Conflict-Driven Answer Set Solving: From Theory to Practice*

Martin Gebser Benjamin Kaufmann
Torsten Schaub†
Universität Potsdam, Institut für Informatik,
August-Bebel-Str. 89, D-14482 Potsdam, Germany
{gebser,kaufmann,torsten}@cs.uni-potsdam.de

May 4, 2012

Abstract

We introduce an approach to computing answer sets of logic programs, based on concepts successfully applied in Satisfiability (SAT) checking. The idea is to view inferences in Answer Set Programming (ASP) as unit propagation on nogoods. This provides us with a uniform constraint-based framework capturing diverse inferences encountered in ASP solving. Moreover, our approach allows us to apply advanced solving techniques from the area of SAT. As a result, we present the first full-fledged algorithmic framework for native conflict-driven ASP solving. Our approach is implemented in the ASP solver *clasp* that has demonstrated its competitiveness and versatility by winning first places at various solver contests.

1 Introduction

Answer Set Programming (ASP; [67, 94, 102, 66, 87, 6, 65]) has become an attractive paradigm for knowledge representation and reasoning, due to its appealing combination of rich yet simple modeling languages[1] with powerful solving engines. Albeit specialized ASP solvers have been highly optimized (cf. [119, 83, 15]), their performance has so far not matched the one of modern solvers for Satisfiability (SAT; [12]) checking. However, computational mechanisms of SAT and ASP solvers are not far-off, as witnessed by the SAT-based ASP solvers *assat* [90] and *cmodels* [71]. Nonetheless, state-of-the-art look-back techniques from SAT, or more generally,

*This paper combines and extends the work presented in [2, 54, 52, 56].

†Corresponding author: phone: (+49) 331 977 3080/3081, fax: (+49) 331 977 3122. Torsten Schaub is also affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

[1]The interested reader is referred to [120, 45, 83] for detailed accounts of ASP's modeling languages.

Constraint Programming (CP; [26, 113]), such as backjumping and conflict-driven learning, were not yet established in native ASP solvers. In fact, previous approaches to adopt such techniques [126, 112, 91] are rather implementation-specific, i.e., they focus on describing modifications of existing ASP solving approaches, and thus lack generality.

We address this deficiency by introducing a novel computational approach to ASP solving, building on Boolean constraints. Apart from the fact that this allows us to easily integrate solving techniques from related areas like SAT, e.g., backjumping, conflict-driven learning, restarts, etc., it also provides us with a uniform characterization of inferences from logic program rules, unfounded sets, and conflict conditions. As major results, we show that all inferences in ASP solving can be reduced to unit propagation on nogoods, and we devise the first self-contained algorithmic framework for native conflict-driven ASP solving. While the general outline of search is the same as in Conflict-Driven Clause Learning (CDCL; [97, 127, 23, 96]), the state-of-the-art algorithm for industrial SAT solving, the integration of unfounded set checking is particular to ASP and owed to its elevated expressiveness (cf. [117, 76, 88]). However, our approach favors "local" unit propagation over unfounded set checks, i.e., tests whether inherent (loop) nogoods are unit or violated. We elaborate upon the formal properties of our conflict-driven algorithmic framework, and we demonstrate its soundness and completeness for ASP solving.

Our approach has led to the implementation of the award-winning ASP solver *clasp*, taking first places at the ASP, CASC, MISC, PB, and SAT contests in 2011 (see [110] for more details). We discuss the major features of *clasp* and provide an empirical evaluation of its performance by comparing it to other state-of-the-art ASP solvers, using the class of NP decision problems from the second ASP competition [28]. Generally, *clasp* has become a powerful native ASP solver, offering various reasoning modes that make it an attractive tool for knowledge representation and reasoning.[2] This is witnessed by an increasing number of applications relying on *clasp* (or derivatives) as reasoning engine, e.g., [99, 13, 80, 122, 75, 64]. Along with the grounder *gringo* [51], *clasp* constitutes a central component of *Potassco* [44], the Potsdam Answer Set Solving Collection bundling tools for ASP developed at the University of Potsdam.

The outline of this paper is as follows. After establishing the formal background, we provide in Section 3 a constraint-based specification of answer sets in terms of nogoods. Based on this uniform characterization, we develop in Section 4 algorithms for ASP solving that incorporate advanced look-back techniques. In Section 5, we describe the award-winning ASP solver *clasp*, implementing our approach. Section 6 provides a systematic empirical evaluation demonstrating the competitiveness of *clasp*. We conclude with related work and summary. Proofs for formal results are provided in the appendix.

2 Background

Given an alphabet $\mathcal{P}$, a (propositional normal) *logic program* is a finite set of rules of the form

[2]Beyond search for one answer set of a propositional normal logic program, detailed in this paper, *clasp* supports so-called extended rules [47], solution enumeration [53, 57], and optimization [48]. Due to its versatile core engine, *clasp* can be run as a solver for ASP, SAT, Maximum Satisfiability (MaxSAT; [85]), and Pseudo-Boolean (PB; [114]) constraint satisfaction/optimization, incorporating dedicated front-ends for diverse input formats.

$$p_0 \leftarrow p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n \tag{1}$$

where $0 \leq m \leq n$ and each $p_i \in \mathcal{P}$ is an *atom* for $0 \leq i \leq n$. A *body literal* is an atom p or its (default) negation $\mathit{not}\ p$. For a rule r as in (1), let $\mathit{head}(r) = p_0$ be the *head* of r and $\mathit{body}(r) = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}$ be the *body* of r. The intuitive reading of r is that $\mathit{head}(r)$ must be true if $\mathit{body}(r)$ holds, i.e., if $p_1, \dots, p_m$ are (provably) true and if $p_{m+1}, \dots, p_n$ are (assumed to be) false. Given a set β of body literals, let $\beta^+ = \{p \in \mathcal{P} \mid p \in \beta\}$ and $\beta^- = \{p \in \mathcal{P} \mid \mathit{not}\ p \in \beta\}$. For $\mathit{body}(r)$, we then have that $\mathit{body}(r)^+ = \{p_1, \dots, p_m\}$ and $\mathit{body}(r)^- = \{p_{m+1}, \dots, p_n\}$. The set of atoms occurring in a logic program Π is denoted by $\mathit{atom}(\Pi)$, and $\mathit{body}(\Pi) = \{\mathit{body}(r) \mid r \in \Pi\}$ is the set of bodies of rules in Π. For regrouping rule bodies sharing the same head p, we define $\mathit{body}_\Pi(p) = \{\mathit{body}(r) \mid r \in \Pi, \mathit{head}(r) = p\}$.

disjunctive,, one of these three conditions holds.

A set $X \subseteq \mathcal{P}$ of atoms is a *model* of a logic program Π, if $\mathit{head}(r) \in X$, $\mathit{body}(r)^+ \not\subseteq X$, or $\mathit{body}(r)^- \cap X \neq \emptyset$ holds for every $r \in \Pi$. In ASP, the semantics of Π is given by its answer sets [67]. The *reduct*, Π^X, of Π relative to X is defined by $\Pi^X = \{\mathit{head}(r) \leftarrow \mathit{body}(r)^+ \mid r \in \Pi, \mathit{body}(r)^- \cap X = \emptyset\}$. Note that Π^X is a Horn program possessing a unique $\subseteq$-minimal model (cf. [30]). Given this, X is an *answer set* of Π, if X itself is the $\subseteq$-minimal model of Π^X. Note that any answer set of Π is a model of Π as well, while the converse does not hold in general.

