SMT-based Answer Set Solver CMODELS(DIFF) (System Description)

Da Shen
Department of Computer Science, University of Nebraska at Omaha
[South 67th Street, Omaha, NE 68182, USA]
dashen@unomaha.edu

Yuliya Lierler
Department of Computer Science, University of Nebraska at Omaha
[South 67th Street, Omaha, NE 68182, USA]
ylierler@unomaha.edu
https://orcid.org/0000-0002-6146-623X

Abstract

Many answer set solvers utilize Satisfiability solvers for search. Satisfiability Modulo Theory solvers extend Satisfiability solvers. This paper presents the CMODELS(DIFF) system that uses Satisfiability Modulo Theory solvers to find answer sets of a logic program. Its theoretical foundation is based on Niemela's characterization of answer sets of a logic program via so called level rankings. The comparative experimental analysis demonstrates that CMODELS(DIFF) is a viable answer set solver.

2012 ACM Subject Classification Computing methodologies → Logic programming and answer set programming, Software and its engineering → Constraint and logic languages, Theory of computation → Constraint and logic programming

Keywords and phrases answer set programming, satisfiability modulo theories, constraint satisfaction processing

Digital Object Identifier 10.4230/OASIcs.ICLP.2018.11

1 Introduction

This paper describes a new answer set solver CMODELS(DIFF). Its theoretical foundation lies on the generalizations of Niemela's ideas. Niemela [19] characterized answer sets of a normal logic program as models of a propositional formula called program's completion that satisfy "level ranking" requirements. In this sense, this system is a close relative of an earlier answer set solver LP2DIFF developed by Janhunen et al. [10]. Yet, LP2DIFF only accepts programs of a very restricted form. For example, neither choice rules nor aggregate expressions are allowed. Solver CMODELS(DIFF) permits such important modeling constructs in its input. Also, unlike LP2DIFF, the CMODELS(DIFF) system is able to generate multiple solutions.

The CMODELS(DIFF) system follows the tradition of answer set solvers such as ASSAT [16] and CMODELS [11]. In place of designing specialized search procedures targeting logic programs, these tools compute a program's completion and utilize Satisfiability solvers [9] – systems for finding satisfying assignments for propositional formulas – for search. Since not all models of a program's completion are answer sets of a program, both ASSAT and CMODELS implement specialized procedures (based on loop formulas [16]) to weed out such models. Satisfiability Modulo Theory (SMT) solvers [2] extend Satisfiability solvers. They process formulas that go beyond propositional logic and may contain, for example, integer linear expressions. The CMODELS(DIFF) system utilizes this fact and translates a logic program into an SMT formula so that any model of this formula corresponds to an answer set of the program. It then uses SMT solvers for search. Unlike CMODELS or ASSAT, the CMODELS(DIFF) system does not need an additional step to weed out unwanted models. Also, it utilizes SMT-LIB – a standard input language of SMT solvers [1] – to interface with these systems. This makes its architecture open towards new developments in the realm of SMT solving. There is practically no effort involved in incorporating a new SMT system into the CMODELS(DIFF) implementation.

Creation of the CMODELS(DIFF) system was inspired by the development of recent constraint answer set programming solver EZSMT [21] that utilizes SMT solvers for finding solutions for "tight" constraint answer set programs. On the one hand, CMODELS(DIFF) restricts its attention to pure answer set programs. On the other hand, it goes beyond tight programs. In the future, we will extend CMODELS(DIFF) to accept non-tight constraint answer set programs. The theory developed in this work paves the way for such an extension.

Lierler and Susman [13] demonstrate that SMT formulas are strongly related to constraint programs [17]. Many efficient constraint solvers[1] exist. Majority of these systems focus on finite-domain constraint problems. The theoretical contributions of this work provide a foundation for developing a novel constraint-solver-based method in processing logic programs. Currently, CMODELS(DIFF) utilizes SMT-LIB to interface with SMT solvers. By producing output in MINIZINC – a standard input language of constraint solvers [18] – in place of SMT-LIB, CMODELS(DIFF) will become a constraint-based answer set solver. This is another direction of future work.

The outline of the paper is as follows. We start by reviewing the concepts of a logic program, a completion, tightness and an SMT logic SMT(IL). We then present a key concept of this work, namely, a level ranking; and state theoretical results. Section 4 presents transformations from logic programs to SMT(IL) by means of variants of level rankings. After that, we introduce the architecture of the CMODELS(DIFF) system and conclude with comparative experimental analysis.

2 Preliminaries

A *vocabulary* is a finite set of propositional symbols also called atoms. As customary, a *literal* is an atom a or its negation, denoted $\neg a$. A *(propositional) logic program*, denoted by Π, over vocabulary σ is a finite set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m,\ \ not\ \ not\ b_{m+1}, \ldots,\ not\ \ not\ b_n \tag{1}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq n$, is an atom or symbol $\top$ and $\bot$ in σ. Sometimes we use the abbreviated form of rule (1)

$$a \leftarrow B \tag{2}$$

where B stands for the right hand side of an arrow in (1) and is also called a *body*. We identify rule (1) with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n \rightarrow a \tag{3}$$

and B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{4}$$

[1] http://www.minizinc.org/

Note that (i) the order of terms in (4) is immaterial, (ii) *not* is replaced with classical negation ($\neg$), and (iii) comma is replaced with conjunction ($\wedge$). When the body is empty it corresponds to the empty conjunction or $\top$. Expression $b_1 \wedge \ldots \wedge b_\ell$ in formula (4) is referred to as the *positive* part of the body and the remainder of (4) as the *negative* part of the body.

