Abstract Answer Set Solvers with Backjumping and Learning

YULIYA LIERLER

Department of Computer Science
University of Texas at Austin
1 University Station C0500
Taylor Hall 2.124
Austin, USA
E-mail: yuliya@cs.utexas.edu

submitted 10th June 2009; revised 15th October 2009; accepted 9th December 2009

Abstract

Nieuwenhuis, Oliveras, and Tinelli (2006) showed how to describe enhancements of the Davis-Putnam-Logemann-Loveland algorithm using transition systems, instead of pseudocode. We design a similar framework for several algorithms that generate answer sets for logic programs: SMODELS, SMODELS$_{cc}$, ASP-SAT with Learning (CMODELS), and a newly designed and implemented algorithm SUP. This approach to describing answer set solvers makes it easier to prove their correctness, to compare them, and to design new systems.

KEYWORDS: answer set programming, inference, learning

1 Introduction

Answer Set Programming (ASP) is a methodology commonly used for solving combinatorial search problems (Lifschitz 2008). In the development of ASP solvers, computational ideas behind SAT solvers (Gomes et al. 2008) play an important role. Influence of SAT solvers development on ASP systems is twofold. On the one hand, such ASP solvers as ASSAT[1] and CMODELS[2] follow the so called SAT-based approach where a SAT solver is invoked for search, possibly multiple times. On the other hand, "native" ASP solvers that implement search procedures specifically suited for logic programs often adopt computational techniques from SAT solvers. For instance, DLV[3] implements backjumping (Ricca et al. 2006), and SMODELS$_{cc}$[4] (Ward and Schlipf 2004) extends the answer set solver SMODELS[5] by introducing restarts, conflict-driven backjumping, learning, and forgetting – techniques widely used in

[1] http://assat.cs.ust.hk/ .
[2] http://www.cs.utexas.edu/users/tag/cmodels .
[3] http://www.dbai.tuwien.ac.at/proj/dlv/ .
[4] http://www.nku.edu/~wardj1/Research/smodels_cc.html .
[5] http://www.tcs.hut.fi/Software/smodels/ .

SAT solvers. The ASP solvers CLASP[6] (Gebser et al. 2007) and SUP[7] (Lierler 2008) implement these features also.

In this paper our main goal is to show how the "abstract" approach to describing SAT solvers proposed in (Nieuwenhuis et al. 2006) can be extended to ASP solvers that use these sophisticated features. Usually computation procedures are described in terms of pseudocode. In (Nieuwenhuis et al. 2006), the authors proposed an alternative approach to describing DPLL-like procedures. They introduced an abstract framework that captures what "states of computation" are, and what transitions between states are allowed. In this way, it 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 branching, some to backtracking. This allows the authors to model a DPLL-like algorithm by a mathematically simple and elegant object, graph, rather than a collection of pseudocode statements. In (Lierler 2008), we extended this framework for describing such ASP algorithms as SMODELS, ASP-SAT with Backtracking, and SUP *without Learning*. In this paper, we expand our previous work on abstract answer set solvers to cover such features as backjumping and learning (and also forgetting and restart). We start by introducing an abstract framework that captures a general mechanism of these sophisticated features in ASP solvers. For instance, this framework provides the transition underlying the process of learning a clause, but it does not suggest which clause shall be learned. Similarly, it provides a general description of backjumping but it does not supply the means for computing a "backjump clause" necessary for an answer set solver to perform backjumping. We then enhance this abstract framework to capture enough information about a state of computation for deriving a backjump clause.

Usually, DPLL-like procedures implement conflict-driven backjumping and learning where a particular learning schema such as, for instance, *Decision* or *FirstUIP* (Mitchell 2005) is applied for computing a special kind of a backjump clause. There are two common methods for describing a backjump clause construction. One employs the implication graph (Marques-Silva and Sakallah 1996) and the other employs resolution (Mitchell 2005). Ward and Schlipf (2004) extended the notion of an implication graph to the SMODELS algorithm. They then defined an algorithm for computing *FirstUIP* backjump clauses utilized by SMODELS_{cc} to implement conflict-driven backjumping and learning. In this paper we introduce the algorithms *BackjumpClause* and *BackjumpClauseFirstUIP* based on resolution and the enhanced abstract framework that compute *Decision* and *FirstUIP*[8] backjump clauses respectively.

In (Lierler 2008), we introduced the basic algorithm underlining the system SUP but neglected some of its features: conflict-driven backjumping, learning, forgetting, and restarts. Here we account for these techniques and use an abstract framework designed in this paper for describing system SUP. We emphasize that the work

[6] `http://www.cs.uni-potsdam.de/clasp/` .
[7] `http://www.cs.utexas.edu/users/tag/sup` .
[8] The names of the backjump clauses follow (Mitchell 2005).

on this abstract framework helped us to develop ASP solver SUP, to incorporate learning into its algorithm, and to prove its correctness.

We start the paper with Section 2 that reviews the abstract DPLL framework introduced in (Nieuwenhuis et al. 2006) and some logic programming concepts. In Section 3, we define a graph representing the application of the algorithm for finding supporting models of a logic program. This paves the way to defining a graph representing the application of the SMODELS algorithm to a program in Section 4. Section 4.2 elaborates on the relationship between previously defined abstract frameworks. Section 5 extends the abstract DPLL framework by introducing an additional inference rule so that the *generate and test* algorithm of the SAT-based ASP system CMODELS may be characterized by this graph. In Section 6, we review the abstract framework that describes DPLL enhanced by backjumping and learning. In Section 7, we define a general abstract framework for describing ASP algorithms that implement such phenomena as backjumping and learning. In Section 7.2 we describe the algorithms of systems SMODELS$_{cc}$ and SUP by means of this framework. In Section 8 we extend the abstract *generate and test* framework to accommodate backjumping and learning, and in Section 8.2 we use these findings to describe the CMODELS algorithm. Section 9 extends the framework for describing ASP algorithms to capture additional information about computation states of a solver, demonstrates the correctness results, and discusses how the frameworks are related to each other. Section 10 provides the proofs for these results. In Section 10.3 and 11 we introduce the algorithms based on the extended framework for computing a backjump clause that are important in implementing conflict-driven backjumping and learning. At last, in Section 12 we introduce the concept of an extended graph for the *generate and test* abstract framework and state the correctness results. Due to the lack of space some of the proofs are omitted here. The interested reader will find the missing proofs in the long version of the paper (Lierler 2010).

2 Review: Abstract DPLL and Logic Programs

2.1 Abstract Classical DPLL

For a set σ of atoms, a *record* M relative to σ is a list of literals over σ where

(i) some literals in M are annotated by Δ that marks them as *decision* literals,
(ii) M contains no repetitions.

The concatenation of two such lists is denoted by juxtaposition. Frequently, we consider a record as a set of literals, ignoring both the annotations and the order between its elements. A literal l is *unassigned by* a record if neither l nor its complement $\overline{l}$ belongs to it.

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 $\{a\}$ of atoms are

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

where by $\emptyset$ we denote the empty list.

$$\textit{Unit Propagate}: \quad M \Longrightarrow M\,l \ \text{ if } \ \begin{cases} C \vee l \in F \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

$$\textit{Decide}: \quad M \Longrightarrow M\,l^{\Delta} \ \text{ if } \ \begin{cases} M \text{ is consistent and} \\ l \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail}: \quad M \Longrightarrow \textit{FailState} \ \text{ if } \ \begin{cases} M \text{ is inconsistent and} \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}: \quad P\,l^{\Delta}\,Q \Longrightarrow P\,\overline{l} \ \text{ if } \ \begin{cases} P\,l^{\Delta}\,Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 1. The transition rules of the graph $\textsc{dp}_F$.

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. We will sometimes identify C with the multi-set of its elements.

For any CNF formula F (a finite set of clauses), we will define its *DPLL graph* $\textsc{dp}_F$. The nodes of $\textsc{dp}_F$ are the states relative to the set of atoms occurring in F. We use the terms "state" and "node" interchangeably. Recall that a node is called *terminal* in a graph if there is no edge leaving this node in the graph. If a state is consistent and complete then it represents a truth assignment for F.

The set of edges of $\textsc{dp}_F$ is described by a set of "transition rules." Each transition rule is an expression $M \Longrightarrow M'$ followed by a condition, where M and M' are nodes of $\textsc{dp}_F$. Whenever the condition is satisfied, the graph contains an edge from node M to M'. Generally, an edge in the graph may be justified by several transition rules. Figure 1 presents four transition rules that characterize the edges of $\textsc{dp}_F$.

This graph can be used for deciding the satisfiability of a formula F simply by constructing an arbitrary path leading from node $\emptyset$ until a terminal node M is reached. The following proposition shows that this process always terminates, that F is unsatisfiable if M is *FailState*, and that M is a model of F otherwise.

Proposition 1

For any CNF formula F,

(a) graph $\textsc{dp}_F$ is finite and acyclic,

(b) any terminal state of $\textsc{dp}_F$ other than *FailState* is a model of F,

(c) *FailState* is reachable from $\emptyset$ in $\textsc{dp}_F$ if and only if F is unsatisfiable.

For instance, let F be the set consisting of the clauses

$$a \vee b$$
$$\neg a \vee c.$$

Here is a path in DP_F:

$$\begin{array}{ll} \emptyset & \Longrightarrow (\mathit{Decide}) \\ a^{\Delta} & \Longrightarrow (\mathit{Unit\ Propagate}) \\ a^{\Delta}\, c & \Longrightarrow (\mathit{Decide}) \\ a^{\Delta}\, c\, b^{\Delta} & \end{array} \tag{1}$$

The name of the transition rule after each $\Longrightarrow$ shows which rule justifies the presence of this edge in the graph. Since the state $a^{\Delta}\, c\, b^{\Delta}$ is terminal, Proposition 1(b) asserts that $\{a, c, b\}$ is a model of F. Here is another path in DP_F from $\emptyset$ to the same terminal node:

$$\begin{array}{ll} \emptyset & \Longrightarrow (\mathit{Decide}) \\ a^{\Delta} & \Longrightarrow (\mathit{Decide}) \\ a^{\Delta} \neg c^{\Delta} & \Longrightarrow (\mathit{Unit\ Propagate}) \\ a^{\Delta} \neg c^{\Delta}\, c & \Longrightarrow (\mathit{Backtrack}) \\ a^{\Delta}\, c & \Longrightarrow (\mathit{Decide}) \\ a^{\Delta}\, c\, b^{\Delta} & \end{array} \tag{2}$$

Path (1) corresponds to an execution of DPLL in the sense of (Davis et al. 1962); path (2) does not, because it applies *Decide* to a^{Δ} even though *Unit Propagate* could be applied in this state.

Note that the graph DP_F is a modification of the *classical DPLL* graph defined in (Nieuwenhuis et al. 2006, Section 2.3). It is different in three ways. First, its states are pairs $M||F$ for all CNF formulas F. For the purposes of this section, it is not necessary to include F. Second, the description of the classical DPLL graph involves a "PureLiteral" transition rule. We dropped this rule because it does not correspond to any of the propagation rules used in answer set solvers whose algorithms we will model in this paper. Third, in the definition of that graph, each M is required to be consistent. In case of DPLL, due to the simple structure of a clause, it is possible to characterize the applicability of *Backtrack* in a simple manner: when some of the clauses become inconsistent with the current partial assignment, *Backtrack* is applicable. In ASP, it is not easy to describe the applicability of *Backtrack* if only consistent states are taken into account. We introduced inconsistent states in the graph DP_F to facilitate our work on extending this graph to model algorithms of answer set solvers.

2.2 Logic Programs

We consider programs consisting of finitely many rules of the form

$$a \leftarrow b_1, \ldots, b_l, \mathit{not}\ b_{l+1}, \ldots, \mathit{not}\ b_m \tag{3}$$

where a is an atom or symbol $\bot$, and each b_i $(1 \leq i \leq m)$ is an atom. We will identify the body of (3) with the conjunction

$$b_1 \wedge \ldots \wedge b_l \wedge \neg b_{l+1} \wedge \ldots \wedge \neg b_m \tag{4}$$

and also with the set of its conjunctive terms. If the head a of a rule (3) is an atom then we will identify (3) with the clause

$$a \vee \neg b_1 \vee \ldots \vee \neg b_l \vee b_{l+1} \vee \ldots \vee b_m. \tag{5}$$

If a is $\bot$ then we call rule (3) a *constraint* and identify (3) with the clause

$$\neg b_1 \vee \ldots \vee \neg b_l \vee b_{l+1} \vee \ldots \vee b_m. \tag{6}$$

We will often omit the symbol $\bot$ when referring to a constraint.