The *positive dependency graph* of a program Π is given by $(\mathit{atom}(\Pi), \leq^+)$, where $\mathit{atom}(\Pi)$ and $\leq^+ = \{(p, \mathit{head}(r)) \mid r \in \Pi, p \in \mathit{body}(r)^+\}$ are the set of vertices and directed edges, respectively. This graph allows us to identify circular positive dependencies among atoms. According to [90], a non-empty $L \subseteq \mathit{atom}(\Pi)$ is a *loop* of Π, if for every pair $p \in L, q \in L$ (including $p = q$), there is a path of non-zero length from p to q in $(\mathit{atom}(\Pi), \leq^+)$ such that all vertices in the path belong to L. We denote the set of all loops of Π by $\mathit{loop}(\Pi)$; if $\mathit{loop}(\Pi) = \emptyset$ (or $\mathit{loop}(\Pi) \neq \emptyset$), Π is a *tight* (or *non-tight*) program. As shown in [40] and exploited in Section 3, the answer sets of a tight program Π coincide with models of the Clark completion of Π [21], also referred to as the supported models of Π [3]. A *strongly connected component* of $(\mathit{atom}(\Pi), \leq^+)$ is a maximal subgraph such that any pair of vertices is connected by some path; it is *non-trivial*, if it contains some edge. Note that, for any loop L of Π, the atoms in L belong to the same non-trivial strongly connected component of $(\mathit{atom}(\Pi), \leq^+)$. Moreover, we have that Π is tight iff $(\mathit{atom}(\Pi), \leq^+)$ does not include any non-trivial strongly connected component.

Example 2.1. *Consider the following logic program:*[3]

$$\Pi_2 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\} \tag{2}$$

This program has two answer sets: $\{a, c\}$ and $\{a, d\}$. Note that Π_2 is non-tight because its positive dependency graph contains the non-trivial strongly connected component $(\{e\}, \{(e, e)\})$.

In practice, propositional logic programs are usually obtained from inputs in some first-order language (cf. [120, 45, 83]) via grounding. We do not detail grounding here, but only mention that off-the-shelf grounders, such as *dlv*'s grounding component [105], *gringo* [51], and *lparse* [120], are available to accomplish this task. Moreover, particular classes of logic programs

[3]Our enumeration scheme for particular logic programs Π follows that of equations.

admit language extensions like classical negation and disjunctions [68], nested expressions [89], propositional formulas [106, 41], cardinality and weight constraints [119], or aggregates [39], to name some of them. In this paper, we focus mainly on normal propositional logic programs and, in particular, on solving the decision problem of answer set existence. For further details and broader overviews about the area of ASP, we refer the interested reader to [6, 65].

The concepts introduced next are central in the context of conflict-driven answer set computation. A Boolean *assignment* $\mathbf{A}$ over a *domain*, $\mathit{dom}(\mathbf{A})$, is a sequence $(\sigma_1, \dots, \sigma_n)$ of *signed literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$, where $v_i \in \mathit{dom}(\mathbf{A})$ for $1 \leq i \leq n$ and $v_i \neq v_j$ for $i < j \leq n$. A literal $\mathbf{T}v$ expresses that v is *true*, and $\mathbf{F}v$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. The sequence obtained by appending a literal σ to $\mathbf{A}$ is denoted by $\mathbf{A} \circ \sigma$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and the false members of $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \in \mathit{dom}(\mathbf{A}) \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \in \mathit{dom}(\mathbf{A}) \mid \mathbf{F}v \in \mathbf{A}\}$. For $\mathbf{A} = (\sigma_1, \dots, \sigma_{i-1}, \sigma_i, \dots, \sigma_n)$, let $\mathbf{A}[\sigma_i] = (\sigma_1, \dots, \sigma_{i-1})$ be the prefix of $\mathbf{A}$ relative to σ_i, and define $\mathbf{A}[\sigma] = \mathbf{A}$ for any $\sigma \notin \mathbf{A}$.

For a canonical representation of Boolean constraints, we rely on the concept of a nogood (cf. [26, 113]), reflecting (partial) assignments that cannot be extended to a solution. In our setting, a *nogood* is a set $\{\sigma_1, \dots, \sigma_m\}$ of signed literals, expressing that any assignment containing $\sigma_1, \dots, \sigma_m$ is unintended.[4] Accordingly, a nogood $\{\sigma_1, \dots, \sigma_m\}$ is *violated* by an assignment $\mathbf{A}$, if $\{\sigma_1, \dots, \sigma_m\} \subseteq \mathbf{A}$. In turn, an assignment $\mathbf{A}$ such that $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = \mathit{dom}(\mathbf{A})$ is a *solution* for a set Δ of nogoods, if no nogood in Δ is violated by $\mathbf{A}$.

Given a nogood δ and an assignment $\mathbf{A}$, we say that a literal $\sigma \notin \mathbf{A}$ is *unit-resulting* for δ wrt $\mathbf{A}$, if $\delta \setminus \mathbf{A} = \{\overline{\sigma}\}$. This designates $\overline{\sigma}$ as the single literal of δ not contained in $\mathbf{A}$, so that σ must necessarily be added to $\mathbf{A}$ for avoiding the violation of δ. Note that a violated nogood does not have any unit-resulting literal, while the prerequisite $\sigma \notin \mathbf{A}$ precludes duplicates: if $\mathbf{A}$ already contains σ, it cannot be unit-resulting.[5] For example, $\mathbf{F}q$ is unit-resulting for nogood $\{\mathbf{F}p, \mathbf{T}q\}$ wrt assignment $(\mathbf{F}p)$, but neither wrt $(\mathbf{F}p, \mathbf{F}q)$ nor $(\mathbf{F}p, \mathbf{T}q)$. Along the lines of SAT, for a set Δ of nogoods, we refer to the iterated process of extending $\mathbf{A}$ by unit-resulting literals as *unit propagation*. We call a nogood δ an *antecedent* of σ wrt $\mathbf{A}$, if σ is unit-resulting for δ wrt prefix $\mathbf{A}[\sigma]$. In turn, $\sigma \in \mathbf{A}$ is *implied* by Δ wrt $\mathbf{A}$, if there is some antecedent of σ wrt $\mathbf{A}$ in Δ.

3 Nogoods of Logic Programs

Inferences in ASP solving rely on truth values of atoms and applicability of program rules, which can be expressed by assignments over atoms and bodies. Given a program Π, we thus fix the domain of assignments $\mathbf{A}$ to $\mathit{dom}(\mathbf{A}) = \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$. Such a hybrid approach may result in exponentially smaller search space traversals than a purely either atom- or body-based

[4]Any nogood $\{\sigma_1, \dots, \sigma_m\}$ can be syntactically represented by a clause $\overline{\sigma_1} \vee \dots \vee \overline{\sigma_m}$ (dropping $\mathbf{T}$ and replacing $\mathbf{F}$ with $\neg$ in $\overline{\sigma_1}, \dots, \overline{\sigma_m}$ to stay in the syntax of propositional logic), while other representations like logic program rules, PB constraints, and Boolean circuit gates are possible as well.

[5]The concept of a unit-resulting literal is closely related to unit clauses considered in SAT (cf. [12]): a clause is unit iff the nogood it represents has some unit-resulting literal.

approach [63, 62]; it moreover allows for a succinct representation of nogoods, as we show in this section. While syntactic translations of logic programs to clauses (cf. [5, 70, 86]) primarily aim at reducing ASP to SAT solving, the nogoods provided below describe semantic conditions under which an assignment over $atom(\Pi) \cup body(\Pi)$ represents an answer set of Π.

