ECAI 2014
T. Schaub et al. (Eds.)
© 2014 The Authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-419-0-165

Abstract Disjunctive Answer Set Solvers

Remi Brochenin[1] and **Yuliya Lierler**[2] and **Marco Maratea**[3]

Abstract. A fundamental task in answer set programming is to compute answer sets of logic programs. Answer set solvers are the programs that perform this task. The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete. The high complexity of reasoning within disjunctive logic programming is responsible for few solvers capable of dealing with such programs, namely DLV, GNT, CMODELS and CLASP. We show that transition systems introduced by Nieuwenhuis, Oliveras, and Tinelli to model and analyze satisfiability solvers can be adapted for disjunctive answer set solvers. In particular, we present transition systems for CMODELS (without backjumping and learning), GNT and DLV (without backjumping). The unifying perspective of transition systems on satisfiability and non-disjunctive answer set solvers proved to be an effective tool for analyzing, comparing, proving correctness of each underlying search algorithm as well as bootstrapping new algorithms. Given this, we believe that this work will bring clarity and inspire new ideas in design of more disjunctive answer set solvers.

1 Introduction

Answer set programming (ASP) is a declarative programming paradigm oriented towards difficult combinatorial search problems [20, 21]. ASP has been applied to many areas of science and technology, from the design of a decision support system for the Space Shuttle [24] to graph-theoretic problems arising in zoology and linguistics [1]. A fundamental task in ASP is to compute answer sets of logic programs. Answer set solvers are the programs that perform this task. There were sixteen answer set solvers participating in the Fourth Answer Set Programming Competition in 2013[4].

Gelfond and Lifschitz introduced logic programs with disjunctive rules [8]. The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete [3]. The high complexity of reasoning within disjunctive logic programming stems from two sources: (i) there is an exponential number of possible candidate models, and (ii) the hardness of checking whether a candidate model is an answer set of a propositional disjunctive logic program is co-NP-complete. Only four answer set systems allow programs with disjunctive rules: DLV [13], GNT [10], CMODELS [14] and CLASP [6].

Recently, several formal approaches have been used to describe and compare search procedures implemented in answer set solvers. These approaches range from a pseudo-code representation of the procedures [9], to tableau calculi [7], to abstract frameworks via transition systems [17, 18]. The last method proved to be particularly suited for the goal. It originates from the work by Nieuwenhuis et al. [23], where authors proposed to use transition systems to describe the DPLL (Davis-Putnam-Logemann-Loveland) procedure [2]. They introduced an abstract framework – a DPLL graph – that captures what "states of computation" are, and what transitions between states are allowed. Every execution of the DPLL procedure corresponds to a path in the DPLL graph. Lierler and Truszczynski [17, 18] adapted this approach to describing answer set solvers for *non-disjunctive* programs including SMODELS, CMODELS, and CLASP. Such an abstract way of presenting algorithms simplifies the analysis of their correctness and facilitates formal reasoning about their properties, by relating algorithms in precise mathematical terms.

In this paper we present transition systems that account for *disjunctive* answer set solvers implementing plain backtracking. We define abstract frameworks for CMODELS (without backjumping and learning), GNT and DLV (without backjumping). We also identify a close relationship between answer set solvers DLV and CMODELS by means of properties of the related graphs. We believe that this work will bring better understanding of the main design features of current disjunctive answer set solvers as well as inspire new algorithms.

The paper is structured as follows. Sec. 2 introduces needed preliminaries. Sec. 3, 4 and 5 show the abstract frameworks of CMODELS, GNT and DLV, respectively. The paper ends in Sec. 6 by discussing related works and with final remarks.

2 Preliminaries

Formulas, Logic Programs, and Program's Completion *Atoms* are Boolean variables over $\{true, false\}$. The symbols $\bot$ and $\top$ are the *false* and the *true* constant, respectively. The letter l denotes a literal, that is an atom a or its negation $\neg a$, and $\bar{l}$ is the complement of l, i.e., literal a for $\neg a$ and literal $\neg a$ for a. *Propositional formulas* are logical expressions defined over atoms and symbols $\bot$, $\top$ that take value in the set $\{true, false\}$. A finite disjunction of literals, is a *clause*. We identify an empty clause with the clause $\bot$. A *CNF formula* is a conjunction (alternatively, a set) of clauses. A conjunction (disjunction) of literals will sometimes be seen as a set, containing each of its literals. Given a conjunction (disjunction) B of literals, by $\overline{B}$ we denote the disjunction (conjunction) of the complements of the elements of B. For example, $\overline{a \vee \neg b}$ denotes $\neg a \wedge b$, while $\overline{a \wedge \neg b}$ denotes $\neg a \vee b$. A *(truth) assignment* to a set X of atoms is a function from X to $\{false, true\}$. A *satisfying assignment* or a *model* for a formula F is an assignment M such that F evaluates to *true* under M. If F evaluates to *false* under M, we say that M contradicts F. If F has no model we say that F is *unsatisfiable*. We often identify a consistent set L of literals (i.e., a set that does not contain complementary literals, for example, a and $\neg a$) with an assignment as follows: if $a \in L$ then a maps to *true*, while if $\neg a \in L$ then a maps to *false*. We also identify a set X of atoms over $At(\Pi)$ with an assignment as follows: if $a \in X$ then a maps to *true*, while if $a \in At(\Pi) \setminus X$ then a maps to *false*.