We will use two abbreviated forms for a rule (3): The first is

$$a \leftarrow B$$

where B stands for $b_1, \ldots, b_l, not\ b_{l+1}, \ldots, not\ b_m$. The second abbreviation is

$$a \leftarrow D, F \tag{7}$$

where D stands for the *positive part of the body* $b_1, \ldots, b_l$, and F stands for the *negative part of the body* $not\ b_{l+1}, \ldots, not\ b_m$.

The *reduct* Π^X of a program Π with respect to a set X of atoms is obtained from Π by

- removing each rule (7) such that $\overline{F} \cap X \neq \emptyset$, and
- replacing each remaining rule (7) by $a \leftarrow D$.

A set X of atoms is an *answer set* for a program Π if X is minimal (with respect to set inclusion) among the sets of atoms that satisfy the reduct Π^X (Gelfond and Lifschitz 1988).

For example, let Π be the program

$$\begin{array}{ll} a \leftarrow not\ b & c \leftarrow a \\ b \leftarrow not\ a & d \leftarrow d. \end{array} \tag{8}$$

Consider set $\{a, c\}$. Reduct $\Pi^{\{a,c\}}$ is

$$\begin{array}{l} a \leftarrow \\ c \leftarrow a \\ d \leftarrow d. \end{array} \tag{9}$$

Set $\{a, c\}$ satisfies the reduct and is minimal, hence $\{a, c\}$ is an answer set of Π. Consider set $\{a, c, d\}$. The reduct $\Pi^{\{a,c,d\}}$ is (9). Set $\{a, c, d\}$ satisfies the reduct but is not minimal and hence it is not an answer set of Π.

By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of Π with head a. For any set M of literals, by M^+ we denote the set of positive literals from M. For any consistent and complete set M of literals (that is, an assignment), if M^+ is an answer set for a program Π, then M is a model of Π. Moreover, in this case M is a *supported* model of Π, in the sense that for every atom $a \in M$, $M \models B$ for some $B \in Bodies(\Pi, a)$.

A set U of atoms occurring in a program Π is said to be *unfounded* (Van Gelder et al. 1991) on a consistent set M of literals w.r.t. Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $\overline{B} \cap M \neq \emptyset$ or $U \cap B^+ \neq \emptyset$. There is a tight relation between unfounded sets and answer sets:

Proposition 2 (*Corollary 2 from (Saccá and Zaniolo 1990)*)
For any model M of a program Π, M^+ is an answer set for Π if and only if M contains no non-empty subsets unfounded on M w.r.t. Π.[9]

For instance, let Π be program (8) and let M be a consistent set $\{a, \neg b, c, d\}$ of literals. We already demonstrated that $M^+ = \{a, c, d\}$ is not an answer set of Π. Accordingly, its subset $\{d\}$ is unfounded on $\{a, \neg b, c, d\}$ w.r.t. Π, because the only rule in Π with d in the head $d \leftarrow d$ is such that $U \cap B^+ = \{d\} \cap \{d\} \neq \emptyset$.

We say that a program Π *entails* a formula F when for any consistent and complete set M of literals, if M^+ is an answer set for Π, then $M \models F$. For instance, any program Π entails each rule occurring in Π.

3 Generating Supported Models

In Section 4 we will define, for an arbitrary program Π, a graph SM_Π representing the application of the SMODELS algorithm to Π; the terminal nodes of SM_Π are answer sets of Π. As a step in this direction, we describe here a simpler graph ATLEAST_Π.

3.1 Graph ATLEAST_Π

The terminal nodes of ATLEAST_Π are supported models of Π. The transition rules defining ATLEAST_Π are closely related to procedure *Atleast* (Simons 2000, Sections 4.1), which is one of the core procedures of the SMODELS algorithm.

The nodes of ATLEAST_Π are the states relative to the set of atoms occurring in Π. The edges of the graph ATLEAST_Π are described by the transition rules *Decide*, *Fail*, *Backtrack* introduced in Section 2.1 and the additional transition rules[10] presented in Figure 2. Note that each of the rules *Unit Propagate LP* and *Backchain False* is similar to *Unit Propagate*: the former corresponds to *Unit Propagate* on $C \vee l$ where l is the head of the rule, and the latter corresponds to *Unit Propagate* on $C \vee l$ where $\bar{l}$ is an element of the body of the rule.

This graph can be used for deciding whether program Π has a supported model by constructing a path from $\emptyset$ to a terminal node:

Proposition 3
For any program Π,

(a) graph ATLEAST_Π is finite and acyclic,
(b) any terminal state of ATLEAST_Π other than *FailState* is a supported model of Π,
(c) *FailState* is reachable from $\emptyset$ in ATLEAST_Π if and only if Π has no supported models.

[9] The Corollary 2 from (Saccá and Zaniolo 1990) refers to "assumption sets" rather than unfounded sets. But as the authors noted, in the context of this corollary the two concepts are equivalent.
[10] The names of some of these rules follow (Ward 2004).

$$\textit{Unit Propagate LP}: \quad M \Longrightarrow M\ a \ \text{ if } \begin{cases} a \leftarrow B \in \Pi \ \text{ and} \\ B \subseteq M \end{cases}$$

$$\textit{All Rules Cancelled}: \quad M \Longrightarrow M\ \neg a \ \text{ if } \overline{B} \cap M \neq \emptyset \text{ for all } B \in \mathit{Bodies}(\Pi, a)$$

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

$$\textit{Backchain False}: \quad M \Longrightarrow M\ \overline{l} \ \text{ if } \begin{cases} a \leftarrow l, B \ \in \ \Pi, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M \end{cases}$$

Fig. 2. The additional transition rules of the graph ATLEAST_Π.

For instance, let Π be program (8). Here is a path in ATLEAST_Π:

$$\begin{array}{ll} \emptyset & \Longrightarrow (\textit{Decide}) \\ a^\Delta & \Longrightarrow (\textit{Unit Propagate LP}) \\ a^\Delta\, c & \Longrightarrow (\textit{All Rules Cancelled}) \\ a^\Delta\, c \neg b & \Longrightarrow (\textit{Decide}) \\ a^\Delta\, c \neg b\, d^\Delta & \end{array} \tag{10}$$

Since the state $a^\Delta\, c \neg b\, d^\Delta$ is terminal, Proposition 3(b) asserts that $\{a, c, \neg b, d\}$ is a supported model of Π.

The assertion of Proposition 3 will remain true if we drop the transition rules *Backchain True* and *Backchain False* from the definition of ATLEAST_Π.

3.2 Relation between DP_F and ATLEAST_Π

It is well known that the supported models of a program can be characterized as models of program's completion in the sense of (Clark 1978). It turns out that the graph ATLEAST_Π is identical to the graph DP_F, where F is the (clausified) completion of Π. To make this claim precise, we first review the notion of completion.

For any program Π, its completion consists of Π and the formulas that can be written as

$$\neg a \vee \bigvee_{B \in \mathit{Bodies}(\Pi, a)} B \tag{11}$$

for every atom a in Π. *CNF-Comp*(Π) is the completion converted to CNF using straightforward equivalent transformations. In other words, *CNF-Comp*(Π) consists of clauses of two kinds:

1. the rules $a \leftarrow B$ of the program written as clauses

$$a \vee \overline{B}, \tag{12}$$

2. formulas (11) converted to CNF using the distributivity of disjunction over conjunction[11].

Proposition 4
For any program Π, the graphs ATLEAST$_\Pi$ and DP$_{CNF\text{-}Comp(\Pi)}$ are equal.

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow b,\ not\ c \\ b. \end{array} \tag{13}$$

Its completion is

$$(a \leftrightarrow b \wedge \neg c) \wedge b \wedge \neg c, \tag{14}$$

and *CNF-Comp*(Π) is

$$(a \vee \neg b \vee c) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg c) \wedge b \wedge \neg c. \tag{15}$$

Proposition 4 asserts that ATLEAST$_\Pi$ coincides with DP$_{CNF\text{-}Comp(\Pi)}$.

From Proposition 4, it follows that applying the *Atleast* algorithm to a program essentially amounts to applying DPLL to its completion.

4 Answer Set Solver Smodels

4.1 Abstract Smodels

We now describe the graph SM$_\Pi$ that represents the application of the SMODELS algorithm to program Π. SM$_\Pi$ is a graph whose nodes are the same as the nodes of the graph ATLEAST$_\Pi$. The edges of SM$_\Pi$ are described by the transition rules of ATLEAST$_\Pi$ and the additional transition rule:

$$\textit{Unfounded}: \quad M \implies M\,\neg a \ \text{ if } \begin{cases} M \text{ is consistent, and} \\ a \in U \text{ for a set } U \text{ unfounded on } M \text{ w.r.t. } \Pi \end{cases}$$

This transition rule of SM$_\Pi$ is closely related to procedure *Atmost* (Simons 2000, Sections 4.2), which together with the procedure *Atleast* forms the core of the SMODELS algorithm.

The graph SM$_\Pi$ can be used for deciding whether program Π has an answer set by constructing a path from $\emptyset$ to a terminal node:

Proposition 5
For any program Π,

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

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

To illustrate the difference between SM_Π and ATLEAST_Π, assume again that Π is program (8). Path (10) in the graph ATLEAST_Π is also a path in SM_Π. But state $a^\Delta\, c \neg b\, d^\Delta$, which is terminal in ATLEAST_Π, is not terminal in SM_Π. This is not surprising, since $\{a, c, \neg b, d\}^+ = \{a, c, d\}$ is not an answer set of Π. To get to a state that is terminal in SM_Π, we need two more steps:

$$\begin{array}{ll} \vdots & \\ a^\Delta\, c \neg b\, d^\Delta & \Longrightarrow (\mathit{Unfounded},\ U = \{d\}) \\ a^\Delta\, c \neg b\, d^\Delta \neg d & \Longrightarrow (\mathit{Backtrack}) \\ a^\Delta\, c \neg b \neg d & \end{array} \tag{16}$$

Proposition 5(b) asserts that $\{a, c\}$ is an answer set of Π.

The assertion of Proposition 5 will remain true if we drop the transition rules *All Rules Cancelled*, *Backchain True*, and *Backchain False* from the definition of SM_Π.

4.2 Smodels Algorithm

We can view a path in the graph SM_Π as a description of a process of search for an answer set for a program Π by applying inference rules. Therefore, we can characterize the algorithm of an answer set solver that utilizes the inference rules of SM_Π by describing a strategy for choosing a path in SM_Π. A strategy can be based, in particular, on assigning priorities to some or all inference rules of SM_Π, 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.

We use this method to describe the SMODELS algorithm. System SMODELS assigns priorities to the inference rules of SM_Π as follows:

Backtrack, *Fail* $\gg$
Unit Propagate LP, *All Rules Cancelled*, *Backchain True*, *Backchain False* $\gg$
Unfounded $\gg$ *Decide*.

For example, let Π be program (8). The SMODELS algorithm may follow a path

$$\begin{array}{ll} \emptyset & \Longrightarrow (\mathit{Decide}) \\ a^\Delta & \Longrightarrow (\mathit{Unit\ Propagate\ LP}) \\ a^\Delta\, c & \Longrightarrow (\mathit{All\ Rules\ Cancelled}) \\ a^\Delta\, c \neg b & \Longrightarrow (\mathit{Unfounded}) \\ a^\Delta\, c \neg b \neg d & \end{array}$$

in the graph SM_Π, whereas it may never follow path (10), because *Unfounded* has a higher priority than *Decide*.

4.3 Tight Programs

We will now review the definitions of a positive dependency graph and a tight program. The *positive dependency graph* of a program Π is the directed graph G such that

- the nodes of G are the atoms occurring in Π, and

- G contains the edges from a to b_i $(1 \leq i \leq l)$ for each rule (3) in Π where a is an atom.

A program is *tight* if its positive dependency graph is acyclic. For instance, program (8) is not tight since its positive dependency graph has a cycle due to the rule $d \leftarrow d$. The program constructed from (8) by removing this rule is tight.