Our approach is guided by the idea of Lin and Zhao [90] and decomposes ASP solving into "local" inferences obtainable from the Clark completion [21] of a program and those obtainable from loop formulas [90]. While Clark completion captures the answer sets of tight programs compactly in terms of theories in propositional logic [40], exponentially many loop formulas may be required in addition to extend this characterization of answer sets to non-tight programs [88].

3.1 Completion Nogoods

We begin with nogoods capturing constraints induced by the Clark completion of a program, where we use $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ for denoting the classical connectives in propositional logic. Then, the *Clark completion* of a program Π can be defined as follows:

$$\begin{aligned} &\{\, p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \mid \\ &\qquad\qquad \beta \in body(\Pi), \beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\} \,\} && (3)\\ \cup\ &\{\, p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k} \mid p \in atom(\Pi), body_\Pi(p) = \{\beta_1, \ldots, \beta_k\} \,\} \ . && (4) \end{aligned}$$

This formulation relies on atoms p as well as auxiliary propositions p_β representing bodies β. Such propositions are also introduced in Conjunctive Normal Form (CNF) transformations as abbreviations avoiding an exponential blow-up [123]. The models of the Clark completion of a program are called *supported models* [3]; on tight programs, they coincide with answer sets [40].

The equivalences in (3) define propositions standing for bodies, while those in (4) define atoms in terms of their supporting bodies. For identifying the underlying constraints, we begin with the body-oriented equivalences in (3). In fact, they can be decomposed into two kinds of implications considered next.

On the one hand, we obtain $(p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \rightarrow p_\beta)$. That is, the body β of a rule holds if all its body literals are true. Conversely, some literal of β must be false if β does not hold. So, given a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, the previous implication expresses the nogood:

$$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \ .$$

Note that nogoods treat atoms and bodies as equitable objects. In terms of unit propagation, $\delta(\beta)$ is a constraint enforcing the truth of β or the falsity of a contained literal. For instance, for body $\{a, not\ d\}$ in Program Π_2, we obtain $\delta(\{a, not\ d\}) = \{\mathbf{F}\{a, not\ d\}, \mathbf{T}a, \mathbf{F}d\}$ (see also Table 1).

On the other hand, a body β must be false if some of its literals is false, or all literals of β must be true if β holds. This is expressed by the second implication obtained from (3), viz., $(p_\beta \rightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n)$. It is equivalent to the conjunction of clauses $(\neg p_\beta \vee p_1), \ldots, (\neg p_\beta \vee p_m), (\neg p_\beta \vee \neg p_{m+1}), \ldots, (\neg p_\beta \vee \neg p_n)$. For $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, such clauses induce the following set of nogoods:

$$\Delta(\beta) = \{\, \{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\} \,\} \ .$$

Body β	$\delta(\beta)$	$\Delta(\beta)$
$\emptyset$	$\{\mathbf{F}\emptyset\}$	
$\{\mathit{not}\ a\}$	$\{\mathbf{F}\{\mathit{not}\ a\}, \mathbf{F}a\}$	$\{\mathbf{T}\{\mathit{not}\ a\}, \mathbf{T}a\}$
$\{a, \mathit{not}\ d\}$	$\{\mathbf{F}\{a, \mathit{not}\ d\}, \mathbf{T}a, \mathbf{F}d\}$	$\{\mathbf{T}\{a, \mathit{not}\ d\}, \mathbf{F}a\}$ $\{\mathbf{T}\{a, \mathit{not}\ d\}, \mathbf{T}d\}$
$\{\mathit{not}\ c, \mathit{not}\ e\}$	$\{\mathbf{F}\{\mathit{not}\ c, \mathit{not}\ e\}, \mathbf{F}c, \mathbf{F}e\}$	$\{\mathbf{T}\{\mathit{not}\ c, \mathit{not}\ e\}, \mathbf{T}c\}$ $\{\mathbf{T}\{\mathit{not}\ c, \mathit{not}\ e\}, \mathbf{T}e\}$
$\{b\}$	$\{\mathbf{F}\{b\}, \mathbf{T}b\}$	$\{\mathbf{T}\{b\}, \mathbf{F}b\}$
$\{e\}$	$\{\mathbf{F}\{e\}, \mathbf{T}e\}$	$\{\mathbf{T}\{e\}, \mathbf{F}e\}$

Table 1: Body-oriented nogoods in $\Delta_{\mathit{body}(\Pi_2)}$.

Taking again body $\{a, \mathit{not}\ d\}$ gives $\Delta(\{a, \mathit{not}\ d\}) = \{\,\{\mathbf{T}\{a, \mathit{not}\ d\}, \mathbf{F}a\}, \{\mathbf{T}\{a, \mathit{not}\ d\}, \mathbf{T}d\}\,\}$.

We now come to constraints primarily aiming at atoms. In analogy to the above, we derive the corresponding nogoods from the equivalences in (4).

To begin with, the implication $(p_{\beta_1} \vee \cdots \vee p_{\beta_k} \rightarrow p)$ tells us that an atom p must be true if some element of $\mathit{body}_\Pi(p)$ holds and, conversely, that all elements of $\mathit{body}_\Pi(p)$ must be false if p is false. For $\mathit{body}_\Pi(p) = \{\beta_1, \ldots, \beta_k\}$, we thus get the set of nogoods:[6]

$$\Delta(p) = \{\,\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\,\}\ .$$

For example, for atom e in Program Π_2 with $\mathit{body}_{\Pi_2}(e) = \{\{b\}, \{e\}\}$, we obtain $\Delta(e) = \{\,\{\mathbf{F}e, \mathbf{T}\{b\}\}, \{\mathbf{F}e, \mathbf{T}\{e\}\}\,\}$ (see also Table 2).

Finally, according to the implication $(p \rightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k})$, some element of $\mathit{body}_\Pi(p)$ must hold if p is true, or p must be false if all elements of $\mathit{body}_\Pi(p)$ are false. For $\mathit{body}_\Pi(p) = \{\beta_1, \ldots, \beta_k\}$, this is reflected by the following nogood:

$$\delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}\ .$$

Taking once more atom e with $\mathit{body}_{\Pi_2}(e) = \{\{b\}, \{e\}\}$, we get $\delta(e) = \{\mathbf{T}e, \mathbf{F}\{b\}, \mathbf{F}\{e\}\}$.

Combining the four types of nogoods stemming from the Clark completion of a program Π, we obtain the following sets of nogoods:

$$\begin{aligned}
\Delta_{\mathit{body}(\Pi)} &= \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} && (5)\\
\Delta_{\mathit{atom}(\Pi)} &= \{\delta(p) \mid p \in \mathit{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \mathit{atom}(\Pi)\} && (6)\\
\Delta_\Pi &= \Delta_{\mathit{body}(\Pi)} \cup \Delta_{\mathit{atom}(\Pi)}\ .
\end{aligned}$$

For illustration, in Table 1 and 2, we provide the set Δ_{Π_2} of nogoods stemming from the Clark completion of Program Π_2. While Table 1 shows the body-oriented nogoods in $\Delta_{\mathit{body}(\Pi_2)}$, Table 2 comprises the atom-oriented ones in $\Delta_{\mathit{atom}(\Pi_2)}$.