[1] University of Genova, Italy, email: remi.brochenin@unige.it
[2] University of Nebraska at Omaha, email: ylierler@unomaha.edu
[3] University of Genova, Italy, email: marco@dibris.unige.it
[4] `https://www.mat.unical.it/aspcomp2013/Participants`

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

$$a_1 \vee \ldots \vee a_i \leftarrow a_{i+1}, \ldots, a_j, \mathit{not}\ a_{j+1}, \ldots, \mathit{not}\ a_k, \\ \mathit{not}\ \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n, \tag{1}$$

where $a_1, \ldots, a_n$ are atoms. The left hand side expression of a rule is called the *head*. We call rule (1) *non-disjunctive* if its head contains not more than one atom. A program is non-disjunctive if it consists of non-disjunctive rules. The letter B often denotes the body

$$a_{i+1}, \ldots, a_j, \mathit{not}\ a_{j+1}, \ldots, \mathit{not}\ a_k,\ \ \mathit{not}\ \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n \tag{2}$$

of a rule (1). We often identify (2) with the conjunction

$$a_{i+1} \wedge \ldots \wedge a_j \wedge \neg a_{j+1} \wedge \ldots \wedge \neg a_k \wedge a_{k+1} \wedge \ldots \wedge a_n.$$

We identify the rule (1) with the clause

$$a_1 \vee \ldots \vee a_i \vee \neg a_{i+1} \vee \cdots \vee \neg a_j \vee \\ a_{j+1} \vee \cdots \vee a_k \vee \neg a_{k+1} \vee \cdots \vee \neg a_n. \tag{3}$$

This allows us to sometimes view a program Π as a CNF formula.

It is important to note the presence of doubly negated atoms in the bodies of rules. This version of logic programs is a special case of programs with nested expressions introduced by Lifschitz et al. [19]. A *choice rule* [22] construct $\{a\} \leftarrow B$, originally employed in the LPARSE[5] and GRINGO[6] languages, can be seen as an abbreviation for a rule $a \leftarrow B,\ \mathit{not}\ \mathit{not}\ a$ [5]. In this work we adopt this abbreviation. We sometime write (1) as

$$A \leftarrow D, F \tag{4}$$

where A is $a_1 \vee \ldots \vee a_i$, D is $a_{i+1}, \ldots, a_j$, and F is

$$\mathit{not}\ a_{j+1}, \ldots, \mathit{not}\ a_k, \mathit{not}\ \mathit{not}\ a_{k+1}, \ldots, \mathit{not}\ \mathit{not}\ a_n.$$

The *reduct* Π^X of a disjunctive program Π w.r.t. a set X of atoms is obtained from Π by deleting each rule (4) such that $X \not\models F$ and replacing each remaining rule (4) with $A \leftarrow D$. A set X of atoms is an *answer set* of Π if X is minimal among the sets of atoms that satisfy Π^X. For any consistent and complete set M of literals, 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 rule $a \leftarrow B$ in Π.

The *completion* $\mathit{Comp}(\Pi)$ of a program Π is a formula

$$\mathit{Comp}(\Pi) = \Pi \cup \{\neg a \vee \bigvee_{C \vee a \leftarrow B \in \Pi} (B \wedge \overline{C}),\ a \in \mathit{At}(\Pi)\}$$

where by $\mathit{At}(\Pi)$ we denote the set of atoms occurring in Π. This formula has the property that any answer set of Π is a model of $\mathit{Comp}(\Pi)$. The converse does not hold in general.

Abstract DPLL. The Davis-Putnam-Logemann-Loveland (DPLL) procedure [2] is a well-known method that exhaustively explores assignments to generate models of a propositional formula. Most modern satisfiability and answer set solvers are based on variations of the DPLL procedure. We now review the abstract transition system for DPLL proposed by Nieuwenhuis et al. [23]. This framework provides an alternative to common pseudo-code descriptions of backtrack search based algorithms.

For a set X of atoms, a *record* relative to X is a string L composed of literals over X or symbol $\bot$ without repetitions where some literals are annotated by Δ. The annotated literals are called *decision* literals. We say that a record L is *inconsistent* if it contains both a literal l and its complement $\bar{l}$, or if it contains $\bot$. We will sometime identify a record with the set containing all its elements disregarding its annotations. For example, we will identify a record $b^\Delta\ \neg a$ with the set $\{\neg a, b\}$ of literals.

A *state* relative to X is either the distinguished state $\mathit{Failstate}$, a record relative to X, or $\mathit{Ok}(L)$ where L is a record relative to X. For instance, states relative to a singleton set $\{a\}$ include

$$\mathit{Failstate},\ \emptyset,\ \bot,\ a\,\bot,\ \bot\,a,\ a,\ \neg a,\ a^\Delta,\ \neg a^\Delta,\ a\,\neg a \\ a^\Delta\,\neg a,\ a\,\neg a^\Delta,\ a^\Delta\,\neg a^\Delta,\ \neg a\,a,\ \neg a^\Delta\,a,\ \neg a\,a^\Delta, \mathit{Ok}(a).$$

Each CNF formula F determines its DPLL *graph* DP_F. The set of nodes of DP_F consists of the states relative to the set of atoms occurring in F. The edges of the graph DP_F are specified by the transition rules:[7]

$\mathit{UnitPropagate}$:

$L \Longrightarrow Ll$ if $\begin{cases} C \vee l \text{ is a clause in } F \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{cases}$

Decide :

$L \Longrightarrow Ll^\Delta$ if $\begin{cases} L \text{ is consistent and} \\ \text{neither } l \text{ nor } \bar{l} \text{ occur in } L \end{cases}$

$\mathit{Conclude}$:

$L \Longrightarrow \mathit{Failstate}$ if $\begin{cases} L \text{ is inconsistent and} \\ L \text{ contains no decision literals} \end{cases}$

$\mathit{Backtrack}$:

$Ll^\Delta L' \Longrightarrow L\bar{l}$ if $\begin{cases} Ll^\Delta L' \text{ is inconsistent and} \\ L' \text{ contains no decision literals} \end{cases}$

OK :

$L \Longrightarrow \mathit{Ok}(L)$ if no other rule applies

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

The following theorem gathers key properties of the graph DP_F.

