with the 3-colorings of the graph (V, E). The first line in (3) intuitively says that each vertex is assigned some colors. The second line says that it is impossible that a vertex is assigned two colors. The third line says that it is impossible that any two adjacent vertexes are assigned the same color.

The formula in spirit of (3) for graph $\mathcal{G}_1$ consists of clauses given in Fig. 2. Horizontal lines separate the clauses that come from distinct "schematic formulas" in (3). This formula has six satisfying interpretations including

$$\{c_{a1},\ c_{b2},\ c_{c1},\ c_{d3},\ \neg c_{a2},\ \neg c_{a3},\ \neg c_{b1},\ \neg c_{b3},\ \neg c_{c2},\ \neg c_{c3},\ \neg c_{d1},\ \neg c_{d2}\}.$$

This model captures solution $\mathcal{S}_{\mathcal{G}_1}$. To encode the 3-coloring problem for graph $\mathcal{G}_2$ one has to extend the set of formulas in Fig. 2 by the following clauses

$$\neg c_{a1} \vee \neg c_{c1} \quad \neg c_{a2} \vee \neg c_{c2} \quad \neg c_{a3} \vee \neg c_{c3}$$

to account for a new edge. This formula is unsatisfiable, which says that it is impossible to color this graph in accordance with the constraints posed by 3-coloring problem.

A word TEST captures a common methodology to model a search problem using SAT. Interpretations over a signature of a CNF formula define space of possible solutions. Each clause of a CNF formula forms a constraint or, in other words, is part of a TEST that each solution to a problem has to satisfy. The specification (3) of the graph coloring problem

$c_{a1} \vee c_{a2} \vee c_{a3}$	$c_{b1} \vee c_{b2} \vee c_{b3}$	$c_{c1} \vee c_{c2} \vee c_{c3}$	$c_{d1} \vee c_{d2} \vee c_{d3}$
$\neg c_{a1} \vee \neg c_{a2}$	$\neg c_{a1} \vee \neg c_{a3}$	$\neg c_{a2} \vee \neg c_{a3}$	
$\neg c_{b1} \vee \neg c_{b2}$	$\neg c_{b1} \vee \neg c_{b3}$	$\neg c_{b2} \vee \neg c_{b3}$	
$\neg c_{c1} \vee \neg c_{c2}$	$\neg c_{c1} \vee \neg c_{c3}$	$\neg c_{c2} \vee \neg c_{c3}$	
$\neg c_{d1} \vee \neg c_{d2}$	$\neg c_{d1} \vee \neg c_{d3}$	$\neg c_{d2} \vee \neg c_{d3}$	
$\neg c_{a1} \vee \neg c_{b1}$	$\neg c_{a2} \vee \neg c_{b2}$	$\neg c_{a3} \vee \neg c_{b3}$	
$\neg c_{d1} \vee \neg c_{a1}$	$\neg c_{d2} \vee \neg c_{a2}$	$\neg c_{d3} \vee \neg c_{a3}$	
$\neg c_{b1} \vee \neg c_{d1}$	$\neg c_{b2} \vee \neg c_{d2}$	$\neg c_{b3} \vee \neg c_{d3}$	
$\neg c_{c1} \vee \neg c_{d1}$	$\neg c_{c2} \vee \neg c_{d2}$	$\neg c_{c3} \vee \neg c_{d3}$	
$\neg c_{b1} \vee \neg c_{c1}$	$\neg c_{b2} \vee \neg c_{c2}$	$\neg c_{b3} \vee \neg c_{c3}$	

Fig. 2 SAT *GC* problem encoding for graph $\mathcal{G}_1$

is a fine example of this methodology. Indeed, each line in (3) corresponds to a group of constraints that an interpretation must satisfy in order to be a solution to the graph coloring problem.

3 Propositional logic programs and generate-and-test methodology

Answer set programming practitioners develop applications that rely on ASP languages, which go beyond propositional/ground atoms. Yet, common ASP solvers, SMODELS[2] [60, 71], CLASP[3] [26, 27], WASP[4] [1, 2] to name a few, process propositional logic programs only. In this section we introduce such programs. Semantics of CNF formulas in earlier section is given by means of interpretations – sets of literals. Semantics of logic programs relies on the notion of answer sets, which are sets of atoms. A set X of atoms over some signature σ can be identified with an interpretation over σ:

$$\{a \mid a \in X\} \cup \{\neg a \mid a \in \sigma \setminus X\}.$$

A *(propositional) logic program* is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_k, \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ a_m, \mathit{not}\ \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n, \tag{4}$$

where a_0 is a ground atom or symbol $\bot$; $a_1, \ldots, a_n$ are ground atoms. We call rule (4) *normal* when $m = n$. A program is *normal* when it is composed of normal rules. The left hand side expression of rule (4) is called the *head*. The right hand side is called the *body*. Expressions

$$a_1, \ldots, a_k$$

and

$$\mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ a_m, \mathit{not}\ \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n$$

constitute *positive part* and *negative part* of the body, respectively. We call rule (4)

- a *fact* when its body is empty (we then drop $\leftarrow$ from a rule);
- a *denial* when its head is symbol $\bot$ (we then drop $\bot$ from a rule).

We call a program *definite* when it is composed of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_k. \tag{5}$$

For instance,

$$\begin{array}{l} p \\ r \leftarrow p, q \end{array} \tag{6}$$

and

$$\begin{array}{l} p \leftarrow \mathit{not}\ q \\ q \leftarrow \mathit{not}\ r \end{array} \tag{7}$$

are both normal programs, where program (6) is also definite.

We say that a set X of atoms is *closed* under a definite program Π if for all rules (5) in Π, $a_0 \in X$ whenever $a_1, \ldots, a_k \in X$. For example, sets $\{p\}$, $\{p, r\}$, and $\{p, q, r\}$ are closed under definite program (6).

[2] http://www.tcs.hut.fi/Software/smodels/.

[3] http://www.cs.uni-potsdam.de/clasp/.

[4] http://alviano.github.io/wasp/.

We say that a set X of atoms *reduces* rule (4), when $X \cap \{a_{k+1}, \ldots, a_m\} = \emptyset$ and $\{a_{m+1}, \ldots, a_n\} \subseteq X$. For instance, sets $\{p\}$ and $\{p, r\}$ reduce the first rule

$$p \leftarrow not\ q$$

of program (7), while set $\{p, q, r\}$ does not. Let Π be a program. The *reduct* Π^X of Π with respect to a set X of atoms is the set of rules (5) for all rules (4) in Π such that X reduces (4). Note that Π^X is a definite program. A set X of atoms is an *answer set* of the program Π if X is minimal among the sets of atoms that are closed under Π^X. For instance, let Π_1 denote program (7) and X_1 denote set $\{q\}$. The reduct $\Pi_1^{X_1}$ consists of a single fact q. Set $\{q\}$ is the minimal set closed under $\Pi_1^{X_1}$. Consequently, X_1 is an answer set of Π_1. Let X_2 denote set $\{p\}$. The reduct $\Pi_1^{X_2}$ consists of two facts p and q. Set $\{p, q\}$ is the minimal set closed under $\Pi_1^{X_2}$. Thus, $\{p\}$ is not an answer set of Π_1.

To the set of atoms that occur in a program Π, we refer as the *signature of* Π. When convenient, we identify answer sets of Π with interpretations over the signature of Π. In these cases we refer to answer sets as *stable models* of Π.

We now consider several special case programs in order to illustrate some interesting properties of answer sets. Let a program consist of facts only. The set of these facts form the only answer set of such a program. Intuitively, each fact states what is known and an answer set reflects this information by asserting that each atom mentioned as a fact is *true*, whereas anything else is *false*. Thus answer sets semantics follows closed world assumption (CWA) – presumption that what is not currently known to be *true* is *false*.

Consider a definite program. It is obvious that the reduct of a definite program Π with respect to any set of atoms coincides with Π. We can trivially simplify the definition of an answer set for a definite program Π: A set X of atoms is an *answer set* of Π if X is minimal among the sets of atoms that are closed under Π. Consequently, definite programs without denials have a unique answer set. For instance, set $\{p\}$ is the only answer set of definite program (6).

It is intuitive to argue that answer set semantics for logic programs generalizes CWA. This is a good place to note that an atom, which does not occur in the head of some rule in a program, will not be a part of any answer set of this program.

Closed world assumption has been recognized important in design of knowledge representation languages, thus it is not surprising that logic programs under answer set semantics "found their home" in knowledge representation and reasoning community and answer set programming is often positioned as a prominent knowledge representation and reasoning formalism.

Let a program consist of the rule

$$p \leftarrow not\ not\ p. \tag{8}$$

This program has two answer sets $\emptyset$ and $\{p\}$. This rule can be used to "eliminate" CWA for an atom p. Intuitively, it states that an atom p *may* be a part of an answer set. Denecker et al. [13] elaborate on this observation.

The version of the language of logic programs that allows doubly negated atoms is a special case of programs with nested expressions introduced by Lifschitz et al. [46]. This extension of logic programs is essential. *Choice rules* [60] of the form

$$\{a_0\} \leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, not\ a_m, \tag{9}$$

are important constructs of common ASP dialects. Ferraris and Lifschitz [18] show that a choice rule (9) can be seen as an abbreviation for a rule with doubly negated atoms of the form

$$a_0 \leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, not\ a_m, not\ not\ a_0.$$

In this work we adapt this abbreviation. The simplest choice rule

$$\{p\}$$

corresponds to rule (8).

Consider a denial $\leftarrow p$. Extending program (6) by this rule will result in a program that has no answer sets. In other words, denial $\leftarrow p$ eliminates the only answer of (6). It is convenient to view any denial

$$\leftarrow a_1, \ldots, a_k, not\ a_{k+1}, \ldots, not\ a_m, not\ not\ a_{m+1}, \ldots, not\ not\ a_n, \tag{10}$$

as a clause

$$\neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m \vee \neg a_{m+1} \vee \cdots \vee \neg a_n. \tag{11}$$

Then, we can state the general property about denials: stable models of a program satisfy the CNF formula composed of its denials. Furthermore, for a program Π and a set Γ of denials the stable models of $\Pi \cup \Gamma$ coincide with the stable models of Π that satisfy Γ [43]. Consequently, denials can be seen as elements of classical logic in logic programs.

