Transition Systems for Model Generators — A Unifying Approach

YULIYA LIERLER and MIROSLAW TRUSZCZYNSKI

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0633, USA
(*e-mail:* yuliya,mirek@cs.uky.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

A fundamental task for propositional logic is to compute models of propositional formulas. Programs developed for this task are called satisfiability solvers. We show that transition systems introduced by Nieuwenhuis, Oliveras, and Tinelli to model and analyze satisfiability solvers can be adapted for solvers developed for two other propositional formalisms: logic programming under the answer-set semantics, and the logic PC(ID). We show that in each case the task of computing models can be seen as "satisfiability modulo answer-set programming," where the goal is to find a model of a theory that also is an answer set of a certain program. The unifying perspective we develop shows, in particular, that solvers CLASP and MINISAT(ID) are closely related despite being developed for different formalisms, one for answer-set programming and the latter for the logic PC(ID).

1 Introduction

A fundamental reasoning task for propositional logic is to compute models of propositional formulas or determine that no models exist. Programs developed for this task are commonly called *model generators* or *satisfiability (SAT) solvers*. In the paper we show that transition systems introduced by Nieuwenhuis et al. (2006) to model and analyze SAT solvers can be adapted for the analysis and comparison of solvers developed for other propositional formalisms. The two formalisms we focus on are logic programming with the answer-set semantics and the logic PC(ID).

Davis-Putnam-Logemann-Loveland (DPLL) procedure is a well-known method that exhaustively explores interpretations to generate models of a propositional formula. Most modern SAT solvers are based on variations of the DPLL procedure. Usually these variations are specified by pseudocode. Nieuwenhuis et al. (2006) proposed an alternative approach based on the notion of a *transition system* that describes "states of computation" and allowed transitions between them. In this way, it defines a directed graph such that every execution of the DPLL procedure corresponds to a path in the graph. This abstract way of presenting DPLL-based algorithms simplifies the analysis of their correctness and facilitates studies of their properties — instead of reasoning about pseudocode constructs, we reason about properties of a graph. For instance, by proving that the graph corresponding to a DPLL-based algorithm is finite and acyclic we show that the algorithm always terminates.

Answer-set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999) is a declarative programming formalism based on the answer-set semantics of logic programs (Gelfond and Lifschitz 1988). Generating answer sets of propositional programs is the

key step in computation with ASP. The logic FO(ID), introduced by Denecker (2000) is another formalism for declarative programming and knowledge representation. As in the case of ASP, most automated reasoning tasks in the logic FO(ID) reduce to reasoning in its propositional core, the logic PC(ID) (Mariën et al. 2008), where generating models is again the key.

In this paper, we show that both computing answer sets of programs and computing models of PC(ID) theories can be considered as testing *satisfiability modulo theories* (SMT), where the objective is to find a model of a set of clauses that is also an answer set of a certain program. We refer to this computational problem as *satisfiability modulo answer-set programming* and denote it by SM(ASP). We identify the propositional formalism capturing SM(ASP) — we use the same term to refer to it — and show that it is a common generalization of ASP and PC(ID). We define a simple transition system for SM(ASP) and show that it can be used as an abstract representation of the solver SMODELS[1] (Niemelä and Simons 2000), an alternative to a similar characterization of SMODELS obtained earlier by Lierler (2011). We then define another more elaborate transition system for SM(ASP) that captures such features of backtracking search as backjumping and learning. We use this transition system to obtain abstract characterizations of the algorithms implemented by the ASP solvers CMODELS[2] (Giunchiglia et al. 2004) and CLASP[3] (Gebser et al. 2007), and the PC(ID) solver MINISAT(ID)[4] (Mariën et al. 2008). Finally, we briefly mention the possibility to regard the introduced transition systems as proof systems. In that setting, transition systems could be used for comparing the solvers they represent in terms of the complexity of the corresponding proof systems.

Our results provide a uniform correctness proof for a broad class of solvers that can be modeled by the transition system for SM(ASP), clarify essential computational principles behind ASP and PC(ID) solvers, and offer insights into how they relate to each other. In particular, our results yield the first abstract representation of CLASP in terms of transition systems (up to now CLASP has been typically specified in pseudocode), and show that at the abstract level, CLASP and MINISAT(ID) are strikingly closely related.

This last point is noteworthy as the two solvers were developed for different propositional formalisms. MINISAT(ID) was developed specifically for the logic PC(ID), where there is no concept of an answer set. The semantics is a natural extension of the notion of a model of a propositional theory to the setting when a theory consists of propositional clauses and *definitions*. Definitions are written as logic programs but they are interpreted by the well-founded semantics and not by the answer-set semantics. There is no indication in the literature that CLASP or MINISAT(ID) were influenced by each other. The two solvers were developed independently and for differently motivated formalisms. It is then of substantial interest that at the level of solving they are closely related.

The appendix to the paper containing proofs of the results is available at `http://arxiv.org/abs/1105.0650` and at the Theory and Practice of Logic Programming website.

[1] `http://www.tcs.hut.fi/Software/smodels/` .
[2] `http://www.cs.utexas.edu/users/tag/cmodels` .
[3] `http://www.cs.uni-potsdam.de/clasp/` .
[4] `http://dtai.cs.kuleuven.be/krr/software/minisatid` .

2 Preliminaries

We now review the abstract transition system framework proposed for the DPLL procedure by Nieuwenhuis et al. (2006), and introduce some necessary terminology concerning logic programs and the logic PC(ID).

Abstract DPLL. Most state-of-the-art SAT solvers are based on variations of the DPLL procedure (Davis et al. 1962). Nieuwenhuis et al. (2006) described DPLL by means of a transition system that can be viewed as an abstract representation of the underlying DPLL computation. In this section we review the abstract DPLL in the form convenient for our purposes, following the presentation proposed by Lierler (2011).

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

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

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 dual, written $\bar{l}$, 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 $b^{\Delta}\neg b$ and $ba\neg b$ are inconsistent.