Theorem 1 (Proposition 1 in [17]) *For any CNF formula F,*

1. *graph DP_F is finite and acyclic,*
2. *any terminal state reachable from $\emptyset$ in DP_F other than $\mathit{Failstate}$ is $\mathit{Ok}(L)$, with L being a model of F,*
3. *Failstate is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable.*

Thus, to decide the satisfiability of a CNF formula F it is enough to find a path leading from node $\emptyset$ to a terminal node. If it is a $\mathit{Failstate}$, F is unsatisfiable. Otherwise, F is satisfiable. For instance, let $F = \{a \vee b, \neg a \vee c\}$. Below we show a path in DP_F with every edge annotated by the name of the transition rule that gives rise to this edge in the graph (UP abbreviates $\mathit{UnitPropagate}$):

$$\emptyset \overset{\mathit{Decide}}{\Longrightarrow} a^\Delta \overset{UP}{\Longrightarrow} a^\Delta\,c \overset{\mathit{Decide}}{\Longrightarrow} a^\Delta\,c\,b^\Delta \overset{OK}{\Longrightarrow} \mathit{Ok}(a^\Delta\,c\,b^\Delta). \tag{5}$$

The state $\mathit{Ok}(a^\Delta\ c\,b^\Delta)$ is terminal. Thus, Theorem 1 asserts that F is satisfiable and $\{a, c, b\}$ is a model of F. Here is another path to the same terminal state

$$\emptyset \overset{\mathit{Decide}}{\Longrightarrow} a^\Delta \overset{\mathit{Decide}}{\Longrightarrow} a^\Delta\,\neg c^\Delta \overset{UP}{\Longrightarrow} a^\Delta\,\neg c^\Delta\,c \\ \overset{\mathit{Backtrack}}{\Longrightarrow} a^\Delta\,c \overset{\mathit{Decide}}{\Longrightarrow} a^\Delta\,c\,b^\Delta \overset{OK}{\Longrightarrow} \mathit{Ok}(a^\Delta\,c\,b^\Delta). \tag{6}$$

A path in the graph DP_F is a description of a process of search for a model of a CNF formula F. The process is captured via applications of transition rules. Therefore, we can characterize the algorithm

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

[7] Recall that, given the definition of a record, a state may have a form Ll only if a literal l or l^Δ is not already in L.

of a solver that utilizes the transition rules of DP_F by describing a strategy for choosing a path in this graph. A strategy can be based on assigning priorities to transition rules of DP_F so that a solver never applies a rule in a state if a rule with higher priority is applicable to the same state. The DPLL procedure is captured by the following priorities

$$Conclude, Backtrack >> UnitPropagate >> Decide.$$

Path (5) complies with the DPLL priorities. Thus it corresponds to an execution of DPLL. Path (6) does not: it uses $Decide$ when $UnitPropagate$ is applicable.

Disjunctive Answer Set Solvers: Discussion The problem of deciding whether a disjunctive program has an answer set is Σ_2^P-complete [3]. This is because: (i) there is an exponential number of possible candidate models, and (ii) the hardness of checking whether a candidate model is an answer set of a disjunctive program is co-NP-complete. The latter condition differentiates disjunctive answer set solving procedures from answer set solvers for non-disjunctive programs. Informally, a disjunctive (answer set) solver requires two "layers" of computation – two solving engines: one that *generates* candidate models, and another that *tests* candidate models. Existing disjunctive solvers differ in underlying technology for each of the solving engines. System CMODELS uses instances of SAT solvers for each of the tasks. System GNT uses instances of non-disjunctive answer set solver SMODELS. System DLV uses the SMODELS-like procedure to generate candidate models, and instances of SAT solvers to test candidate models. These substantial differences obscure the thorough analysis and understanding of similarities and differences between the existing disjunctive solvers. To elevate this difficulty, we generalize the graph-based framework for capturing DPLL-like procedures to the case of disjunctive answer set solving.

3 Abstract CMODELS

We start by introducing a graph $DP^2_{F,f}$ based on two instances of DPLL graph. We then describe how it can be used to capture the CMODELS procedure for disjunctive programs.

Abstract Solver via DPLL. We call a function $f : M \to F$ from a set M of literals to a CNF formula F a *witness-(formula)* function. Intuitively, a CNF formula resulting from a witness function is a *witness (formula)* with respect to M. Informally, a witness formula is what is tested by a solver after generating a candidate model so as to know whether this candidate is good.

An (extended) state relative to sets X and X' of atoms is a pair (L, R) or distinguished states $Failstate$ or $Ok(L)$, where L and R are records relative to X and X', respectively. We often drop the word extended before state, when it is clear from a context. A state $(\emptyset, \emptyset)$ is called *initial*. For a formula F, by $At(F)$ we denote the set of atoms occurring in F. For a formula F and a witness function f, by $At(F, f)$ we denote the union of $At(f(L))$ for all possible consistent records L over $At(F)$. It is not necessarily equal to $At(F)$ as f may, for instance, introduce additional variables.