Recall that for any program Π and any assignment M, if M^+ is an answer set of Π then M is a supported model of Π. For the case of tight programs, the converse holds also: M^+ is an answer set for Π if and only if M is a supported model of Π (Fages 1994) or, in other words, is a model of the completion of Π.

It turns out that for tight programs the graph SM_Π is "almost identical" to the graph DP_F, where F is the clausified completion of Π. To make this claim precise, we need the following terminology.

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

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

For instance, let Π be the program

$$\begin{array}{l} a \leftarrow \ b \\ b \leftarrow \ c. \end{array}$$

The edge

$$\begin{array}{ll} a^\Delta\, b^\Delta\, \neg c^\Delta & \Longrightarrow (\textit{Unfounded},\ U = \{a, b\}) \\ a^\Delta\, b^\Delta\, \neg c^\Delta\, \neg a & \end{array}$$

in the graph SM_Π is singular, because the edge

$$\begin{array}{ll} a^\Delta\, b^\Delta\, \neg c^\Delta & \Longrightarrow (\textit{All Rules Cancelled}) \\ a^\Delta\, b^\Delta\, \neg c^\Delta\, \neg b & \end{array}$$

belongs to SM_Π also.

With respect to the actual SMODELS algorithm (Simons 2000), singular edges of the graph SM_Π are inessential: in view of priorities for choosing a path in SM_Π described in Section 4.2 SMODELS never follows a singular edge. Indeed, the transition rule *Unfounded* has the lower priority than any other transition rule but *Decide*. By SM^-_Π we denote the graph obtained from SM_Π by removing all singular edges.

Proposition 6
For any tight program Π, the graph SM^-_Π is equal to each of the graphs ATLEAST_Π and $\text{DP}_{\textit{CNF-Comp}(\Pi)}$.

For instance, let Π be the program (13). This program is tight, its completion is (14), and *CNF-Comp*(Π) is formula (15). Proposition 6 asserts that, SM^-_Π coincides with $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ and with ATLEAST_Π.

From Proposition 6, it follows that applying the SMODELS algorithm to a tight program essentially amounts to applying DPLL to its completion. A similar relationship, in terms of pseudocode representations of SMODELS and DPLL, is established in (Giunchiglia and Maratea 2005).

5 Generate and Test

In this section, we present a modification of the graph DP_F (Section 2.1) that includes testing "partial" assignments of F found by DPLL.

Let F be a CNF formula, and let G be a formula formed from atoms occurring in F. The terminal nodes of the graph $\text{GT}_{F,G}$ defined below are models of formula $F \wedge G$. This modification of the graph DP_F is of interest, for example, in connection with the fact that answer sets of a program Π can be characterized as models of its completion extended by so called loop formulas of Π (Lin and Zhao 2002). If *CNF-Comp*(Π), as above, is the completion converted to CNF, and $LF(\Pi)$ is the conjunction of all loop formulas of Π, then for any assignment M, M^+ is an answer set of Π iff M is a model of *CNF-Comp*$(\Pi) \wedge LF(\Pi)$. Hence, the terminal nodes of the graph $\text{GT}_{\textit{CNF-Comp}(\Pi),LF(\Pi)}$ will correspond to answer sets of Π.

The nodes of the graph $\text{GT}_{F,G}$ are the same as the nodes of the graph DP_F. The edges of $\text{GT}_{F,G}$ are described by the transition rules of DP_F and the additional transition rule:

$$\textit{Test}: \quad M \Longrightarrow M\,\overline{l} \;\text{ if }\; \begin{cases} M \text{ is consistent,} \\ G \models \overline{M}, \\ l \in M \end{cases}$$

It is easy to see that the graph DP_F is a subgraph of $\text{GT}_{F,G}$. The latter graph can be used for deciding whether a formula $F \wedge G$ has a model by constructing a path from $\emptyset$ to a terminal node:

Proposition 7
For any CNF formula F and a formula G formed from atoms occurring in F,

(a) graph $\text{GT}_{F,G}$ is finite and acyclic,
(b) any terminal state of $\text{GT}_{F,G}$ other than *FailState* is a model of $F \wedge G$,
(c) *FailState* is reachable from $\emptyset$ in $\text{GT}_{F,G}$ if and only if $F \wedge G$ is unsatisfiable.

Note that to verify the applicability of the new transition rule *Test* we need a procedure for testing whether G entails a clause, but there is no need to explicitly write out G. This is important because $LF(\Pi)$ can be very long (Lin and Zhao 2002).

For instance, let Π be the nontight program $d \leftarrow d$. Its completion is $d \leftrightarrow d$, and *CNF-Comp*(Π) is $(d \vee \neg d)$. This program has one loop formula $d \rightarrow \bot$. Proposition 7 asserts that a terminal state $\neg d$ of $\text{GT}_{\textit{CNF-Comp}(\Pi),\, d\rightarrow\bot}$ is a model of *CNF-Comp*$(\Pi) \wedge LF(\Pi)$. It follows that $\{\neg d\}^+ = \emptyset$ is an answer set of Π. To compare with the graph $\text{DP}_{\textit{CNF-Comp}(\Pi)}$: state d is a terminal state in $\text{DP}_{\textit{CNF-Comp}(\Pi)}$ whereas d is not a terminal state in $\text{GT}_{\textit{CNF-Comp}(\Pi),\, d\rightarrow\bot}$ because the transition rule *Test* is applicable to this state.

ASP-SAT with Backtracking (Giunchiglia et al. 2006) is a procedure that computes models of the completion of the given program using DPLL, and tests them until an answer set is found. The application of this procedure to a program Π can be viewed as constructing a path from $\emptyset$ to a terminal node in the graph $\text{GT}_{\textit{CNF-Comp}(\Pi),LF(\Pi)}$ by adopting a strategy that *Test* is applied to a state M only when M is an assignment.

$$\textit{Unit Propagate } \lambda\text{:} \quad M||\Gamma \Longrightarrow M\,l||\Gamma \;\text{ if } \; \begin{cases} C \vee l \in F \cup \Gamma \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

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

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

Fig. 3. The additional transition rules of the graph DPL_F.

6 Review: Abstract DPLL with Learning

Most modern SAT solvers implement such sophisticated techniques as backjumping and learning:

Backjumping: Chronological Backtracking (used in classical DPLL) can be seen as a prototype of Backjumping. Unlike Backtracking that undoes only the previously made decision, Backjumping is generally able to backtrack further in the search tree by undoing several decisions at once.

Learning: Most modern SAT solvers implement so called *conflict-driven backjumping and learning*: whenever backjumping is performed they add (learn) a "backjump clause" to the clause database of a solver. Learning backjump clauses prevents a solver from reaching "similar" inconsistent states.

In this section we will extend the graph DP_F to capture the ideas behind backjumping and learning. The new graph will be closely related to the *DPLL System with Learning* graph introduced in (Nieuwenhuis et al. 2006, Section 2.4).

We first note that the graph DP_F is not adequate to capture such technique as learning since it is incapable to reflect a change in a state of computation related to newly learned clauses. We start by redefining a state so that it incorporates information about changes performed on a clause database.

For a CNF formula F, an *augmented state* relative to F 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 Γ is a (multi-)set of clauses over atoms of F that are entailed by F.

We now define a graph DPL_F for any CNF formula F. Its nodes are the augmented states relative to F. The transition rules *Decide* and *Fail* of DP_F are extended to DPL_F as follows: $M||\Gamma \Longrightarrow M'||\Gamma$ ($M||\Gamma \Longrightarrow \textit{FailState}$) is an edge in DPL_F justified by *Decide* (*Fail*) if and only if $M \Longrightarrow M'$ ($M \Longrightarrow \textit{FailState}$) is an edge in DP_F justified by *Decide* (*Fail*). Figure 3 presents the other transition rules of DPL_F. We refer to the transition rules *Unit Propagate* λ, *Backjump*, *Decide*, and *Fail* of the graph DPL_F as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn* is applicable to it. We will omit the word "augmented" before "state" when this is clear from a context.

The graph DPL_F can be used for deciding the satisfiability of a formula F simply by constructing an arbitrary path from node $\emptyset||\emptyset$ to a semi-terminal node:

Proposition 8
For any CNF formula F,

(a) every path in DPL_F contains only finitely many edges justified by Basic transition rules,
(b) for any semi-terminal state $M||\Gamma$ of DPL_F reachable from $\emptyset||\emptyset$, M is a model of F,
(c) *FailState* is reachable from $\emptyset||\emptyset$ in DPL_F if and only if F is unsatisfiable.

On the one hand, Proposition 8 (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 will be eventually reached. On the other hand, Proposition 8 (b) and (c) assert that as soon as a semi-terminal state is reached the problem of deciding whether formula F is satisfiable is solved.

For instance, let F be the formula

$$a \vee b$$
$$\neg a \vee c.$$

Here is a path in DPL_F:

$$\begin{array}{ll} \emptyset||\emptyset & \Longrightarrow (\textit{Learn}) \\ \emptyset||b \vee c & \Longrightarrow (\textit{Decide}) \\ \neg b^{\Delta}||b \vee c & \Longrightarrow (\textit{Unit Propagate } \lambda) \\ \neg b^{\Delta}\, c||b \vee c & \Longrightarrow (\textit{Unit Propagate } \lambda) \\ \neg b^{\Delta}\, c\, a||b \vee c & \end{array} \tag{17}$$

Since the state $\neg b^{\Delta}\, c\, a$ is semi-terminal, Proposition 8 (b) asserts that $\{\neg b, c, a\}$ is a model of F.

Recall that the transition rule *Backtrack* of the graph DP_F – a prototype of *Backjump* – is applicable in any inconsistent state with a decision literal in DP_F. The transition rule *Backjump*, on the other hand, is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$ (the proof of this statement is similar to the proof of Lemma 2.8 from (Nieuwenhuis et al. 2006)). The application of *Backjump* where l^{Δ} is the last decision literal and l' is $\bar{l}$ can be seen as an application of *Backtrack*. This fact shows that *Backjump* is essentially a generalization of *Backtrack*. The subgraph of DP_F induced by the nodes reachable from $\emptyset$ is basically a subgraph of DPL_F.

7 Answer Set Solver with Learning

In this section we will extend the graph SM_Π to capture backjumping and learning. As a result we will be able to model the algorithms of systems SMODELS_{cc} and SUP.

7.1 Graph SML_Π

An *(augmented) state* relative to a program Π is either a distinguished state *FailState* or a pair of the form $M||\Gamma$ where M is a record relative to the set of atoms

$$\textit{Backchain False } \lambda\text{:} \quad M||\Gamma \Longrightarrow M\ \bar{l}||\Gamma \ \text{ if } \begin{cases} a \leftarrow l, B\ \in\ \Pi \cup \Gamma, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M \end{cases}$$

$$\textit{Backjump LP}\text{:} \qquad P\, l^{\Delta}\, Q||\Gamma \Longrightarrow P\, l'||\Gamma \ \text{ if } \begin{cases} P\, l^{\Delta}\, Q \text{ is inconsistent and} \\ \Pi \text{ entails } l' \vee \overline{P} \end{cases}$$

$$\textit{Learn LP}\text{:} \qquad M||\Gamma \Longrightarrow M|| \leftarrow B, \Gamma \ \text{ if } \ \Pi \text{ entails } \overline{B}$$

Fig. 4. The additional transition rules of the graph SML_Π.

occurring in Π, and Γ is a (multi-)set of constraints formed from atoms occurring in Π that are entailed by Π.

For any program Π, we will define a graph SML_Π. Its nodes are the augmented states relative to Π. The transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Unfounded*, *Decide* and *Fail* of SM_Π are extended to SML_Π as follows: $M||\Gamma \Longrightarrow M'||\Gamma$ ($M||\Gamma \Longrightarrow \mathit{FailState}$) is an edge in SML_Π justified by a transition rule T if and only if $M \Longrightarrow M'$ ($M \Longrightarrow \mathit{FailState}$) is an edge in SM_Π justified by T. Figure 4 presents the other transition rules of SML_Π.

We refer to the transition rules *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Backchain False* λ, *Unfounded*, *Backjump LP*, *Decide*, and *Fail* of the graph SML_Π as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn LP* is applicable to it.

The graph SML_Π can be used for deciding whether a program Π has an answer set by constructing a path from $\emptyset||\emptyset$ to a semi-terminal node:

Proposition 9

For any program Π,

(a) every path in SML_Π contains only finitely many edges labeled by Basic transition rules,

(b) for any semi-terminal state $M||\Gamma$ of SML_Π reachable from $\emptyset||\emptyset$, M^+ is an answer set of Π,

(c) *FailState* is reachable from $\emptyset||\emptyset$ in SML_Π if and only if Π has no answer sets.