We now present how a propositional logic program under answer set semantics encodes the graph coloring problem *GC* so that answer sets of this program are in one to one correspondence with the 3-colorings of a given graph (V, E). As in case of SAT encoding (3), we take atoms of the form c_{vi} to stand for an assertion that vertex v is assigned color i. Consider a following program

$$\begin{aligned}
&\{c_{vi}\} && (v \in V, 1 \leq i \leq 3)\\
&\leftarrow c_{vi}, c_{vj} && (v \in V,\ 1 \leq i < j \leq 3),\\
&\leftarrow c_{vi}, c_{wi} && (\{v, w\} \in E,\ 1 \leq i \leq 3),\\
&\leftarrow not\ c_{v1}, not\ c_{v2}, not\ c_{v3}. && (v \in V).
\end{aligned} \tag{12}$$

A collection of choice rules for each vertex v captured by the first line in (12) intuitively says that vertex v may be assigned some colors. The second line says that it is impossible that a vertex is assigned two colors. The third line states that it is impossible that any two adjacent vertexes are assigned the same color. The last line states that it is impossible that a vertex is not assigned a color.

Recall graph $\mathcal{G}_1$ in Fig. 1. In Fig. 2 we illustrated SAT encoding stated in (3) for this graph. Figure 3 presents a logic program in spirit of (12) for $\mathcal{G}_1$. Horizontal lines separate the clauses that come from distinct "schematic rules" in (12). This program has six answer sets including

$$\{c_{a1},\ c_{b2},\ c_{c1},\ c_{d3}\},$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. To encode the 3-coloring problem for graph $\mathcal{G}_2$ one has to extend the set of rules in Fig. 3 with rules

$$\leftarrow c_{a1}, c_{c1} \qquad \leftarrow c_{a2}, c_{c2} \qquad \leftarrow c_{a3}, c_{c3}.$$

This program has no answer sets, which captures the fact that this graph has no 3-colorings.

$\{c_{a1}\}$	$\{c_{a2}\}$	$\{c_{a3}\}$	$\{c_{b1}\}$	$\{c_{b2}\}$	$\{c_{b3}\}$	$\{c_{c1}\}$	$\{c_{c2}\}$	$\{c_{c3}\}$	$\{c_{d1}\}$	$\{c_{d2}\}$	$\{c_{d3}\}$
$\leftarrow c_{a1},\ c_{a2}$			$\leftarrow c_{a1},\ c_{a3}$			$\leftarrow c_{a2},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{b2}$			$\leftarrow c_{b1},\ c_{b3}$			$\leftarrow c_{b2},\ c_{b3}$					
$\leftarrow c_{c1},\ c_{c2}$			$\leftarrow c_{c1},\ c_{c3}$			$\leftarrow c_{c2},\ c_{c3}$					
$\leftarrow c_{d1},\ c_{d2}$			$\leftarrow c_{d1},\ c_{d3}$			$\leftarrow c_{d2},\ c_{d3}$					
$\leftarrow c_{a1},\ c_{b1}$			$\leftarrow c_{a2},\ c_{b2}$			$\leftarrow c_{a3},\ c_{b3}$					
$\leftarrow c_{d1},\ c_{a1}$			$\leftarrow c_{d2},\ c_{a2}$			$\leftarrow c_{d3},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{d1}$			$\leftarrow c_{b2},\ c_{d2}$			$\leftarrow c_{b3},\ c_{d3}$					
$\leftarrow c_{c1},\ c_{d1}$			$\leftarrow c_{c2},\ c_{d2}$			$\leftarrow c_{c3},\ c_{d3}$					
$\leftarrow c_{b1},\ c_{c1}$			$\leftarrow c_{b2},\ c_{c2}$			$\leftarrow c_{b3},\ c_{c3}$					
$\leftarrow not\ c_{a1}, not\ c_{a2}, not\ c_{a3}.$											
$\leftarrow not\ c_{b1}, not\ c_{b2}, not\ c_{b3}.$											
$\leftarrow not\ c_{c1}, not\ c_{c2}, not\ c_{c3}.$											
$\leftarrow not\ c_{d1}, not\ c_{d2}, not\ c_{d3}.$											

Fig. 3 Logic program for 3-coloring for graph $\mathcal{G}_1$

Just as the SAT specification (3) of the graph coloring problem *GC* is a fine example of the TEST methodology by means of SAT, the ASP specification (12) illustrates the use of the so-called GENERATE and TEST methodology within answer set programming paradigm. The GENERATE part of the specification "defines" a collection of answer sets (interpretations) that can be seen as potential solutions. The TEST part consists of conditions that eliminate the answer sets of the GENERATE part that do not correspond to solutions. The first line in (3) corresponds to GENERATE, whereas the remaining lines correspond to TEST. Observe, how denials are used to formulate TEST.

It is interesting to note how the GENERATE part is non-existent in SAT specifications. Any interpretation of the signature of a problem forms a potential solution in SAT: so in a sense the GENERATE part is implicit. CWA present in answer set programming poses a need for the explicit GENERATE part. Choice rules provide a convenient tool in ASP for formulating GENERATE.

4 Completion

The SAT and ASP formulations (3) and (12) look closely related. This is not by chance. Fages [17] showed that for a large syntactic class of "tight" programs, their stable models coincide with the models of programs' "completion" – a propositional formula defined by Clark [10]. It turns out that

i. the completion of program (12) is equivalent to formula (3) (in fact, a simple equivalent transformation on the completion of program (12) will result in formula (3)), and
ii. program (12) is tight.

We now recall the definitions of tightness and Clark's completion.

For any propositional program Π, the *dependency graph* of Π is the directed graph that

- has all atoms occurring in Π as its vertexes, and
- for each rule (4) in Π has an edge from a_0 to a_i, where $1 \leq i \leq k$.

We say that a program is *tight* if its dependency graph is acyclic.

Fig. 4 Sample dependency graphs for programs (6) and (7)

Sample programs (6) and (7) are tight. Figure 4 presents the dependency graph for these programs. The encoding (12) of the graph coloring *GC* problem is also a tight program as its dependency graph has no edges. Many practical logic programs are tight.

The simple non-tight program follows

$$p \leftarrow p. \tag{13}$$

Indeed, its dependency graph consists of a single vertex p that has an edge from this vertex to itself.

Let us identify the rule (4) with the clause

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_k \vee a_{k+1} \vee \cdots \vee a_m \vee \neg a_{m+1} \vee \cdots \vee \neg a_n.$$

As before, in case if the rule is denial (10), then we identify this rule with the clause (11). This allows us to view a program Π as a CNF formula. It is also convenient to identify the *body*

$$a_1, \ldots, a_k, \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ a_m, \mathit{not}\ \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n, \tag{14}$$

of a rule (4) with the conjunction

$$a_1 \wedge \cdots \wedge a_k \wedge \neg a_{k+1} \wedge \cdots \wedge \neg a_m \wedge a_{m+1} \wedge \cdots \wedge a_n.$$

If a body is empty (for a rule that is a fact), we identify it with the empty conjunction $\top$.

The *completion* of a program Π, $comp(\Pi)$, is a conjunction of

(i) equivalences of the form

$$a \leftrightarrow \bigvee_{a \leftarrow B \in \Pi} B \tag{15}$$

for each atom a that occurs in Π and

(ii) the clauses that correspond to denials of Π.

If there is an atom a in a program Π that never occurs in the head of any rule then we view (15) as the clause $\neg a$. Process of completion captures the intuition that the collection of rules in a program that share the same atom in their head *defines* this atom or, in other words, provides complete list of specifications under which this atom holds. Thus, the fact that a clause $\neg a$ is added for any atom a that does not occur in any head of a program captures CWA explicitly: if there is no reason to deduce an atom a, it is then *false*.

Recall sample programs (6) and (7). The completion of the former program is

$$\begin{array}{l} p \leftrightarrow \top \\ r \leftrightarrow p \wedge q \\ \neg q, \end{array} \tag{16}$$

while the completion of the latter is

$$\begin{array}{l} p \leftrightarrow \neg q \\ q \leftrightarrow \neg r \\ \neg r. \end{array} \tag{17}$$

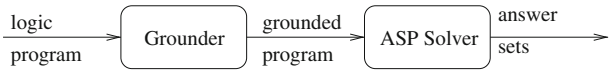


Fig. 5 Common Architecture of ASP Systems

Consider another sample program

$$\begin{array}{l} \{p\} \\ \{s\} \\ q \leftarrow not\ r \\ q \leftarrow r, s \\ \leftarrow q,\ not\ p. \end{array} \tag{18}$$

Its completion follows

$$\begin{array}{l} p \leftrightarrow p \\ s \leftrightarrow s \\ q \leftrightarrow \neg r \vee (r \wedge s) \\ \neg r \\ \neg q \vee p. \end{array} \tag{19}$$

The only model of (16) is set $\{p, \neg q, \neg r\}$. The only model of (17) is set $\{\neg p, q, \neg r\}$. There are two models of (19): sets $\{p, q, \neg r, \neg s\}$ and $\{p, q, \neg r, s\}$. These models are also stable models of respective programs. This is not by chance. Programs (6), (7), and (18) are tight and hence their models of completion and stable models coincide.

The completion of program (13) consists of a single equivalence

$$p \leftrightarrow p.$$

This formula has two models $\{p\}$ and $\{\neg p\}$, while program (13) has only one stable model $\{\neg p\}$. It is a general fact that any stable model of a program Π is a model of $comp(\Pi)$. The converse does not hold generally (but in case of tight programs the converse also holds).

Recall program (12). Its completion is composed of (3) and equivalence

$$c_{vi} \leftrightarrow c_{vi} \ \ (v \in V, 1 \leq i \leq 3).$$

This formula is a tautology in propositional logic and can be safely dropped form the completion.

5 Programs with variables

As mentioned earlier, ASP practitioners develop applications that rely on languages, which go beyond propositional/ground atoms. Figure 5 presents a typical architecture of an answer set programming tool that encompasses two parts: a system called *grounder* and a system called *solver*. The former is responsible for eliminating variables in a program. The latter is responsible for finding answer sets of a respective propositional (ground) program. Systems LPARSE [74] and GRINGO[5] [23, 29] are two well known grounders that serve as front-ends for many solvers including SMODELS [71], CLASP [27], and WASP [2].

In this section we refer to non-ground atoms that are constructed as customary in predicate logic: they may contain variables, object constants, and function symbols.

[5] http://potassco.sourceforge.net/.