The expression a is the *head* of the rule. When a is $\perp$, we often omit it and say that the head is empty. We denote the set of nonempty heads of rules in Π by $hd(\Pi)$. We call a rule whose body is empty a *fact*. In such cases, we drop the arrow. We sometimes may identify a set X of atoms with the set of facts $\{a. \mid a \in X\}$.

We say that a set X of atoms *satisfies* a rule (1) if X satisfies a formula (3). The reduct Π^X of a program Π relative to a set X of atoms is obtained by first removing all rules (1) such that X does not satisfy its negative part $\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n$ and replacing all of its remaining rules with $a \leftarrow b_1, \ldots, b_\ell$. A set X of atoms is an *answer set*, if it is a minimal set that satisfies all rules of Π^X [15].

Ferraris and Lifschitz [6] show that a choice rule $\{a\} \leftarrow B$ can be seen as an abbreviation for a rule $a \leftarrow$ *not not* a, B. We adopt this abbreviation here. Choice rules were introduced in [20] and are commonly used in answer set programming languages.

It is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value *true* to every atom in X and *false* to every atom in $\sigma \setminus X$.

Consider sample programs listed in Figure 1. Program Π_1 has two answer sets, namely, $\{a, c\}$ and an empty set. Program Π_2 has two answer sets: $\{a, b, c\}$ and an empty set.

Π_1	Π_2
$\{c\}.$	$\{c\}.$
$a \leftarrow c.$	$a \leftarrow c.$
	$a \leftarrow b.$
	$b \leftarrow a.$

$Comp(\Pi_1)$	$Comp(\Pi_2)$
$\neg\neg c \rightarrow c.$	$\neg\neg c \rightarrow c.$
$c \rightarrow a.$	$c \rightarrow a.$
$c \rightarrow \neg\neg c.$	$b \rightarrow a.$
$a \rightarrow c.$	$a \rightarrow b.$
	$c \rightarrow \neg\neg c.$
	$a \rightarrow c \vee b.$

Figure 1 Sample programs and their completions.

Completion and Tightness

Let σ be a vocabulary and Π be a program over σ. For every atom a in Π, by $Bodies(\Pi, a)$ we denote the set composed of the bodies B appearing in the rules of the form $a \leftarrow B$ in Π. The *completion* of Π [3], denoted by $Comp(\Pi)$, is the set of classical formulas that consists of the rules (1) in Π (recall that we identify rule (1) with implication (3)) and the implications

$$a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B \tag{5}$$

for all atoms a in σ. When set $Bodies(\Pi, a)$ is empty, the implication (5) has the form $a \rightarrow \perp$. When a rule (2) is a fact $a.$, then we identify this rule with the unit clause a.

For example, completions of programs Π_1 and Π_2 are presented in Figure 1.

For the large class of logic programs, called *tight*, their answer sets coincide with models of their completion [5, 4]. This is the case for program Π_1 (we illustrate that Π_1 is tight, shortly). Yet, for non-tight programs, every answer set is a model of completion but not necessarily the other way around. For instance, set $\{a, b\}$ is a model of $Comp(\Pi_2)$, but not an answer set of Π_2. It turns out that Π_2 is not tight.

Tightness is a syntactic condition on a program that can be verified by means of program's dependency graph. The *dependency graph* of Π is the directed graph G such that

- the nodes of G are the atoms occurring in Π, and
- for every rule (1) in Π whose head is an atom, G has an edge from atom a to each atom $b_1, \dots, b_\ell$.

A program is called *tight* if its dependency graph is acyclic.

For example, the dependency graph of program Π_1 consists of two nodes, namely, a and c, and a single edge from a to c . This graph is acyclic and hence Π_1 is tight. On the other hand, it is easy to see that the graph of Π_2 is not acyclic.

Logic SMT(IL)

We now introduce the notion of Satisfiability Modulo Theory (SMT) [2] for the case when Linear Integer Arithmetic is a considered theory. We denote this SMT instance by SMT(IL).

Let σ be a vocabulary and χ be a finite set of integer variables. The set of *atomic formulas* of SMT(IL) consists of propositions in σ and linear constraints of the form

$$a_1x_1 \pm \dots \pm a_nx_n \bowtie a_{n+1} \tag{6}$$

where $a_1, \dots, a_{n+1}$ are integers and $x_1, \dots, x_n$ are variables in χ, $\pm$ stands for $+$ or $-$, and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. The set of SMT(IL) formulas is the smallest set that contains the atomic formulas and is closed under $\neg$ and conjunction $\wedge$. Other connectives such as $\top$, $\bot$, $\vee$, $\rightarrow$, and $\leftrightarrow$ can be defined in terms of $\neg$ and $\wedge$ as customary.

A valuation τ consists of a pair of functions

- $\tau_\sigma : \sigma \rightarrow \{true, false\}$ and
- $\tau_\chi : \chi \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers.

A valuation interprets all SMT(IL) formulas by defining

- $\tau(p) = \tau_\sigma(p)$ when $p \in \sigma$,
- $\tau(a_1x_1 \pm \dots \pm a_nx_n \bowtie a_{n+1}) = true$ iff $a_1\tau_\chi(x_1) \pm \dots \pm a_n\tau_\chi(x_n) \bowtie a_{n+1}$ holds,

and applying the usual rules for the Boolean connectives.