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

In this paper, a *clause* is a *non-empty* disjunction of literals and a CNF formula is a conjunction (alternatively, a set) of clauses. Each CNF formula F determines its *DPLL graph* DP_F. The set of nodes of DP_F consists of the states relative to the set of atoms occurring in F. The edges of the graph DP_F are specified by four transition rules:

Unit Propagate:	$M \Longrightarrow Ml$	if $C \vee l \in F$ and $\overline{C} \subseteq M$
Decide:	$M \Longrightarrow Ml^{\Delta}$	if l is unassigned by M
Fail:	$M \Longrightarrow \mathit{FailState}$	if $\begin{cases} M \text{ is inconsistent, and} \\ M \text{ contains no decision literals} \end{cases}$
Backtrack:	$Pl^{\Delta}Q \Longrightarrow P\bar{l}$	if $\begin{cases} Pl^{\Delta}Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals.} \end{cases}$

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 $F = \{a \vee b, \neg a \vee c\}$. Below we show a path in DP_F with every edge annotated by the name of the transition rule that gives rise to this edge in the graph:

$$\emptyset \quad \overset{\mathit{Decide}}{\Longrightarrow} \quad a^{\Delta} \quad \overset{\mathit{Unit\ Propagate}}{\Longrightarrow} \quad a^{\Delta}c \quad \overset{\mathit{Decide}}{\Longrightarrow} \quad a^{\Delta}cb^{\Delta}.$$

The state $a^{\Delta}cb^{\Delta}$ is terminal. Thus, Proposition 1(b) asserts that F is satisfiable and $\{a,c,b\}$ is a model of F.

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

$$a_0 \leftarrow a_1, \ldots, a_l, \mathit{not}\ a_{l+1}, \ldots, \mathit{not}\ a_m, \mathit{not\ not}\ a_{m+1}, \ldots, \mathit{not\ not}\ a_n, \tag{1}$$

where a_0 is an atom or $\bot$ and each a_i, $1 \leq i \leq n$, is an atom.[5] If a_0 is an atom then a rule (1) is *weakly normal*. If, in addition, $n = m$ then it is *normal*. Programs consisting of weakly normal (normal, respectively) rules only are called *weakly normal* (*normal*, respectively). If Π is a program, by $\mathit{At}(\Pi)$ we denote the set of atoms that occur in Π.

The expression a_0 is the *head* of the rule. If $a_0 = \bot$ we say that the head of the rule is *empty* and we often omit $\bot$ from the notation. In such case we require that $n > 0$. We call a rule with the empty head a *constraint*. We write $\mathit{Head}(\Pi)$ for the set of nonempty heads of rules in a program Π.

We call the expression $a_1, \ldots, a_l, \mathit{not}\ a_{l+1}, \ldots, \mathit{not}\ a_m, \mathit{not\ not}\ a_{m+1}, \ldots, \mathit{not\ not}\ a_n$ in a rule (1) the *body* of the rule and often view it as the set of all elements that occur in it. If a is an atom, we set $s(a) = s(\mathit{not\ not}\ a) = a$, and $s(\mathit{not}\ a) = \neg a$, and we define $s(B) = \{s(l) \mid l \in B\}$. More directly,

$$s(B) = \{a_1, \ldots, a_l, \neg a_{l+1}, \ldots, \neg a_m, a_{m+1}, \ldots, a_n\}.$$

We also frequently identify the body B of (1) with the conjunction of elements in $s(B)$:

$$a_1 \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge a_{m+1} \wedge \cdots \wedge a_n.$$

By $\mathit{Bodies}(\Pi, a)$ we denote the set of the bodies of all rules of Π with the head a (including the empty body). If B is the body of (1), we write B^{pos} for the *positive* part of the body, that is, $B^{pos} = \{a_1, \ldots a_l\}$.

We often interpret a rule (1) as a propositional clause

$$a_0 \vee \neg a_1 \vee \ldots \vee \neg a_l \vee a_{l+1} \vee \ldots \vee a_m \vee \neg a_{m+1} \vee \ldots \vee \neg a_n \tag{2}$$

(in the case when the rule is a constraint, a_0 is absent in (2)). Given a program Π, we write Π^{cl} for the set of clauses (2) corresponding to all rules in Π.

This version of the language of logic programs is a special case of programs with nested expressions (Lifschitz et al. 1999). It is essential for our approach as it yields an alternative definition of the logic PC(ID), which facilitates connecting it to ASP. We assume that the reader is familiar with the definition of an answer set of a logic program and refer to the paper by Lifschitz et al. (1999) for details.

[5] In the paper, we do not use the term *literal* for expressions a, *not* a and *not not* a. We reserve the term *literal* exclusively for propositional literals a and $\neg a$.

Well-Founded Semantics and the Logic PC(ID). Let M be a set of (propositional) literals. By $\overline{M}$ we understand the set of the duals of the literals in M. A set U of atoms occurring in a program Π is *unfounded* on a consistent set M of literals with respect to Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $M \cap \overline{s(B)} \neq \emptyset$ or $U \cap B^{pos} \neq \emptyset$. For every program Π and for every consistent set M of literals, the union of sets that are unfounded on M with respect to Π is also unfounded on M with respect to Π. Thus, under the assumptions above, there exists the *greatest unfounded set* on M with respect to Π. We denote this set by $GUS(M, \Pi)$.

For every weakly normal program Π we define an operator W_Π on a set M of literals as follows

$$W_\Pi(M) = \begin{cases} M \cup \{a \mid a \leftarrow B \in \Pi \text{ and } s(B) \subseteq M\} \cup \overline{GUS(M, \Pi)} & \text{if } M \text{ is consistent} \\ At(\Pi) \cup \overline{At(\Pi)} & \text{otherwise.} \end{cases}$$

By $W_\Pi^{fix}(M)$ we denote a fixpoint of the operator W_Π over a set M of literals. One can show that it always exists since W_Π is not only monotone but also increasing (for any set M of literals, $M \subseteq W_\Pi(M)$). The least fixpoint of W_Π, $W_\Pi^{fix}(\emptyset)$, is consistent and yields the *well-founded model* of Π, which in general is three-valued. It is also written as $lfp(W_\Pi)$. These definitions and properties were initially introduced for normal programs only (Van Gelder et al. 1991). They extend to programs in our syntax in a straightforward way, no changes in statements or arguments are needed (Lee 2005).

Let Π be a program and A a set of atoms. An atom a is *open* with respect to Π and A if $a \in A \setminus Head(\Pi)$. We denote the set of atoms that are open with respect to Π and A by O_A^Π. By Π_A we denote the logic program Π extended with the rules $a \leftarrow$ *not not* a for each atom $a \in O_A^\Pi$. For instance, let Π be a program

$$\begin{array}{l} a \leftarrow b, \mathit{not}\ c \\ b. \end{array} \tag{3}$$

Then, $\Pi_{\{c\}}$ is

$$\begin{array}{l} c \leftarrow \mathit{not\ not}\ c \\ a \leftarrow b, \mathit{not}\ c \\ b. \end{array}$$

We are ready to introduce the logic PC(ID) (Denecker 2000). A *PC(ID) theory* is a pair (F, Π), where F is a set of clauses and Π is a weakly normal logic program. For a PC(ID) theory (F, Π), by Π^o we denote $\Pi_{At(F \cup \Pi)}$ and by O^Π we denote $O^\Pi_{At(F \cup \Pi)}$ (where $At(F \cup \Pi)$ stands for the set of atoms that occur in F and Π). Moreover, for a set M of literals and a set A of atoms, by M^A we denote the set of those literals in M whose atoms occur in A. A set M of literals is *complete* over the set At of atoms if every atom in At occurs (possibly negated) in M and no other atoms occur in M.