In what follows, we develop the result of this section that solutions for Δ_Π coincide with supported models, and if Π is tight, also with answer sets of Π. As an auxiliary property, we have that the truth values of bodies are uniquely determined by those of atoms.

[6]For the sake of simplicity, we leave an underlying program Π implicit in the notations $\Delta(p)$ and $\delta(p)$.

Atom p	$\Delta(p)$	$\delta(p)$
a	$\{\mathbf{F}a, \mathbf{T}\emptyset\}$	$\{\mathbf{T}a, \mathbf{F}\emptyset\}$
b	$\{\mathbf{F}b, \mathbf{T}\{\mathit{not}\ a\}\}$	$\{\mathbf{T}b, \mathbf{F}\{\mathit{not}\ a\}\}$
c	$\{\mathbf{F}c, \mathbf{T}\{a, \mathit{not}\ d\}\}$	$\{\mathbf{T}c, \mathbf{F}\{a, \mathit{not}\ d\}\}$
d	$\{\mathbf{F}d, \mathbf{T}\{\mathit{not}\ c, \mathit{not}\ e\}\}$	$\{\mathbf{T}d, \mathbf{F}\{\mathit{not}\ c, \mathit{not}\ e\}\}$
e	$\{\mathbf{F}e, \mathbf{T}\{b\}\}$ $\{\mathbf{F}e, \mathbf{T}\{e\}\}$	$\{\mathbf{T}e, \mathbf{F}\{b\}, \mathbf{F}\{e\}\}$

Table 2: Atom-oriented nogoods in $\Delta_{atom(\Pi_2)}$.

Lemma 3.1. *Let* Π *be a logic program and* $X \subseteq atom(\Pi)$.
Then, we have that

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in atom(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in body(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in body(\Pi), (\beta^+ \cap (atom(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for $\Delta_{body(\Pi)}$ *such that* $\mathbf{A}^{\mathbf{T}} \cap atom(\Pi) = X$.

Observe that, for a given $X \subseteq atom(\Pi)$, the unique solution $\mathbf{A}$ for $\Delta_{body(\Pi)}$ must assign $\mathbf{T}$ or $\mathbf{F}$ to $\beta \in body(\Pi)$ according to the semantics of conjunction, as it is expected.

The next auxiliary result establishes one-to-one correspondence between supported models of Π, satisfying the equivalences in (3) and (4), and solutions for Δ_Π.

Lemma 3.2. *Let* Π *be a logic program and* $X \subseteq atom(\Pi) \cup body(\Pi)$.
Then, we have that $(X \cap atom(\Pi)) \cup \{p_\beta \mid \beta \in X \cap body(\Pi)\}$ *is a supported model of* Π *iff* $\{\mathbf{T}v \mid v \in X\} \cup \{\mathbf{F}v \mid v \in (atom(\Pi) \cup body(\Pi)) \setminus X\}$ *is a solution for* Δ_Π.

Since supported models and answer sets coincide on tight programs Π, we further obtain the following correspondence between answer sets of Π and solutions for Δ_Π.

Theorem 3.3. *Let* Π *be a tight logic program and* $X \subseteq atom(\Pi)$.
Then, we have that X *is an answer set of* Π *iff*

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in atom(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in body(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in body(\Pi), (\beta^+ \cap (atom(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for Δ_Π *such that* $\mathbf{A}^{\mathbf{T}} \cap atom(\Pi) = X$.

Example 3.1. *For illustration, let us inspect the supported models and answer sets of* Π_2 *from Example 2.1, which is non-tight because of rule* $(e \leftarrow e)$. *The equivalences of the Clark completion of* Π_2 *are:*

$$\begin{array}{lllll} a \leftrightarrow p_\emptyset & b \leftrightarrow p_{\{not\ a\}} & c \leftrightarrow p_{\{a, not\ d\}} & d \leftrightarrow p_{\{not\ c, not\ e\}} & e \leftrightarrow p_{\{b\}} \vee p_{\{e\}} \\ p_\emptyset \leftrightarrow \top & p_{\{not\ a\}} \leftrightarrow \neg a & p_{\{a, not\ d\}} \leftrightarrow a \wedge \neg d & p_{\{not\ c, not\ e\}} \leftrightarrow \neg c \wedge \neg e & p_{\{b\}} \leftrightarrow b \quad p_{\{e\}} \leftrightarrow e\ . \end{array}$$

Supported Model	Assignment
$\{\, a, d \,\} \cup$ $\{\, p_\emptyset, p_{\{not\ c, not\ e\}} \,\}$	$\{\, \mathbf{T}a, \mathbf{T}d, \mathbf{F}c, \mathbf{F}e, \mathbf{F}b \,\} \cup$ $\{\, \mathbf{T}\emptyset, \mathbf{T}\{not\ c, not\ e\}, \mathbf{F}\{a, not\ d\}, \mathbf{F}\{e\}, \mathbf{F}\{not\ a\}, \mathbf{F}\{b\} \,\}$
$\{\, a, c \,\} \cup$ $\{\, p_\emptyset, p_{\{a, not\ d\}} \,\}$	$\{\, \mathbf{T}a, \mathbf{F}d, \mathbf{T}c, \mathbf{F}e, \mathbf{F}b \,\} \cup$ $\{\, \mathbf{T}\emptyset, \mathbf{F}\{not\ c, not\ e\}, \mathbf{T}\{a, not\ d\}, \mathbf{F}\{e\}, \mathbf{F}\{not\ a\}, \mathbf{F}\{b\} \,\}$
$\{\, a, c, e \,\} \cup$ $\{\, p_\emptyset, p_{\{a, not\ d\}}, p_{\{e\}} \,\}$	$\{\, \mathbf{T}a, \mathbf{F}d, \mathbf{T}c, \mathbf{T}e, \mathbf{F}b \,\} \cup$ $\{\, \mathbf{T}\emptyset, \mathbf{F}\{not\ c, not\ e\}, \mathbf{T}\{a, not\ d\}, \mathbf{T}\{e\}, \mathbf{F}\{not\ a\}, \mathbf{F}\{b\} \,\}$

Table 3: Supported models of Π_2 and corresponding solutions for Δ_{Π_2}.

The supported models of Π_2 and corresponding solutions for Δ_{Π_2} are shown in Table 3. Note that the atoms belonging to the first two supported models correspond to answer sets of Π_2, but not those in the third one. The reason for this mismatch is rule $(e \leftarrow e)$, which makes Π_2 non-tight. When dropping this rule from Π_2, the first two supported models remain valid, while e in the third one is no longer supported. By Lemma 3.2, this allows us to conclude that the first two assignments (without $\mathbf{F}\{e\}$) are solutions for $\Delta_{\Pi_2 \setminus \{e \leftarrow e\}}$. Since $\Pi_2 \setminus \{e \leftarrow e\}$ is tight, we can further use Theorem 3.3 to see that $\{a, d\}$ and $\{a, c\}$ are the two answer sets of $\Pi_2 \setminus \{e \leftarrow e\}$.

As pointed out at the beginning of this section, the nogoods contributing to $\Delta_{body(\Pi)}$ in (5) and $\Delta_{atom(\Pi)}$ in (6) are directly linked to the clauses obtained when decomposing the equivalences in the Clark completion of a program Π in the straightforward way. Hence, the nogoods in Δ_Π essentially characterize supported models in terms of assignments over atoms as well as rule bodies. Note that each atom and each body is defined by some equivalence, given implicitly through (the semantics of) Π or written explicitly in the Clark completion of Π. In a sense, one can view Π as a shorthand representation for the conditions that all rules must be fulfilled and that any true atom must be supported via some rule whose body holds.