We say that an SMT(IL) formula Φ is satisfied by a valuation τ when $\tau(\Phi) = true$. A set of SMT(IL) formulas is satisfied by a valuation when every formula in the set is satisfied by this valuation. We call a valuation that satisfies an SMT(IL) formula a *model*.

3 Level Rankings

Niemela [19] characterized answer sets of "normal" logic programs in terms of "level rankings." *Normal* programs consist of rules of the form (1), where $n = m$ and a is an atom. Lierler and Susman [13] generalized the concept of level ranking to programs considered in this paper that include choice rules and denials (rules with empty head).

By $\mathbb{N}$ we denote the set of natural numbers. For a rule (2), by B^+ we denote its positive part and sometimes identify it with the set of atoms that occur in it, i.e., $\{b_1, \dots, b_l\}$. For a program Π, by $At(\Pi)$ we denote the set of atoms occurring in it.

▶ **Definition 1.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \rightarrow \mathbb{N}$ is a *level ranking* of X for Π when for each $a \in X$, there is B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+$ it holds that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$.

Niemela [19] observed that for a normal logic program, a model X of its completion is also its answer set when there is a level ranking of X for the program. Lierler and Susman [13] generalized this result to programs with double negation *not not*:

▶ **Theorem 2** (Theorem 1 [13]). *For a program* Π *and a set* X *of atoms that is a model of its completion* $Comp(\Pi)$*,* X *is an answer set of* Π *if and only if there is a level ranking of* X *for* Π*.*

The nature of a level ranking is such that there is an infinite number of level rankings for the same answer set of a program. Theorem below illustrates that we can add a *single* linear constraint to limit the number of level rankings by utilizing the size of a program.

▶ **Theorem 3.** *For a logic program* Π *and its answer set* X*, we can always construct a level ranking of* X *for* Π *such that, for every* $a \in X$*,* $lr(a) \leq |At(\Pi)|$*.*

Proof. Since there is an answer set X, by Theorem 2 there exists some level ranking lr' of X for Π. Then, we can always use the level ranking lr' to construct a level ranking lr of X for Π such that, for every $a \in X$, $lr(a) \leq |At(\Pi)|$. Below we describe the method.

For an integer y, by $s(y)$ we denote the following set of atoms

$$\{a \mid a \in X, lr'(a) = y\}.$$

Let Y be the set of integers so that

$$\{y \mid a \in X, lr'(a) = y\}.$$

Let Y^s denote the sorted list $[y_1, \ldots, y_k]$ constructed from all integers of Y, such that $y_1 < y_2 < \ldots < y_k$. Note that $y_i > y_j$ if and only if $i > j$. Obviously, $|Y| \leq |At(\Pi)|$. Thus, $k \leq |At(\Pi)|$. For every element y_i in Y^s and every atom $a \in s(y_i)$, we assign $lr(a) = i$. Consequently, $lr(a) \leq |At(\Pi)|$.

Now we prove that lr is indeed a level ranking. According to the definition of lr', for each atom $a \in X$, there exists B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+$ it holds that $\mathrm{lr}'(a) - 1 \geq \mathrm{lr}'(b)$. We show that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$ also holds for each b in this B^+. Atoms a, b belong to some sets $s(y_{k_a})$ and $s(y_{k_b})$ respectively, where $k_a, k_b \leq k$. By the definition of $s(\cdot)$, $y_{k_a} = lr'(a)$ and $y_{k_b} = lr'(b)$. Since $lr'(a) > lr'(b)$, $y_{k_a} > y_{k_b}$. Since for any i and j, $y_i > y_j$ if and only if $i > j$, we derive that $k_a > k_b$. By the construction of lr, $lr(a) = k_a$ and $lr(b) = k_b$. Consequently, $lr(a) - 1 \geq lr(b)$ also holds. Thus, lr is a level ranking by definition. ◀

Strong level ranking

Niemela [19] introduced the concept of a strong level ranking so that only one strong level ranking exists for an answer set. It is obviously stricter than the condition captured in Theorem 3. Yet, the number of linear constraints in formulating the conditions of strong level ranking is substantially greater. We now generalize the concept of a strong level ranking to the case of logic programs considered here and then state the formal result on the relation of answer sets and strong level rankings.

▶ **Definition 4.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \to \mathbb{N}$ is a *strong level ranking* of X for Π when lr is a level ranking and for each $a \in X$ the following conditions hold:

1. If there is B in $Bodies(\Pi, a)$ such that X satisfies B and B^+ is empty, then $\mathrm{lr}(a) = 1$.

2. For every B in $Bodies(\Pi, a)$ such that X satisfies B and B^+ is not empty, there is at least one $b \in B^+$ such that $\mathrm{lr}(b) + 1 \geq \mathrm{lr}(a)$.

▶ **Theorem 5.** *For a program* Π *and a set* X *of atoms that is a model of its completion* $Comp(\Pi)$*,* X *is an answer set of* Π *if and only if there is a strong level ranking of* X *for* Π*.*

Proof. This proof follows the argument provided for Theorem 2 in [19], but respects the terminology used here. We start by defining an operator $T_\Pi(I)$ for a program Π and a set I over $At(\Pi) \cup \bot$ as follows:

$$T_\Pi(I) = \{a \mid a \leftarrow B \in \Pi, I \text{ satisfies } B\}.$$

For this operator we define

$$T_\Pi \uparrow 0 = \emptyset,$$

and for i= 0,1,2, ...

$$T_\Pi \uparrow (i+1) = T_\Pi(T_\Pi \uparrow i).$$

Left-to-right: Assume X is an answer set of Π. We can construct a strong level ranking lr of X for Π using the $T_{\Pi^X}(\cdot)$ operator. As X is an answer set of Π, we know that $X = T_{\Pi^X} \uparrow \omega$ and for each $a \in X$ there is a unique i such that $a \in T_{\Pi^X} \uparrow i$, but $a \notin T_{\Pi^X} \uparrow (i-1)$. Let $lr(a) = i$. We now illustrate that lr is indeed a strong level ranking.