Definition 1
Let (F, Π) be a PC(ID) theory. A consistent and complete (over $At(F \cup \Pi)$) set M of literals is called a *model* of (F, Π) if

(i) M is a model of F, and
(ii) $M = W_{\Pi^o}^{fix}(M^{O^\Pi})$.

For instance, let F be a clause $b \vee \neg c$ and Π be program (3). The PC(ID) theory (F, Π) has two models $\{b, \neg c, a\}$ and $\{b, c, \neg a\}$. We note that although sets $\{\neg b, \neg c, a\}$ and $\{\neg b, \neg c, \neg a\}$ satisfy the condition (i), that is, are models of F, they do not satisfy the condition (ii) and therefore are not models of (F, Π).

The introduced definition of a PC(ID) theory differs from the original one (Denecker 2000). Specifically, for us the second component of a PC(ID) theory is a weakly normal program rather than a set of normal programs (definitions). Still, the two formalisms are closely related.

Proposition 2
For a PC(ID) theory (F, Π) such that Π is a normal program, M is a model of (F, Π) if and only if M is a model of $(F, \{\Pi\})$ according to the definition by Denecker (2000).

As the restriction to a single program in PC(ID) theories is not essential (Mariën et al. 2008), Proposition 2 shows that our definition of the logic PC(ID) can be regarded as a slight generalization of the original one (more general programs can appear as definitions in PC(ID) theories).

3 Satisfiability Modulo ASP: a unifying framework for ASP and PC(ID) solvers

For a theory T the *satisfiability modulo theory* (SMT) problem is: given a formula F, determine whether F is T-satisfiable, that is, whether there exists a model of F that is also a model of T. We refer the reader to the paper by Nieuwenhuis et al. (2006) for an introduction to SMT. Typically, a theory T that defines a specific SMT problem is a first-order formula. The SMT problem that we consider here is different. The theory T is a logic program under the (slightly modified) answer-set semantics. We show that the resulting version of the SMT problem can be regarded as a joint extension of ASP and PC(ID).

We start by describing the modification of the answer-set semantics that we have in mind.

Definition 2
Given a logic program Π, a set X of atoms is an *input answer set* of Π if X is an answer set of $\Pi \cup (X \setminus Head(\Pi))$.

Informally, the atoms of X that cannot possibly be defined by Π as they do not belong to $Head(\Pi)$ serve as "input" to Π. A set X is an input answer set of Π if it is an answer set of the program Π extended with these "input" atoms from X. Input answer sets are related to stable models of a propositional logic program module (Oikarinen and Janhunen 2006).

For instance, let us consider program (3). Then, sets $\{b, c\}$, $\{a, b\}$ are input answer sets of the program whereas set $\{a, b, c\}$ is not.

There are two important cases when input answer sets of a program are closely related to answer sets of the program.

Proposition 3
For a logic program Π and a set X of atoms:

(a) $X \subseteq Head(\Pi)$ and X is an input answer set of Π if and only if X is an answer set of Π.

(b) If $(X \setminus Head(\Pi)) \cap At(\Pi) = \emptyset$, then X is an input answer set of Π if and only if $X \cap Head(\Pi)$ is an answer set of Π.

We now introduce a propositional formalism that we call *satisfiability modulo ASP* and denote by SM(ASP). Later in the paper we show that SM(ASP) can be viewed as a common generalization of both ASP and PC(ID). *Theories* of SM(ASP) are pairs $[F, \Pi]$, where F is a set of clauses and Π is a program. In the definition below and in the remainder of the paper, for a set M of literals we write M^+ to denote the set of atoms (non-negated literals) in M. For instance, $\{a, \neg b\}^+ = \{a\}$.

Definition 3
For an SM(ASP) theory $[F, \Pi]$, a consistent and complete (over $At(F \cup \Pi)$) set M of literals is a *model* of $[F, \Pi]$ if M is a model of F and M^+ is an input answer set of Π.

For instance, let F be a clause $b \vee \neg c$ and Π be program (3). The SM(ASP) theory $[F, \Pi]$ has two models $\{b, \neg c, a\}$ and $\{b, c, \neg a\}$.

The problem of finding models of pairs $[F, \Pi]$ can be regarded as an SMT problem in which, given a formula F and a program Π, the goal is to find a model of F that is (its representation by the set of its true atoms, to be precise) an input answer set of Π. This observation motivated our choice of the name for the formalism.

As for PC(ID) theories, also for an SM(ASP) theory $[F, \Pi]$ we write Π^o for the program $\Pi_{At(\Pi \cup F)}$. We have the following simple observation.

Proposition 4
A set M of literals is a model of an SM(ASP) theory $[F, \Pi]$ if and only if M is a model of an SM(ASP) theory $[F, \Pi^o]$.

It is evident that a set M of literals is a model of F if and only if M is a model of $[F, \emptyset]$. Thus, SM(ASP) allows us to express the propositional satisfiability problem. We now show that the SM(ASP) formalism captures ASP. Let Π be a program. We say that a set F of clauses is Π-*safe* if

1. $F \models \neg a$, for every $a \in O^{\Pi}_{At(\Pi)}$, and
2. for every answer set X of Π there is a model M of F such that $X = M^+ \cap Head(\Pi)$.

Proposition 5
Let Π be a program. For every Π-safe set F of clauses, a set X of atoms is an answer set of Π if and only if $X = M^+ \cap At(\Pi)$, for some model M of $[F, \Pi]$.

This result shows that for an appropriately chosen theory F, answer sets of a program Π can be derived in a direct way from models of an SM(ASP) theory $[F, \Pi]$. There are several possible choices for F that satisfy the requirement of Π-safety. One of them is the Clark's *completion* of Π (Clark 1978). We recall that the completion of a program Π consists of clauses in Π^{cl} and of the formulas that can be written as

$$\neg a \vee \bigvee_{B \in Bodies(\Pi, a)} B \tag{4}$$

for every atom a in Π that is not a fact (that is, the set $Bodies(\Pi, a)$ contains no empty body). Formulas (4) can be clausified in a straightforward way by applying distributivity.