3.2 Loop Nogoods

Every answer set of a program Π is also a supported model of Π, while the converse does not hold in general. In fact, the mismatch on non-tight programs is due to the potential of circular support (or positive recursion) among true atoms, which is admissible with supported models, but not with answer sets. Hence, such improper support must be suppressed to distinguish supported models that are answer sets from the rest, and there are several approaches to accomplish this. On the one hand, well-founded semantics is based on unfounded sets [124], viz., sets of atoms that cannot be non-circularly supported and must thus be false. While unfounded sets are traditionally determined wrt partial interpretations over atoms, an alternative approach identifying unfounded sets wrt (false) rule bodies is described in [2]. On the other hand, loop formulas can be utilized to characterize answer sets by classical models of propositional theories. Here, the main focus is on restricting the consideration of unfounded sets to particular (syntactic) classes of sets of atoms, namely, loops [90, 82] or elementary sets [61, 58]. In this section, we introduce the concept of an unfounded set in our setting and relate it to traditional approaches [124, 84]. We further exploit unfounded sets to extend our constraint-based characterization of answer sets to non-tight programs, which yields a close relationship to loop formulas.

To begin with, for a program Π and some $U \subseteq atom(\Pi)$, we define the *external bodies* of U for Π as

$$EB_\Pi(U) = \{body(r) \mid r \in \Pi, head(r) \in U, body(r)^+ \cap U = \emptyset\} \ .$$

A body in $EB_\Pi(U)$ can provide non-circular support for U, as it does not (positively) contain any atom of U. Then, U is unfounded, if all its external bodies are false, that is, if there is no non-circular support left for U. In our setting, this amounts to the following definition.

Definition 3.1. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi)$.
Then, we define U *as an unfounded set of* Π *wrt* $\mathbf{A}$*, if* $EB_\Pi(U) \subseteq \mathbf{A^F}$.

In more detail, this definition determines U as unfounded for Π wrt $\mathbf{A}$, if for every $r \in \Pi$, we have that $head(r) \notin U$, $body(r)^+ \cap U \neq \emptyset$, or $body(r) \in \mathbf{A^F}$. For comparison, the traditional unfounded set definition by Van Gelder, Ross, and Schlipf [124] can be reformulated wrt assignments as follows.

Definition 3.2. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi)$.
Then, we define U *as a GRS-unfounded set of* Π *wrt* $\mathbf{A}$*, if*

$$EB_\Pi(U) \subseteq \{\beta \in body(\Pi) \mid (\beta^+ \cap \mathbf{A^F}) \cup (\beta^- \cap \mathbf{A^T}) \neq \emptyset\} \ .$$

Note that this definition requires an external body to contain a false literal in order to witness the unavailability of non-circular support.

For comparing our concept of an unfounded set to the traditional one, we define the following properties for a program along with an assignment.

Definition 3.3. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment.*
Then, we define $\mathbf{A}$ *as*

1. *body-saturated for* Π*, if*

$$\{\beta \in body(\Pi) \mid (\beta^+ \cap \mathbf{A^F}) \cup (\beta^- \cap \mathbf{A^T}) \neq \emptyset\} \subseteq \mathbf{A^F} \ ;$$

2. *body-synchronized for* Π*, if*

$$\{\beta \in body(\Pi) \mid (\beta^+ \cap \mathbf{A^F}) \cup (\beta^- \cap \mathbf{A^T}) \neq \emptyset\} = \mathbf{A^F} \cap body(\Pi) \ .$$

In words, body-saturation requires that bodies containing false literals must likewise be assigned to false; if the converse holds as well, we have body-synchronization.

Based on these properties, we now formalize the relationships between GRS-unfounded sets and our unfounded set notion.

Proposition 3.4. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi)$.
If $\mathbf{A}$ *is body-saturated for* Π*, then we have that* U *is an unfounded set of* Π *wrt* $\mathbf{A}$ *if* U *is a GRS-unfounded set of* Π *wrt* $\mathbf{A}$.

$$\begin{tikzcd}
 & & b \arrow[dl] \arrow[d] \arrow[dr] & \\
a \arrow[r] & c \arrow[r, bend left] \arrow[rr, bend right] & d \arrow[l, bend left] \arrow[r, bend left] & e \arrow[l, bend left]
\end{tikzcd}$$

Figure 1: The positive dependency graph of Π_7.

Proposition 3.5. *Let Π be a logic program, $\mathbf{A}$ an assignment, and $U \subseteq atom(\Pi)$.*

If $\mathbf{A}$ is body-synchronized for Π, then we have that U is an unfounded set of Π wrt $\mathbf{A}$ iff U is a GRS-unfounded set of Π wrt $\mathbf{A}$.

These results show that any GRS-unfounded set can be turned into an unfounded set by assigning bodies containing false literals to false as well, in this way establishing body-saturation. For a body β, the nogoods in $\Delta(\beta)$ enable such forwarding of falsity by unit propagation (cf. Section 2). As the following example illustrates, there is no straightforward converse "transformation" to turn unfounded sets into GRS-unfounded sets.

Example 3.2. *Consider the following (non-tight) program:*

$$\Pi_7 = \left\{ \begin{array}{llll} a \leftarrow not\ b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, not\ a \\ b \leftarrow not\ a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\} \tag{7}$$

The positive dependency graph of Π_7 is shown in Figure 1. Observe that $\{c, d\}$, $\{d, e\}$, and $\{c, d, e\}$ are all non-empty sets of atoms such that their elements reach one another via (loop-internal) paths of non-zero length, i.e., $loop(\Pi_7) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$. In particular, $U = \{d, e\}$ is unfounded for Π_7 wrt $\mathbf{A} = (\mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\})$ in view of $EB_{\Pi_7}(U) = \{\{b, c\}, \{b, not\ a\}\}$. This tells us that any answer set of Π_7 such that rules $(d \leftarrow b, c)$ and $(e \leftarrow b, not\ a)$ are inapplicable (i.e., their bodies $\{b, c\}$ and $\{b, not\ a\}$ do not hold) must not contain d or e. In fact, the remaining rules supporting d and e, $(d \leftarrow e)$ and $(e \leftarrow c, d)$, are circular and do thus not provide external support for U. However, U is not a GRS-unfounded set of Π_7 wrt $\mathbf{A}$, and neither nogood $\delta(\{b, c\}) = \{\mathbf{F}\{b, c\}, \mathbf{T}b, \mathbf{T}c\}$ nor $\delta(\{b, not\ a\}) = \{\mathbf{F}\{b, not\ a\}, \mathbf{T}b, \mathbf{F}a\}$ allows for deriving the falsity of any body literal by unit propagation wrt $\mathbf{A}$. That is, the fact that the remaining supports for d and e are circular is not reflected by GRS-unfounded sets. On the other hand, we have that U is a GRS-unfounded set of Π_7 wrt $\mathbf{B} = (\mathbf{F}b)$ because positive body literal b in $\{b, c\}$ and $\{b, not\ a\}$ is false wrt $\mathbf{B}$. The mismatch that U is not an unfounded set of Π_7 wrt $\mathbf{B}$ is due to $\mathbf{B}$ not being body-saturated for Π_7. Yet a body-saturated assignment $\mathbf{B}'$ containing $\mathbf{F}\{b, c\}$ and $\mathbf{F}\{b, not\ a\}$ is easily derived from $\mathbf{B}$ by unit propagation, in view of nogoods $\{\mathbf{T}\{b, c\}, \mathbf{F}b\}$ and $\{\mathbf{T}\{b, not\ a\}, \mathbf{F}b\}$ belonging to $\Delta(\{b, c\})$ and $\Delta(\{b, not\ a\})$, respectively. Then, we have that U is an unfounded set of Π_7 wrt $\mathbf{B}'$.