A *logic program with variables* is a finite set of rules of the form (4), where a_0 is symbol $\bot$ or a non-ground atom; and $a_1, \ldots, a_n$ are non-ground atoms. Grounding a logic program replaces each rule with all its instances obtained by substituting ground terms, formed from the object constants and function symbols occurring in the program, for all variables. For a program Π, by $ground(\Pi)$ we denote the result of its grounding. (We use the convention common in logic programming: variables are represented by capitalized identifiers.) For example, let Π be a program

$$\begin{array}{l} \{a(1)\}\ \{a(2)\}\ \{b(1)\} \\ c(X) \leftarrow a(X), b(X), \end{array} \tag{20}$$

$ground(\Pi)$ follows

$$\begin{array}{l} \{a(1)\}\ \{a(2)\}\ \{b(1)\} \\ c(1) \leftarrow a(1), b(1) \\ c(2) \leftarrow a(2), b(2). \end{array}$$

The answer sets of a program Π with variables are answer sets of $ground(\Pi)$ [30]. For instance, there are eight answer sets of program (20) including $\emptyset$ and set $\{a(1)\ b(1)\ c(1)\}$.

Given a program Π with variables, grounders often produce a variable-free program that is smaller than $ground(\Pi)$, but still has the same answer sets as $ground(\Pi)$; we call any such program an *image* of Π. For example, program

$$\begin{array}{l} \{a(1)\}\ \{a(2)\}\ \{b(1)\} \\ c(1) \leftarrow a(1), b(1) \end{array}$$

is an image of (20).

When a program Π with variables has at least one function symbol and at least one object constant, grounding results in infinite $ground(\Pi)$. Yet, even for an input program of this kind, grounders often find an image that is a finite propositional program (finite image). For instance, for program

$$\begin{array}{l} p(0) \\ q(f(X)) \leftarrow p(X) \end{array} \tag{21}$$

grounding results in infinite program outlined below

$$\begin{array}{l} p(0) \\ q(f(0)) \leftarrow p(0) \\ q(f(f(0))) \leftarrow p(f(0)) \\ q(f(f(f(0)))) \leftarrow p(f(f(0))) \\ \ldots \end{array}$$

A finite image of program (21) follows

$$\begin{array}{l} p(0) \\ q(f(0)) \leftarrow p(0). \end{array}$$

Program

$$\begin{array}{l} p(0) \\ q(f(0)) \end{array}$$

is another image of (21). In fact, given program (21) as an input grounders LPARSE and GRINGO will generate the latter image.

To produce images for input programs, grounders follow techniques exemplified by intelligent grounding [9]. Different grounders implement distinct procedures so that they may generate different images for the same input program. One can intuitively measure the quality of a produced image by its size so that the smaller the image is the better. A common

syntactic restriction that grounders pose on input programs is "safety". A program Π is *safe* if every variable occurring in a rule of Π also occurs in positive body of that rule. For instance, programs (20) and (21) are safe. The safety requirement suggests that positive body of a rule must contain information on the values that should be substituted for a variable in the process of grounding. Safety is instrumental in designing grounding techniques that utilize knowledge about the structure of a program for constructing smaller images [9]. The GRINGO grounder and the grounder of the DLV system expect programs to be safe. For programs with function symbols, to guarantee that the grounding process terminates, grounders pose additional syntactic restrictions (in other words, to guarantee that a grounder is able to construct a finite image). For example, grounder LPARSE expects given programs to belong to a so-called ω-restricted class [73]. Calautti et al. [8] defined a class of programs that is more general than ω-restricted class and for which bottom-up approaches for grounding implemented in DLV and GRINGO are guaranteed to terminate.

Often a set of propositional rules that follow a simple pattern can be represented concisely by means of logic programs with variables. We support this statement by an example. Recall program (12). We now capture atoms of the form c_{vi} by expressions $c(v, i)$, where c is a predicate symbol and v, i are object constants denoting a vertex v and color i respectively. Atom of the form $vtx(v)$, intuitively, states that an object constant v is a vertex, while atom $e(v, w)$ states that there is an edge from vertex v to vertex w in a given graph. Atom $color(i)$ states that an object constant i represents a color. Recall graph coloring problem *GC* for an input graph (V, E). We now present a program with variables that encodes a solution to this problem. First, this program consists of facts that encode graph (V, E):

$$\begin{array}{ll} vtx(v) & (v \in V) \\ e(v, w) & (\{v, w\} \in E) \end{array} \tag{22}$$

Second, facts

$$color(c) \qquad (c \in 1, 2, 3) \tag{23}$$

enumerate three colors of the problem. The following rules conclude the description of the program:

$$\{c(V, I)\} \leftarrow vtx(V),\ color(I) \tag{24}$$

$$\leftarrow c(V, I),\ c(V, J),\ I < J,\ vtx(V),\ color(I),\ color(J) \tag{25}$$

$$\leftarrow c(V, I),\ c(W, I),\ vtx(V),\ vtx(W),\ color(I),\ e(V, W) \tag{26}$$

$$\leftarrow notc(V, 1), notc(V, 2), notc(V, 3),\ vtx(V) \tag{27}$$

These rules are the counterparts of groups of rules in propositional program (12). Indeed, rule (24) states that every vertex may be assigned some colors; rule (25) says that it is impossible that a vertex is assigned two colors; rule (26) says that it is impossible that any two adjacent vertexes are assigned the same color; and the last rule (27) states that it is impossible that a vertex is not assigned a color.

Programs with variables permit for a concise encoding of an instance of a search problem. Indeed, size of a program composed of rules (22–27) is almost identical to the size of a given graph (V, E). There are $|V|+|E|+7$ rules in this program. On the other hand, the line

$$\leftarrow c_{vi}, c_{wi}\ \ (\{v, w\} \in E,\ 1 \leq i \leq 3)$$

of program (12) alone encapsulates $3|E|$ rules.

6 Modeling of search problems in ASP

Answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. We follow the lines of [7] in defining a search problem abstractly. A *search problem* P consists of a set of instances with each *instance* I assigned a finite set $S_P(I)$ of solutions. In answer set programming to solve a search problem P, we construct a program Π_P that captures problem specifications so that when extended with facts D_I representing an instance I of the problem, the answer sets of $\Pi_P \cup D_I$ are in one to one correspondence with members in $S_P(I)$. In other words, answer sets describe all solutions of problem P for the instance I. Thus solving of a search problem is reduced to finding a uniform encoding of its specifications by means of a logic program with variables.

For example, an instance of the graph coloring search problem *GC* is a graph. All 3-colorings for a given graph form its solutions set. Consider any graph (V, E). By $D_{(V,E)}$ we denote facts in (22) that encode graph (V, E). By Π_{gc} we denote a program composed of rules in (23–27). This program captures specifications of 3-coloring problem so that answer sets of $\Pi_{gc} \cup D_{(V,E)}$ correspond to solutions to instance graph (V, E) of a problem. Recall graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ presented in Fig. 1. Facts $D_{\mathcal{G}_1}$ representing $\mathcal{G}_1$ follow

$$vtx(a)\ \ vtx(b)\ \ vtx(c)\ \ vtx(d)\ \ e(a,b)\ \ e(b,c)\ \ e(c,d)\ \ e(d,a)\ \ e(b,d).$$

Program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ has six answer sets, including

$$\begin{array}{l}\{vtx(a)\ vtx(b)\ vtx(c)\ vtx(d)\\ e(a,b)\ e(b,c)\ e(c,d)\ e(d,a)\ e(b,d)\\ color(1)\ color(2)\ color(3)\\ c(a,1)\ c(b,2)\ c(c,1)\ c(d,3)\},\end{array}$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. Similarly, we can use encoding Π_{gc} to establish whether 3-colorings exist for graph $\mathcal{G}_2$. Facts $D_{\mathcal{G}_2}$ representing $\mathcal{G}_2$ consists of facts in $D_{\mathcal{G}_1}$ and an additional fact $e(a, c)$. Program $\Pi_{gc} \cup D_{\mathcal{G}_2}$ has no answer sets suggesting that no 3-colorings exist for graph $\mathcal{G}_2$.

It is important to mention that the languages supported by ASP grounders and solvers go beyond rules presented here. For instance, GRINGO versions 4.5+ support such constructs as aggregates, cardinality expressions, intervals, pools [20, 22]. It is beyond the scope of this paper to formally discuss these constructs, but it is worth mentioning that they generally allow us more concise, intuitive, and elaboration tolerant encodings of problems. Also, they often permit to utilize more sophisticated and efficient procedures in solving [24]. It is interesting to mention pseudo-Boolean solvers [68] – systems closely related to SAT solvers. These systems implement specialized solving techniques for processing an extension of propositional CNF logic that supports expressions in spirit of cardinality constructs. Thus, it is well recognized that these constructs bring modeling and solving advances into formalisms.

For instance, a single rule

$$\leftarrow not\, 1\{c(V, I) : color(I)\}1,\ \ vtx(V). \tag{28}$$

can replace two rules (25) and (27) in program Π_{gc}. This shorter program will result in smaller groundings for instances of the *GC* problem paving the way to more efficient solving. Cardinality construct $1\{c(V, I) : color(I)\}1$ intuitively suggests us to count, for a given value of V the atoms of the form $c(V, I)$ that belong to the answer set. Number 1 to the right and to the left of this aggregate expression tells us a specific condition on the count,

in particular, that it has to be exactly 1. Cardinality expressions form only one example of multitude of constructs that GRINGO language offers for effective modeling of problem specifications.

In this section, we illustrated one essential difference between ASP and SAT. Propositional SAT language does not allow variables and thus for each considered graph a separate complete SAT encoding must be formulated. In case of ASP, it is possible to formulate a logic program in a way that to solve graph coloring problem it is sufficient to only specify details of a new graph in question. In introduction, we mentioned that typically to use a SAT solver to compute solutions for an instance of a search problem a specialized program in imperative language is written to generate specific propositional formula that encodes given instance as a CNF formula. Answer set programming provides another way to utilize SAT technology. For example, constructed logic program Π_{gc} with variables for solving graph coloring problem *GC* can be used to invoke a SAT solver for solving *GC* as follows. Consider some instance graph (V, E) encoded by means of the facts $D_{(V,E)}$. Applying a grounder on the union $\Pi_{gc} \cup D_{(V,E)}$ results in a tight propositional program. Applying a SAT solver on a clausified completion of this propositional program allows us to enumerate the solutions to graph coloring problem for graph (V, E). In Section 8, we present details behind several modern answer set solvers. It is remarkable that several systems including CMODELS[6] [31] and CLASP compute answer sets of tight programs in a described fashion. In particular, for a tight program these systems start by computing a clausified completion. System CMODELS then invokes a SAT solver (e.g., SAT solver MINISAT [15]) to find models of a computed formula. System CLASP has an internal implementation of a SAT solver that is invoked on computed completion.