The set of all the resulting clauses and of those in Π^{cl} forms the *clausified* completion of Π, which we will denote by $Comp(\Pi)$.

The theory $Comp(\Pi)$ does not involve any new atoms but it can be exponentially larger than the completion formula before clausification. We can avoid the exponential blow-up by introducing new atoms. Namely, for each body B of a rule in Π with $|B| > 1$, we introduce a fresh atom f_B. If $|B| = 1$, then we define $f_B = s(l)$, where l is the only element of B. By $ED\text{-}Comp(\Pi)$, we denote the set of the following clauses:

1. all clauses in Π^{cl}
2. all clauses $\neg a \vee \bigvee_{B \in Bodies(\Pi,a)} f_B$, for every $a \in At(\Pi)$ such that a is not a fact in Π and $|Bodies(\Pi,a)| > 1$
3. all clauses $\neg a \vee s(l)$, where $a \in At(\Pi)$, $Bodies(\Pi,a) = \{B\}$ and $l \in B$,
4. all clauses $\neg a$, where $|Bodies(\Pi,a)| = 0$
5. all clauses obtained by clausifying in the obvious way formulas $f_B \leftrightarrow B$, where $B \in Bodies(\Pi,a)$, for some atom a that is not a fact in Π and $|Bodies(\Pi,a)| > 1$.

Clearly, the restrictions of models of the theory $ED\text{-}Comp(\Pi)$ to the original set of atoms are precisely the models of $Comp(\Pi)$ (and of the completion of Π). However, the size of $ED\text{-}Comp(\Pi)$ is linear in the size of Π. The theory $ED\text{-}Comp(\Pi)$ has long been used in answer-set computation. Answer set solvers such as CMODELS (Giunchiglia et al. 2004) and CLASP (Gebser et al. 2007) start their computation by transforming the given program Π into $ED\text{-}Comp(\Pi)$.

For instance, let Π be program (3). The completion of Π is the formula

$$(a \vee \neg b \vee c) \wedge b \wedge \neg c \wedge (\neg a \vee (b \wedge \neg c)),$$

its clausified completion $Comp(\Pi)$ is the formula

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg c) \wedge b \wedge \neg c,$$

and, finally, $ED\text{-}Comp(\Pi)$ is the formula

$$(a \vee \neg b \vee c) \wedge (\neg a \vee f_{b \wedge \neg c}) \wedge (f_{b \wedge \neg c} \vee \neg b \vee c) \wedge$$
$$(\neg f_{b \wedge \neg c} \vee b) \wedge (\neg f_{b \wedge \neg c} \vee \neg c) \wedge b \wedge \neg c.$$

We now have the following corollary from Proposition 5.

Corollary 1
For a logic program Π and a set X of atoms, the following conditions are equivalent:

(a) X is an answer set of Π,
(b) $X = M^+$ for some model M of the SM(ASP) theory $[\{\neg a \mid a \in O^{\Pi}_{At(\Pi)}\}, \Pi]$,
(c) $X = M^+$ for some model M of the SM(ASP) theory $[Comp(\Pi), \Pi]$,
(d) $X = M^+ \cap At(\Pi)$ for some model M of the SM(ASP) theory $[ED\text{-}Comp(\Pi), \Pi]$.

It is in this sense that ASP can be regarded as a fragment of SM(ASP). Answer sets of a program Π can be described in terms of models of SM(ASP) theories. Moreover, answer-set computation can be reduced in a straightforward way to the task of computing models of SM(ASP) theories.

Remark 1
Corollary 1 specifies three ways to describe answer sets of a program in terms of models of SM(ASP) theories. This offers an interesting view into answer-set generation. The CNF formulas appearing in the SM(ASP) theories in the conditions (b) - (d) make explicit some of the "propositional satisfiability inferences" that may be used when computing answer sets. The condition (b) shows that when computing answer sets of a program, atoms not occurring as heads can be inferred as false. The theory in (c) makes it clear that a much broader class of inferences can be used, namely those that are based on the clauses of the completion. The theory in (d) describes still additional inferences, as now, thanks to new atoms, we can explicitly infer whether bodies of rules must evaluate to true or false. In each case, some inferences needed for generating answer sets are still not captured by the respective CNF theory and require a reference to the program Π. We note that it is possible to express these "answer-set specific" inferences in terms of clauses corresponding to loop formulas (Lin and Zhao 2004; Lee 2005). We do not consider this possibility in this paper.

Next, we show that SM(ASP) encompasses the logic PC(ID). The well-founded model M of a program Π is *total* if it assigns all atoms occurring in Π. For a PC(ID) theory (F,Π), a program Π is *total on a model* M of F if $W^{fix}_{\Pi^o}(M^{O^\Pi})$ is total. A program Π is *total* if Π is total on every model M of F. The PC(ID) theories (F,Π) where Π is total form an important class of *total* PC(ID) theories.

There is a tight relation between models of a total PC(ID) theory (F,Π) and models of an SM(ASP) theory $[F,\Pi]$.

Proposition 6
For a total PC(ID) theory (F,Π) and a set M of literals over the set $At(F\cup\Pi)$ of atoms, the following conditions are equivalent:

(a) M is a model of (F,Π),
(b) M is a model of the SM(ASP) theory $[F,\Pi]$,
(c) M is a model of the SM(ASP) theory $[Comp(\Pi_{At(\Pi)})\cup F,\Pi]$,
(d) for some model M' of the SM(ASP) theory $[ED\text{-}Comp(\Pi_{At(\Pi)})\cup F,\Pi]$, $M = M'\cap At(F\cap\Pi)$.

The conditions (b), (c), (d) state that the logic PC(ID) restricted to total theories can be regarded as a fragment of the SM(ASP) formalism. The comments made in Remark 1 pertain also to generation of models in the logic PC(ID).

We now characterize models of SM(ASP) theories, and computations that lead to them, in terms of transition systems. Later we discuss implications this characterization has for ASP and PC(ID) solvers.