First, we illustrate that lr is a level ranking. For $a \in X$ there is a rule $a \leftarrow B$ of the form (1) such that $a \leftarrow b_1, \ldots, b_l \in \Pi^X$ and $T_{\Pi^X} \uparrow (i-1)$ satisfies $b_1 \wedge \cdots \wedge b_l$. Consequently, for every b_j in $\{b_1, \ldots, b_l\}$, $lr(b_j) \leq i-1$. Thus, $lr(a) - 1 \geq lr(b_j)$. Also, from the way the reduct is constructed, it follows that X satisfies body B of rule $a \leftarrow B$.

Second, we show that Condition 1 of the definition of strong level ranking holds for lr. If there is $a \leftarrow B \in \Pi$ such that X satisfies B and B^+ is empty, then $a \leftarrow \top$ is in Π^X. By definition of the $T_{\Pi^X}(\cdot)$ operator, $a \in T_{\Pi^X} \uparrow 1$. Consequently, $lr(a) = 1$ holds.

Third, we demonstrate that Condition 2 holds for lr. For $a \in X$, by the construction of lr using the $T_{\Pi^X}(\cdot)$ operator we know that there is a unique i such that $lr(a) = i$, $a \in T_{\Pi^X} \uparrow i$, but $a \notin T_{\Pi^X} \uparrow (i-1)$. Proof by contradiction. Assume that there is a rule $a \leftarrow B \in \Pi$ such that X satisfies B and B^+ is not empty, but for all $b \in B^+$, $lr(b) + 1 < lr(a)$ holds. Then for all $b \in B^+$, $lr(b) < lr(a) - 1$. Thus, $lr(b) < i - 1$. It follows that all $b \in B^+$ belong to $T_{\Pi^X} \uparrow (i-2)$. Hence, by the definition of $T_{\Pi^X}(\cdot)$ operator, $a \in T_{\Pi^X} \uparrow (i-1)$, which contradicts that $a \notin T_{\Pi^X} \uparrow (i-1)$. Thus, there is at least one $b \in B^+$ such that $\mathrm{lr}(b) + 1 \geq \mathrm{lr}(a)$.

Right-to-left: Assume that there is a strong level ranking of X for Π. By the definition, it is also a level ranking. Recall that X is a model of $Comp(\Pi)$. By Theorem 2, X is an answer set of Π. ◀

SCC level ranking

Niemela [19] illustrated how one can utilize the structure of the dependency graph corresponding to a normal program to reduce the number of linear constraints in capturing conditions similar to these of level ranking. We now generalize these results to logic programs with doubly negated atoms and denials.

Recall that a strongly connected component of a directed graph is a maximal set V of nodes such that each pair of nodes in V is reachable from each other. We call a set of atoms in a program Π a *strongly connected component* (SCC) of Π when it is a strongly connected component in the dependency graph of Π. The SCC including an atom a is denoted by $SCC(a)$. A *non-trivial* SCC is an SCC that consists of at least two atoms. We denote the set of atoms in all non-trivial SCCs of Π by $NT(\Pi)$.

▶ **Definition 6.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \cap NT(\Pi) \to \mathbb{N}$ is a *SCC level ranking* of X for Π when for each $a \in X \cap NT(\Pi)$, there is B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+ \cap SCC(a)$ it holds that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$.

The byproduct of the definition of SCC level rankings is that for tight programs SCC level ranking trivially exists since it is a function whose domain is empty. Thus no linear constraints are produced.

▶ **Theorem 7.** *For a program Π and a set X of atoms that is a model of its completion $Comp(\Pi)$, X is an answer set of Π if and only if there is an SCC level ranking of X for Π.*

This is a generalization of Theorem 4 in [19]. Its proof follows the lines of the proof presented there with the reference to Theorem 2.

▶ **Theorem 8.** *For a satisfiable logic program Π and its answer set X, we can always construct an SCC level ranking of X for Π such that, for every $a \in X$, $lr(a) \leq |SCC(a)|$.*

This theorem can be proved by applying the similar argument as in the proof of Theorem 3 to each SCC. This result allows us to set minimal upper bounds for $lr(a)$ in order to reduce search space.

Further, Niemela [19] introduces the concept of strong SCC level ranking and states a similar result to Theorem 7 for that concept. It is straightforward to generalize these results to logic programs considered here.

4 From Logic Programs to SMT(IL)

In this section we present a mapping from a logic program to SMT(IL) such that the models of a constructed SMT(IL) theory are in one-to-one correspondence with answer sets of the program. Thus, any SMT solver capable of processing SMT(IL) expressions can be used to find answer sets of logic programs. The developed mappings generalize the ones presented by Niemela [19].

For a rule $a \leftarrow B$ of the form (1), the auxiliary atom β_B, equivalent to its body, is defined as

$$\beta_B \leftrightarrow b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge b_{m+1} \wedge \ldots \wedge b_n \tag{7}$$

When the body of a rule consist of a single element, no auxiliary atom is introduced (the single element itself serves the role of an auxiliary atom).