Thus if we construct a path from $\emptyset||\emptyset$ so that Basic transition rules periodically appear in it then some semi-terminal state will be eventually reached; as soon as a semi-terminal state is reached the problem of finding an answer set is solved.

For instance, let Π be program (8). Here is a path in SML_Π with every edge

annotated by the name of a transition rule that justifies the presence of this edge in the graph :

$$\begin{array}{ll}
\emptyset||\emptyset & \Longrightarrow (\mathit{Decide}) \\
a^{\Delta}||\emptyset & \Longrightarrow (\mathit{Unit\ Propagate\ LP}) \\
a^{\Delta}\, c||\emptyset & \Longrightarrow (\mathit{All\ Rules\ Cancelled}) \\
a^{\Delta}\, c\neg b||\emptyset & \Longrightarrow (\mathit{Decide}) \\
a^{\Delta}\, c\neg b\, d^{\Delta}||\emptyset & \Longrightarrow (\mathit{Unfounded}) \\
a^{\Delta}\, c\neg b\, d^{\Delta}\neg d||\emptyset & \Longrightarrow (\mathit{Backjump\ LP}) \\
a^{\Delta}\, c\neg b\neg d||\emptyset & \Longrightarrow (\mathit{Learn\ LP}) \\
a^{\Delta}\, c\neg b\neg d||\neg a \vee \neg c \vee b \vee \neg d &
\end{array} \tag{18}$$

Since the state $a^{\Delta}\, c\neg b\neg d$ is semi-terminal, Proposition 9 (b) asserts that

$$\{a, c, \neg b, \neg d\}^{+} = \{a, c\}$$

is an answer set for Π.

Proof of Proposition 9 is in Section 10.1.

As in case of the graphs DP_F and DPL_F, *Backjump LP* is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$ (Proposition 12 from Section 9), and is essentially a generalization of the transition rule *Backtrack* of the graph SM_{Π}.

Modern SAT solvers often implement such sophisticated techniques as restart and forgetting in addition to backjumping and learning:

Restart: A solver restarts the DPLL procedure whenever the search is not making "enough" progress. The idea is that upon a restart a solver will explore a new part of the search space using the clauses that have been learned.

Forgetting: This technique is usually implemented in relation with conflict-driven backjumping and learning. When a solver "notes" that earlier learned clauses are not helpful anymore it removes (forgets) them from the clause database. Forgetting allows a solver to avoid a possible exponential space blow-up introduced by learning.

We may extend the graph SML_{Π} with the following transition rules that capture the ideas behind these technique:

$$\begin{array}{ll}
\mathit{Restart}: & M||\Gamma \Longrightarrow \emptyset||\Gamma \\
\mathit{Forget\ LP}: & M|| \leftarrow B, \Gamma \Longrightarrow M||\Gamma.
\end{array}$$

The transition rules *Restart* and *Forget LP* are similar to the analogous rules in (Nieuwenhuis et al. 2006) for extending DPLL procedure with restart and forgetting techniques. It is easy to prove a result similar to Proposition 9 for the graph SML_{Π} with *Restart* and *Forget LP* (for such graph a state is semi-terminal if no rule other than *Learn LP*, *Restart*, *Forget LP* is applicable to it.)

7.2 Smodels$_{cc}$ and Sup Algorithms

In Section 4.2 we demonstrated a method for specifying the algorithm of an answer set solver by means of the graph SM_{Π}. In particular, we described the SMODELS algorithm by assigning priorities to transition rules of SM_{Π}. In this section we use

this method to describe the SMODELS$_{cc}$ (Ward and Schlipf 2004) and SUP (Lierler 2008) algorithms by means of SML$_\Pi$.

System SMODELS$_{cc}$ enhances the SMODELS algorithm with conflict-driven backjumping and learning. Its strategy for choosing a path in the graph SML$_\Pi$ is similar to that of SMODELS. System SMODELS$_{cc}$ assigns priorities to inference rules of SML$_\Pi$ as follows:

Backjump LP, *Fail* $\gg$
Unit Propagate LP, *All Rules Cancelled*, *Backchain True*, *Backchain False* $\lambda \gg$
Unfounded $\gg$ *Decide*.

Also, SMODELS$_{cc}$ always applies the transition rule *Learn LP* in a non-semi-terminal state reached by an application of *Backjump LP*, because it implements conflict-driven backjumping and learning.[12]

In (Lierler 2008), we introduced the simplified SUP algorithm that relies on backtracking rather than conflict-driven backjumping and learning that are actually implemented in the system. We now present the SUP algorithm that takes these sophisticated techniques into account.

System SUP assigns priorities to inference rules of SML$_\Pi$ as follows:

Backjump LP, *Fail* $\gg$
Unit Propagate LP, *All Rules Cancelled*, *Backchain True*, *Backchain False* $\lambda \gg$
Decide $\gg$ *Unfounded*.

Similarly to SMODELS$_{cc}$, SUP always applies the transition rule *Learn LP* in a non-semi-terminal state reached by an application of *Backjump LP*. In Section 11 we discuss details on which clause is being learned during an application of *Learn LP*.

For example, let Π be program (8). Path (18) corresponds to an execution of system SUP, but does not correspond to any execution of SMODELS$_{cc}$ because for the latter *Unfounded* is a rule of higher priority than *Decide*. Here is another path in SML$_\Pi$ from $\emptyset||\emptyset$ to the same semi-terminal node:

$$\begin{array}{ll} \emptyset||\emptyset & \Longrightarrow (\mathit{Decide}) \\ a^\Delta||\emptyset & \Longrightarrow (\mathit{Unit\ Propagate\ LP}) \\ a^\Delta\, c||\emptyset & \Longrightarrow (\mathit{All\ Rules\ Cancelled}) \\ a^\Delta\, c \neg b||\emptyset & \Longrightarrow (\mathit{Unfounded}) \\ a^\Delta\, c \neg b \neg d||\emptyset & \end{array} \tag{19}$$

Path (19) corresponds to an execution of system SMODELS$_{cc}$, but does not correspond to any execution of system SUP because for the latter *Decide* is a rule of higher priority than *Unfounded*.

The strategy of SUP of assigning the transition rule *Unfounded* the lowest priority may be reasonable for many problems. For instance, it is easy to see that transition rule *Unfounded* is redundant for tight programs. The SUP algorithm is similar

[12] System SMODELS$_{cc}$ (SUP) also implements restarts and forgetting that may be modeled by the transition rules *Restart* and *Forget LP*. An application of these transition rules in SML$_\Pi$ relies on particular heuristics implemented by the solver.

to SAT-based answer set solvers such as ASSAT (Lin and Zhao 2004) and CMODELS (Giunchiglia et al. 2006) (see Section 8.2) in the fact that it will first compute a supported model of a program and only then will test whether this model is indeed an answer set, i.e., whether *Unfounded* is applicable in this state.

8 Generate and Test with Learning

In this section we model backjumping and learning for the *generate and test* procedure by defining a graph $\mathrm{GTL}_{F,G}$ that extends $\mathrm{GT}_{F,G}$ (Section 5) in a similar manner as DPL_F (Section 6) extends DP_F.

8.1 Graph $\mathrm{GTL}_{F,G}$

An *(augmented) state* relative to a CNF formula F and a formula G formed from atoms occurring in F is either a distinguished state *FailState* or a pair of the form $M||\Gamma$, where M is a record (Section 2.1) relative to the set of atoms occurring in F, and Γ is a (multi-)set of clauses formed from atoms occurring in F that are entailed by $F \wedge G$.

The nodes of the graph $\mathrm{GTL}_{F,G}$ are the augmented states relative to a CNF formula F and a formula G formed from atoms occurring in F. The edges of $\mathrm{GTL}_{F,G}$ are described by the transition rules *Unit Propagate* λ, *Decide*, *Fail* of DPL_F, the transition rules

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

$$\textit{Learn } GT: \quad M||\Gamma \Longrightarrow M||\, C, \Gamma \ \text{ if } \ \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ and} \\ F \wedge G \models C \end{cases}$$

and the transition rule *Test* of $\mathrm{GT}_{F,G}$ that is extended to $\mathrm{GTL}_{F,G}$ as follows: $M||\Gamma \Longrightarrow M'||\Gamma$ is an edge in $\mathrm{GTL}_{F,G}$ justified by *Test* if and only if $M \Longrightarrow M'$ is an edge in $\mathrm{GT}_{F,G}$ justified by *Test*.

We refer to the transition rules *Unit Propagate* λ, *Test*, *Decide*, *Fail*, *Backjump GT* of the graph $\mathrm{GTL}_{F,G}$ as *Basic*. We say that a node in the graph is *semi-terminal* if no rule other than *Learn GT* is applicable to it.

The graph $\mathrm{GTL}_{F,G}$ can be used for deciding whether a formula $F \wedge G$ has a model by constructing a path from $\emptyset||\emptyset$ to a terminal node:

Proposition 10
For any CNF formula F and a formula G formed from atoms occurring in F,

(a) every path in $\mathrm{GTL}_{F,G}$ contains only finitely many edges labeled by Basic transition rules,
(b) for any semi-terminal state $M||\Gamma$ of $\mathrm{GTL}_{F,G}$ reachable from $\emptyset||\emptyset$, M is a model of $F \wedge G$,
(c) *FailState* is reachable from $\emptyset||\emptyset$ in $\mathrm{GTL}_{F,G}$ if and only if $F \wedge G$ is unsatisfiable.

As in case of the graph DPL_F, the transition rule *Backjump GT* is applicable in any inconsistent state with a decision literal that is reachable from $\emptyset||\emptyset$. We call such states *backjump* states.

Proposition 11
For any CNF formula F and a formula G formed from atoms occurring in F, the transition rule *Backjump GT* is applicable in any backjump state in $\text{GTL}_{F,G}$.

8.2 Cmodels Algorithm

System CMODELS implements an algorithm called ASP-SAT with Learning (Giunchiglia et al. 2006) that extends ASP-SAT with Backtracking by backjumping and learning.

The application of CMODELS to a program Π can be viewed as constructing a path from $\emptyset||\emptyset$ to a terminal node in the graph $\text{GTL}_{F,G}$, where

- F is the completion of Π converted to conjunctive normal form, and
- G is $LF(\Pi)$ defined in Section 5.

In Sections 4.2 we demonstrated a method for specifying the algorithm of an answer set solver by means of the graph SM_Π. We use this method to describe the CMODELS algorithm using the graph $\text{GTL}_{F,G}$. System CMODELS assigns priorities to the inference rules of $\text{GTL}_{F,G}$ as follows:

$$\textit{Backjump GT}, \textit{Fail} \gg \textit{Unit Propagate } \lambda \gg \textit{Decide} \gg \textit{Test}.$$

Also, CMODELS always applies the transition rule *Learn GT* in a non-semi-terminal state reached by an application of *Backjump GT*.

The priorities imposed on the rules by CMODELS guarantee that the transition rule *Test* is applied to a model of $F \cup \Gamma$ (clausified completion F extended by learned clauses Γ). This allows CMODELS to proceed with its search in case if a found model is not an answer set. Furthermore, the CMODELS strategy guarantees that in a state reached by an application of *Test*, first *Backjump GT* will be applied and then in the resulting state *Learn GT* will be applied. The clause learned due to this application of *Learn GT* is derived by means of loop formulas (see (Giunchiglia et al. 2006)). In this sense CMODELS uses loop formulas to guide its search.

Systems SAG (Lin et al. 2006) and CLASP (Gebser et al. 2007) are answer set solvers that are enhancements of CMODELS. First, they compute and clausify program's completion and then use unit propagate on resulting propositional formula as an inference mechanism. Second, they guide their search by means of loop formulas. Third, they implement conflict-driven backjumping and learning. Also, SAG uses SAT solvers for search. The systems differ from CMODELS in the following:

- they maintain the data structure representing an input logic program through out the whole computation,
- in addition to implementing inference rules of the graph $\text{GTL}_{F,G}$ they also implement the inference rule *Unfounded* of SM_Π. A hybrid graph combining the inference rule *Unfounded* of SM_Π and the inference rules of $\text{GTL}_{F,G}$ may be used to describe the SAG and CLASP algorithms.