We define the transition graph $\mathrm{SM(ASP)}_{F,\Pi}$ for an SM(ASP) theory $[F,\Pi]$ as follows. The set of nodes of the graph $\mathrm{SM(ASP)}_{F,\Pi}$ consists of the states relative to $At(F\cup\Pi)$. There are five transition rules that characterize the edges of $\mathrm{SM(ASP)}_{F,\Pi}$. The transition rules *Unit Propagate*, *Decide*, *Fail*, *Backtrack* of the graph $\mathrm{DP}_{F\cup\Pi^{cl}}$, and the transition rule

$$\textit{Unfounded}: \quad M \implies M\neg a \ \text{ if } a\in U \text{ for a set } U \text{ unfounded on } M \text{ w.r.t. } \Pi^o.$$

The graph $\mathrm{SM(ASP)}_{F,\Pi}$ can be used for deciding whether an SM(ASP) theory $[F,\Pi]$ has a model.

Proposition 7
For any SM(ASP) theory $[F, \Pi]$,

(a) graph $\text{SM(ASP)}_{F,\Pi}$ is finite and acyclic,
(b) for any terminal state M of $\text{SM(ASP)}_{F,\Pi}$ other than *FailState*, M is a model of $[F, \Pi]$,
(c) *FailState* is reachable from $\emptyset$ in $\text{SM(ASP)}_{F,\Pi}$ if and only if $[F, \Pi]$ has no models.

Proposition 7 shows that algorithms that correctly find a path in the graph $\text{SM(ASP)}_{F,\Pi}$ from $\emptyset$ to a terminal node can be regarded as SM(ASP) solvers. It also provides a proof of correctness for every SM(ASP) solver that can be shown to work in this way.

One of the ways in which SM(ASP) encompasses ASP (specifically, Corollary 1(c)) is closely related to the way the answer-set solver SMODELS works. We recall that to represent SMODELS Lierler (2011) proposed a graph SM_Π. We note that the rule *Unfounded* above is closely related to the transition rule with the same name used in the definition of SM_Π (Lierler 2011). In fact, if $\Pi = \Pi^o$ then these rules are identical.

Lierler (2011) observed that SMODELS as it is implemented never follows certain edges in the graph SM_Π, and called such edges *singular*. Lierler (2011) denoted by SM^-_Π the graph obtained by removing from SM_Π all its singular edges and showed that SM^-_Π is still sufficient to serve as an abstract model of a class of ASP solvers including SMODELS. The concept of a singular edge extends literally to the case of the graph $\text{SM(ASP)}_{F,\Pi}$. An edge $M \Longrightarrow M'$ in the graph $\text{SM(ASP)}_{F,\Pi}$ is *singular* if:

1. the only transition rule justifying this edge is *Unfounded*, and
2. some edge $M \Longrightarrow M''$ can be justified by a transition rule other than *Unfounded* or *Decide*.

We define $\text{SM(ASP)}^-_{F,\Pi}$ as the graph obtained by removing all singular edges from $\text{SM(ASP)}_{F,\Pi}$. Proposition 8 below can be seen as an extension of Proposition 4 given by Lierler (2011) to non-tight programs.

Proposition 8
For every program Π, the graphs SM^-_Π and $\text{SM(ASP)}^-_{Comp(\Pi),\Pi}$ are equal.

It follows that the graph $\text{SM(ASP)}^-_{Comp(\Pi),\Pi}$ provides an abstract model of SMODELS. We recall though that $Comp(\Pi)$ can be exponentially larger than the completion formula before clausification. Using ASP specific propagation rules such as *Backchain True* and *All Rules Cancelled* (Lierler 2011) allows SMODELS to avoid explicit representation of the clausified completion and infer all the necessary transitions directly on the basis of the program Π.

A similar relationship, in terms of pseudocode representations of SMODELS and DPLL, is established in the paper by Giunchiglia and Maratea (2005) for tight programs.

The answer-set solvers CMODELS, CLASP and the PC(ID) solver MINISAT(ID) cannot be described in terms of the graph SM(ASP) nor its subgraphs. These solvers implement such advanced features of SAT and SMT solvers as learning (forgetting), backjumping and restarts (Nieuwenhuis et al. (2006) give a good overview of these techniques). In the next section we *extend* the graph $\text{SM(ASP)}_{F,\Pi}$ with propagation rules that capture these techniques. In the subsequent section, we discus how this new graph models solvers CMODELS, CLASP, and MINISAT(ID). Then we provide insights into how they are related.

4 Backjumping and Learning for SM(ASP)

Nieuwenhuis et al. (2006, Section 2.4) defined the *DPLL System with Learning* graph that can be used to describe most of the modern SAT solvers, which typically implement such sophisticated techniques as learning and backjumping. We demonstrate how to extend these findings to capture SM(ASP) framework with learning and backjumping.

Let $[F,\Pi]$ be an SM(ASP) theory and let G be a formula over $At(F\cup\Pi)$. We say that $[F,\Pi]$ *entails* G, written $F,\Pi\models G$, if for every model M of $[F,\Pi]$, $M\models G$.

For an SM(ASP) theory $[F,\Pi]$, an *augmented state* relative to F and Π is either a distinguished state *FailState* or a pair $M||\Gamma$ where M is a record relative to the set of atoms occurring in F and Π, and Γ is a set of clauses over $At(F\cup\Pi)$ such that $F,\Pi^o\models\Gamma$.

We now define a graph $\text{SML(ASP)}_{F,\Pi}$ for an SM(ASP) theory $[F,\Pi]$. Its nodes are the augmented states relative to F and Π. The rules *Decide*, *Unfounded*, and *Fail* of $\text{SM(ASP)}_{F,\Pi}$ are extended to $\text{SML(ASP)}_{F,\Pi}$ as follows: $M||\Gamma\Longrightarrow M'||\Gamma$ ($M||\Gamma\Longrightarrow$ *FailState*, respectively) is an edge in $\text{SML(ASP)}_{F,\Pi}$ justified by *Decide* or *Unfounded* (*Fail*, respectively) if and only if $M\Longrightarrow M'$ ($M\Longrightarrow$ *FailState*) is an edge in $\text{SM(ASP)}_{F,\Pi}$ justified by *Decide* or *Unfounded* (*Fail*, respectively). The other transition rules of $\text{SML(ASP)}_{F,\Pi}$ follow:

$$\textit{Unit Propagate Learn}:\quad M||\Gamma\Longrightarrow Ml||\Gamma\ \text{ if }\ \begin{cases} C\vee l\in F\cup\Pi^{cl}\cup\Gamma \text{ and}\\ \overline{C}\subseteq M\end{cases}$$

$$\textit{Backjump}:\quad Pl^{\Delta}Q||\Gamma\Longrightarrow Pl'||\Gamma\ \text{ if }\ \begin{cases} Pl^{\Delta}Q \text{ is inconsistent and}\\ F,\Pi^o\models l'\vee\overline{P}\end{cases}$$

$$\textit{Learn}:\quad M||\Gamma\Longrightarrow M||C,\Gamma\ \text{ if }\ \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ and}\\ F,\Pi^o\models C.\end{cases}$$

We refer to the transition rules *Unit Propagate Learn*, *Unfounded*, *Backjump*, *Decide*, and *Fail* of the graph $\text{SML(ASP)}_{F,\Pi}$ as *basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn* is applicable to it. We omit the word "augmented" before "state" when this is clear from a context.