We now define a graph $DP^2_{F,f}$ for a CNF formula F and a witness function f. The set of nodes of $DP^2_{F,f}$ consists of the states relative to $At(F)$ and $At(F, f)$. The edges of the graph $DP^2_{F,f}$ are specified by the transition rules presented in Figure 1. We use the following abbreviations in stating these rules. Expression $up(L, l, F)$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{Left-rules:}} \\
UnitPropagate_L & (L, \emptyset) & \Longrightarrow (Ll, \emptyset) & \text{if } up(L, l, F) \\
Decide_L & (L, \emptyset) & \Longrightarrow (Ll^{\Delta}, \emptyset) & \text{if } de(L, l, F) \\
Conclude_L & (L, \emptyset) & \Longrightarrow Failstate & \text{if } fa(L) \\
Backtrack_L & (Ll^{\Delta}L', \emptyset) & \Longrightarrow (L\bar{l}, \emptyset) & \text{if } ba(L, l, L') \\
\\
\multicolumn{4}{l}{\text{Right-rules, applicable when no left-rule applies:}} \\
UnitPropagate_R & (L, R) & \Longrightarrow (L, Rl) & \text{if } up(R, l, f(L)) \\
Decide_R & (L, R) & \Longrightarrow (L, Rl^{\Delta}) & \text{if } de(R, l, f(L)) \\
Conclude_R & (L, R) & \Longrightarrow Ok(L) & \text{if } fa(R) \\
Backtrack_R & (L, Rl^{\Delta}R') & \Longrightarrow (L, R\bar{l}) & \text{if } ba(R, l, R') \\
\\
\multicolumn{4}{l}{\text{Crossing-rules, applicable when no right-rule and no left-rule applies:}} \\
Conclude_{LR} & (L, R) & \Longrightarrow Failstate & \\
 & \multicolumn{3}{l}{\text{if } L \text{ contains no decision literal}} \\
Backtrack_{LR} & (Ll^{\Delta}L', R) & \Longrightarrow (L\bar{l}, \emptyset) & \\
 & \multicolumn{3}{l}{\text{if } L' \text{ contains no decision literal}}
\end{array}
$$

Figure 1. The transition rules of the graph $DP^2_{F,f}$.

holds when the condition of the transition rule $UnitPropagate$ of the graph DP_F holds, i.e., when

$$\begin{array}{l} C \vee l \text{ is a clause in } F \text{ and} \\ \text{all the literals of } \overline{C} \text{ occur in } L \end{array}$$

Similarly, $de(L, l, F)$, $fa(L)$, and $ba(L, l, L')$ hold when the conditions of $Decide$, $Conclude$, and $Backtrack$ of DP_F hold, respectively.

A graph $DP^2_{F,f}$ can be used for deciding whether a CNF formula F has a model M such that witness formula defined by f with respect to M is unsatisfiable.

Theorem 2 *For any CNF formula F and a witness function f:*

1. *graph $DP^2_{F,f}$ is finite and acyclic,*
2. *any terminal state of $DP^2_{F,f}$ reachable from the initial state and other than $Failstate$ is $Ok(L)$, with L being a model of F such that $f(L)$ is unsatisfiable,*
3. *$Failstate$ is reachable from the initial state if and only if F has no model such that its witness is unsatisfiable.*

This graph can be used to capture two layers of computation – *generate* and *test* – by combining two DPLL procedures as follows. The generate layer applies the DPLL procedure to a given formula F (see left-rules). It turns out that left-rules no longer apply to a state (L, R) only when L is a model for F. Thus, when a model L for F is found, then a witness formula with respect to L is built. The test layer applies the DPLL procedure to the witness formula (see right-rules). If no model is found for the witness formula, then $Conclude_R$ rule applies bringing us to a terminal state $Ok(L)$ suggesting that L represents a solution to a given search problem. It turns out that no left-rules and no right-rules apply in a state (L, R) only when R is a model for the witness formula. Thus, the set L of literals is not a solution and the DPLL procedure of the generate layer proceeds with the search (see crossing-rules).

CMODELS via the Abstract Solver. We now relate the graph $DP^2_{F,f}$ to the CMODELS procedure, DP-ASSAT-PROC, described by Lierler [14]. We start by introducing some required notation.

For a set M of literals, by M^+ we denote atoms that occur positively in M. For example, $\{\neg a, b\}^+ = \{b\}$. For set σ of atoms and set M of literals, by $M_{|\sigma}$ we denote the maximal subset of M over σ. For example, $\{a, \neg b, c\}_{|\{a,b\}} = \{a, \neg b\}$. We say that a set M of literals *covers* a set σ of atoms if for each atom a in σ either a or $\neg a$ is in M. For example, set $\{\neg a\}$ of literals covers set $\{a\}$ of atoms while $\{\neg a\}$ does not cover $\{a, b\}$. Given a program Π and a consistent set M of literals that covers $At(\Pi)$, a witness function f_{min} maps M into a formula composed of the clause $\overline{M^+}$, one clause $\neg a$ for each literal $\neg a \in M$, and the clauses of Π^{M^+}. Recall that we identify a program with a CNF formula.