System SAG assigns the same priorities to the inference rules of the hybrid graph as CMODELS. Also, SAG at random decides whether to apply the inference rule *Unfounded* in a state.

On the other hand, system CLASP assigns priorities to the inference rules of the hybrid graph as follows:

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

Like CMODELS, both SAG and CLASP always apply the transition rule *Learn GT* in a non-semi-terminal state reached by an application of *Backjump GT*.

9 Backjumping and Extended Graph

Recall the transition rule *Backjump LP* of SML_Π

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

A state in the graph SML_Π is a *backjump state* if it is inconsistent, contains a decision literal, and is reachable from $\emptyset||\emptyset$. Note that it may be not clear a priori whether *Backjump LP* is applicable to a backjump state and if so to which state the edge due to the application of *Backjump LP* leads. These questions are important if we want to base an algorithm on this framework. It turns out that *Backjump LP* is always applicable to a backjump state:

Proposition 12
For a program Π, the transition rule *Backjump LP* is applicable to any backjump state in SML_Π.

Proposition 12 guarantees that a backjump state in SML_Π is never semi-terminal. In the end of this section we show how Proposition 12 can be derived from the results proved later in this paper. Next question to answer is how to continue choosing a path in the graph after reaching a backjump state. To answer this question we introduce the notions of reason and extended graph.

For a program Π, we say that a clause $l \vee C$ is a *reason* for l to be in a list of literals $P\, l\, Q$ w.r.t Π if Π entails $l \vee C$ and $\overline{C} \subseteq P$. We can equivalently restate the second condition of *Backjump LP* "Π entails $l' \vee \overline{P}$" as "there exists a reason for l' to be in $P\, l'$ w.r.t. Π" (note that $l' \vee \overline{P}$ is a reason for l' to be in $P\, l'$). We call a reason for l' to be in $P\, l'$ a *backjump clause*. Note that Proposition 12 asserts that a backjump clause always exists for a backjump state. It is clear that we may continue choosing a path in the graph after reaching a backjump state if we know how to compute a backjump clause for this state. We now define a graph $\text{SML}_\Pi^{\uparrow}$ that shares many properties of SML_Π but allows us to give a simpler procedure for computing a backjump clause.

An *extended record* M relative to a program Π is a list of literals over the set of atoms occurring in Π where

(i) each literal l in M is annotated either by Δ or by a reason for l to be in M w.r.t. Π,

(ii) M contains no repetitions,
(iii) for any inconsistent prefix of M its last literal is annotated by a reason.

For instance, let Π be the program

$$a \leftarrow \ not\ b$$
$$c.$$

The list of literals

$$b^{\Delta}\, a^{\Delta} \neg b^{\neg b \vee \neg a}$$

is an extended record relative to Π. On the other hand, the lists of literals

$$a^{\Delta} \neg a^{\Delta} \qquad a^{\Delta} \neg b^{\neg b \vee \neg a}\, b^{\Delta} \qquad b^{\Delta}\, a^{\Delta} \neg b^{\neg b \vee \neg a}\, c^{\Delta}$$

are not extended records.

An *extended state* relative to a program Π is either a distinguished state *FailState* or a pair of the form $M||\Gamma$ where M is an extended record relative to Π, and Γ is the same as in the definition of an augmented state (i.e., Γ is a (multi-)set of constraints formed from atoms occurring in Π that are entailed by Π.) It is easy to see that for any extended state S relative to a program Π, the result of removing annotations from all nondecision literals of S is a state of SML_Π: we will denote this state by $S^{\downarrow}$.

For instance, consider program $a \leftarrow \ not\ b$. All pairs

$$\mathit{FailState} \quad \emptyset||\emptyset \qquad a^{\Delta} \neg b^{\neg b \vee \neg a}||\emptyset \qquad \neg a^{\Delta}\, b^{b \vee a}||\emptyset$$

are among valid extended states relative to this program. The corresponding states $S^{\downarrow}$ are

$$\mathit{FailState} \qquad \emptyset||\emptyset \qquad a^{\Delta} \neg b||\emptyset \qquad \neg a^{\Delta}\, b||\emptyset.$$

We now define a graph $\text{SML}^{\uparrow}_\Pi$ for any program Π. Its nodes are the extended states relative to Π. The transition rules of SML_Π are extended to $\text{SML}^{\uparrow}_\Pi$ as follows: $S_1 \Longrightarrow S_2$ is an edge in $\text{SML}^{\uparrow}_\Pi$ justified by a transition rule T if and only if $S_1^{\downarrow} \Longrightarrow S_2^{\downarrow}$ is an edge in SML_Π justified by T.

We will omit the word "extended" before "record" and "state" when this is clear from a context.

The following lemma formally states the relationship between nodes of the graphs SML_Π and $\text{SML}^{\uparrow}_\Pi$:

Lemma 1
For any program Π, if S' is a state reachable from $\emptyset||\emptyset$ in the graph SML_Π then there is a state S in the graph $\text{SML}^{\uparrow}_\Pi$ such that $S^{\downarrow} = S'$.

The definitions of Basic transition rules and semi-terminal states in $\text{SML}^{\uparrow}_\Pi$ are similar to their definitions for SML_Π.

Proposition 9$^{\uparrow}$
For any program Π,

(a) every path in $\text{SML}^{\uparrow}_\Pi$ contains only finitely many edges labeled by Basic transition rules,

(b) for any semi-terminal state $M||\Gamma$ of $\text{SML}_\Pi^\uparrow$, M^+ is an answer set of Π,
(c) $\text{SML}_\Pi^\uparrow$ contains an edge leading to *FailState* if and only if Π has no answer sets.

Note that Proposition $9^\uparrow$ (b), unlike Proposition 9 (b), is not limited to semi-terminal states that are reachable from $\emptyset||\emptyset$. As in the case of the graph SML_Π, $\text{SML}_\Pi^\uparrow$ can be used for deciding whether a program Π has an answer set. Furthermore, the new graph provides the means for computing a backjump clause that permits practical application of the transition rule *Backjump LP*: Sections 10.3 and 11 describe the *BackjumpClause* (Algorithm 1) and *BackjumpClauseFirstUIP* (Algorithm 2) procedures that compute *Decision* and *FirstUIP* backjump clauses respectively.

We say that a state in the graph $\text{SML}_\Pi^\uparrow$ is a *backjump state* if its record is inconsistent and contains a decision literal. Unlike the definition of a backjump state in SML_Π, this definition does not require a backjump state to be reachable from $\emptyset||\emptyset$ in $\text{SML}_\Pi^\uparrow$. As in case of the graph SML_Π, any backjump state in $\text{SML}_\Pi^\uparrow$ is not semi-terminal:

Proposition 12$^\uparrow$
For a program Π, the transition rule *Backjump LP* is applicable to any backjump state in $\text{SML}_\Pi^\uparrow$.

Proposition 12 easily follows from Lemma 1 and Proposition $12^\uparrow$.

Next section will present the proofs for Proposition $9^\uparrow$, Lemma 1, and Proposition $12^\uparrow$. It is interesting to note that the proofs of Lemma 1 and Proposition $12^\uparrow$ implicitly provide the means for choosing a path in the graph $\text{SML}_\Pi^\uparrow$:

- given a state $M||\Gamma$ and a transition rule *Unit Propagate LP*, *All Rules Cancelled*, *Backchain True*, *Backchain False* λ, or *Unfounded* applicable to $M||\Gamma$, the proof of Lemma 1 describes a clause that may be used to construct a record M' so that there is an edge $M||\Gamma \Longrightarrow M'||\Gamma$ due to this transition rule,
- given a backjump state $M||\Gamma$, the proof of Proposition $12^\uparrow$ describes a backjump clause that can be used to construct a record M' so that there is an edge $M||\Gamma \Longrightarrow M'||\Gamma$ due to *Backjump LP*.

Furthermore, the construction of the proof of Proposition $12^\uparrow$ paves the way for procedure *BackjumpClause* presented in Algorithm 1.

10 Proofs of Proposition $9^\uparrow$, Lemma 1, Proposition $12^\uparrow$

10.1 Proof of Proposition $9^\uparrow$

Lemma 2
For any program Π, an extended record M relative to Π, and every assignment X such that X^+ is an answer set for Π, if X satisfies all decision literals in M then $X \models M$.

Proof
By induction on the length of M. The property trivially holds for $\emptyset$. We assume that the property holds for any state with n elements. Consider any state M with $n+1$ elements. Let X be an assignment such that X^+ is an answer set for Π and X satisfies all decision literals in M. We will now show that $X \models M$.

Case 1. M has the form $P\, l^{\Delta}$. By the inductive hypothesis, $X \models P$. Since X satisfies all decision literals in M, $X \models l$.

Case 2. M has the form $P\, l^{l \vee C}$. By the inductive hypothesis, $X \models P$. By the definition of a reason, (i) Π entails $l \vee C$ and (ii) $\overline{C} \subseteq P$. From (ii) it follows that $P \models \neg C$. Consequently, $X \models \neg C$. From (i) it follows that for any assignment X such that X^+ is an answer set, $X \models l \vee C$. Consequently, $X \models l$. $\square$

The proof of Proposition $9^{\uparrow}$ assumes the correctness of Proposition $12^{\uparrow}$ that we demonstrate in Section 10.3. Note that Proposition 9 (b), (c) easily follow from Lemma 1 and Proposition $9^{\uparrow}$ (b), (c). Proof of Proposition 9 (a) is similar to the proof of Proposition $9^{\uparrow}$ (a).

Proposition $9^{\uparrow}$
For any program Π,

(a) every path in $\text{SML}^{\uparrow}_{\Pi}$ contains only finitely many edges labeled by Basic transition rules,
(b) for any semi-terminal state $M||\Gamma$ of $\text{SML}^{\uparrow}_{\Pi}$, M^+ is an answer set of Π,
(c) $\text{SML}^{\uparrow}_{\Pi}$ contains an edge leading to *FailState* if and only if Π has no answer sets.

Proof
(a) For any list N of literals by $|N|$ we denote the length of N. Any state $M||\Gamma$ has the form $M_0\, l_1^{\Delta} M_1 \ldots l_p^{\Delta} M_p||\Gamma$, where $l_1^{\Delta} \ldots l_p^{\Delta}$ are all decision literals of M; we define $\alpha(M||\Gamma)$ as the sequence of nonnegative integers $|M_0|, |M_1|, \ldots, |M_p|$, and $\alpha(\mathit{FailState}) = \infty$. For any states S and S' of $\text{SML}^{\uparrow}_{\Pi}$, we understand $\alpha(S) < \alpha(S')$ as the lexicographical order. We first note that for any state $M||\Gamma$, value of α is based only on the first component M of the state. Second, there is a finite number of distinct values of α due to the fact that there is a finite number of distinct Ms over Π. We conclude that there is a finite number of distinct values of α for the states of $\text{SML}^{\uparrow}_{\Pi}$, even though the number of distinct states in $\text{SML}^{\uparrow}_{\Pi}$ is infinite.

By the definition of the transition rules of $\text{SML}^{\uparrow}_{\Pi}$, if there is an edge from $M||\Gamma$ to $M'||\Gamma'$ in $\text{SML}^{\uparrow}_{\Pi}$ formed by any Basic transition rule then $\alpha(M||\Gamma) < \alpha(M'||\Gamma')$. Then, due to the fact that there is a finite number of distinct values of α, it follows that there is only a finite number of edges due to the application of Basic rules possible in any path.
(b) Let $M||\Gamma$ be a semi-terminal state so that none of the Basic rules are applicable. From the fact that *Decide* is not applicable, we conclude that M assigns all literals.

Furthermore, M is consistent. Indeed, assume that M is inconsistent. Then, since *Fail* is not applicable, M contains a decision literal. Consequently, $M||\Gamma$ is a backjump state. By Proposition $12^{\uparrow}$, the transition rule *Backjump LP* is applicable in $M||\Gamma$. This contradicts our assumption that $M||\Gamma$ is semi-terminal.

Also, M is a model of Π: since *Unit Propagate LP* is not applicable in $M||\Gamma$, it follows that for every rule $a \leftarrow B \in \Pi$, if $B \subseteq M$ then $a \in M$.

Assume that M^+ is not an answer set. Then, by Proposition 2, there is a non-empty unfounded set U on M w.r.t. Π such that $U \subseteq M$. It follows that *Unfounded* is applicable (with an arbitrary $a \in U$) in $M||\Gamma$. This contradicts the assumption that $M||\Gamma$ is semi-terminal.