Let Π be a program. We say that an atom a is a *head atom* in Π if it is the head of some rule. Any atom a in Π such that

- it is a head atom, or
- it occurs in some positive part of the body of some rule whose head is an atom,

we associate with an integer variable denoted by lr_a. We call such variables level ranking variables. For each head atom a in Π, we construct an SMT(IL) formula

$$a \to \bigvee_{a \leftarrow B \in \Pi} (\beta_B \wedge \bigwedge_{b \in B^+} lr_a - 1 \geq lr_b). \tag{8}$$

We call the conjunction of formulas (8) for the head atoms in program Π a *level ranking formula* of Π.

For example, the level ranking formula of program Π_2 in Figure 1 follows

$$(c \to \neg\neg c) \;\wedge\; \big(a \to (c \wedge lr_a - 1 \geq lr_c) \vee (b \wedge lr_a - 1 \geq lr_b)\big) \;\wedge\; \big(b \to a \wedge lr_b - 1 \geq lr_a\big). \tag{9}$$

▶ **Theorem 9.** *For a logic program* Π *and the set* F *of* SMT(IL) *formulas composed of* $Comp(\Pi)$ *and a level ranking formula of* Π

1. *If a set* X *of atoms is an answer set of* Π*, then there is a satisfying valuation* τ *for* F *such that* $X = \{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$.
2. *If valuation* τ *is satisfying for* F*, then the set* $\{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$ *is an answer set for* Π.

This is a generalization of Theorem 6 in [19]. Its proof follows the lines of the proof presented there with the reference to Theorem 2.

SCC level ranking

For each atom a in the set $NT(\Pi)$, we introduce an auxiliary atom ext_a. If there exists some rule $a \leftarrow B$ in Π such that $B^+ \cap SCC(a) = \emptyset$, then we construct an SMT(IL) formula

$$ext_a \leftrightarrow \bigvee_{a\leftarrow B\in\Pi \text{ and } B^+\cap SCC(a)=\emptyset} \beta_B; \tag{10}$$

otherwise, we construct a formula

$$\neg ext_a. \tag{11}$$

We also introduce an SMT(IL) formula:

$$a \rightarrow ext_a \vee \bigvee_{a\leftarrow B\in\Pi \text{ and } B^+\cap SCC(a)\neq\emptyset} (\beta_B \wedge \bigwedge_{b\in B^+\cap SCC(a)} lr_a - 1 \geq lr_b). \tag{12}$$

We call the conjunction of formulas (10), (11) and (12) a *SCC level ranking formula* of Π.

For instance, $NT(\Pi_1)$ is empty, so we introduce no SCC level ranking formula for program Π_1. The SCC level ranking formula of program Π_2 follows

$$\big(ext_a \leftrightarrow c\big) \wedge \neg ext_b \wedge \big(a \rightarrow ext_a \vee (b \wedge lr_a - 1 \geq lr_b)\big) \wedge \big(b \rightarrow ext_b \vee (a \wedge lr_b - 1 \geq lr_a)\big). \tag{13}$$

The claim of Theorem 9 holds also when we replace *a level ranking formula of* Π with *an SCC level ranking formula of* Π in its statement.

Strong level ranking

For each rule $a \leftarrow B$ in program Π we construct an SMT(IL) formula

$$\begin{array}{ll} a \wedge \beta_B \rightarrow lr_a = 1 & \text{when } B^+ = \emptyset, \\ a \wedge \beta_B \rightarrow \bigvee_{b\in B^+} lr_b + 1 \geq lr_a & \text{otherwise.} \end{array} \tag{14}$$

We call the conjunction of formulas (8) and (14) a *strong level ranking formula* of Π.

For example, the strong level ranking formula of program Π_2 is a conjunction of formula (9) and formula

$$\begin{array}{l} \big(c \wedge \neg\neg c \rightarrow lr_c = 1\big) \wedge \big(a \wedge c \rightarrow lr_c + 1 \geq lr_a\big) \wedge \\ \big(a \wedge b \rightarrow lr_b + 1 \geq lr_a\big) \wedge \big(b \wedge a \rightarrow lr_a + 1 \geq lr_b\big). \end{array}$$

We now state a similar result to Theorem 9 that makes an additional claim on one-to-one correspondence between the models of a constructed SMT(IL) formula with the use of strong level ranking formula and answer sets of a program.

▶ **Theorem 10.** *For a logic program* Π *and the set* F *of* SMT(IL) *formulas composed of* $Comp(\Pi)$ *and a strong level ranking formula of* Π

1. *If a set* X *of atoms is an answer set of* Π*, then there is a satisfying valuation* τ *for* F *such that* $X = \{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$.
2. *If valuation* τ *is satisfying for* F*, then the set* $\{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$ *is an answer set for* Π.
3. *If valuations* τ *and* τ' *satisfy* F *and are distinct, then*

$$\{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\} \neq \{a \mid a \in At(\Pi) \text{ and } \tau'(a) = true\}.$$

Strong SCC level ranking

For each atom $a \in NT(\Pi)$, we construct a formula

$$ext_a \rightarrow lr_a = 1, \tag{15}$$

and for each rule $a \leftarrow B$ such that $B^+ \cap SCC(a) \neq \emptyset$, we introduce a formula

$$a \wedge \beta_B \rightarrow \bigvee_{b \in B^+ \cap SCC(a)} lr_b + 1 \geq lr_a. \tag{16}$$

We call the conjunction of formulas (10), (11), (12), (15) and (16) a *strong SCC level ranking formulas* of Π.