7 The generate-define-and-test modeling methodology of ASP

Section 3 presented how the GENERATE and TEST methodology is applicable within answer set programming. Yet, an essential feature of logic programs is their ability to elegantly and concisely "define" predicates. Denecker [12] argues that normal logic programs provide a convenient language for expressing inductive definitions. He considers normal logic programs under parametrized well-founded semantics. Truszczyński [76] illustrates that for practical subset of logic programs answer set semantics and parametrized well-founded semantics coincide. Thus, logic programs under answer set semantics can be seen as a formalism that allows one to combine classical logic (recall our discussion about denials) with inductive definitions. Denecker et al. [13] elaborate on this subject and provide a detailed account on intuitive readings of connectives in logic programs. We direct interested readers to Denecker et al. work for the presentation of Tarskian informal semantics of logic programs.

The GENERATE, DEFINE, and TEST is a typical methodology used by ASP practitioners in designing programs. Lifschitz [44] coined this term. It generalizes the GENERATE and TEST methodology discussed earlier. The roles of the GENERATE and TEST parts of a program stay the same so that, informally, GENERATE defines a large collection of answer sets that could be seen as potential solutions, while TEST consists of rules that eliminate the answer sets of the GENERATE part that do not correspond to solutions. The DEFINE section expresses additional, auxiliary concepts and connects the GENERATE and TEST parts.

[6] http://www.cs.utexas.edu/users/tag/cmodels.html.

To illustrate the essence of DEFINE, consider a *Hamiltonian cycle* search problem:

Given a directed graph (V, E), the goal is to find a *Hamiltonian cycle* — a set of edges that induce in (V, E) a directed cycle going through each vertex exactly once.

This is an important combinatorial search problem related to Traveling Salesperson problem.

We now formalize the specifications of Hamiltonian cycle problem by means of GENERATE, DEFINE, and TEST methodology. As in the encoding Π_{gc} of the graph coloring problem, we use expressions of the form $vtx(v)$ and $e(v, w)$ to encode an input graph. To formulate the GENERATE part, we introduce a predicate symbol in so that the expression $in(v, w)$ states that edge (v, w) is part of found Hamiltonian cycle. Choice rule

$$\{in(X, Y)\} \leftarrow e(X, Y) \tag{29}$$

forms the GENERATE part of the problem. This rule states that any subset of edges of a given graph may form a Hamiltonian cycle. Answer sets of a program composed of this rule and a set of facts encoding an input graph will correspond to all subsets of edges of the graph. For instance, program composed of facts $D_{\mathcal{G}_1}$ that encode directed graph $\mathcal{G}_1$ introduced in Fig. 1 extended by rule (29) has 32 answer sets each representing a different subset of its edges.

In order to state the TEST part, an auxiliary concept of $reachable$ is required so that we can capture the restriction that a found subset of edges of the graph is also a cycle. The DEFINE part follows

$$\begin{array}{l} reachable(V, V) \leftarrow vtx(V) \\ reachable(U, W) \leftarrow in(U, V),\ reachable(V, W), \\ \qquad\qquad\qquad\qquad vtx(U),\ vtx(V),\ vtx(W) \end{array} \tag{30}$$

These rules define *transitive closure* of the predicate in: all pairs of vertexes (u, v) such that v can be reached from u by following zero or more edges that are in. The in edges form a Hamiltonian cycle if and only if every pair of vertexes is in the transitive closure.

We are now ready to state the TEST part composed of three rules

$$\begin{array}{l} \leftarrow in(U, V),\ in(U, W),\ V \neq W,\ vtx(U),\ vtx(V),\ vtx(W) \\ \leftarrow in(U, V),\ in(W, V),\ U \neq W,\ vtx(U),\ vtx(V),\ vtx(W) \\ \leftarrow not\ reachable(U, V),\ vtx(U),\ vtx(V). \end{array} \tag{31}$$

The former two rules state that no two selected edges start or end in the same vertex, respectively. The last rule states that it is impossible for a pair of vertexes not to belong to a transitive closure of the predicate in. In other words, a cycle formed by in-edges must include every vertex. Rules (29), (30), and (31) form a program Π_{hc} that captures specifications of Hamiltonian cycle search problem. Extending Π_{hc} with facts representing a directed graph results in a program whose answer sets describe all Hamiltonian cycles of this graph. For example, program $\Pi_{hc} \cup D_{\mathcal{G}_1}$ has only one answer set. Set

$$\{in(a, b)\ in(b, c)\ in(c, d)\ in(d, a)\}$$

contains all in-edges of that answer set stating that edges (a, b), (b, c), (c, d), and (d, a) form the only Hamiltonian cycle for graph $\mathcal{G}_1$.

Concise encoding of transitive closure is a feature of answer set programming that constitutes an essential difference between ASP and SAT or effectively propositional logic — first-oder generalization of pure propositional satisfiability (discussed in more detail in Section 9). For instance, to solve Hamiltonian cycle problem, known encodings of reachability lead to propositional logic representations of larger size than these of respective

propositional logic programs. East and Truszczyński [14] demonstrated that the performance of SAT solvers on propositional encodings of the Hamiltonian cycle problem lags dramatically behind that of the answer set solver ASPPS. Lierler and Lifschitz [41] say that effectively propositional logic reasoning can be described as the common part of classical first-order logic and logic programming under the answer set semantics. Transitive closure is not expressible by first-order formulas. Thus any subset of first-order logic taken as the language with variables for modeling search problems declaratively will fail at defining (directly) concepts that rely on transitive closure.

8 Satisfiability and answer set solving

Most modern answer set solvers are close relatives to SAT solvers based on the CDCL algorithm [51]. The CDCL algorithm can be seen as an enhancement of classical DPLL procedure [11]. Clause learning and backjumping are features that distinguish CDCL from DPLL. These techniques proved to be truly essential in building effective solvers. Yet, here we avoid describing these features and instead direct interested readers to [51, 61] for more details on learning and backjumping in SAT and to [27, 40] for more details on learning and backjumping in ASP. In practice, key ideas in learning and backjuming are identical for both SAT and ASP solvers. Thus, for our purposes it is sufficient to present DPLL and then extend DPLL to a procedure that can be used for finding answer sets of a program. We call this procedure ASET. An ASET algorithm is in the same relation to modern answer set solving algorithms as DPLL to CDCL. By contrasting and comparing the DPLL and ASET algorithms we are able to point out key difference between SAT and ASP solving methods.

Answer set solvers, such as SMODELS [60], SMODELS$_{cc}$ [78], and DLV [38], are so-called *native* systems. They are based on specialized search procedures in spirit of the DPLL algorithm. The core of DPLL consists of performing three basic operations: decision, unit propagate, and backtrack. Unit propagate operation is based on a simple inference rule in propositional logic that given a CNF formula F allows to utilize knowledge about unit clauses (clauses that consist of a single literal) occurring in F or being inferred so far by the DPLL procedure, in order to conclude new inferences. Native answer set solvers replace (or augment) unit propagate of DPLL by specialized operations based on inference rules suitable in the context of logic programs. For example, SMODELS implements four propagators *Unit Propagate* , *All Rules Cancelled* , *Backchain True* , and *Unfounded* that we discuss in detail later in this section. Solver SMODELS$_{cc}$ extends the algorithm of SMODELS by backjumping and learning. Clause learning is an advanced solving technique that originated in SAT [51, 56] and proved to be extremely powerful in practice. The distinguishing feature about the answer set solver DLV is its ability to handle *disjunctive* answer set programs. In rules of such programs a disjunction of atoms in place of a single atom is allowed in heads. The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete [16].

Native answer set solvers form one core group of ASP systems. Another group is formed by so-called SAT-based answer set solvers. To explain intuitions behind the latter we present some auxiliary terminology.

In this section we use terms an atom, a literal, and a clause to refer to ground instances of these objects. For a set M of literals, by M^+ we denote atoms that occur positively in M. For example, $\{\neg a, b\}^+ = \{b\}$. For set σ of atoms and set M of literals, by $M_{|\sigma}$ we denote the maximal subset of M over σ. For example, $\{a, \neg b, c\}_{|\{a,b\}} = \{a, \neg b\}$. For a program Π, by σ_Π we denote the set of all atoms occurring in Π or, in other words, the signature of Π.

There exists a number of transformations from logic programs under answer set semantics to SAT. Given a propositional logic program Π, there are two kinds of transformations:

- transformations that form a formula F_Π, which may contain "new atoms" so that
  - (i) $\{M^+_{|\sigma_\Pi} \mid M \text{is a model of } F_\Pi\} = \{M \mid M \text{ is an answer set of } \Pi\}$, and
  - (ii) there are no distinct models M and M' of F_Π such that $M^+_{|\sigma_\Pi} = M'^+_{|\sigma_\Pi}$.
- transformations that preserve the vocabulary of Π and form a formula F_Π so that M is a model of F_Π if and only if M^+ is an answer set of Π.

Answer set solver LP2SAT[7] [34, 35] is an example of a system that relies on a transformation of the former kind. It starts its computation by producing a SAT instance based on a given normal logic program. The length of the SAT instance as well as the transformation time are of order $||\Pi|| \times log_2|\sigma_\Pi|$, where $||\Pi||$ is the length of the program Π. System LP2SAT then invokes a SAT solver to compute models' of the SAT instance that are in one to one correspondence with the answer sets of a given program.

Completion [10] is a remarkable transformation of the later kind for the large class of tight programs. As mentioned earlier, for these programs the models of program's completion coincide with the stable models of a program. This fact is exploited in several state-of-the-art answer set solvers including CLASP [27], CMODELS [31], IDP[8] [80]. For tight programs these solvers apply off-the-shelve SAT-solvers "as is" to clausified program's completion.[9] Nontight logic programs can be translated into propositional formulas preserving their vocabulary only with exponential blow up in general case [45] (assuming $P \not\subseteq NC^1/poly$, a conjecture from the theory of computational complexity that is widely believed to be true). One such transformation relies on extending program's completion with so-called loop formulas [48]. Loop formulas play essential role in many modern answer set solvers, including CLASP, CMODELS, and IDP, in case of non-tight programs. They are directly related to the concept of unfounded sets presented later [37].