Given a disjunctive program Π, the answer set solver CMODELS starts its computation by converting program's completion $Comp(\Pi)$ into a CNF formula that we call $EDcomp(\Pi)$. Lierler (Section 13.2, [16]) describes the details of the transformation. The graph $DP^2_{EDcomp(\Pi),f_{min}}$ captures the search procedure of DP-ASSAT-PROC of CMODELS. The DP-ASSAT-PROC algorithm follows the priorities on its transition rules listed below

$$\begin{array}{l} Backtrack_L, Conclude_L >> UnitPropagate_L >> Decide_L >> \\ Backtrack_R, Conclude_R >> UnitPropagate_R >> Decide_R >> \\ Backtrack_{LR}, Conclude_{LR}. \end{array}$$

A proof of correctness and termination of the DP-ASSAT-PROC procedure results from Theorem 2 and two conditions on formula $EDcomp(\Pi)$ and function f_{min}: (i) for any answer set X of Π there is a model M of $EDcomp(\Pi)$ such that $X = M^+_{|At(\Pi)}$, and (ii) for any consistent set M of literals covering $At(\Pi)$, $M^+_{|At(\Pi)}$ is an answer set of Π if and only if $f_{min}(M)$ results in an unsatisfiable formula.

We now capture,for the graph $DP^2_{EDcomp(\Pi),f_{min}}$, general properties which guarantee that a similar solving strategy that uses the DPLL procedure for generate and test layers results in a correct answer set solver. We say that a propositional formula F *DP-approximates* a program Π if for any answer set X of Π there is a model M of F such that $X = M^+_{|At(\Pi)}$. For instance, completion of Π DP-approximates Π. We say that a witness-formula function f *DP-ensures* a program Π if for any consistent set M of literals that covers $At(\Pi)$, $M^+_{|At(\Pi)}$ is an answer set of Π if and only if $f(M)$ results in an unsatisfiable formula. For example, the witness-formula function f_{min} DP-ensures Π. It turns out that for any program Π, given any formula F that DP-approximates Π and any witness function f that DP-ensures Π, the graph $DP^2_{F,f}$ captures a correct algorithm for establishing whether Π has answer sets.

Theorem 3 *For a disjunctive program* Π*, a CNF formula* F *that DP-approximates* Π*, and a witness-formula function* f *that DP-ensures* Π*,*

1. *graph* $DP^2_{F,f}$ *is finite and acyclic,*
2. *any terminal state of* $DP^2_{F,f}$ *reachable from the initial state and other than* $Failstate$ *is* $Ok(L)$*, with* $L^+_{|At(\Pi)}$ *being an answer set of* Π*,*
3. $Failstate$ *is reachable from the initial state if and only if* Π *has no answer sets.*

4 Abstract GNT

We illustrated how the graph $DP^2_{F,f}$ captures the basic CMODELS procedure. This section describes a respective graph for the procedure underlying disjunctive solver GNT. Recall that unlike solver CMODELS that uses the DPLL procedure for generating and testing, system GNT uses the SMODELS procedure – an algorithm for finding answer sets of non-disjunctive logic programs – for respective tasks. Lierler [17] introduced the graph SM$_\Lambda$ that captures the computation underlying the SMODELS algorithm just as the graph DP_F captures the computation underlying DPLL. The graph SM$_\Lambda$ forms a basis for devising the transition system suitable to describe GNT.

$$ac(L,a,\Lambda) \quad \text{if} \begin{cases} \text{for each rule } a \leftarrow B \text{ of } \Lambda \\ \quad B \text{ is contradicted by } L \end{cases}$$

$$bt(L,l,\Lambda) \quad \text{if} \begin{cases} \text{there is a rule } a \leftarrow l, B \text{ of } \Lambda \text{ such that} \\ \quad a \text{ is a literal of } L \text{ and} \\ \text{for each other rule } a \leftarrow B' \text{ of } \Lambda \\ \quad B' \text{ is contradicted by } L \end{cases}$$

$$uf(L,a,\Lambda) \quad \text{if} \begin{cases} L \text{ is consistent and} \\ \text{there is a set } M \text{ containing } a \text{ such that} \\ \quad M \text{ is unfounded on } L \text{ w.r.t. } \Lambda \end{cases}$$

Figure 2. The properties for rules of the graph $SM^2_{\Lambda,p}$.

Left-rules:				
$AllRulesCancelled_L$	$(L, \emptyset)$	$\Longrightarrow$	$(L\neg a, \emptyset)$	if $ac(L, a, \Lambda)$
$BackchainTrue_L$	$(L, \emptyset)$	$\Longrightarrow$	$(Ll, \emptyset)$	if $bt(L, l, \Lambda)$
$Unfounded_L$	$(L, \emptyset)$	$\Longrightarrow$	$(L\neg a, \emptyset)$	if $uf(L, a, \Lambda)$
Right-rules, applicable when no left-rule applies:				
$AllRulesCancelled_R$	(L, R)	$\Longrightarrow$	$(L, R\neg a)$	if $ac(R, a, p(\Lambda))$
$BackchainTrue_R$	(L, R)	$\Longrightarrow$	(L, Rl)	if $bt(R, l, p(\Lambda))$
$Unfounded_R$	(L, R)	$\Longrightarrow$	$(L, R\neg a)$	if $uf(R, a, p(\Lambda))$

Figure 3. Transition rules of the graph $SM^2_{\Lambda,p}$

Abstract Solver via SMODELS. We abuse some terminology, by calling a function $p : M \to \Lambda$ from a set M of literals to a non-disjunctive program Λ a *witness-(program)* function. Intuitively, a program resulting from a witness function is a *witness (program)* with respect to M. For a program Λ and a witness function p, by $At(\Lambda, p)$ we denote the union of $At(p(L))$ for all possible consistent records L over $At(\Lambda)$.