The graph $\text{SML(ASP)}_{F,\Pi}$ can be used for deciding whether an SM(ASP) theory $[F,\Pi]$ has a model.

Proposition 9
For any SM(ASP) theory $[F,\Pi]$,

(a) every path in $\text{SML(ASP)}_{F,\Pi}$ contains only finitely many edges justified by basic transition rules,
(b) for any semi-terminal state $M||\Gamma$ of $\text{SML(ASP)}_{F,\Pi}$ reachable from $\emptyset||\emptyset$, M is a model of $[F,\Pi]$,
(c) *FailState* is reachable from $\emptyset||\emptyset$ in $\text{SML(ASP)}_{F,\Pi}$ if and only if $[F,\Pi]$ has no models.

On the one hand, Proposition 9 (a) asserts that if we construct a path from $\emptyset||\emptyset$ so that basic transition rules periodically appear in it then some semi-terminal state is eventually reached. On the other hand, parts (b) and (c) of Proposition 9 assert that as soon as a semi-terminal state is reached the problem of deciding whether $[F,\Pi]$ has a model is solved. In other words, Proposition 9 shows that the graph $\text{SML(ASP)}_{F,\Pi}$ gives rise to a class of correct algorithms for computing models of an SM(ASP) theory $[F,\Pi]$. It gives a proof

of correctness to every SM(ASP) solver in this class and a proof of termination under the assumption that basic transition rules periodically appear in a path constructed from $\emptyset||\emptyset$.

Nieuwenhuis et al. (2006) proposed the transition rules to model such techniques as forgetting and restarts. The graph $\text{SML(ASP)}_{F,\Pi}$ can easily be extended with such rules.

5 Abstract CMODELS, CLASP and MINISAT(ID)

We can view a path in the graph $\text{SML(ASP)}_{F,\Pi}$ as a description of a process of search for a model of an SM(ASP) theory $[F,\Pi]$ by applying transition rules. Therefore, we can characterize the algorithm of a solver that utilizes the transition rules of $\text{SML(ASP)}_{F,\Pi}$ by describing a strategy for choosing a path in this graph. A strategy can be based, in particular, on assigning priorities to transition rules of $\text{SML(ASP)}_{F,\Pi}$, so that a solver never applies a rule in a state if a rule with higher priority is applicable to the same state.

We use this approach to describe and compare the algorithms implemented in the solvers CMODELS, CLASP and MINISAT(ID). We stress that we talk here about characterizing and comparing algorithms and not their specific implementations in the solvers. We refer to these algorithms as *abstract* CMODELS, CLASP and MINISAT(ID), respectively. Furthermore, we only discuss the abstract MINISAT(ID) for the case of the total PC(ID) theories whereas the MINISAT(ID) system implements additional *totality check* propagation rule to deal with the non-total theories. Given a program Π, abstract CMODELS and abstract CLASP construct first $\textit{ED-Comp}(\Pi)$. Afterwards, they search the graph $\text{SML(ASP)}_{\textit{ED-Comp}(\Pi),\Pi}$ for a path to a semi-terminal state. In other words, both algorithms, while in a node of the graph $\text{SML(ASP)}_{\textit{ED-Comp}(\Pi),\Pi}$, progress by selecting one of the outgoing edges. By Proposition 9 and Corollary 1, each algorithm is indeed a method to compute answer sets of programs.

However, abstract CMODELS selects edges according to the priorities on the transition rules of the graph that are set as follows:

$$\textit{Backjump}, \textit{Fail} \gg \textit{Unit Propagate} \gg \textit{Decide} \gg \textit{Unfounded},$$

while abstract CLASP uses a different prioritization:

$$\textit{Backjump}, \textit{Fail} \gg \textit{Unit Propagate} \gg \textit{Unfounded} \gg \textit{Decide}.$$

The difference between the algorithms boils down to when the rule *Unfounded* is used.

We now describe the algorithm behind the PC(ID) solver MINISAT(ID) (Mariën et al. 2008) for total PC(ID) theories — the abstract MINISAT(ID). Speaking precisely, MINISAT(ID) assumes that the program Π of the input PC(ID) theory (F,Π) is in the *definitional normal form* (Mariën 2009). Therefore, in practice MINISAT(ID) is always used with a simple preprocessor that converts programs into the definitional normal form. We will assume here that this preprocessor is a part of MINISAT(ID). Under this assumption, given a PC(ID) theory (F,Π), MINISAT(ID) can be described as constructing the completion $\textit{ED-Comp}(\Pi^o)$ (the new atoms are introduced by the preprocessor when it converts Π into the definitional normal form, the completion part is performed by the MINISAT(ID) proper), and then uses the transitions of the graph $\text{SML(ASP)}_{\textit{ED-Comp}(\Pi^o)\cup F,\Pi^o}$ to search for a path to a semi-terminal state. In other words, the graph $\text{SML(ASP)}_{\textit{ED-Comp}(\Pi^o)\cup F,\Pi^o}$ represents the abstract MINISAT(ID). The strategy used by the algorithm follows the prioritization:

$$\textit{Backjump}, \textit{Fail} \gg \textit{Unit Propagate} \gg \textit{Unfounded} \gg \textit{Decide}.$$

By Propositions 4 and 6, the algorithm indeed computes models of total PC(ID) theories.

Systems CMODELS, CLASP, and MINISAT(ID) implement conflict-driven backjumping and learning. They apply the transition rule *Learn* only when in a non-semi-terminal state reached by an application of *Backjump*. Thus, the rule *Learn* does not differentiate the algorithms and so we have not taken it into account when describing these algorithms.