In order to identify constraints induced by unfounded sets, we inspect loop formulas. Reusing auxiliary propositions for rule bodies, as given in (3), for a program Π and $U \subseteq atom(\Pi)$, the (disjunctive) *loop formula* can be written as follows:

$$\left(\bigvee\nolimits_{p \in U} p\right) \rightarrow \left(\bigvee\nolimits_{\beta \in EB_{\Pi}(U)} p_\beta\right) .$$

Such a loop formula stipulates at least one body in $EB_\Pi(U)$ to hold whenever some atom of U is true. An alternative reading is that all elements of U must be false if U is unfounded. For Program Π_7 and $U = \{d, e\}$, we get loop formula $(d \vee e \rightarrow p_{\{b,c\}} \vee p_{\{b,not\ a\}})$; the corresponding clauses are $(\neg d \vee p_{\{b,c\}} \vee p_{\{b,not\ a\}})$ and $(\neg e \vee p_{\{b,c\}} \vee p_{\{b,not\ a\}})$.

To capture unit propagation on loop formulas, for a program Π, a non-empty $U \subseteq atom(\Pi)$, and some $p \in U$, we define a *loop nogood* by

$$\lambda(p, U) = \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}$$

where $EB_\Pi(U) = \{\beta_1, \ldots, \beta_k\}$. For Program Π_7 and $U = \{d, e\}$, we obtain $\lambda(d, U) = \{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\}\}$ and $\lambda(e, U) = \{\mathbf{T}e, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\}\}$. Notice that literals of the form $\mathbf{F}\beta$, where $\beta \in EB_\Pi(U)$, are the same in $\lambda(p, U)$ for all $p \in U$.

Overall, we get the following set of loop nogoods for a program Π:

$$\Lambda_\Pi = \bigcup_{\emptyset \subset U \subseteq atom(\Pi)} \{\lambda(p, U) \mid p \in U\} \ .$$

The next result describes the relationship between loop nogoods and unfounded sets.

Proposition 3.6. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment such that* $\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F} = atom(\Pi) \cup body(\Pi)$.

Then, we have that $\mathbf{A}$ *is a solution for* Λ_Π *iff* $U \subseteq \mathbf{A}^\mathbf{F}$ *for every unfounded set* U *of* Π *wrt* $\mathbf{A}$.

In combination with Proposition 3.5, the previous result tells us that a body-synchronized total assignment $\mathbf{A}$ is *unfounded-free* [84] iff $\mathbf{A}$ is a solution for Λ_Π. Along with Lemma 3.1, which establishes that any solution for Δ_Π is body-synchronized, this now allows us to extend Theorem 3.3 to non-tight programs.

Theorem 3.7. *Let* Π *be a logic program and* $X \subseteq atom(\Pi)$.

Then, we have that X *is an answer set of* Π *iff*

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in atom(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in body(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in body(\Pi), (\beta^+ \cap (atom(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for $\Delta_\Pi \cup \Lambda_\Pi$ *such that* $\mathbf{A}^\mathbf{T} \cap atom(\Pi) = X$.

We have thus established that the nogoods in $\Delta_\Pi \cup \Lambda_\Pi$ describe a set of constraints that need to be checked for identifying answer sets. However, while the size of Δ_Π is linear in the size of Π, the one of Λ_Π is, in general, exponential. As shown by Lifschitz and Razborov [88], the latter is not a defect in the construction of Λ_Π, but an implication of widely accepted assumptions in complexity theory. Hence, most answer set solvers work on logic programs as succinct representations of loop nogoods (or formulas, respectively) and check them efficiently by determining unfounded sets relative to assignments. To this end, program structure, viz., loops or elementary sets, can be used to confine unfounded set checking to necessary parts.

In the remainder of this section, we detail the theoretical foundations for the completeness of our loop-oriented unfounded set detection algorithm, presented in Section 4.3. To begin with, we note that, under the assumption of body-saturation, we may eliminate false atoms from an unfounded set in order to obtain an unfounded set of non-false atoms only.

Proposition 3.8. *Let Π be a logic program, $\mathbf{A}$ an assignment, and U an unfounded set of Π wrt $\mathbf{A}$.*

If $\mathbf{A}$ is body-saturated for Π, then we have that $U \setminus \mathbf{A}^{\mathbf{F}}$ is an unfounded set of Π wrt $\mathbf{A}$.

For instance, $U = \{b, d, e\}$ is an unfounded set of Π_7 wrt body-saturated assignment $\mathbf{A} = (\mathbf{F}\{not\ a\}, \mathbf{F}b, \mathbf{F}\{b, d\}, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\})$, and Proposition 3.8 tells us that $U \setminus \mathbf{A}^{\mathbf{F}} = \{d, e\}$ remains unfounded for Π_7 wrt $\mathbf{A}$. That is, we may limit the attention to unfounded sets containing exclusively non-false atoms.

In what follows, we exploit loops to confine the consideration of unfounded sets, essentially reproducing results similar to those in [90, 2] in our setting. To accomplish this, we introduce atom-saturation as a property dual to body-saturation.

Definition 3.4. *Let Π be a logic program and $\mathbf{A}$ an assignment.*

Then, we define $\mathbf{A}$ as atom-saturated for Π, if

$$\{p \in atom(\Pi) \mid body_{\Pi}(p) \subseteq \mathbf{A}^{\mathbf{F}}\} \subseteq \mathbf{A}^{\mathbf{F}} .$$

This definition requires that atoms p without support must be assigned to false, as it is also stipulated by nogood $\delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\}$ (where $body_{\Pi}(p) = \{\beta_1, \dots, \beta_k\}$).

Given an atom-saturated assignment, we have that every non-empty unfounded set of non-false atoms contains some unfounded loop.

Proposition 3.9. *Let Π be a logic program, $\mathbf{A}$ an assignment, and $U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ a non-empty unfounded set of Π wrt $\mathbf{A}$.*

If $\mathbf{A}$ is atom-saturated for Π, then there is some unfounded set $L \subseteq U$ of Π wrt $\mathbf{A}$ such that $L \in loop(\Pi)$.

For illustration, note that $U = \{d, e\}$ is an unfounded loop of Π_7 wrt atom- and body-saturated assignment $\mathbf{A} = (\mathbf{F}\{not\ a\}, \mathbf{F}b, \mathbf{F}\{b, d\}, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\})$. Moreover, $\{a\}$ is the only non-empty unfounded set of Π_7 wrt $\mathbf{B} = (\mathbf{F}\{not\ b\})$, which is not atom-saturated because $a \notin \mathbf{B}^{\mathbf{F}}$. The fact that a must be false when given $\mathbf{B}$ is expressed by $\delta(a) = \{\mathbf{T}a, \mathbf{F}\{not\ b\}\}$, and $\lambda(a, \{a\}) = \delta(a)$ does not provide additional information for the "non-loop" $\{a\}$.