For instance, $NT(\Pi_1)$ is empty, so we introduce no strong SCC level ranking formula for program Π_1. The strong SCC level ranking formula of program Π_2 is a conjunction of formula (13) and formula

$$(ext_a \rightarrow lr_a = 1) \wedge (ext_b \rightarrow lr_b = 1) \wedge (a \wedge b \rightarrow lr_b + 1 \geq lr_a) \wedge (b \wedge a \rightarrow lr_a + 1 \geq lr_b).$$

The claim of Theorem 10 holds also when we replace *a strong level ranking formula of* Π with *a strong SCC level ranking formula of* Π in its statement.

5 The CMODELS(DIFF) system

We are now ready to describe the the CMODELS(DIFF)[2] system in detail. It is an extension of the CMODELS [11] system. Figure 2 illustrates the pipeline architecture of CMODELS(DIFF). This system takes an arbitrary (tight or non-tight) logic program in the language supported by CMODELS as an input. These logic programs may contain such features as choice rules and aggregate expressions. The rules with these features are translated by CMODELS [11] into rules considered here. The CMODELS(DIFF) system translates a logic program into SMT(IL) formulas, after which an SMT solver is called to find models of these formulas (that correspond to answer sets).

[2] CMODELS(DIFF) is posted at https://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/cmodels-diff.php

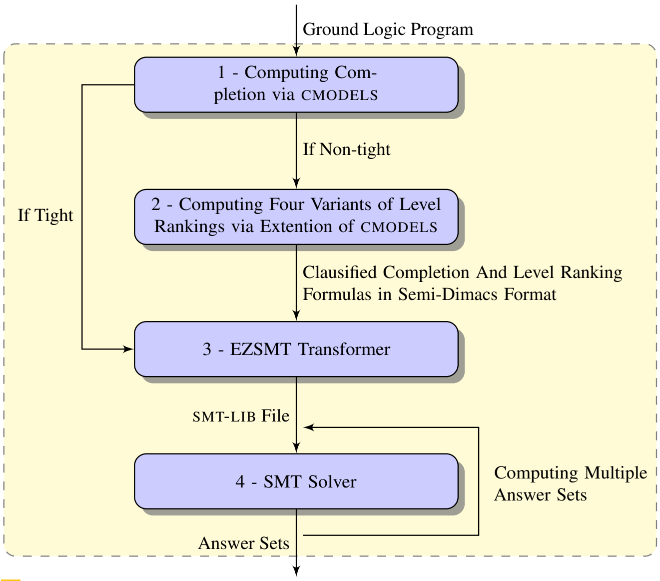

Figure 2 CMODELS(DIFF) Pipeline

(1,2) Computing Completion and Level Ranking Formulas

The CMODELS(DIFF) system utilizes the original algorithm of CMODELS to compute completion, during which CMODELS determines whether the program is tight or not. If the program is not tight, the corresponding level ranking formula is added.

Flags `-levelRanking`, `-levelRankingStrong`, `-SCClevelRanking`, and `-SCClevelRankingStrong` instruct CMODELS(DIFF) to construct a level ranking formula, a strong level ranking formula, a SCC level ranking formula, and a strong SCC level ranking formula, respectively. And, `-SCClevelRanking` is chosen by default. Finally, completion and the level ranking formula are clausified using the same technique as in original CMODELS. The CMODELS(DIFF) system outputs the resulting clauses into a text file in semi-Dimacs format [21].

(3, 4) Transformation and Solving

The transformer is taken from EZSMT v1.1. It converts the semi- Dimacs output from step (2) into SMT-LIB syntax (SMT-LIB is a standard input language for SMT solvers [1]). By default, the SMT-LIB output contains an instruction that sets the logic of SMT solvers to Linear Integer Arithmetic. If the transformer is invoked with the parameter `difference-logic`, then the SMT-LIB output sets the logic of SMT solvers to Difference Logic instead.

Finally, one of the SMT solvers CVC4, Z3, or YICES is called to compute models by using flags `-cvc4`, `-z3`, or `-yices`. (In fact, any other SMT solver supporting SMT-LIB can be utilized.) The CMODELS(DIFF) system post-processes the output of the SMT solvers mentioned above to produce answer sets in a typical format disregarding any auxiliary atoms or integer variables that are created during the system's execution.

The CMODELS(DIFF) system allows us to compute multiple answer sets. Currently, SMT solvers typically find only a single model. We design a process to enumerate all models. For a logic program Π, after an SMT solver finds a model and exits, the CMODELS(DIFF) system constructs a clause that consists of (i) atoms in $At(\Pi)$ that are assigned *false* by the model and (i) negations of atoms in $At(\Pi)$ that are assigned *true* by the model. This clause is added into the SMT-LIB formula previously computed. Then, the SMT solver is called again taking the new input. The process is performed repeatedly, until the SMT-LIB formula becomes unsatisfiable.

In summary, CMODELS(DIFF) has eight possible configurations. We can choose one from the four variants of level ranking formulas, and choose a logic from either Linear Integer Arithmetic or Difference Logic for the invoked SMT solver.