The major benefit of the first approach exemplified by LP2SAT is in the fact that it immediately benefits from any advances in SAT technology as it considers any SAT solver as a "black box". In other words, it communicates with a SAT solver via its input-output interface without interacting with any internal components of the system. Thus any new SAT solver can easily be integrated into the LP2SAT framework. The second approach exemplified by CLASP, CMODELS, and IDP requires development of specialized procedures. Systems CMODELS and IDP are implemented as extensions of the SAT solver MINISAT [15]. Although, these systems are able to utilize the sophisticated features of MINISAT, an appearance of any novel SAT solver does not translate into immediate advances for them. Yet, the inference related to loop formulas (or unfounded sets) implemented in these solvers proved to play an essential role in the success of the second approach. The answer set solver ASSAT [47] is a proponent of a third intermediate approach. Similarly to the LP2SAT framework, ASSAT utilizes a SAT solver as a black box. Similarly to CLASP and its peers, ASSAT utilizes the concepts of completion and loop formula in its search. An execution of ASSAT can be visualized as a sequence of calls to a SAT solver so that at each call completion of a program is extended with additional loop formulas, which permits a more precise

[7] http://www.tcs.hut.fi/Software/lp2sat/.

[8] http://dtai.cs.kuleuven.be/krr/software/idp.

[9] System CLASP is both a SAT solver and an answer set solver. SAT solver MINISAT [15] forms a backbone of systems CMODELS and IDP.

description of the original problem. In the worst case scenario, it is possible that exponential number of loop formulas have to be added in this process and, respectively, a SAT solver must be called an exponential number of times.

In the rest of this section we present the DPLL algorithm [11], a classical SAT procedure. We then provide an extension of this algorithm that captures the basis of modern answer set solvers including such solvers as SMODELS, CMODELS, and CLASP. We conclude by drawing a parallel between SAT and answer set solvers.

DPLL Nieuwenhuis et al. [61] described DPLL by means of a transition system that can be viewed as an abstract representation of the underlying DPLL computation. This transition system captures what "states of computation" are, and what transitions between states are allowed. In this way, a transition system defines a directed graph such that every execution of the DPLL procedure corresponds to a path in this graph. Some edges may correspond to unit propagation steps, some to decision, some to backtracking. We follow this approach for describing search algorithms. We now review the abstract DPLL in the form convenient for our purposes.

For a set σ of atoms, a *record* relative to σ is a sequence M of distinct literals over σ, some possibly annotated by Δ, which marks them as *decision* literals. A *state* relative to σ is either a distinguished state *FailState* or a record relative to σ. For instance, the states relative to a singleton set $\{p\}$ are

$$\begin{array}{l} \mathit{FailState},\ \emptyset,\ p,\ \neg p,\ p^{\Delta},\ \neg p^{\Delta},\ p\ \neg p,\ p^{\Delta}\ \neg p, \\ p\ \neg p^{\Delta},\ p^{\Delta}\ \neg p^{\Delta},\ \neg p\ p,\ \neg p^{\Delta}\ p,\ \neg p\ p^{\Delta},\ \neg p^{\Delta}\ p^{\Delta}. \end{array}$$

Note how a sequence of literals such as $p\ p$ or $p\ p^{\Delta}$ does not form a record. Frequently, we consider M as a set of literals, ignoring both the annotations and the order among its elements. If neither a literal l nor its complement, written $\overline{l}$, occurs in M, then l is *unassigned* by M. We say that M is *inconsistent* if both an atom a and its negation $\neg a$ occur in it. For instance, states $p^{\Delta}\ \neg p$ and $p\ q\ \neg p$ are inconsistent. Also both q and $\neg q$ are unassigned by state $p^{\Delta}\ \neg p$, whereas both of them are assigned by $p\ q\ \neg p$.

If C is a disjunction (conjunction) of literals then by $\overline{C}$ we understand the conjunction (disjunction) of the complements of the literals occurring in C. In some situations, we will identify disjunctions and conjunctions of literals with the sets of these literals.

Each CNF formula F determines its *DPLL graph* DP_F. The set of nodes of DP_F consists of the states relative to the signature of F. The edges of the graph DP_F are specified by four transition rules:

$$\mathit{UnitPropagate}:\ M\ \Rightarrow\ M\ l \qquad \text{if} \begin{cases} C \in F,\ l \in C, \text{ and for every} \\ \text{literal } l' \in C \text{ so that } l' \neq l, \\ \overline{l'} \in M \end{cases}$$

$$\mathit{Decide}:\qquad M\ \Rightarrow\ M\ l^{\Delta} \qquad \text{if } l \text{ is unassigned by } M$$

$$\mathit{Fail}:\qquad M\ \Rightarrow\ \mathit{FailState} \text{ if} \begin{cases} M \text{ is inconsistent, and} \\ M \text{ contains no decision literals} \end{cases}$$

$$\mathit{Backtrack}:\qquad P\ l^{\Delta}\ Q \Rightarrow\ P\ \overline{l} \text{ if} \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals.} \end{cases}$$

A node (state) in the graph is *terminal* if no edge originates in it.

The following proposition gathers key properties of the graph DP_F.

Proposition 1 *For any CNF formula F,*

(a) *graph* DP_F *is finite and acyclic,*
(b) *any terminal state of* DP_F *other than FailState is a model of F,*
(c) *FailState is reachable from $\emptyset$ in* DP_F *if and only if F is unsatisfiable.*

Thus, to decide the satisfiability of a CNF formula F it is enough to find a path leading from node $\emptyset$ to a terminal node M. If $M = \mathit{FailState}$, F is unsatisfiable. Otherwise, F is satisfiable and M is a model of F.

For instance, let $F_1 = \{p \vee q, \neg p \vee r\}$. Below we show a path in DP_{F_1} with every edge annotated by the name of the transition rule that gives rise to this edge in the graph:

$$\emptyset \overset{\mathit{Decide}}{\Rightarrow} p^{\Delta} \overset{\mathit{UnitPropagate}}{\Rightarrow} p^{\Delta}\, r \overset{\mathit{Decide}}{\Rightarrow} p^{\Delta}\, r\, q^{\Delta}. \tag{32}$$

The state $p^{\Delta}\ r\ q^{\Delta}$ is terminal. Thus, Proposition 1(b) asserts that F_1 is satisfiable and $\{p, r, q\}$ is a model of F_1. Another path in DP_{F_1} that leads us to concluding that set $\{p, r, q\}$ is a model of F_1 follows

$$\emptyset \overset{\mathit{Decide}}{\Rightarrow} p^{\Delta} \overset{\mathit{Decide}}{\Rightarrow} p^{\Delta}\, r^{\Delta} \overset{\mathit{Decide}}{\Rightarrow} p^{\Delta}\, r^{\Delta}\, q^{\Delta}. \tag{33}$$

We can view a path in the graph DP_F as a description of a process of search for a model of a formula F by applying transition rules of the graph. Therefore, we can characterize an algorithm of a SAT solver that utilizes the inference rules of DP_F by describing a strategy for choosing a path in DP_F. A strategy can be based, in particular, on assigning priorities to some or all transition rules of DP_F, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state. The DPLL algorithm can be captured by the following priorities:

$$\mathit{Backtrack}, \mathit{Fail} >> \mathit{Unit\ Propagate} >> \mathit{Decide}.$$

Note how path (32) in the graph DP_{F_1} respects priorities above, while path (33) does not. Thus DPLL will never explore the latter search trajectory given input F_1.

Answer set solving We are now ready to present an extension to the DPLL algorithm that captures a family of backtrack search procedures for finding answer sets of a propositional logic program.

For a program Π, we call an interpretation M over σ_Π a *classical model* of Π if it is a model of the set of rules in Π seen as clauses. For example, program (6) has three classical models $\{p, \neg q, \neg r\}$, $\{p, \neg q, r\}$, and $\{p, q, r\}$.

By $\mathit{Bodies}(\Pi, a)$ we denote the set of the bodies of all rules of program Π with the head a (including the empty body). A set U of atoms occurring in a propositional program Π is *unfounded* on a consistent set M of literals with respect to Π if for every $a \in U$ and every $B \in \mathit{Bodies}(\Pi, a)$, $M \cap \overline{B} \neq \emptyset$ or $U \cap B^{pos} \neq \emptyset$, where B^{pos} denotes positive part of the body B. For instance, set $\{r\}$ is unfounded on set $\{p, \neg q, r\}$ with respect to program (6), while set $\{q\}$ is unfounded on $\{p, q, r\}$ with respect to program (6).

We are now ready to restate the result from [39, Theorem 4.6] that relates the notions of unfounded set and stable models.[10] This result is crucial for understanding key inference rules used in propagators of modern answer set solvers.

[10] Here we present the result from [39] applicable to programs with doubly negated atoms. Lee [37] illustrated the soundness of this generalization.

Proposition 2 *For a program* Π *and an interpretation* M *over* σ_Π*,* M *is a stable model of* Π *if and only if* M *is a classical model of* Π *and no non-empty subset of* M^+ *is an unfounded set on* M *with respect to* Π*.*

This proposition gives an alternative characterization of stable models. For example, it asserts that (i) classical models of program (6) are the only interpretations that may be stable models of (6) and (ii) classical models $\{p, \neg q, r\}$ and $\{p, q, r\}$ are not stable models of the program due to unfounded sets $\{r\}$ and $\{q\}$ respectively, while classical model $\{p, \neg q, \neg r\}$ is a stable model (since $\{p\}$ is not an unfounded set on $\{p, \neg q, \neg r\}$ with respect to program (6)).

By $f : \Pi \to F$ we denote a function from a propositional logic program Π to a CNF formula F. We say that function f *approximates* program Π when

- for any stable model N of Π there is a model M of $f(\Pi)$ such that $N = M_{|\sigma_\Pi}$,
- any model M of $f(\Pi)$ is such that $M_{|\sigma_\Pi}$ is a classical model of Π.

The following proposition follows immediately from Proposition 2 and the definition of an approximating function.