Given that a program may yield exponentially many loops, which can be unfounded separately wrt different assignments, it is impractical to identify (arbitrary) loops a priori. However, the non-trivial strongly connected components of a positive dependency graph limit the atoms that can jointly belong to (unfounded) loops, and the scope of unfounded set checking procedures [119, 17, 2, 31, 47] can thus be restricted to them. In our setting, the fact that the consideration of unfounded sets can be confined to non-trivial strongly connected components is an immediate consequence of Proposition 3.9.

Corollary 3.10. *Let Π be a logic program, $\mathbf{A}$ an assignment, and $U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ a non-empty unfounded set of Π wrt $\mathbf{A}$.*

If $\mathbf{A}$ is atom-saturated for Π, then there is some non-empty unfounded set $U' \subseteq U$ of Π wrt $\mathbf{A}$ such that all $p \in U'$ belong to the same non-trivial strongly connected component of $(atom(\Pi), \leq^{+})$.

Finally, we can combine Proposition 3.8 and 3.9 to establish the formal basis for the completeness of our unfounded set detection algorithm in Section 4.3.

Theorem 3.11. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment.*

If $\mathbf{A}$ *is both atom- and body-saturated for* Π *and if there is some unfounded set* U *of* Π *wrt* $\mathbf{A}$ *such that* $U \not\subseteq \mathbf{A}^{\mathbf{F}}$*, then there is some unfounded set* $L \subseteq U \setminus \mathbf{A}^{\mathbf{F}}$ *of* Π *wrt* $\mathbf{A}$ *such that* $L \in \mathit{loop}(\Pi)$.

Since fixpoints of unit propagation on Δ_Π are both atom- and body-saturated for Π, Theorem 3.11 tells us that unfounded set checking can focus on loops (of non-false atoms). For Program Π_7, where $\mathit{loop}(\Pi_7) = \{\{c,d\},\{d,e\},\{c,d,e\}\}$ (cf. Figure 1), we can thus in principle restrict unfounded set checking to its three loops in order to test all nogoods in Λ_{Π_7}.

As mentioned above, it is impractical to consider (arbitrary) loops as long as they are not unfounded, while strongly connected components can easily be determined statically [121]. The role of such components as a means to limit the scope of unfounded set checks is summarized in the following immediate consequence of Theorem 3.11.

Corollary 3.12. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment.*

If $\mathbf{A}$ *is both atom- and body-saturated for* Π *and if there is some unfounded set* U *of* Π *wrt* $\mathbf{A}$ *such that* $U \not\subseteq \mathbf{A}^{\mathbf{F}}$*, then there is some non-empty unfounded set* $U' \subseteq U \setminus \mathbf{A}^{\mathbf{F}}$ *of* Π *wrt* $\mathbf{A}$ *such that all* $p \in U'$ *belong to the same non-trivial strongly connected component of* $(\mathit{atom}(\Pi), \leq^{+})$.

With the characterization of answer sets in terms of nogoods along with relevant background knowledge on unfounded sets at hand, the next section provides our conflict-driven approach to the computation of solutions representing answer sets.

4 Conflict-Driven ASP Solving

Given the specification of answer sets in terms of nogoods, we can now make use of advanced search techniques from SAT for developing equally advanced ASP solving procedures. But while SAT deals with plain nogoods, represented by clauses, our algorithms work on logic programs, inducing several kinds of nogoods. In particular, the exponentially many nogoods stemming from unfounded sets are succinctly given by a program, and the algorithms devised below determine individual ones only when used for unfounded set falsification. The main purpose of associating nogoods with a logic program is to provide reasons for literals derived by (unit) propagation. This puts ASP solving on the same logical fundament as SAT solving, so that similar reasoning strategies can be applied without relying on translation to SAT or proprietary techniques (apart from unfounded set checking).

In what follows, we first present our main conflict-driven ASP solving procedure. We then detail its subroutines for propagation and unfounded set checking, which is the main particularity of ASP (compared to SAT). Furthermore, we describe resolution-based conflict analysis in our setting. Finally, we outline the derivation of soundness and completeness results.

Algorithm 1: CDNL-ASP

```
Input  : A logic program $\Pi$.
Output: An answer set of $\Pi$ or "no answer set".
$\mathbf{A} := \emptyset$                                   // assignment over $atom(\Pi) \cup body(\Pi)$
$\nabla := \emptyset$                                       // set of (dynamic) nogoods
$dl := 0$                                                   // decision level
loop
    $(\mathbf{A}, \nabla) := \textsc{NogoodPropagation}(dl, \Pi, \nabla, \mathbf{A})$
    if $\varepsilon \subseteq \mathbf{A}$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ then    // conflict
        if $dl = 0$ then return no answer set
        $(\delta, dl) := \textsc{ConflictAnalysis}(\varepsilon, \Pi, \nabla, \mathbf{A})$
        $\nabla := \nabla \cup \{\delta\}$                                  // learning
        $\mathbf{A} := \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid dl < dl(\sigma)\}$       // backjumping
    else if $\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F} = atom(\Pi) \cup body(\Pi)$ then       // answer set
        return $\mathbf{A}^\mathbf{T} \cap atom(\Pi)$
    else
        $\sigma_d := \textsc{Select}(\Pi, \nabla, \mathbf{A})$                        // decision
        $dl := dl + 1$
        $dl(\sigma_d) := dl$
        $\mathbf{A} := \mathbf{A} \circ \sigma_d$
```

4.1 Conflict-Driven Nogood Learning

Our main procedure for deciding whether a program has an answer set is similar to CDCL with *First-UIP* scheme [97, 127, 23, 96]. In fact, clauses can be viewed as particular syntactic representations of nogoods, but other representations (e.g., gates, inequalities, rules, etc.) can be used as well. Hence, to abstract from syntax, we present our conflict-driven algorithm for ASP solving in terms of nogoods and call it *Conflict-Driven Nogood Learning* for ASP (CDNL-ASP).

Given a program Π, CDNL-ASP, shown in Algorithm 1, starts from an empty assignment $\mathbf{A}$ and an empty set ∇ of recorded nogoods. The latter include nogoods derived from conflicts encountered during search, and if Π is non-tight, also loop nogoods explaining inferences due to unfounded sets. Moreover, the *decision level* dl, initialized with 0, is used to count *decision literals*, that is, literals in $\mathbf{A}$ that are heuristically selected (cf. Line 14–17), while literals derived via propagation in Line 5 are implied. For any literal $\sigma \in \mathbf{A}$, we access via $dl(\sigma)$ the decision level of σ, that is, the value dl had when σ was added to $\mathbf{A}$; such values are relevant for conflict analysis in Line 8 and backjumping in Line 10.

Algorithm 1 follows the standard proceeding of CDCL. First, NOGOODPROPAGATION (detailed in Section 4.2) deterministically extends $\mathbf{A}$, and possibly also records loop nogoods from Λ_Π in ∇. Afterwards, one of the following three cases applies:

Conflict If propagation led to a conflict, as checked in Line 6, there are two possibilities. If the current decision level is 0, it means that the conflict occurred independently of any heuristic decision; that is, the input program Π has no answer set. Otherwise, CONFLICTANALYSIS (detailed in Section 4.4) is performed in Line 8 to determine a conflict nogood δ, recorded in ∇ in Line 9, and a decision level to jump back to.