We now define a graph $SM^2_{\Lambda,p}$ for a non-disjunctive program Λ and a witness function p. The set of nodes of $SM^2_{\Lambda,p}$ consists of the states relative to $At(\Lambda)$ and $At(\Lambda, p)$. The edges of the graph $SM^2_{\Lambda,p}$ are specified by the transition rules of the $DP^2_{\Lambda,p}$ graph extended with the transition rules presented in Figure 3 and based on the properties listed in Figure 2. We refer the reader to [12] for the definition of "unfounded" sets.

A graph $SM^2_{\Lambda,p}$ can be used for deciding whether a non-disjunctive program Λ has an answer set X such that witness program defined by $p(X)$ has no answer sets.

Theorem 4 *For any non-disjunctive program* Λ *and a witness function* p*:*

1. *graph* $SM^2_{\Lambda,p}$ *is finite and acyclic,*
2. *any terminal state of* $SM^2_{\Lambda,p}$ *reachable from the initial state and other than* $Failstate$ *is* $Ok(L)$*, with* L^+ *being an answer set of* Λ *such that* $p(L)$ *has no answer set,*
3. $Failstate$ *is reachable from the initial state if and only if there is no set* L *of literals such that* L^+ *is an answer set of* Λ *and* $p(L)$ *has no answer set.*

Similarly to the graph $DP^2_{F,f}$, the graph $SM^2_{\Lambda,p}$ has two layers. It combines two SMODELS procedures in place of DPLL procedures.

GNT via the Abstract Solver. Let us illustrate how GNT is described by this graph. We need some additional notations for that. For a disjunctive program Π, by Π_N we denote the set of non-disjunctive rules of Π, by Π_D we denote $\Pi \setminus \Pi_N$. For each atom a in $At(\Pi)$ let

a^s be a new atom. For a set X of atoms by X^s we denote a set $\{a^s \mid a \in X\}$ of atoms. The non-disjunctive program $Gen(\Pi)$ defined by Janhunen et al. [10][8] consists of the rules below

$$\begin{array}{l}\{\{a\} \leftarrow B \mid a, A \leftarrow B \in \Pi_D\}\cup\\ \{\leftarrow \overline{A}, B \mid A \leftarrow B \in \Pi_D\}\cup\\ \Pi_N\cup\\ \{a^s \leftarrow \overline{A\setminus\{a\}}, B \mid A \leftarrow B \in \Pi; a \in A; a \vee A' \leftarrow B' \in \Pi_D\}\cup\\ \{\leftarrow a, not\ a^s \mid a \vee A \leftarrow B \in \Pi\}\end{array}$$

Janhunen et al. [10] defined a witness-program function that they call $Test$. The graph $SM^2_{Gen(\Pi),Test}$ captures the GNT procedure in a similar way as $DP^2_{EDcomp(\Pi),f_{min}}$ captures the CMODELS procedure of DP-ASSAT-PROC. The precedence order

$$\begin{array}{l}Backtrack_L, Conclude_L >>\\ UnitPropagate_L, AllRulesCancelled_L,\\ BackchainTrue_L >> Unfounded_L >> Decide_L >>\\ Backtrack_R, Conclude_R >>\\ UnitPropagate_R, AllRulesCancelled_R,\\ BackchainTrue_R >> Unfounded_R >> Decide_R >>\\ Backtrack_{LR}, Conclude_{LR}\end{array} \quad (7)$$

on the rules of the graph $SM^2_{Gen(\Pi),Test}$ describes GNT.[9]

We say that a non-disjunctive program Λ *SM-approximates* a program Π (resp. *SM′-approximates*) if for any answer set X of Π there is a consistent and complete set M of literals such that M^+ is an answer set of Λ (resp. M is a supported model of Λ) such that $X = M^+_{|At(\Pi)}$. The program $Gen(\Pi)$ both SM-approximates Π and SM′-approximates Π. We say that a witness-program function p *SM-ensures* a program Π if for any consistent set M of literals that covers $At(\Pi)$, $M^+_{|At(\Pi)}$ is an answer set of Π if and only if $p(M)$ results in a program that has no answer sets. The function $Test$ SM-ensures Π. We also define the graph $SM' \times SM_{\Lambda,p}$ as the graph $SM^2_{\Lambda,p}$ minus the rule $Unfounded_L$. It turns out that for any program Π, given a witness-program function p that SM-ensures Π and a nondisjunctive program Λ that SM-approximates Π (resp. SM′-approximates Π), the graph $SM^2_{\Lambda,p}$ (resp. $SM' \times SM_{\Lambda,p}$) captures a correct algorithm for establishing whether Π has answer sets.

Theorem 5 *For a disjunctive program Π, a non-disjunctive program Λ that SM-approximates Π (resp. SM′-approximates Π), and a witness-program function p that SM-ensures Π,*

1. *graph $SM^2_{\Lambda,p}$ (resp. $SM' \times SM_{\Lambda,p}$) is finite and acyclic,*
2. *any terminal state of $SM^2_{\Lambda,p}$ (resp. $SM' \times SM_{\Lambda,p}$) reachable from the initial state and other than $Failstate$ is $Ok(L)$, with $L^+_{|At(\Pi)}$ being an answer set of Π,*
3. *$Failstate$ is reachable from the initial state if and only if Π has no answer sets.*

Gelfond and Lifschitz [8] defined a mapping from a disjunctive program Π to a non-disjunctive program Π_{sh}, the *shifted variant* of Π, by replacing each rule (1) in Π by i new rules:

$$a_m \leftarrow B, not\ a_1, \ldots, not\ a_{m-1}, not\ a_{m+1}, \ldots, not\ a_i \quad (8)$$

where $1 < m \leq i$, B stands for the body (2) of the rule (1). Program Π_{sh} SM′-approximates Π. Theorem 5 ensures the graph $SM' \times SM_{\Pi_{sh},Test}$ captures a correct procedure for establishing whether a program Π has answer sets.