Proposition 3 *For a program* Π*, a function* f *that approximates* Π*, and an interpretation* N *over* σ_Π*,* N *is a stable model of* Π *if and only if (i) there is a model* M *of* $f(\Pi)$ *such that* $N = M_{|\sigma_\Pi}$ *and no non-empty subset of* N^+ *is an unfounded set on* N *with respect to* Π*.*

We define the transition graph $\text{ASET}_{f,\Pi}$ for a program Π and a function f that approximates Π as follows. The set of nodes of the graph $\text{ASET}_{f,\Pi}$ consists of the states relative to atoms occurring in $f(\Pi)$ and Π. There are five transition rules that characterize the edges of $\text{ASET}_{f,\Pi}$. The transition rules *Unit Propagate* , *Decide* , *Fail* , *Backtrack* of the graph $\text{DP}_{f(\Pi)}$, and the transition rule

$$\textit{Unfounded}: \ M \ \Rightarrow \ M \ \neg a \ \text{ if } \begin{cases} a \in U \text{ for a set } U \text{ unfounded on } M \\ \text{with respect to } \Pi. \end{cases}$$

The graph $\text{ASET}_{f,\Pi}$ can be used for deciding whether a logic program has answer sets:

Proposition 4 *For any program* Π *and a function* f *that approximates* Π*,*

(a) *graph* $\text{ASET}_{f,\Pi}$ *is finite and acyclic,*
(b) *for any terminal state* M *of* $\text{ASET}_{f,\Pi}$ *other than FailState,* $M^+_{|\sigma_\Pi}$ *is an answer set of* Π*,*
(c) *FailState is reachable from* $\emptyset$ *in* $\text{ASET}_{f,\Pi}$ *if and only if* Π *has no answer sets.*

Obviously, any program approximates itself (recall that we identify rules with clauses). By f_Π we denote an identity function $f_\Pi(\Pi) = \Pi$. Extending graph $\text{ASET}_{f_\Pi,\Pi}$ with transition rules

$$\textit{AllRulesCancelled}: M \ \Rightarrow \ M \ \neg a \ \text{ if } \ \overline{B} \cap M \neq \emptyset \text{ for all } B \in \textit{Bodies}(\Pi, a),$$

$$\textit{BackchainTrue}: M \ \Rightarrow \ M \ l \ \text{ if } \begin{cases} a \leftarrow B \in \Pi, \\ a \in M, \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \textit{Bodies}(\Pi, a) \setminus \{B\}, \\ l \in B \end{cases}$$

results in a graph that captures the computation procedure of answer set solver SMODELS. We denote the resulting graph by SM_Π. Similar statement to Proposition 4 holds for the

graph SM_Π [40]. The system SMODELS assigns priorities to the inference rules of SM_Π as follows:

Backtrack, *Fail* >>
Unit Propagate, *All Rules Cancelled*, *Backchain True* >>
Unfounded >>
Decide.

We say that an edge $M \Rightarrow M'$ in the graph SM_Π is *singular* if

- the only transition rule justifying this edge is *Unfounded* , and
- some edge $M \Rightarrow M''$ can be justified by a transition rule other than *Unfounded* .

For instance, let Π be the program

$$p \leftarrow q$$
$$q \leftarrow r.$$

The edge

$$p^\Delta\ q^\Delta\ \neg r^\Delta \quad \Rightarrow\ (\textit{Unfounded},\ U = \{p, q\})$$
$$p^\Delta\ q^\Delta\ \neg r^\Delta\ \neg p$$

in the graph SM_Π is singular, because the edge

$$p^\Delta\ q^\Delta\ \neg r^\Delta \quad \Rightarrow\ (\textit{All Rules Cancelled})$$
$$p^\Delta\ q^\Delta\ \neg r^\Delta\ \neg q$$

belongs to SM_Π also.

From the point of view of actual execution of the SMODELS algorithm, singular edges of the graph SM_Π are inessential: SMODELS never follows a singular edge. By SM_Π^- we denote the graph obtained from SM_Π by removing all singular edges.

It is easy to see that the completion of program Π can also be seen as a conjunction of clauses in Π and implications

$$a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B \tag{34}$$

for each atom a that occurs in Π. Formulas (34) can be written as

$$\neg a \vee \bigvee_{a \leftarrow B \in \Pi} B \tag{35}$$

for every atom a in Π. $CNF - Comp_\Pi$ is a function that maps program Π into its completion converted to CNF using straightforward equivalent transformations. In other words, $CNF - Comp_\Pi$ consists of clauses of two kinds:

1. the rules of the program seen as clauses, and
2. formulas (35) converted to CNF using the distributivity of disjunction over conjunction.[11]

The following result from [40] illustrates that applying the SMODELS algorithm to a tight program Π essentially amounts to applying DPLL to its clausified completion $CNF - Comp_\Pi$. A similar relationship, in terms of pseudocode representations of SMODELS and DPLL, was also established in [32].

Proposition 5 *For any tight program* Π*, the graph* SM_Π^- *is equal to* $\text{DP}_{CNF-Comp_\Pi}$.

[11] It is essential that repetitions are not removed in the process of clausification. For instance, $CNF - Comp_{p \leftarrow not\ p}$ is the formula $(p \vee p) \wedge (\neg p \vee \neg p)$.

For instance, let Π be the program

$$\begin{array}{l} p \leftarrow q,\ not\ r \\ q. \end{array}$$

This program is tight. Its completion is

$$(p \leftrightarrow q \wedge \neg r) \wedge q \wedge \neg r,$$

and $CNF - Comp_{\Pi}$ is

$$(p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg r) \wedge q \wedge \neg r.$$

Proposition 5 asserts that, for this formula F, SM^{-}_{Π} coincides with DP_{F}.

Proposition 5 conveys that combination of completion of a program and unit propagate capture propagators *All Rules Cancelled* and *Backchain True* unique to the graph SM^{-}_{Π}. In a sense, these propagators are present implicitly in $\mathrm{DP}_{CNF-Comp_{\Pi}}$.

Truszczyński and Lierler [42] generalize Proposition 5 to the case of arbitrary programs:

Proposition 6 *For any program* Π*, the graph* SM^{-}_{Π} *is equal to* $\mathrm{ASET}_{CNF-Comp,\Pi}$.

As mentioned earlier several answer set solvers, including CMODELS and CLASP, start their computation by forming program's completion. It is clear that $CNF - Comp_{\Pi}$ can be exponentially larger than $Comp(\Pi)$. Systems CMODELS and CLASP define an ED-transformation procedure that converts implications (34) into a CNF formula by means of explicit definitions and avoids exponential growth. It is a special case of the Tseitin procedure [77] that efficiently transforms any given propositional formula to CNF form by adding new atoms corresponding to its subformulas. It does not produce a CNF equivalent to the input formula, but it gives us its conservative extension. The ED-transformation procedure adds explicit definitions for the disjunctive terms in (34) whenever they contain more than one atom. In other words, it introduces auxiliary atoms as abbreviations for these disjunctive terms. It then applies equivalent transformation to resulting formula and replaces these disjunctive terms by their corresponding auxiliary atoms. At last, the ED-transformation procedure converts this formula to CNF using straightforward equivalent transformations.

For instance, consider the program

$$\begin{array}{l} p \leftarrow q_1,\ r_1 \\ p \leftarrow q_2,\ r_2 \\ p \leftarrow s. \end{array}$$

For atom p, the implication (34) is

$$p \rightarrow (q_1 \wedge r_1) \vee (q_2 \wedge r_2) \vee s. \tag{36}$$

First, ED-transformation introduces following explicit definitions

$$\begin{array}{l} aux_1 \leftrightarrow q_1 \wedge r_1 \\ aux_2 \leftrightarrow q_2 \wedge r_2 \end{array} \tag{37}$$

Second, ED-transformation turns implication (36) into the formula

$$p \rightarrow aux_1 \vee aux_2 \vee s \tag{38}$$

that contains two auxiliary atoms aux_1, aux_2. Third, ED-transformation clausifies these formulas (37) and (38) as follows:

$$\begin{aligned}
&\neg p \vee aux_1 \vee aux_2 \vee s\\
&\neg aux_1 \vee q_1\\
&\neg aux_1 \vee r_1\\
&\neg q_1 \vee \neg r_1 \vee aux_1\\
&\neg aux_2 \vee q_2\\
&\neg aux_2 \vee r_2\\
&\neg q_2 \vee \neg r_2 \vee aux_2.
\end{aligned}$$

We now define for a program Π, a function $ED - Comp_\Pi$ that maps Π into a CNF formula that is the completion $Comp(\Pi)$ converted to CNF using the ED-transformation: It consists of clauses of two kinds

1. the rules of the program seen as clauses, and
2. formulas (34) converted to CNF using the ED-transformation.

The graph $\text{ASET}_{ED-Comp,\Pi}$ can be used to capture basic ideas behind such answer set solvers as CMODELS and CLASP. These solvers also implement backjumping and learning that are not captured by the graph $\text{ASET}_{ED-Comp,\Pi}$. Yet, Nieuwenhuis et al. [61] and Lierler and Truszczyński [42] demonstrate how transition systems DP and ASET, respectively can be extended to capture these features. Here we focus on basic variants of CMODELS and CLASP based on backtracking. Basic CMODELS is modeled by the graph $\text{ASET}_{ED-Comp,\Pi}$ and the priorities:

$$\textit{Backtrack}, \textit{Fail} >> \textit{Unit Propagate} >> \textit{Decide} >> \textit{Unfounded}.$$

Basic CLASP is captured by $\text{ASET}_{ED-Comp,\Pi}$ and the priorities:

$$\textit{Backtrack}, \textit{Fail} >> \textit{Unit Propagate} >> \textit{Unfounded} >> \textit{Decide}.$$

Proposition 6 asserts that systems SMODELS is characterized by the graph $\text{ASET}_{CNF-Comp,\Pi}$. Furthermore, its priorities coincide with these of CLASP. In this sense, system CLASP is closer than CMODELS to answer set solver SMODELS as CLASP and SMODELS both apply inference rule *Unfounded* eagerly (unlike CMODELS that uses this rule only on states that correspond to interpretations). Despite the similar look of graphs $\text{ASET}_{CNF-Comp,\Pi}$ and $\text{ASET}_{ED-Comp,\Pi}$ (indeed, $ED - Comp$ and $CNF - Comp$ serve a similar purpose), they are substantially different. Anger et al. [3] illustrated theoretically and experimentally that DPLL applied to $ED - Comp_\Pi$ is superior to DPLL applied to $CNF - Comp_\Pi$.