(c) Left-to-right: There is a state $M||\Gamma$ in $\text{SML}_\Pi^\uparrow$ such that there is an edge between $M||\Gamma$ and *FailState*. By the definition of $\text{SML}_\Pi^\uparrow$, this edge is due to the transition rule *Fail*. Consequently, state $M||\Gamma$ is such that M is inconsistent and contains no decision literals. By Lemma 2, for every assignment X such that X^+ is an answer set for Π, X satisfies M. Since M is inconsistent we conclude that Π has no answer sets.

Right-to-left: Consider the process of constructing a path consisting only of edges due to Basic transition rules. By (a), it follows that this path will eventually reach a semi-terminal state. By (b), this semi-terminal state cannot be different from *FailState*, because Π has no answer sets. We conclude that there is an edge leading to *FailState*. □

10.2 Proof of Lemma 1

The proof uses the notion of loop formula (Lin and Zhao 2004).

Given a set A of atoms by $Bodies(\Pi, A)$ we denote the set that consists of the elements of $Bodies(\Pi, a)$ for all a in A. Let Π be a program. For any set Y of atoms, the *external support formula* (Lee 2005) for Y is

$$\bigvee_{B \in Bodies(\Pi, Y),\, B^+ \cap Y = \emptyset} B. \tag{20}$$

We will denote the external support formula by $ES_{\Pi, Y}$. For any set Y of atoms, the *loop formula* for Y is the implication

$$\bigvee_{a \in Y} a \rightarrow ES_{\Pi, Y}.$$

We can rewrite this formula as the disjunction

$$\bigwedge_{a \in Y} \neg a \vee ES_{\Pi, Y}. \tag{21}$$

From the *Main Theorem* in (Lee 2005) we conclude:

Lemma on Loop Formulas
For any program Π, Π entails loop formulas (21) for all sets Y of atoms that occur in Π.

For a state S in the graph $\text{SML}_\Pi^\uparrow$, we say that $S^\downarrow$ in SML_Π is the *image* of S.

Lemma 1
For any program Π, if S' is a state reachable from $\emptyset||\emptyset$ in the graph SML_Π then there is a state S in the graph $\text{SML}_\Pi^\uparrow$ such that $S^\downarrow = S'$.

Proof
Since the property trivially holds for the initial state $\emptyset||\emptyset$, we only need to prove that all transition rules of SML_Π preserve it.

Consider an edge $M||\Gamma \Longrightarrow M'||\Gamma'$ in the graph SML_Π such that there is a state $M_1||\Gamma$ in the graph $\text{SML}_\Pi^\uparrow$ satisfying the condition $(M_1||\Gamma)^\downarrow = M||\Gamma$. We need to show that there is a state in the graph $\text{SML}_\Pi^\uparrow$ such that $M'||\Gamma'$ is its image in SML_Π. Consider several cases that correspond to a transition rule leading from $M||\Gamma$ to $M'||\Gamma'$:

Unit Propagate LP: $M||\Gamma \Longrightarrow M\ a||\Gamma$ if $\left\{\begin{array}{l} a \leftarrow B \in \Pi \text{ and} \\ B \subseteq M. \end{array}\right.$

$M'||\Gamma'$ is $M\,a||\Gamma$. It is sufficient to prove that $M_1\,a^{a \vee \overline{B}}||\Gamma$ is a state of $\text{SML}_\Pi^\uparrow$. It is enough to show that a clause $a \vee \overline{B}$ is a reason for a to be in $M\,a$. By applicability conditions of *Unit Propagate LP*, $B \subseteq M$. Since Π entails its rule $a \leftarrow B$, Π entails $a \vee \overline{B}$.

All Rules Cancelled: $M||\Gamma \Longrightarrow M\ \neg a||\Gamma$ if $\overline{B} \cap M \neq \emptyset$ for all $B \in \mathit{Bodies}(\Pi, a)$. $M'||\Gamma'$ is $M\neg a||\Gamma$. Consider any $B \in \mathit{Bodies}(\Pi, a)$. Since $\overline{B} \cap M \neq \emptyset$, B contains a literal from $\overline{M}$: call it $f(B)$. It is sufficient to show that

$$\neg a \vee \bigvee_{B \in \mathit{Bodies}(\Pi, a)} f(B) \tag{22}$$

is a reason for $\neg a$ to be in $M\neg a$.

First, by the choice of $f(B)$, $f(B) \in \overline{M}$; consequently,

$$\overline{\bigvee_{B \in \mathit{Bodies}(\Pi, a)} f(B)} \subseteq M.$$

Second, since $f(B) \in B$, the loop formula $\neg a \vee \mathit{ES}_{\Pi,\{a\}}$ entails (22). By *Lemma on Loop Formulas*, it follows that Π entails (22).

Backchain True: $M||\Gamma \Longrightarrow M\,l||\Gamma$ if $\left\{\begin{array}{l} a \leftarrow B \in \Pi, \\ a \in M, \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus \{B\}, \\ l \in B. \end{array}\right.$

$M'||\Gamma'$ is $M\,l||\Gamma$. Consider any $B' \in \mathit{Bodies}(\Pi, a) \setminus B$. Since $\overline{B}' \cap M \neq \emptyset$, B' contains a literal from $\overline{M}$: call it $f(B')$. A clause

$$l \vee \neg a \vee \bigvee_{B' \in \mathit{Bodies}(\Pi, a) \setminus B} f(B')\cdot \tag{23}$$

is a reason for l to be in $M\,l$. The proof of this statement is similar to the case of *All Rules Cancelled*.

Backchain False λ: $M||\Gamma \Longrightarrow M\ \overline{l}||\Gamma$ if $\left\{\begin{array}{l} a \leftarrow l, B\ \in\ \Pi \cup \Gamma, \\ \neg a \in M \text{ or } a = \bot, \\ B \subseteq M. \end{array}\right.$

$M'||\Gamma'$ is $M\ \overline{l}||\Gamma$. A clause $\overline{l} \vee \overline{B} \vee a$ is a reason for $\overline{l}$ to be in $M\overline{l}$. The proof of this statement is similar to the case of *Unit Propagate LP*.

Unfounded: $M||\Gamma \Longrightarrow M\,\neg a||\Gamma$ if $\left\{\begin{array}{l} M \text{ is consistent and} \\ a \in U \text{ for a set } U \text{ unfounded on } M \text{ w.r.t. } \Pi. \end{array}\right.$

$M'||\Gamma'$ is $M\ \neg a||\Gamma$. Consider any $B \in \mathit{Bodies}(\Pi, U)$ such that $U \cap B^+ = \emptyset$. By the definition of an unfounded set, it follows that $\overline{B} \cap M \neq \emptyset$. Consequently, B contains a literal from $\overline{M}$: call it $f(B)$. The clause

$$\neg a \vee \bigvee_{\mathit{Bodies}(\Pi,U),\, B^+ \cap U = \emptyset} f(B) \tag{24}$$

is a reason for $\neg a$ to be in $M\,\neg a$. The proof of this statement is similar to the case of *All Rules Cancelled*.

Backjump LP, *Decide*, *Fail*, and *Learn LP*: obvious. □

The process of turning a state of SML_Π reachable from $\emptyset||\emptyset$ into a corresponding state of $\text{SML}_\Pi^\uparrow$ can be illustrated by the following example: Consider a program Π

$$\begin{array}{ll} a \leftarrow \mathit{not}\ b & k \leftarrow l,\ \mathit{not}\ b \\ b \leftarrow \mathit{not}\ a,\ \mathit{not}\ c & \leftarrow m,\ \mathit{not}\ l,\ \mathit{not}\ b \\ c \leftarrow \mathit{not}\ f & m \leftarrow\ \mathit{not}\ k,\ \mathit{not}\ l \\ \leftarrow k, d & \end{array} \tag{25}$$

and a path in SML_Π

$$\begin{array}{l} \emptyset||\emptyset \Longrightarrow (\mathit{Decide}) \\ a^\Delta||\emptyset \Longrightarrow (\mathit{All\ Rules\ Cancelled}) \\ a^\Delta \neg b||\emptyset \Longrightarrow (\mathit{Decide}) \\ a^\Delta \neg b\, c^\Delta||\emptyset \Longrightarrow (\mathit{Backchain\ True}) \\ a^\Delta \neg b\, c^\Delta \neg f||\emptyset \Longrightarrow (\mathit{Decide}) \\ a^\Delta \neg b\, c^\Delta \neg f\, d^\Delta||\emptyset \Longrightarrow (\mathit{Backchain\ False}\ \lambda) \\ a^\Delta \neg b\, c^\Delta \neg f\, d^\Delta \neg k||\emptyset \Longrightarrow (\mathit{Backchain\ False}\ \lambda) \\ a^\Delta \neg b\, c^\Delta \neg f\, d^\Delta \neg k \neg l||\emptyset \Longrightarrow (\mathit{Backchain\ False}\ \lambda) \\ a^\Delta \neg b\, c^\Delta \neg f\, d^\Delta \neg k \neg l \neg m||\emptyset \Longrightarrow (\mathit{Unit\ Propagate\ LP}) \\ a^\Delta \neg b\, c^\Delta \neg f\, d^\Delta \neg k \neg l \neg m\, m||\emptyset \end{array} \tag{26}$$

The construction in the proof of Lemma 1 applied to the nodes in this path gives following states of $\text{SML}_\Pi^\uparrow$:

$$\begin{array}{l} \emptyset||\emptyset \\ a^\Delta||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}\, d^\Delta||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}\, d^\Delta \neg k^{\neg k \vee \neg d}||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}\, d^\Delta \neg k^{\neg k \vee \neg d} \neg l^{\neg l \vee b \vee k}||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}\, d^\Delta \neg k^{\neg k \vee \neg d} \neg l^{\neg l \vee b \vee k} \neg m^{\neg m \vee l \vee b}||\emptyset \\ a^\Delta \neg b^{\neg b \vee \neg a}\, c^\Delta \neg f^{\neg f \vee \neg c}\, d^\Delta \neg k^{\neg k \vee \neg d} \neg l^{\neg l \vee b \vee k} \neg m^{\neg m \vee l \vee b}\, m^{m \vee k \vee l}||\emptyset \end{array} \tag{27}$$

It is clear that these nodes form a path in $\text{SML}_\Pi^\uparrow$ with every edge justified by the same transition rule as the corresponding edge in path (26) in SML_Π.

10.3 Proof of Proposition 12[↑]

In this section Π is an arbitrary and fixed logic program.

For a record M, by $lcp(M)$ we denote its largest consistent prefix. We say that a clause C is *conflicting* on a list M of literals if Π entails C, and $\overline{C} \subseteq lcp(M)$. For example, let M be the first component of the last state in (27):

$$a^{\Delta}\, \neg b^{\neg b \vee \neg a}\, c^{\Delta}\, \neg f^{\neg f \vee \neg c}\, d^{\Delta}\, \neg k^{\neg k \vee \neg d}\, \neg l^{\neg l \vee b \vee k}\, \neg m^{\neg m \vee l \vee b}\, m^{m \vee k \vee l}. \tag{28}$$

Then, $lcp(M)$ is obtained by dropping the last element $m^{m \vee k \vee l}$ of M. It is clear that the reason $m \vee k \vee l$ for m to be in M is a conflicting clause on M.

Lemma 3
The literal that immediately follows $lcp(M)$ in an inconsistent record M, has the form l^C where C is a conflicting clause on M.

For any inconsistent record $l_1 \cdots l_n$ and any conflicting clause C on this record, by $\beta_{l_1 \cdots l_n}(C)$ we denote the set of numbers i such that $l_i \in \overline{C}$. (It is clear that every element from $\overline{C}$ equals to one of the literals in $l_1 \cdots l_n$.) The relation $I < J$ between subsets I, J of $\{1 \cdots n\}$ is understood here as the lexicographical order between I and J sorted in descending order. For instance, $\{2\ 6\ 7\} < \{6\ 7\ 8\}$ because $\{7\ 6\ 2\} < \{8\ 7\ 6\}$ in lexicographical order.

Recall that the *resolution rule* can be applied to clauses $C \vee l$ and $C' \vee \neg l$ and produces the clause $C \vee C'$, called the *resolvent* of $C \vee l$ and $C' \vee \neg l$ on l.