[8] The presented program $Gen(\Pi)$ captures the essence of a program defined under this name by Janhunen et al., but is not identical to it. Our language of programs includes rules with empty heads as well as choice rules. This allows us a more concise description of $Gen(\Pi)$.

[9] Sec. 5.1 of [10] describes the "early minimality test" optimization implemented in GNT. The introduced abstract framework does not account for this feature of GNT. It is a direction of future work to enhance the framework to this case.

$dAllRulesCancelled_L$:

$$(L,\emptyset) \Longrightarrow (L\neg a,\emptyset) \text{ if } \begin{cases}\text{for each rule } a \vee A \leftarrow B \text{ of } \Pi\\ \quad B \text{ is contradicted by } L\end{cases}$$

$dBackchainTrue_L$:

$$(L,\emptyset) \Longrightarrow (Ll,\emptyset) \text{ if } \begin{cases}\text{there is a rule } a \vee A \leftarrow l, B \text{ of } \Pi\\ \text{or a rule } a \vee \bar{l} \vee A \leftarrow B \text{ of } \Pi \text{ such that}\\ a \text{ is a literal of } L \text{ and}\\ \text{for each other rule } a, A' \leftarrow B' \text{ of } \Pi\\ \quad B' \text{ is contradicted by } L\end{cases}$$

Figure 4. The new transition rules of the graph $SM^{\vee} \times DP_{\Pi,f}$

5 Abstract DLV and More

We illustrated how procedures behind CMODELS and GNT are captured by the graphs $DP^2_{F,f}$ and $SM^2_{\Lambda,p}$ respectively. We now introduce a graph that captures answer set solver DLV.

We define a graph $SM^{\vee} \times DP_{\Pi,f}$ for a program Π and a witness-formula function f. The set of nodes of $SM^{\vee} \times DP_{\Pi,f}$ consists of the states relative to $At(\Pi)$ and $At(\Pi,f)$. The edges of the graph $SM^{\vee} \times DP_{\Pi,f}$ are specified by the rules of $DP^2_{\Pi,f}$ and the rules presented in Figure 4. We note that the new rules are in spirit of some left-rules of the $SM^2_{\Lambda,p}$ graph.

Theorem 6 *For any program Π and a witness-formula function f that DP-ensures Π:*

1. *graph $SM^{\vee} \times DP_{\Pi,f}$ is finite and acyclic,*
2. *any terminal state of $SM^{\vee} \times DP_{\Pi,f}$ reachable from the initial state and other than $Failstate$ is $Ok(L)$, with L^+ being an answer set of Π,*
3. *$Failstate$ is reachable from the initial state if and only if Π has no answer set.*

The graph $SM^{\vee} \times DP_{\Pi,f}$ has two layers. The generate layer, i.e., the left-rule layer, is reminiscent to the SMODELS algorithm without $Unfounded_L$. The test layer applies the DPLL procedure to the witness formula. We refer the reader to [11] for the details of the specific witness function Γ employed in DLV.

It differs from f_{min} used in CMODELS. The graph $SM^{\vee} \times DP_{\Pi,\Gamma}$, along with the precedence order (7) trivially extended to the rules of $SM^{\vee} \times DP_{\Pi,\Gamma}$ describes DLV, as in [4] and [11].

It turns out that systems DLV and CMODELS share a lot in common: the transition systems that capture DLV and CMODELS fully coincide in their left-rules.

Theorem 7 *For a disjunctive program Π, the edge-induced subgraph of $SM^{\vee} \times DP_{\Pi,f}$ w.r.t. left-edges is equal to the edge-induced subgraph of $DP^2_{CNF-Comp(\Pi),f}$ w.r.t. left-edges.*

Additionally, the precedence orders on their left-rules coincide. The proof of this fact illustrates that $UnitPropagate_L$ is applicable in a state of $DP^2_{CNF-Comp(\Pi),f}$ whenever one of the rules $UnitPropagate_L$, $dAllRulesCancelled_L$, $dBackchainTrue_L$ is applicable in the same state in $SM^{\vee} \times DP_{\Pi,f}$. The last result is remarkable as it illustrates close relation between solving technology for different propositional formalisms.

Alternative Solvers We now illustrate how transition systems introduced earlier may inspire the design of new solving procedures. We start by defining a graph that is a "symbiosis" of graphs $DP^2_{F,f}$ and $SM^2_{\Lambda,p}$.

A graph $DP \times SM_{F,p}$ for a CNF formula F and a witness-program function p is defined as follows. The set of nodes of $DP \times SM_{F,p}$ consists of the states relative to $At(F)$ and $At(F, p)$. The edges of the graph $DP \times SM_{F,p}$ are specified by (i) the Left-rules and Crossing-rules of the $DP^2_{F,p}$ graph, and (ii) the Right-rules of $SM^2_{F,p}$. This graph allows us to define a new procedure for deciding whether disjunctive answer set program has an answer set.