Lierler and Truszczyński [42] provide a comprehensive account of major answer set solvers by means of transition systems. (Proposition 4 follows immediately from their work.) They use transition systems to contrast and compare various solvers in detail and include an account on such techniques as learning. Here we present a glimpse to these results that is sufficient to illustrates key differences between DP-based SAT solvers and ASET-based ASP solvers, in particular, presence of additional propagators in answer set solvers:

- presence of a propagator *Unfounded* is unique to ASP technology. Propagator *Unfounded* is responsible for taming recursive definitions, such as transitive closure, in solving.

- explicit or implicit presence of propagators *All Rules Cancelled* and *Backchain True* in answer set solvers is a mechanism for effective processing of inductive definitions encoded in programs.

9 Effectively propositional logic (or direct first-order extension of SAT) for modeling search problems

Effectively propositional logic [70], also known as Bernays-Schönfinkel class and EPR, provides a generalization of pure propositional satisfiability. An *EPR formula* is the universal closure of a conjunction of clauses. The EPR language can be seen as an alternative to the language of logic programs with variables for developing concise formulations of search problems. For example, recall SAT representation (3). The EPR language allows us to generalize (3) as a set of the EPR clauses as follows. Consider any graph (V, E). Ground clauses consisting of single literals presented in (22) and a collection of ground clauses

$$\neg e(v, w) \qquad (\{v, w\} \not\in E;\ v, w \in V) \tag{39}$$

encode the graph (V, E). We denote the collection of clauses in (22) and (39) by $D^{EPR}_{(V,E)}$. Three ground clauses (23) enumerate three colors of the problem. The following EPR clauses conclude the description of the EPR encoding of *GC* problem:

$$c(V, 1) \vee c(V, 2) \vee c(V, 3) \vee \neg vtx(V) \tag{40}$$
$$\neg c(V, I) \vee \neg c(V, J) \vee \neg color(I) \vee \neg color(J) \vee \neg vtx(V) \vee I \geq J \tag{41}$$
$$\neg c(V, I) \vee \neg c(W, I) \vee \neg vtx(V) \vee \neg vtx(W) \vee \neg color(I) \vee \neg e(V, W). \tag{42}$$
$$\neg color(V) \vee \neg vtx(V) \tag{43}$$
$$\neg e(V, W) \vee vtx(V) \tag{44}$$
$$\neg e(V, W) \vee vtx(W) \tag{45}$$

By Γ_{gc}, we denote the EPR formula composed of clauses (23,40–45). EPR formula Γ_{gc} captures specifications of graph coloring problem *GC* so that given a graph (V, E), Herbrand models of $\Gamma_{gc} \cup D^{EPR}_{(V,E)}$ correspond to 3-colorings of (V, E). The three clauses (40–42) of the EPR encoding Γ_{gc} are the counterparts of groups of clauses in propositional formula (3). Clause (40) says that each vertex is assigned some colors. Clause (41) states that it is impossible that a vertex is assigned two colors. Clause (42) says that it is impossible that any two adjacent vertexes are assigned the same color. The clauses (43–45) in Γ_{gc} intuitively say that (i) a set of object constants representing colors is disjoint from a set of object constants representing vertexes and (ii) edges are only possible between object constants that stand for vertexes. These clauses together with clauses (39) in $D^{EPR}_{(V,E)}$ explicitly encode close world assumption for predicates $color$, vtx and e. Note how logic programming encoding $\Pi_{gc} \cup D_{(V,E)}$ of graph coloring problem incorporated this assumption implicitly.

Software systems called EPR solvers or model builders can be used for finding models of EPR theories. The annual CADE ATP System Competition [72] contains the EPR division track promoting advances in such systems. Navaro-Perez and Voronkov [57, 58] illustrate how EPR theories can be used for modeling search problems in various domains including planning. Many of the EPR systems rely on two computational stages: grounding and solving by means of SAT.

10 Conclusions and related work

In this paper we illustrate how answer set programming can be seen as a convenient declarative programming language for utilizing SAT and SAT-like technology. It can be used to declaratively model problem's specifications and allows for elaboration tolerant solutions that follow the GENERATE, DEFINE, and TEST modeling methodology. We paid attention to essential component of answer set programming tool-set, namely, grounding. We also presented key accounts in design of answer set solvers and draw a parallel to the DPLL procedure of SAT solvers. We believe that this comprehensive account on ASP versus SAT helps to understand the similarities and differences between these research directions and research concerns that fields pose. For instance, since answer set programming provides a programming language to their users, large body of research in the field concerns language features that go beyond logic programs discussed here. Such concerns are nonexistent in SAT.

Logic programs under answer set semantics is not the only formalism that serves a role of declarative programming front-end for SAT-like technology. For instance, East and Truszczyński [14], Ternovska and Mitchell [55, 75], Wittocx, Mariën, and Denecker [79, 80] introduce other formalisms. The aim of the efforts by Ternovska and Mitchell is to devise a declarative constraint programming framework that (i) takes the problem specification of a search problem formulated in classical first-order logic, (ii) performs the grounding and then (iii) uses a SAT solver to solve the ground formula. This framework implements *model-expansion* task introduced in [55]. Efforts by East and Truszczyński were inspired by answer set programming. They introduce a language that syntactically is closer to classical first-order logic than the language of logic programs. They then show how theories in this language can be (i) grounded, (ii) transformed into a propositional formula, and (iii) solved by means of SAT solvers. Systems by Wittocx, Maarten, and Denecker support the language called FO(ID) that extends classical first-order logic with inductive definitions [12]. Truszczyński [76] illustrates that despite syntactic and semantic differences between logic FO(ID) and logic programs under answer set semantics these frameworks are closely related and can be seen as different dialects of the same formalism. One property that all these languages share in common: they implement close world assumption – presumption that what is not currently known to be *true* is *false*. This observation hints that close world assumption is essential in design of logic-based declarative programming formalisms, in particular it allows effective grounding techniques. Also, Gebser et al. [21] propose another declarative approach, where the intended SAT clauses are specified in terms of logic rules with variables. This allows the user to write first-order specifications for intended clauses in a schematic way by exploiting term variables. They define the semantics of such specifications and provide an implementation harnessing ASP grounders to accomplish the grounding step of clauses. As we have seen grounding is an essential component of so far mentioned formalisms. Metodi and Codish [54] argue for another alternative declarative front-end for utilizing SAT technology. They propose to, first, use a general purpose constraint programming language BEE for formulating a problem. Second, these BEE specifications are translated into propositional CNF formulas so that SAT solvers are used to find solutions to a given problem.

The relation between propositional logic programs and propositional formulas in terms of expressiveness has been studied earlier in [35, 45, 69]. For example, Janhunen shows that translations from logic programs under answer set semantics to propositional formulas in classical logic cannot be done modularly. This is further underpinned by a complexity result given in [45]. In contrast, this paper focuses on key features of answer set programming

that go beyond propositional logic programs. It highlights aspects of ASP, which form the basis for this prominent declarative programming paradigm. In particular, we note how the presence of first-order language enriched with numerous constructs enables effective modeling of search problem specifications. Also, an ability to express transitive closure by means of logic programs further enriches this modeling tool. Then, the language of logic programs is supported by two distinct processing tools: one responsible for grounding and another responsible for solving. Last but not least, we highlight the crucial role of a built-in closed world assumption that proved to be crucial not only in concise and intuitive modeling of problems, but also in the development of effective grounding and solving procedures.

Acknowledgments We are grateful for discussions with Remi Brochenin, Marc Denecker, Vladimir Lifschitz, Marco Maratea, Cesare Tinelli, Mirek Truszczyński, and Peter Schüller on the topics related to the ideas presented in this paper. Lecture notes by Lifschitz on *Reducing graph coloring to SAT* available at https://www.cs.utexas.edu/users/vl/teaching/lbai/coloring.pdf inspired the running examples in Sections 2 and 3. We are also thankful to anonymous reviewers for valuable comments.

References

1. Alviano, M., Dodaro, C., Faber, W., Leone, N., & Ricca, F. (2013). WASP: a native ASP solver based on constraint learning. In *Proc. of LPNMR 2013. Vol. 8148 of LNCS* (pp. 54–66). Springer-Verlag.
2. Alviano, M., Dodaro, C., Faber, W., Leone, N., & Ricca, F. (2015). Advances in WASP. In *Proc. of LPNMR 2015. Vol. 9345 of LNCS* (pp. 40–54). Springer-Verlag.
3. Anger, C., Gebser, M., Janhunen, T., & Schaub, T. (2006). What's a head without a body? In *Proceedings of the European conference on artificial intelligence (ECAI'06)* (pp. 769–770).
4. Atserias, A., Fichte, J.K., & Thurley, M. (2011). Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Inteligence Research (JAIR)*, *40*, 353–373.
5. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., & Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, *58*, 117–148.
6. Biere, A., Heule, M., van Maaren, H., & Walsh, T. (2009). *Handbook of satisfiability: volume 185 frontiers in artificial intelligence and applications*. Amsterdam: IOS Press.
7. Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92–103.
8. Calautti, M., Greco, S., Spezzano, F., & Trubitsyna, I. (2015). Checking termination of bottom-up evaluation of logic programs with function symbols. *Theory and Practice of Logic Programming*, *15*, 854–889. http://journals.cambridge.org/article_S1471068414000623.
9. Calimeri, F., Cozza, S., Ianni, G., & Leone, N. (2008). Computable functions in ASP: theory and implementation. In *Proceedings of international conference on logic programming (ICLP)* (pp. 407–424).
10. Clark, K. (1978). Negation as failure, In Gallaire, H., & Minker, J. (Eds.) *Logic and data bases* (pp. 293–322). New York: Plenum Press.
11. Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem proving. *Communications of the ACM*, *5*(7), 394–397.
12. Denecker, M. (2000). Extending classical logic with inductive definitions. In *LNAI* (pp. 703–717). Springer-Verlag.
13. Denecker, M., Lierler, Y., Truszczyński, M., & Vennekens, J. (2012). A tarskian informal semantics for answer set programming. In *Technical communications of the 28th international conference on logic programming (ICLP)* (pp. 277–289).
14. East, D., & Truszczyński, M. (2006). Predicate-calculus-based logics for modeling and solving search problems. *ACM Transactions on Computing Logic*, *7*(1), 38–83.
15. Een, N., & Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *SAT*.