Lemma 4
Let M be a record and let l^B be a nondecision literal from $lcp(M)$. If clause D is the resolvent of B and a clause C conflicting on M then

(i) D is a clause conflicting on M,
(ii) $\beta_M(D) < \beta_M(C)$.

For instance, let M be (28), let reason $\neg m \vee l \vee b$ for $\neg m$ in $lcp(M)$ be B, and let conflicting clause $m \vee k \vee l$ on M be C. Then D, the result of resolving B together with C, is clause $k \vee l \vee b$. Lemma 4 asserts that $k \vee l \vee b$ is a conflicting clause on M and that $\beta_M(D) < \beta_M(C)$. Indeed, $\beta_M(D) = \{2\ 6\ 7\}$ and $\beta_M(C) = \{6\ 7\ 8\}$.

Let record M be $l_1 \cdots l_i \cdots l_n$, the *decision level* of a literal l_i is the number of decision literals in $l_1 \cdots l_i$: we denote it by $dec_M(l_i)$. We will also use this notation to denote the decision level of a set of literals: For a set $P \subseteq M$ of literals, $dec_M(P)$ is the decision level of the literal in P that occurs latest in M. For record M and a decision level j by M^j we denote the prefix of M that consists of the literals in M that belong to decision level less than j and by $M^{j]}$ we denote the prefix of M that consists of the literals in M that belong to decision level less than or equal to j. For instance, let M be record (28) then $dec_M(\neg k) = 3$, $dec_M(\neg b\ c\ \neg k) = 3$, M^3 is $a^{\Delta}\, \neg b^{\neg b \vee \neg a}\, c^{\Delta}\, \neg f^{\neg f \vee \neg c}$, and $M^{3]}$ is M itself.

Lemma 5
For an inconsistent record M and a conflicting clause $l \vee C$ on M, if $dec_M(\bar{l}) > dec_M(\overline{c})$ for all $c \in C$ then $lcp(M)^{dec_M(\overline{C})]}\, l^{l \vee C}$ is a record.

Proposition 12$^{\uparrow}$
For a program Π, the transition rule *Backjump LP* is applicable to any backjump state in $\text{SML}^{\uparrow}_{\Pi}$.

Proof
Let $M||\Gamma$ be a backjump state in $\text{SML}^{\uparrow}_{\Pi}$. Let R be the list of reasons that are assigned to the nondecision literals in $lcp(M)$.

Consider the process of building a sequence $C_1, C_2, \ldots$ of clauses so that

- C_1 is the reason of the member of M that immediately follows $lcp(M)$, and
- C_j $(j > 1)$ is a resolvent of C_{j-1} and some clause in R

while derivation of new clauses is possible. From Lemma 4 (i) and the choice of C_1 and R, it follows that any clause in $C_1, C_2 \ldots$ is conflicting. By Lemma 4 (ii) we conclude that $\beta_M(C_j) < \beta_M(C_{j-1})$ $(j > 1)$. It is clear that this process will terminate after deriving some clause C_m, since the number of conflicting clauses on M is finite. It is clear that clause C_m cannot be resolved against any clause in R.

Case 1. C_m is the empty clause. Since $M||\Gamma$ is a backjump state, M contains a decision literal l^{Δ}. By part (iii) of the definition of a record, l belongs to $lcp(M)$. Consequently, M can be represented in the form $lcp(M)^{dec_M(l)}\, l^{\Delta}\, Q$.

By the choice of C_1, C_1 is a reason and must consist of at least one literal. Consequently, $m > 1$. Clause C_m is derived from clauses C_{m-1} and some clause in R. Since C_m is empty, C_{m-1} is a unit clause l'. We will show that

$$lcp(M)^{dec_M(l)}\, l^{\Delta}\, Q||\Gamma \Longrightarrow lcp(M)^{dec_M(l)}\, l'^{l'}||\Gamma$$

is an application of *Backjump LP*. It is sufficient to demonstrate that $lcp(M)^{dec_M(l)}\, l'^{l'}$ is a record. Since $lcp(M)^{dec_M(l)}\, l^{\Delta}\, Q$ is a record, we only need to show that $l' \notin lcp(M)^{dec_M(l)}$ and clause l' is a reason for l' to be in $lcp(M)^{dec_M(l)}\, l'$. Recall that C_{m-1}, i.e., l', is a conflicting clause. Consequently, Π entails l' and $\overline{l'} \in lcp(M)$. Since $lcp(M)$ is consistent, $l' \notin lcp(M)$ so that $l' \notin lcp(M)^{dec_M(l)}$. On the other hand, from the fact that Π entails l' it immediately follows that clause l' is a reason for l' to be in $lcp(M)^{dec_M(l)}\, l'$.

Case 2. C_m is not empty. Since C_m is a conflicting clause on M, the complement of any literal in C_m belongs to $lcp(M)$. Furthermore, every such complement is a decision literal in $lcp(M)$. Indeed, if this complement is $\overline{l}^{\overline{l} \vee B} \in lcp(M)$ then $\overline{l} \vee B$ is one of the clauses B_i, and it can be resolved against C_m.

By the definition of a decision level, there is at most one decision literal that belongs to any decision level. It follows that C_m can be written as $l \vee C'_m$ so that $dec_M(\overline{l}) > dec_M(\overline{c})$ for any $c \in C'_m$. Consequently, M can be written as $lcp(M)^{dec_M(\overline{l})}\, \overline{l}^{\Delta}\, Q$. Note that

$$lcp(M)^{dec_M(\overline{l})}\, \overline{l}^{\Delta}\, Q||\Gamma \Longrightarrow lcp(M)^{dec_M(\overline{C'_m})]}\, l^{C_m}||\Gamma$$

is an application of *Backjump LP*. Indeed, by Lemma 5 $lcp(M)^{dec_M(\overline{C}'_m)]}\, l^{C_m}$ is a record. $\square$

BackjumpClause$(M||\Gamma)$;
Arguments : $M||\Gamma$ is a backjump state
Return Value : C is a backjump clause
begin
  $C \leftarrow$ the reason of the member of M that immediately follows $lcp(M)$;
  $N \leftarrow$ the list of the nondecision literals in $lcp(M)$;
  $R \leftarrow$ the list of the reasons that are assigned to the literals in N;
  while $\overline{C} \cap N \neq \emptyset$ **do**
    $l \leftarrow$ a literal in $\overline{C} \cap N$;
    $B \leftarrow$ the clause in R that contains l;
    $C' \leftarrow$ the resolvent of C and B on l;
    if $C' = \emptyset$ **then**
      return C
    $C \leftarrow C'$
  return C ;
end

Algorithm 1: A procedure for generating a backjump clause.

Algorithm 1 presents procedure *BackjumpClause* that computes a backjump clause for any backjump state in the graph $\text{SML}^{\uparrow}_{\Pi}$. The algorithm follows from the construction of the proof of Proposition $12^{\uparrow}$. It is based on the iterative application of the resolution rule on reasons of the smallest inconsistent prefix of a state. The proof of Proposition $12^{\uparrow}$ allows to conclude the termination of *BackjumpClause* and asserts that a clause returned by the procedure is a backjump clause on a backjump state.

For instance, let Π be (25). Consider an execution of *BackjumpClause* on Π and backjump state (28). The table below gives the values of $lcp(M)$, C, N, and R during the execution of the *BackjumpClause* algorithm. By C_i we denote a value of C before the i-th iteration of the **while** loop.

$lcp(M)$	$a^{\Delta}\,\neg b^{\neg b\vee\neg a}\,c^{\Delta}\,\neg f^{\neg f\vee\neg c}\,d^{\Delta}\,\neg k^{\neg k\vee\neg d}\,\neg l^{\neg l\vee b\vee k}\,\neg m^{\neg m\vee l\vee b}$
C_1	$m \vee k \vee l$
N	$\neg b^{\neg b\vee\neg a}\,\neg f^{\neg f\vee\neg c}\,\neg k^{\neg k\vee\neg d}\,\neg l^{\neg l\vee b\vee k}\,\neg m^{\neg m\vee l\vee b}$
R	$\neg b \vee \neg a,\ \neg f \vee \neg c,\ \neg k \vee \neg d,\ \neg l \vee b \vee k,\ \neg m \vee l \vee b$
C_2	$k \vee l \vee b$ is the resolvent of C_1 and $\neg m \vee l \vee b$
C_3	$k \vee b$ is the resolvent of C_2 and $\neg l \vee b \vee k$
C_4	$\neg d \vee b$ is the resolvent of C_3 and $\neg k \vee \neg d$
C_5	$\neg d \vee \neg a$ is the resolvent of C_4 and $\neg b \vee \neg a$

(29)

The algorithm will terminate with the clause $\neg d \vee \neg a$. Proof of Proposition $12^{\uparrow}$ asserts that (i) this clause is a backjump clause such that d and a are decision literals in M and (ii) the transition

$$a^{\Delta}\neg b^{\neg b\vee\neg a}\,c^{\Delta}\neg f^{\neg f\vee\neg c}\,d^{\Delta}\neg k^{\neg k\vee\neg d}\neg l^{\neg l\vee b\vee k}\neg m^{\neg m\vee l\vee b}\,m^{m\vee k\vee l}||\emptyset \Longrightarrow \\ a^{\Delta}\,\neg b^{\neg b\vee\neg a}\,\neg d^{\neg d\vee\neg a}||\emptyset \tag{30}$$

in $\text{SML}^{\uparrow}{}_{\Pi}$ is an application of *Backjump LP*. Indeed, by Lemma 5 $lcp(M)^{dec_M(\overline{\neg a})]}\,\neg d^{\neg d\vee\neg a}$, in other words $a^{\Delta}\neg b^{\neg b\vee\neg a}\neg d^{\neg d\vee\neg a}$, is a record.

Note that a backjump clause may be derived in other ways than captured by *BackjumpClause* algorithm: the transition rule *Backjump LP* is applicable with an arbitrary backjump clause. Usually, DPLL-like procedures implement conflict-driven backjumping and learning where a particular learning schema such as, for instance, *Decision* or *FirstUIP* (Mitchell 2005) is applied for computing a special kind of a backjump clause. It turns out that the *BackjumpClause* algorithm captures the *Decision* learning schema for ASP. Typically, SAT solvers impose an order for resolving the literals during the process of *Decision* backjump clause derivation. We can impose similar order by replacing the line

$$l \leftarrow \text{a literal in } \overline{C} \cap N$$

in the algorithm *BackjumpClause* with

$$l \leftarrow \text{a literal in } \overline{C} \cap N \text{ that occurs latest in } lcp(M).$$

In fact, the sample application of *BackjumpClause* algorithm described in (29) follows this ordering.

11 FirstUIP Conflict-Driven Backjumping and Learning

Conflict-driven backjumping and learning proved to be a highly successful technique in modern SAT solving. Furthermore, in (Zhang et al. 2001) the authors investigated the performance of various learning schemes and established experimentally that *FirstUIP* clause is the most useful single clause to learn. Success of conflict-driven learning led to the implementation of its ASP counterpart in systems SMODELS$_{cc}$, CLASP, and SUP. There are two common methods for describing a backjump clause construction in the SAT literature. The first one employes the implication graph (Marques-Silva and Sakallah 1996) and the second one employes resolution (Mitchell 2005). Ward and Schlipf (Ward and Schlipf 2004) extended the definition of an implication graph to the SMODELS algorithm and implemented *FirstUIP* learning schema in answer set solver SMODELS$_{cc}$. In the previous section we used SML$^{\uparrow}_{\Pi}$ formalism and resolution to describe the *BackjumpClause* algorithm for computing an ASP counterpart of a *Decision* backjump clause. In (Gebser et al. 2007) the authors used the concepts from constraint programming to implement *FirstUIP* learning schema in answer set solver CLASP.

The Algorithm 2 presents *BackjumpClauseFirstUIP* procedure for computing an ASP counterpart of a *FirstUIP* backjump clause by means of SML$^{\uparrow}_{\Pi}$ formalism and resolution. The algorithm computes a *FirstUIP* backjump clause for any backjump state in the graph SML$^{\uparrow}_{\Pi}$. *BackjumpClauseFirstUIP* is employed by the system SUP in its implementation of conflict-driven backjumping and learning.