6 Experiments

We benchmark CMODELS(DIFF) on seven problems, to compare its performance with that of other ASP solvers, namely CMODELS and CLASP [7]. All considered benchmarks are *non-tight* programs. The first two benchmarks are Labyrinth and Connected Still Life, which are obtained from the Fifth Answer Set Programming Competition[3]. We note that the original encoding of Still Life is an optimization problem, and we turn it into a decision one. The next three benchmarks originate from Asparagus[4]. The selected problems are RandomNonTight, Hamiltonian Cycle and Wire Routing. We also consider five instances of Wire Routing from RST Construction[5]. Then, we use Bounded Models as the sixth benchmark[6]. Our last benchmark, Mutual Exclusion, comes from Synthesis Benchmarks[7]. We rewrite the seven encodings to fit the syntax of GRINGO 4, and call GRINGO v. 4.5.3[8] to produce ground programs serving as input to all benchmarked systems. All benchmarks are posted at the CMODELS(DIFF) website provided at Footnote 2.

All benchmarks are run on an Ubuntu 16.04.1 LTS (64-bit) system with an Intel core i5-4250U processor. The resource allocated for each benchmark is limited to one cpu core and 4GB RAM. We set a timeout of 1800 seconds. No problems are solved simultaneously.

Numbers of instances are shown in parentheses after names of benchmarks. We present cumulative time of all instances for each benchmark with numbers of unsolved instances due to timeout or insufficient memory inside parentheses. All the steps involved, including grounding and transformation, are reported as parts of solving time.

Five distinct solvers are benchmarked: (1) CMODELS(DIFF) invoking SMT solver CVC4 v. 1.4; (2) CMODELS(DIFF) invoking SMT solver Z3 v. 4.5.1; (3) CMODELS(DIFF) invoking SMT solver YICES v. 2.5.4; (4) CLASP v. 3.1.3; (5) CMODELS v. 3.86.1 with Satisfiability solver Minisat v. 2.0 beta. We use DIFF-CVC4, DIFF-Z3, and DIFF-YICES to denote three variants of CMODELS(DIFF) used in the experiments.

Table 1 Experimental Summary

Benchmark	DIFF-CVC4 LIA	DIFF-Z3 LIA	DIFF-YICES LIA	DIFF-Z3 DL	DIFF-YICES DL	CMODELS	CLASP
Still Life (26)	731	5423(1)	203	899	194	<u>647</u>	**10.8**
Ham. Cycl. (50)	15.39	9.78	4.54	6.61	3.57	<u>1.19</u>	**0.53**
Wire Rout. (10)	1378	562.36	1562	2983(1)	2089(1)	<u>409</u>	**12.5**
Bound. Mod. (8)	6.08	4.30	2.34	2.93	2.20	<u>1.59</u>	**1.38**
Labyrinth (30)	19543(8)	27794(12)	20425(10)	22023(9)	21836(9)	<u>16408</u>(7)	**5826**(2)
Rand. Nont. (20)	27.8	8.65	6.84	7.72	6.47	<u>**1.39**</u>	3.52
Mut. Excl. (5)	5.26	2.72	1.70	2.28	1.50	<u>0.30</u>	**0.13**

Table 1 summarizes main results. Under the name of variants of the CMODELS(DIFF) systems, we state the configuration used for this solver. Namely, "LIA" and "DL" denote that the logic of SMT solvers is set to Linear Integer Arithmetic and Difference Logic, respectively. All DIFF systems in the table are invoked with flag `-SCClevelRanking`. Systems CLASP

[3] https://www.mat.unical.it/aspcomp2014/
[4] https://asp.haiti.cs.uni-potsdam.de/
[5] http://people.sabanciuniv.edu/~esraerdem/ASP-benchmarks/rst-basic.html
[6] http://users.ics.aalto.fi/~kepa/experiments/boundsmodels/
[7] http://www2.informatik.uni-stuttgart.de/fmi/szs/research/projects/synthesis/benchmarks030923.html
[8] http://potassco.sourceforge.net/

and CMODELS are run with default settings. We benchmarked CMODELS(DIFF) with all eight possible configurations. Yet, we do not present all of the data here. CMODELS(DIFF) invoked with `-levelRanking` and `-levelRankingStrong` flags shows worse performance than settings `-SCClevelRanking` and `-SCClevelRankingStrong`, respectively. That is why we avoid presenting the results on configurations `-levelRanking` and `-levelRankingStrong`. Also, adding constraints for strong level ranking typically slightly degrades the performance so we do not present the results for the `-SCClevelRankingStrong` configuration. We note that SMT solver CVC4 implements the same procedure for processing Difference Logic statement and Linear Integer Arithmetic statements.

Observations

We observe that system CLASP almost always displays the best results. This is not surprising as this is one of the best *native* answer set solvers currently available. Its search method is attuned towards processing logic programs. Given that SMT solvers are agnostic towards specifics of logic programs it is remarkable how good the performance of CMODELS(DIFF) is. In some cases it is comparable to that of CLASP.

It is the case that many Satisfiability solvers and answer set solvers share a lot in common [12]. For example, answer set solver CLASP starts by computing clausified programs completion and then later applies to it *Unit* propagator search technique stemming from Satisfiability solving. That is reminiscent of the process that system CMODELS(DIFF) undertakes. It also computes program's completion so that *Unit* propagator of SMT solvers is applicable to it.

We conjecture that the greatest difference between CMODELS(DIFF) and CLASP lies in the fact that

- in CMODELS(DIFF) integer linear constraints encode the conditions to weed out unwanted models of completion; SMT solvers implement search techniques/propagators to target these integer linear constraint;
- in CLASP the structure of the program is taken into account by the so called *Unfounded* propagator for this task.