16. Eiter, T., & Gottlob, G. (1993). Complexity results for disjunctive logic programming and application to nonmonotonic logics, In Miller, D. (Ed.) *Proceedings of international logic programming symposium (ILPS)* (pp. 266–278).
17. Fages, F. (1991). A fixpoint semantics for general logic programs compared with the well-supported and stable model semantics. *New Generation Computing*, *9*, 425–443.
18. Ferraris, P., & Lifschitz, V. (2005). Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, *5*, 45–74.
19. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., & Veber, P. (2010). Repair and prediction (under inconsistency) in large biological networks with answer set programming, In Lin, F., Sattler, U., & Truszczynski, M. (Eds.) *Proceedings of the 12th international conference on principles of knowledge representation and reasoning KR 2010*: AAAI Press.
20. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., & Schaub, T. (2015). Abstract gringo, (Vol. 15. http://journals.cambridge.org/article_S1471068415000150.
21. Gebser, M., Janhunen, T., Kaminski, R., Schaub, T., & Tasharrofi, S. (2015). Writing declarative specifications for clauses. In *Proceedings of the 3rd workshop on grounding, transforming, and modularizing theories with variables*.
22. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., & Thiele, S. (2015). Potassco user guide., available at https://sourceforge.net/projects/potassco/files/guide/2.0/guide-2.0.pdf.
23. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2010). A user's guide to gringo, clasp, clingo, and iclingo., available at http://potassco.sourceforge.net.
24. Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2009). *On the implementation of weight constraint rules in conflict-driven ASP solvers*, (pp. 250–264). Berlin, Heidelberg: Springer Berlin Heidelberg.
25. Gebser, M., Kaminski, R., & Schaub, T. (2011). aspcud: a linux package configuration tool based on answer set programming. In *Second international workshop on logics for component configuration (LoCoCo)*.
26. Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007). Conflict-driven answer set solving. In *Proceedings of 20th international joint conference on artificial intelligence (IJCAI'07)* (pp. 386–392). MIT Press.
27. Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: from theory to practice. *Artificial Intelligence*, *187*, 52–89.
28. Gebser, M., König, A., Schaub, T., Thiele, S., & Veber, P. (2010). The bioASP library: ASP solutions for systems biology. In *22nd IEEE International conference on tools with artificial intelligence, ICTAI 2010* (pp. 383–389). IEEE Computer Society.
29. Gebser, M., Schaub, T., & Thiele, S. (2007). Gringo: a new grounder for answer set programming. In *Proceedings of the ninth international conference on logic programming and nonmonotonic reasoning* (pp. 266–271).
30. Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming, In Kowalski, R., & Bowen, K. (Eds.) *Proceedings of international logic programming conference and symposium* (pp. 1070–1080): MIT Press.
31. Giunchiglia, E., Lierler, Y., & Maratea, M. (2006). Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, *36*, 345–377.
32. Giunchiglia, E., & Maratea, M. (2005). On the relation between answer set and SAT procedures (or, between smodels and cmodels). In *Proceedings of 21st international conference on logic programming (ICLP)* (pp. 37–51). Springer-Verlag.
33. Gomes, C.P., Kautz, H., Sabharwal, A., & Selman, B. (2008). Satisfiability solvers, In van Harmelen, F., Lifschitz, V., & Porter, B. (Eds.) *Handbook of knowledge representation* (pp. 89–134): Elsevier.
34. Janhunen, T. (2004). Representing normal programs with clauses. In *Proceedings of 16th European conference on artificial intelligence, ECAI 2004* (pp. 358–362). IOS Press.
35. Janhunen, T. (2006). Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 35–86.
36. Kautz, H., & Selman, B. (1992). Planning as satisfiability. In *Proceedings of European conference on artificial intelligence (ECAI)* (pp. 359–363).
37. Lee, J. (2005). A model-theoretic counterpart of loop formulas. In *Proceedings of international joint conference on artificial intelligence (IJCAI)* (pp. pp. 503–508). Professional Book Center.
38. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, *7*(3), 499–562.

39. Leone, N., Rullo, P., & Scarcello, F. (1997). Disjunctive stable models: unfounded sets, fixpoint semantics, and computation. *Information and Computation*, *135*(2), 69–112.
40. Lierler, Y. (2011). Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming*, *11*, 135–169.
41. Lierler, Y., & Lifschitz, V. (2013). Logic programs vs. first-order formulas in textual inference. In *10th International conference on computational semantics (IWCS)*.
42. Lierler, Y., & Truszczyński, M. (2011). Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP) Special Issue*, *11*(4–5), 629–646.
43. Lifschitz, V. (1996). Foundations of logic programming, In Brewka, G. (Ed.) *Principles of knowledge representation* (pp. 69–128): CSLI Publications.
44. Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, *138*, 39–54.
45. Lifschitz, V., & Razborov, A. (2006). Why are there so many loop formulas? *ACM Transactions on Computational Logic*, *7*, 261–268.
46. Lifschitz, V., Tang, L.R., & Turner, H. (1999). Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, *25*, 369–389.
47. Lin, F., & Zhao, Y. (2002). ASSAT: computing answer sets of a logic program by SAT solvers. In *Proceedings of national conference on artificial intelligence (AAAI)* (pp. 112–117). MIT Press.
48. Lin, F., & Zhao, Y. (2004). ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, *157*, 115–137.
49. Marek, V., & Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In *The logic programming paradigm: a 25-Year perspective* (pp. 375–398). Springer Verlag.
50. Marques-Silva, J., & Sakallah, K.A. (May 1999). GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*(5), 506–521.
51. Marques Silva, J.P., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability* (pp. 131–153). IOS Press.
52. McCarthy, J. (1988). Mathematical logic in artificial intelligence. Dædalus, 297–311. Reproduced in [53].
53. McCarthy, J. (1990). Formalizing Common Sense: Papers by John McCarthy. Ablex, Norwood.
54. Metodi, A., & Codish, M. (2012). Compiling finite domain constraints to sat with bee*. *Theory and Practice of Logic Programming*, *12*(4–5), 465–483. doi:10.1017/S1471068412000130.
55. Mitchell, D.G., & Ternovska, E. (2005). A framework for representing and solving np search problems. In *Twentieth national conference on artificial intelligence and the seventeenth innovative applications of artificial intelligence conference (AAAI)* (pp. 430–435). AAAI Press / The MIT Press.
56. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: engineering an efficient SAT solver. In *Proceedings DAC-01*.
57. Navarro-Pérez, J. (2007). Encoding and solving problems in effectively propositional logic. Ph.D. thesis, University of Manchester.
58. Navarro-Pérez, J., & Voronkov, A. (2013). Planning with effectively propositional logic, In Voronkov, A., & Weidenbach, C. (Eds.) *Programming logics. Vol. 7797 of lecture notes in computer science* (pp. 302–316). Berlin Heidelberg: Springer. doi:10.1007/978-3-642-37651-1_13.
59. Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, *25*, 241–273.
60. Niemelä, I., & Simons, P. (2000). Extending the Smodels system with cardinality and weight constraints, In Minker, J. (Ed.) *Logic-based artificial intelligence* (pp. 491–521). Kluwer.
61. Nieuwenhuis, R., Oliveras, A., & Tinelli, C. (2006). Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, *53*(6), 937–977.
62. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., & Barry, M. (2001). An A-Prolog decision support system for the Space Shuttle. In *Proceedings of international symposium on practical aspects of declarative languages (PADL)* (pp. 169–183).
63. Pipatsrisawat, K., & Darwiche, A. (2011). On the power of clause-learning sat solvers as resolution engines, (Vol. 175 pp. 512–525). doi:10.1016/j.artint.2010.10.002.
64. Prasad, M.R., Biere, A., & Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *STTT*, *7*(2), 156–173.
65. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., & Leone, N. (2012). Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming*, *12*(3), 361–381.
66. Rintanen, J. (2012). Planning as satisfiability: heuristics. *Artificial Intelligence*, *193*, 45–86.

67. Rossi, F., Beek, P.v., & Walsh, T. (2006). *Handbook of constraint programming (foundations of artificial intelligence)*. New York: Elsevier Science Inc.
68. Roussel, O., & Manquinho, V.M. (2009). Pseudo-boolean and cardinality constraints. In *Handbook of satisfiability* (pp. 695–733). IOS Press.
69. Schlipf, J. (1995). The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, *51*(1), 64–86. http://www.sciencedirect.com/science/article/pii/S0022000085710537.
70. Schulz, S. (2002). A comparison of different techniques for grounding near-propositional CNF formulae. In *Proceedings of the 15th international FLAIRS conference* (pp. 72–76).
71. Simons, P., Niemelä, I., & Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, *138*, 181–234.
72. Sutcliffe, G. (2013). The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Communications*, *26*(2), 211–223.
73. Syrjänen, T. (2001). Omega-restricted logic programs. In *Proceedings of international conference on logic programming and nonmonotonic reasoning* (pp. 267–279).
74. Syrjanen, T. (2003). Lparse manual, available at http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.
75. Ternovska, E., & Mitchell, D.G. (2009). Declarative programming of search problems with built-in arithmetic. In *21st International joint conference on artificial intelligence (IJCAI)* (pp. 942–947).
76. Truszczyński, M. (2012). Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct reasoning: essays on logic-based AI in honor of Vladimir Lifschitz*. Springer-Verlag.
77. Tseitin, G. (1968). On the complexity of derivation in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic Part II*.
78. Ward, J., & Schlipf, J. (2004). Answer set programming with clause learning. In *Proceedings of international conference on logic programming and nonmonotonic reasoning (LPNMR'04)* (pp. 302–313).
79. Wittocx, J., Mariën, M., & Denecker, M. (2008). GidL: a grounder for FO+, In Thielscher, M., & Pagnucco, M. (Eds.) *Proceedings of the twelfth international workshop on non-monotonic reasoning, international workshop on non-monotonic reasoning, Sydney, 13-15 September 2008* (pp. 189–198). https://lirias.kuleuven.be/handle/123456789/197022.
80. Wittocx, J., Mariën, M., & Denecker, M. (2008). The IDP system: a model expansion system for an extension of classical logic. In: *Proceedings of workshop on logic and search, computation of structures from declarative descriptions (LaSh). electronic* (pp. 153–165), available at https://lirias.kuleuven.be/bitstream/123456789/229814/1/lash08.pdf.
81. Zhang, L., Madigan, C.F., Moskewicz, M.W., & Malik, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings ICCAD-01* (pp. 279–285).