6 PC(ID) Theories as Logic Programs with Constraints

For a clause $C = \neg a_1 \vee \ldots \vee \neg a_l \vee a_{l+1} \vee \ldots \vee a_m$ we write C^r to denote the corresponding rule constraint

$$\leftarrow a_1, \ldots, a_l, \mathit{not}\ a_{l+1}, \ldots, \mathit{not}\ a_m.$$

For a set F of clauses, we define $F^r = \{C^r \mid C \in F\}$. Finally, for a PC(ID) theory (F, Π) we define a logic program $\pi(F, \Pi)$ by setting

$$\pi(F, \Pi) = \Pi^o \cup F^r.$$

The representation of a PC(ID) theory (F, Π) as $\pi(F, \Pi)$ is similar to the translation of FO(ID) theories into logic programs with variables given by Mariën et al. (2004). The difference is in the way atoms are "opened." We do it by means of rules of the form $a \leftarrow \mathit{not}\ \mathit{not}\ a$, while Mariën et al. use pairs of rules $a \leftarrow \mathit{not}\ a^*$ and $a^* \leftarrow \mathit{not}\ a$.

There is a close relation between models of a PC(ID) theory (F, Π) and answer sets of a program $\pi(F, \Pi)$.

Proposition 10
For a total PC(ID) theory (F, Π) and a consistent and complete (over $At(F \cup \Pi)$) set M of literals, M is a model of (F, Π) if and only if M^+ is an answer set of $\pi(F, \Pi)$.

A *choice rule* construct $\{a\}$ (Niemelä and Simons 2000) of the LPARSE[6] and GRINGO[7] languages can be seen as an abbreviation for a rule $a \leftarrow \mathit{not}\ \mathit{not}\ a$ (Ferraris and Lifschitz 2005). Thus, in view of Proposition 10, any answer set solver implementing language of LPARSE or GRINGO is also a PC(ID) solver (an input total PC(ID) theory (F, Π) needs to be translated into $\pi(F, \Pi)$).

The reduction implied by Proposition 10 by itself does not show how to relate particular solvers. However, we recall that abstract MINISAT(ID) is captured by the graph $\mathrm{SML(ASP)}_{\mathit{ED\text{-}Comp}(\Pi^o) \cup F, \Pi^o}$. Moreover, we have the following property.

Proposition 11
For a PC(ID) theory (F, Π), we have

$$\mathrm{SML(ASP)}_{\mathit{ED\text{-}Comp}(\pi(F,\Pi)), \pi(F,\Pi)} = \mathrm{SML(ASP)}_{\mathit{ED\text{-}Comp}(\Pi^o) \cup F, \Pi^o}.$$

The graph $\mathrm{SML(ASP)}_{\mathit{ED\text{-}Comp}(\pi(F,\Pi)), \pi(F,\Pi)}$ captures the way CLASP works on the program $\pi(F, \Pi)$. In addition, the MINISAT(ID) and CLASP algorithms use the same prioritization. Thus, Proposition 11 implies that the abstract CLASP used as a PC(ID) solver coincides with the abstract MINISAT(ID).

[6] `http://www.tcs.hut.fi/Software/smodels/` .
[7] `http://potassco.sourceforge.net/` .

7 Related Work and Discussion

Lierler (2011) introduced the graphs SML and GTL that extended the graphs SM and GT (Lierler 2011), respectively, with transition rules *Backjump* and *Learn*. The graph SML was used to characterize the computation of such answer set solvers implementing learning as SMODELS$_{cc}$[8] (Ward and Schlipf 2004) and SUP[9] (Lierler 2011) whereas the graph GTL was used to characterize CMODELS. These graphs are strongly related to our graph SML(ASP) but they are not appropriate for describing the computation behind answer set solver CLASP or PC(ID) solver MINISAT(ID). The graph SML reflects only propagation steps based on a program whereas CLASP and MINISAT(ID) proceed by considering both the program and a propositional theory. The graph GTL, on the other hand, does not seem to provide a way to imitate the behavior of the *Unfounded* rule in the SML(ASP) graph.

Giunchiglia and Maratea (2005) studied the relation between the answer set solver SMODELS and the DPLL procedure for the case of tight programs by means of pseudocode analysis. Giunchiglia et al (2008) continued this work by comparing answer set solvers SMODELS, DLV[10] (Eiter et al. 1997), and CMODELS via pseudocode. In this paper we use a different approach to relate solvers that was proposed by Lierler (2011). That is, we use graphs to represent the algorithms implemented by solvers, and study the structure of these graphs to find how the corresponding solvers are related. We use this method to state the relation between the answer set solvers CMODELS, CLASP, and the PC(ID) solver MINISAT(ID) designed for different knowledge representation formalisms.

Gebser and Schaub (2006) introduced a deductive system for describing inferences involved in computing answer sets by tableaux methods. The abstract framework presented in this paper can be viewed as a deductive system also, but a very different one. For instance, we describe backtracking and backjumping by inference rule, while the Gebser-Schaub system does not. Also the Gebser-Schaub system does not take learning into account. Accordingly, the derivations considered in this paper describe a search process, while derivations in the Gebser-Schaub system do not. Further, the abstract framework discussed here does not have any inference rule similar to Cut; this is why its derivations are paths rather than trees.

Mariën (2009) (Section 5.7) described a MINISAT(ID) transition system to model a computation behind the PC(ID) solver MINISAT(ID). We recall that we modeled the abstract MINISAT(ID) with the graph SML(ASP). The graphs SML(ASP) and MINISAT(ID) are defined using different sets of nodes and transition rules. For instance, SML(ASP) allows states containing inconsistent sets of literals whereas the MINISAT(ID) graph considers consistent states only. Due to this difference the MINISAT(ID) graph requires multiple versions of "backjump" and "fail" transition rules.

We used transition systems to characterize algorithms for computing answer sets of logic programs and models of PC(ID) theories. These transition systems are also suitable for formal comparison of the strength or power of reasoning methods given rules that specify them. An approach to do so was proposed by Mariën (2009) (Section 5.7), who introduced

[8] http://www.nku.edu/∼wardj1/Research/smodels_cc.html .
[9] http://www.cs.utexas.edu/users/tag/sup .
[10] http://www.dbai.tuwien.ac.at/proj/dlv/ .

the concept of *decide-efficiency* for such analysis. We outline below how standard concepts of *proof complexity* (Cook et al. 1979) can be adapted to the setting of transition systems.