One can use this framework to define a theorem in the spirit of Theorem 6, in order to prove the correctness of, for instance, a procedure based on the graph $DP \times SM_{EDcomp(\Pi),Test}$.

6 Related Work and Conclusions

Lierler [15] introduced and compared the transition systems for the answer set solvers SMODELS and CMODELS for non-disjunctive programs. We extend that work as we design and compare transition systems for ASP procedures for disjunctive programs. Lierler [17] considered another extension of her earlier work by introducing transition rules that capture backjumping and learning techniques common in design of modern solvers. It is a direction of future work to extend the transition systems presented in this paper to capture backjumping and learning. This extension will allow us to model answer set solver CLASP for disjunctive programs as well as CMODELS that implements these features.

The approach based on transition systems for describing and comparing ASP procedures is one of the three main alternatives studied in the literature. The other methods include pseudo-code presentation of algorithms [9] and tableau calculi [7]. Giunchiglia et al. [9] presented pseudo-code descriptions of CMODELS (without backjumping and learning), SMODELS and DLV (without backjumping) restricted to non-disjunctive programs. They note the relation between solvers CMODELS and DLV on tight non-disjunctive programs. Gebser et al. [7] considered formal proof systems based on tableau methods for characterizing the operations and the strategies of ASP procedures for disjunctive programs. These proof systems also allow cardinality constraints in the language of logic programs, yet they do not capture backjumping and learning.

In this work we focused on developing graph-based representation for disjunctive answer set solvers GNT, DLV, and CMODELS implementing plain backtracking to allow simpler analysis and comparison of these systems. Similar effort for the case of non-disjunctive solvers resulted in design of a novel answer set solver SUP [17]. We believe that this work is a stepping stone towards clear, comprehensive articulation of main design features of current disjunctive answer set solvers that will inspire new solving algorithms. Sections 4 and 5 hint at some of the possibilities.

An extended version of this paper with proofs of the theorems in available at:

`http://works.bepress.com/yuliya_lierler/51/`

REFERENCES

[1] D. R. Brooks, E. Erdem, S. T. Erdoğan, J. W. Minett, and D. Ringe, 'Inferring phylogenetic trees using answer set programming', *Journal of Automated Reasoning*, **39**, 471–511, (2007).

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

[3] T. Eiter and G. Gottlob, 'Complexity results for disjunctive logic programming and application to nonmonotonic logics', in *Proc. ILPS*, ed., Dale Miller, pp. 266–278, (1993).

[4] W. Faber, *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*, Ph.D. dissertation, Vienna University of Technology, 2002.

[5] P. Ferraris and V. Lifschitz, 'Weight constraints as nested expressions', *TPLP*, **5**, 45–74, (2005).

[6] M. Gebser, B. Kaufmann, and T. Schaub, 'Advanced conflict-driven disjunctive answer set solving', in *Proc. IJCAI 2013*, ed., Francesca Rossi. IJCAI/AAAI, (2013).

[7] M. Gebser and T. Schaub, 'Tableau calculi for logic programs under answer set semantics', *ACM Transaction on Computational Logic*, **14**(2), 15, (2013).

[8] M. Gelfond and V. Lifschitz, 'Classical negation in logic programs and disjunctive databases', *NGC*, **9**, 365–385, (1991).

[9] E. Giunchiglia, N. Leone, and M. Maratea, 'On the relation among answer set solvers', *AMAI*, **53**(1-4), 169–204, (2008).

[10] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J. You, 'Unfolding partiality and disjunctions in stable model semantics', *ACM TOCL*, **7**(1), 1–37, (2006).

[11] C. Koch, N. Leone, Nicola and G. Pfeifer, 'Enhancing disjunctive logic programming systems by SAT checkers', *Artificial Intelligence*, **151**(1-2), 177–212, (2003).

[12] J. Lee, 'A model-theoretic counterpart of loop formulas', in *Proc. of IJCAI*, pp. 503–508, (2005).

[13] N. Leone, W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, 'The DLV system for knowledge representation and reasoning', *ACM TOCL*, **7**(3), 499–562, (2006).

[14] Y. Lierler, 'Cmodels: SAT-based disjunctive answer set solver', in *Proc. of LPNMR*, pp. 447–452, (2005).

[15] Y. Lierler, 'Abstract answer set solvers', in *Proc. of ICLP*, pp. 377–391. Springer, (2008).

[16] Y. Lierler, *SAT-based Answer Set Programming*, Ph.D. dissertation, University of Texas at Austin, 2010.

[17] Y. Lierler, 'Abstract answer set solvers with backjumping and learning', *TPLP*, **11**, 135–169, (2011).

[18] Y. Lierler and M. Truszczynski, 'Transition systems for model generators – a unifying approach', *TPLP*, **11**(4-5), 629–646, (2011).

[19] V. Lifschitz, L. R. Tang, and H. Turner, 'Nested expressions in logic programs', *AMAI*, **25**, 369–389, (1999).

[20] V. Marek and M. Truszczyński, 'Stable models and an alternative logic programming paradigm', in *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398, Springer, (1999).

[21] I. Niemelä, 'Logic programs with stable model semantics as a constraint programming paradigm', *AMAI*, **25**, 241–273, (1999).

[22] I. Niemelä and P. Simons, 'Extending the Smodels system with cardinality and weight constraints', in *Logic-Based Artificial Intelligence*, ed., Jack Minker, 491–521, Kluwer, (2000).

[23] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, '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, (2006).

[24] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, 'An A-Prolog decision support system for the Space Shuttle', in *Proc. of PADL*, pp. 169–183, (2001).