In case of Still Life, Hamiltonian Cycle, Wire Routing, and Bounded Models benchmarks (marked in bold in Table 1) there is *one more substantial difference.* These encodings contain aggregates. CLASP implements specialized search techniques to benefit from the compact representations that aggregates provide. System CMODELS(DIFF) translates aggregates away, which often results in a bigger problem encoding that the system has to deal with. System CMODELS also translates aggregates away. This is why we underline the solving times of CMODELS, as it is insightful to compare the performance of CMODELS to that of CMODELS(DIFF) alone. Indeed, CMODELS(DIFF) utilizes the routines of CMODELS for eliminating aggregates and computing the completion of the resulting program. Thus, the only difference between these systems is in how they eliminate models of completion that are not answer sets. System CMODELS(DIFF) utilizes level rankings for that. System CMODELS implements a propagator in spirit of *Unfounded* propagator of CLASP, but the propagator of CMODELS is only used when a model of completion is found; CLASP utilizes this propagator as frequently as it utilizes *Unit* propagator [14, Section 5]. We believe that when we observe a big difference in performance of CMODELS(DIFF) and CLASP, this attributes to the benefits gained by the utilization of specialized *Unfounded* and "aggregate" propagators by CLASP. Yet, level ranking formulas seem to provide a viable alternative to *Unfounded* propagator and open a door for utilization of SMT solvers for dealing with non-tight programs. This gives us grounds to believe that the future work on extending constraint answer set solver

EZSMT to accept non-tight programs is a viable direction.

As we noted earlier SCC level rankings yield best performance among the four variants of level rankings. Furthermore, Table 1 illustrates the following. The logic of SMT solvers does not make an essential difference. Overall, CMODELS(DIFF)-YICES with Linear Integer Arithmetic logic performs best within the presented CMODELS(DIFF) configurations. Obviously, utilizing better SMT solvers can improve the performance of CMODELS(DIFF) in the future. Notably, this does not require modifications to CMODELS(DIFF), since SMT-LIB used by CMODELS(DIFF) is a standard input language of SMT solvers.

7 Conclusion

In this paper we presents the CMODELS(DIFF) system that takes a logic program and translates it into an SMT-LIB formula which is then solved by an SMT solver to find answer sets of the given program. Our work parallels the efforts of an earlier answer set solver LP2DIFF [10]. The CMODELS(DIFF) system allows richer syntax such as choice rules and aggregate expressions, and enables computation of multiple solutions. (In this work we extended the theory of level rankings to the case of programs with choice rules and denials.) We note that the LP2NORMAL[9] tool can be used as a preprocessor for LP2DIFF in order to enable this system to process logic programs with richer syntax. In the future, we will compare performance of CMODELS(DIFF) and LP2DIFF experimentally. Yet, we do not expect to see great difference in their performance when the same SMT solver is used as a backend. Also, we would like to conduct more extensive experimental analysis to support our conjecture on the benefits of specialized "aggregate" propagator and *Unfounded* propagator employed by CLASP.

The technique implemented by CMODELS(DIFF) for enumerating multiple answer sets of a program is basic. In the future we would like to adopt the nontrivial methods for model enumeration discussed in [8] to our settings. The theory developed in this paper provides a foundation to extend the recent constraint answer set programming solver EZSMT [21] to accept non-tight constraint answer set programs. The contributions of this work also open a door to the development of a novel constraint-based method in processing logic programs by producing intermediate output in MINIZINC [18] in place of SMT-LIB. We believe our work will boost the cross-fertilization between the three areas: SMT, constraint answer set programming, and constraint programming.

Acknowledgements

We are grateful to Cesare Tinelli for valuable discussions on the subject of the paper and for the insights on the CVC4 system. We are also thankful to Ben Susman. Da Shen was supported by the 2017-FUSE (Fund for Undergraduate Scholarly Experiences) Grant from the University of Nebraska at Omaha. Yuliya Lierler was partially supported by the NSF 1707371 grant.

References

1. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.
2. Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014.

[9] https://research.ics.aalto.fi/software/asp/lp2normal/

3 Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

4 Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 242–254, 2001.

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

6 Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

7 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in CLASP series 3. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 2015.

8 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set enumeration. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pages 136–148, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1758481.1758496.

9 Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

10 Tomi Janhunen, Ilkka Niemelä, and Mark Sevalnev. Computing stable models via reductions to difference logic. In *Logic Programming and Nonmonotonic Reasoning*, pages 142–154. Springer Berlin Heidelberg, 2009.

11 Yuliya Lierler. *SAT-based Answer Set Programming.* PhD thesis, University of Texas at Austin, 2010.

12 Yuliya Lierler. What is answer set programming to propositional satisfiability. *Constraints*, pages 1–31, 2016. URL: http://dx.doi.org/10.1007/s10601-016-9257-7, doi:10.1007/s10601-016-9257-7.

13 Yuliya Lierler and Benjamin Susman. On relation between constraint answer set programming and satisfiability modulo theories. *Theory and Practice of Logic Programming*, 17(4):559–590, 2017.

14 Yuliya Lierler and Miroslaw Truszczyński. Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP) Special Issue*, 11(4-5):629–646, 2011.

15 Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

16 Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157:115–137, 2004.

17 Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

18 N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, , and G. Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, page 529–543, 2007.

19 Ilkka Niemela. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53:313–329, 2008.

20 Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

21 Benjamin Susman and Yuliya Lierler. SMT-Based Constraint Answer Set Solver EZSMT (System Description). In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52, pages 1:1–1:15, 2016.