Let $\mathscr{A}$ be an infinite set of atoms. We define a *node over* $\mathscr{A}$ to be a symbol *FailState*, or a finite sequence of literals over $\mathscr{A}$ with annotations. For a propositional formalism $\mathscr{F}$ over $\mathscr{A}$, a *proof procedure* $\mathscr{P}_{\mathscr{F}}$ consists of graphs G_T, where T ranges over all theories in $\mathscr{F}$, such that for every theory T (i) G_T is composed of nodes over $\mathscr{A}$ and (ii) T is *unsatisfiable* if and only if there is a path p in G_T from the empty (*start*) node to the *FailState* node. We call each such path p *a proof*. We say that a proof system S is *based* on a proof procedure $\mathscr{P}_{\mathscr{F}}$ if (i) $S \subseteq \mathscr{F} \times \mathscr{R}$, where $\mathscr{R}$ denotes the set of all finite sequences of nodes over $\mathscr{A}$, and (ii) $S(T,p)$ holds if and only if p is a proof in the graph G_T in $\mathscr{P}_{\mathscr{F}}$. Predicate S is indeed a proof system in the sense of Cook (1979) because (i) S is polynomial-time computable, and (ii) T is unsatisfiable if and only if there exists a proof p such that $S(T,p)$ holds.

In this sense, each of the graphs (transition systems) we introduced in this paper can be regarded as a proof procedure for SM(ASP) (for those involving the rule *Learn*, under additional assumptions to ensure the rule can be efficiently implemented). Thus, transition systems determine proof systems. Consequently, they can be compared, as well as solvers that they capture, in terms of the complexity of the corresponding proof systems.

8 Conclusions

In the paper, we proposed a formalism SM(ASP) that can be regarded as a common generalization of (clausal) propositional logic, ASP, and the logic PC(ID). The formalism offers an elegant *satisfiability modulo theories* perspective on the latter two. We present several characterizations of these formalisms in terms of SM(ASP) theories that differ in the explicitly identified "satisfiability" component. Next, we proposed transition systems for SM(ASP) to provide abstract models of SM(ASP) model generators. The transition systems offer a clear and uniform framework for describing model generation algorithms in SM(ASP). As SM(ASP) subsumes several propositional formalisms, such a uniform approach provides a general proof of correctness and termination that applies to a broad class of model generators designed for these formalisms. It also allows us to describe in precise mathematical terms relations between algorithms designed for reasoning with different logics such as propositional logic, logic programming under answer-set semantics and the logic PC(ID), the latter two studied in detail in the paper. For instance, our results imply that at an abstract level of transition systems, CLASP and MINISAT(ID) are essentially identical. Finally, we note that this work gives the first description of CLASP in the abstract framework rather than in pseudocode. Such high level view on state-of-the-art solvers in different, yet, related propositional formalisms will further their understanding, and help port advances in solver technology from one area to another.

Acknowledgments

We are grateful to Marc Denecker and Vladimir Lifschitz for useful discussions. We are equally grateful to the reviewers who helped eliminate minor technical problems and improve the presentation. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship. Miroslaw Truszczynski was supported by the NSF grant IIS-0913459.

References

BEAME, P., KAUTZ, H., AND SABHARWAL, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research 22*, 319–351.

CLARK, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

COOK, S. A., ROBERT, AND RECKHOW, A. 1979. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic 44*, 36–50.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem proving. *Communications of the ACM 5(7)*, 394–397.

DENECKER, M. 2000. Extending classical logic with inductive definitions. In *Proceedings of the 1st International Conference on Computational Logic, CL 2000*. Lecture Notes in Computer Science, vol. 1861. Springer, Berlin, 703–717.

EITER, T., LEONE, N., MATEIS, C., PFEIFER, G., AND SCARCELLO, F. 1997. A deductive system for non-monotonic reasoning. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 1997*. Lecture Notes in Computer Science, vol. 1265. Springer, Berlin, 363–374.

FERRARIS, P. AND LIFSCHITZ, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 45–74.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*. 386–392.

GEBSER, M. AND SCHAUB, T. 2006. Tableau calculi for answer set programming. In *Proceedings of the 22nd International Conference on Logic Programming, ICLP 2006*. Lecture Notes in Computer Science, vol. 4079. Springer, Berlin, 11–25.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Logic Programming Conference and Symposium*, R. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1070–1080.

GIUNCHIGLIA, E., LEONE, N., AND MARATEA, M. 2008. On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence 53*, 1-4, 169–204.

GIUNCHIGLIA, E., LIERLER, Y., AND MARATEA, M. 2004. SAT-based answer set programming. In *Proceedings of the 19th National Conference on Artificial Intelligence, AAAI 2004*. AAAI Press, Menlo Park, CA, 61–66.

GIUNCHIGLIA, E. AND MARATEA, M. 2005. On the relation between answer set and SAT procedures (or, between smodels and cmodels). In *Proceedings of the 21st International Conference on Logic Programming, ICLP 2005*. Lecture Notes in Computer Science, vol. 3668. Springer, Berlin, 37–51.

LEE, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005*. Professional Book Center, 503–508.

LEONE, N., RULLO, P., AND SCARCELLO, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation 135(2)*, 69–112.

LIERLER, Y. 2010. Sat-based answer set programming. Ph.D. thesis, University of Texas at Austin.

LIERLER, Y. 2011. Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming 11*, 135–169.

LIFSCHITZ, V., TANG, L. R., AND TURNER, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25*, 369–389.

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157*, 115–137.

MAREK, V. AND TRUSZCZYŃSKI, M. 1999. Stable models and an alternative logic programming

paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, Berlin, 375–398.

MARIËN, M. 2009. Model generation for ID-logic. Ph.D. thesis, Katholieke Universiteit Leuven.

MARIËN, M., GILIS, D., AND DENECKER, M. 2004. On the relation between ID-logic and answer set programming. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence, JELIA 2004*. Lecture Notes in Computer Science, vol. 3229. Springer, Berlin, 108–120.

MARIËN, M., WITTOCX, J., DENECKER, M., AND BRUYNOOGHE, M. 2008. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing, SAT 2008*. Lecture Notes in Computer Science, vol. 4996. Springer, Berlin, 211–224.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 241–273.

NIEMELÄ, I. AND SIMONS, P. 2000. Extending the Smodels system with cardinality and weight constraints. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer, Dordrecht, 491–521.

NIEUWENHUIS, R., OLIVERAS, A., AND 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.

OIKARINEN, E. AND JANHUNEN, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*. IOS Press, Amsterdam, 412–416.

VAN GELDER, A., ROSS, K., AND SCHLIPF, J. 1991. The well-founded semantics for general logic programs. *Journal of ACM 38*, 3, 620–650.

WARD, J. AND SCHLIPF, J. 2004. Answer set programming with clause learning. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2004*. Lecture Notes in Computer Science, vol. 2923. Springer, Berlin, 302–313.