We now state the correctness of the algorithm *BackjumpClauseFirstUIP*. We start by showing its termination. By C_1 we will denote the initial value assigned to clause C. From Lemma 4 (i) and the choice of C_1 we conclude that at any point of computation clause C is conflicting on M. By Lemma 4 (ii), the value of $\beta_M(C)$ decreases with each new assignment of clause C in the **while** loop. It follows that the **while** loop will terminate since the number of conflicting clauses C on M such

BackjumpClauseFirstUIP$(M||\Gamma)$
Arguments : $M||\Gamma$ is a backjump state
Return Value : C is a backjump clause
begin
  $C \leftarrow$ the reason of the member of M that immediately follows $lcp(M)$
  $l \leftarrow$ the literal in $\overline{C}$ that occurs latest in $lcp(M)$
  $P \leftarrow$ the sublist of $lcp(M)$ that consists of the literals that belong to the decision level $dec(l)$
  $R \leftarrow$ the list of the reasons that are assigned to the literals in P
  while $|\overline{C} \cap P| > 1$ **do**
    $l \leftarrow$ the literal in $\overline{C}$ that occurs latest in P
    $B \leftarrow$ the clause in R that contains l
    $C \leftarrow$ the resolvent of C and B on l
  return C
end

Algorithm 2: A procedure for generating a FirstUIP backjump clause.

that $|\overline{C} \cap P| > 1$ is finite. By C_m we will denote the clause C with which the **while** loop terminates. In other words *BackjumpClauseFirstUIP* returns C_m. We now show that C_m is indeed a backjump clause. We already concluded that C_m is a conflicting clause on M. Furthermore, from the termination condition of the **while** loop $|\overline{C_m} \cap P| \leq 1$. From the choice of C_1 and P it follows that $|\overline{C_m} \cap P| = 1$. Consequently, C_m can be written as $l \vee C'_m$ where $\overline{l}$ is in singleton $\overline{C_m} \cap P$. By Lemma 4 (ii), $\beta(C_m) \leq \beta(C_1)$. From the definition of β and the choice of P it follows that $dec_M(\overline{l}) > dec_M(\overline{c})$ for all $c \in C'_m$. By Lemma 5, $lcp(M)^{dec_M(\overline{C'_m})]}\, l^{C_m}$ is a record. In other words, transition

$$M||\Gamma \Longrightarrow lcp(M)^{dec_M(\overline{C'_m})]}\, l^{C_m}||\Gamma$$

is an application of *Backjump LP*. Consequently, C_m is a backjump clause.

For instance, let Π be (25). Consider an execution of *BackjumpClauseFirstUIP* on Π and a backjump state (28). The table below gives the values of $lcp(M)$, C, P, and R during the execution of *BackjumpClauseFirstUIP*. By C_i we denote a value of C before the i-th iteration of the **while** loop.

$lcp(M)$	$a^{\Delta}\, \neg b^{\neg b \vee \neg a}\, c^{\Delta}\, \neg f^{\neg f \vee \neg c}\, d^{\Delta}\, \neg k^{\neg k \vee \neg d}\, \neg l^{\neg l \vee b \vee k}\, \neg m^{\neg m \vee l \vee b}$
C_1	$m \vee k \vee l$
P	$d^{\Delta}\, \neg k^{\neg k \vee \neg d}\, \neg l^{\neg l \vee b \vee k}\, \neg m^{\neg m \vee l \vee b}$
R	$\neg k \vee \neg d$, $\neg l \vee b \vee k$, $\neg m \vee l \vee b$
C_2	$k \vee l \vee b$ is the resolvent of C_1 and $\neg m \vee l \vee b$
C_3	$k \vee b$ is the resolvent of C_2 and $\neg l \vee b \vee k$.

The *BackjumpClauseFirstUIP* algorithm will terminate with the clause $k \vee b$.

12 Extended Graph: Generate and Test

In this section we introduce an extended graph $\mathrm{GTL}^{\uparrow}_{F,G}$ for the *generate and test* abstract framework $\mathrm{GTL}_{F,G}$ similar as in Section 9 we introduced $\mathrm{SML}^{\uparrow}_{\Pi}$ for SML_{Π}.

For a formula H, we say that a clause $l \vee C$ is a *reason* for l to be in a list $P\,l\,Q$ of literals w.r.t. H if $H \models l \vee C$ and $\overline{C} \subseteq P$.

An *(extended) record* M relative to a formula H is a list of literals over the set of atoms occurring in H where

(i) each literal l in M is annotated either by Δ or by a reason for l to be in M w.r.t. H,
(ii) M contains no repetitions,
(iii) for any inconsistent prefix of M its last literal is annotated by a reason.

An *(extended) state* relative to a CNF formula F, and a formula G formed from atoms occurring in F is either a distinguished state *FailState* or a pair of the form $M||\Gamma$, where M is an extended record relative to $F \wedge G$, and Γ is the same as in the definition of an augmented state (i.e., Γ is a (multi-)set of clauses formed from atoms occurring in F that are entailed by $F \wedge G$.) For any extended state S relative to F and G, the result of removing annotations from all nondecision literals of S is a state of $\mathrm{GTL}_{F,G}$: we will denote this state by $S^{\downarrow}$.

For a CNF formula F and a formula G formed from atoms occurring in F, we will define a graph $\mathrm{GTL}^{\uparrow}_{F,G}$. The set of the nodes of $\mathrm{GTL}^{\uparrow}_{F,G}$ consists of the extended states relative to F and G. The transition rules of $\mathrm{GTL}_{F,G}$ are extended to $\mathrm{GTL}^{\uparrow}_{F,G}$ as follows: $S_1 \Longrightarrow S_2$ is an edge in $\mathrm{GTL}^{\uparrow}_{F,G}$ justified by a transition rule T if and only if $S_1^{\downarrow} \Longrightarrow S_2^{\downarrow}$ is an edge in $\mathrm{GTL}_{F,G}$ justified by T.

The lemma below formally states the relationship between nodes of the graphs $\mathrm{GTL}_{F,G}$ and $\mathrm{GTL}^{\uparrow}_{F,G}$:

Lemma 6
For any CNF formula F and a formula G formed from atoms occurring in F, if S' is a state reachable from $\emptyset||\emptyset$ in the graph $\mathrm{GTL}_{F,G}$ then there is a state S in the graph $\mathrm{GTL}^{\uparrow}_{F,G}$ such that $S^{\downarrow} = S'$.

The definitions of Basic transition rules and semi-terminal states in $\mathrm{GTL}^{\uparrow}_{F,G}$ are similar to their definitions for $\mathrm{GTL}_{F,G}$.

Proposition 10$^{\uparrow}$
For any CNF formula F and a formula G formed from atoms occurring in F,

(a) every path in $\mathrm{GTL}^{\uparrow}_{F,G}$ contains only finitely many edges labeled by Basic transition rules,
(b) for any semi-terminal state $M||\Gamma$ of $\mathrm{GTL}^{\uparrow}_{F,G}$, M is a model of $F \wedge G$,
(c) $\mathrm{GTL}^{\uparrow}_{F,G}$ contains an edge leading to *FailState* if and only if $F \wedge G$ is unsatisfiable.

A state in the graph $\mathrm{GTL}^{\uparrow}_{F,G}$ is a *backjump state* if its record is inconsistent and contains a decision literal. Any backjump state in $\mathrm{GTL}^{\uparrow}_{F,G}$ is not semi-terminal:

Proposition 11$^{\uparrow}$
For any CNF formula F and a formula G formed from atoms occurring in F, the transition rule *Backjump GT* is applicable in any backjump state in $\mathrm{GTL}^{\uparrow}_{F,G}$.

Algorithms *BackjumpClause* and *BackjumpClauseFirstUIP* are applicable to the backjump states of the graph $\mathrm{GTL}^{\uparrow}_{F,G}$.

13 Related Work

Simons (2000) and Ward (2004) described the SMODELS and SMODELS$_{cc}$ algorithms, respectively, by means of pseudocode and demonstrated their correctness. In this paper we designed an abstract framework that was used as an alternative method for describing these algorithms and demonstrating their correctness.

Gebser and Schaub (2006) provided a deductive system for describing inferences involved in computing answer sets by tableaux methods. The abstract framework presented here can be viewed as a deductive system also, but of a very different kind. First, it accounts for phenomena such as backjumping and learning (and also forgetting and restart) whereas the Gebser-Schaub system does not. Second, we describe backtracking by an inference rule, and the Gebser-Schaub system does not. Accordingly, the derivations considered in this paper describe search process, and derivations in the Gebser-Schaub system do not. Also, the abstract framework discussed here does not have any inference rule similar to Cut; this is why its derivations are paths, rather than trees.

14 Conclusions

In this paper we showed how to model advanced algorithms for computing answer sets of a program by means of simple mathematical objects, graphs. We extended the abstract frameworks proposed in (Lierler 2008) for describing native and SAT-based ASP algorithms to capture such sophisticated features as backjumping and learning. We characterized the algorithms of systems SMODELS$_{cc}$, SUP, and CMODELS that implement these features. We note that the work on this abstract framework helped us design the new answer set solver SUP, and preliminary experimental analysis showed that SUP is a competitive representative in the family of answer set solvers. We hope that in the future this framework will suggest designs of other systems for computing answer sets. The abstract approach to describing algorithms simplifies the analysis of their correctness and allows us to study the relationship between various algorithms by analyzing the differences in strategies of choosing a path in the graph. For example, the description of the SMODELS$_{cc}$ and SUP algorithms in this framework reflects their differences in a simple manner via distinct assignments of priorities to edges of the graph that characterize these systems. Also we used this framework to describe two algorithms for computing *Decision* and *FirstUIP* backjump clauses for the implementation of conflict-driven backjumping and learning in answer set solvers. This formalism provided the transparent means for specifying these algorithms. We believe that the development of this abstract framework powerful enough to describe advanced features of answer set solvers in a simple manner will promote the use of these sophisticated features in more solvers.

Acknowledgments

We are grateful to Marco Maratea for bringing to our attention the work by Nieuwenhuis et al. (2006), to Vladimir Lifschitz for the numerous discussions, to

Mirosław Truszczyński for valuable ideas about the SUP algorithm, to Martin Gebser, and Michael Gelfond for important comments, to anonymous referees for their suggestions. The author was supported by the National Science Foundation under Grant IIS-0712113.

References

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

Davis, M., Logemann, G., and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM 5(7)*, 394–397.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

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'07)*. MIT Press, 386–392.

Gebser, M. and Schaub, T. 2006. Tableau calculi for answer set programming. In *Proceedings of 22nd International Conference on Logic Programming (ICLP'06)*. Springer, 11–25.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

Giunchiglia, E., Lierler, Y., and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36*, 345–377.

Giunchiglia, E. and 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'05)*. Springer, 37–51.

Gomes, C. P., Kautz, H., Sabharwal, A., and Selman, B. 2008. Satisfiability solvers. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 89–134.

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

Lierler, Y. 2008. Abstract answer set solvers. In *Proceedings of International Conference on Logic Programming (ICLP'08)*. Springer, 377–391.

Lierler, Y. 2010. Abstract answer set solvers with learning (long version). arxiv:1001.0820v1 [cs.ai].

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*. MIT Press, 1594–1597.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*. MIT Press, 112–117.

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

Lin, Z., Zhang, Y., and Hernandez, H. 2006. Fast SAT-based answer set solver. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*. MIT Press, 92–97.

Marques-Silva, J. P. and Sakallah, K. A. 1996. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of IEEE Conference on Tools with Artificial Intelligence*.

Mitchell, D. G. 2005. A SAT solver primer. In *EATCS Bulletin (The Logic in Computer Science Column)*. Vol. 85. 112–133.

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.

Ricca, F., Faber, W., and Leone, N. 2006. A backjumping technique for disjunctive logic programming. *AI Commun. 19,* 2, 155–172.

Saccá, D. and Zaniolo, C. 1990. Stable models and non-determinism in logic programs with negation. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*. 205–217.

Simons, P. 2000. Extending and implementing the stable model semantics. Ph.D. thesis, Helsinki University of Technology. Adviser-Niemelä, Ilkka.

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. 2004. Answer set programming with clause learning. [13]. Ph.D. thesis. Adviser-Long, Timothy J. and Adviser-Schlipf, Johns S.

Ward, J. and Schlipf, J. 2004. Answer set programming with clause learning. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*. 302–313.

Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malik, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings ICCAD-01*. 279–285.

[13] `http://www.nku.edu/~wardj1/research/thesis.pdf`