Note that we assume δ to be *asserting*, i.e., some literal must be unit-resulting for δ wrt $\mathbf{A}$ after backjumping in Line 10. This condition, which is guaranteed by CONFLICTANALYSIS, makes sure that, after backjumping, CDNL-ASP traverses the search space differently from before (without explicitly flipping any decision literal).

Solution If $\mathbf{A}$ is not conflicting ($\varepsilon \not\subseteq \mathbf{A}$ for all $\varepsilon \in \Delta_\Pi \cup \nabla$) and total ($\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F} = atom(\Pi) \cup body(\Pi)$), as checked in Line 11, the atoms that are true in $\mathbf{A}$ form an answer set of Π.

Decision Finally, if $\mathbf{A}$ is neither conflicting nor total, a decision literal σ_d is selected according to some heuristic (see Section 5 for further details) and added to $\mathbf{A}$. We assume that $\sigma_d = \mathbf{T}v$ or $\sigma_d = \mathbf{F}v$ for some $v \in (atom(\Pi) \cup body(\Pi)) \setminus (\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F})$, i.e., v must belong to $dom(\mathbf{A})$ and be yet unassigned. Then, σ_d becomes assigned at the new decision level $dl + 1$.

Example 4.1. *Although we have not yet detailed the subroutines used in Algorithm 1, let us consider a full-fledged computation of answer set $\{b, c, d, e\}$ of Program Π_7. To this end, Table 4 shows the current assignment $\mathbf{A}$ at different stages of* CDNL-ASP(Π_7)*, where columns provide the value of dl, viz., the current decision level, and the line of Algorithm 1 at which particular contents of $\mathbf{A}$ and/or some nogood δ are inspected. Note that literals added to $\mathbf{A}$ in Line 17 of Algorithm 1 are decision literals, not implied by any nogood. Unlike them, each literal added to $\mathbf{A}$ in Line 5, that is, within an execution of* NOGOODPROPAGATION*, has some antecedent $\delta \in \Delta_{\Pi_7} \cup \nabla$. Furthermore, we indicate successes of the test for a violated nogood performed in Line 6, and we show the nogood δ to be recorded in ∇ along with the decision level dl to jump back to as returned by* CONFLICTANALYSIS *when invoked in Line 8.*

In detail, a computation of CDNL-ASP(Π_7) *can start by successively picking decision literals $\mathbf{T}d$, $\mathbf{F}\{b, not\ a\}$, $\mathbf{T}c$, and $\mathbf{F}\{not\ a\}$ at levels 1, 2, 3, and 4, respectively. Observe that there is exactly one decision literal per level, and each decision is immediately followed by a propagation step, performed before making the next decision. At the start, propagation cannot derive any literals at decision levels 1 and 2, and thus assignment $\mathbf{A}$ stays partial. After the third decision, the literals shown below the horizontal (single) line are unit-resulting for respective nogoods $\delta \in \Delta_{\Pi_7}$ wrt $\mathbf{A}$. Hence, they are added to $\mathbf{A}$ at decision level 3. Since $\mathbf{A}$ is still partial, decision literal $\mathbf{F}\{not\ a\}$ is picked at level 4. The following propagation step yields a total assignment, whose true atoms, viz., a, c, d, and e, belong to a supported model of Π_7. However, we have that $\{d, e\}$ is unfounded for Π_7 wrt $\mathbf{A}$, that is, the corresponding loop nogoods $\lambda(d, \{d, e\})$ and $\lambda(e, \{d, e\})$ are violated. Such violations are detected by* NOGOODPROPAGATION *and, for some unfounded atom, lead to the recording of an associated loop nogood in ∇. In Table 4, we assume that $\lambda(d, \{d, e\}) = \{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\}\}$ is recorded, so that a conflict is encountered in Line 6 of Algorithm 1. Note that $\mathbf{F}\{b, c\}$ is the single literal in $\lambda(d, \{d, e\})$ assigned at*

dl	$\mathbf{A}$	δ		Line
1	$\mathbf{T}d$			17
2	$\mathbf{F}\{b, \mathit{not}\ a\}$			17
3	$\mathbf{T}c$			17
	$\mathbf{T}\{c, d\}$	$\{\mathbf{F}\{c, d\}, \mathbf{T}c, \mathbf{T}d\}$	$= \delta(\{c, d\})$	5
	$\mathbf{T}e$	$\{\mathbf{F}e, \mathbf{T}\{c, d\}\}$	$\in \Delta(e)$	5
	$\mathbf{T}\{e\}$	$\{\mathbf{F}\{e\}, \mathbf{T}e\}$	$= \delta(\{e\})$	5
4	$\mathbf{F}\{\mathit{not}\ a\}$			17
	$\mathbf{T}a$	$\{\mathbf{F}\{\mathit{not}\ a\}, \mathbf{F}a\}$	$= \delta(\{\mathit{not}\ a\})$	5
	$\mathbf{T}\{a\}$	$\{\mathbf{F}\{a\}, \mathbf{T}a\}$	$= \delta(\{a\})$	5
	$\mathbf{T}\{\mathit{not}\ b\}$	$\{\mathbf{T}a, \mathbf{F}\{\mathit{not}\ b\}\}$	$= \delta(a)$	5
	$\mathbf{F}b$	$\{\mathbf{T}b, \mathbf{F}\{\mathit{not}\ a\}\}$	$= \delta(b)$	5
	$\mathbf{F}\{b, c\}$	$\{\mathbf{T}\{b, c\}, \mathbf{F}b\}$	$\in \Delta(\{b, c\})$	5
	$\mathbf{F}\{b, d\}$	$\{\mathbf{T}\{b, d\}, \mathbf{F}b\}$	$\in \Delta(\{b, d\})$	5
		$\{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, \mathit{not}\ a\}\}$	$= \lambda(d, \{d, e\})$	6
		$\{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, \mathit{not}\ a\}\}$	$dl = 2$	8
2	$\mathbf{F}\{b, \mathit{not}\ a\}$			
	$\mathbf{T}\{b, c\}$	$\{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, \mathit{not}\ a\}\}$	$\in \nabla$	5
	$\mathbf{T}b$	$\{\mathbf{T}\{b, c\}, \mathbf{F}b\}$	$\in \Delta(\{b, c\})$	5
	$\mathbf{T}a$	$\{\mathbf{F}\{b, \mathit{not}\ a\}, \mathbf{T}b, \mathbf{F}a\}$	$= \delta(\{b, \mathit{not}\ a\})$	5
	$\mathbf{T}\{\mathit{not}\ a\}$	$\{\mathbf{T}b, \mathbf{F}\{\mathit{not}\ a\}\}$	$= \delta(b)$	5
		$\{\mathbf{T}\{\mathit{not}\ a\}, \mathbf{T}a\}$	$\in \Delta(\{\mathit{not}\ a\})$	6
		$\{\mathbf{F}\{b, \mathit{not}\ a\}, \mathbf{T}d\}$	$dl = 1$	8
1	$\underline{\mathbf{T}d}$			
	$\mathbf{T}\{b, \mathit{not}\ a\}$	$\{\mathbf{F}\{b, \mathit{not}\ a\}, \mathbf{T}d\}$	$\in \nabla$	5
	$\underline{\mathbf{T}b}$	$\{\mathbf{T}\{b, \mathit{not}\ a\}, \mathbf{F}b\}$	$\in \Delta(\{b, \mathit{not}\ a\})$	5
	$\mathbf{F}a$	$\{\mathbf{T}\{b, \mathit{not}\ a\}, \mathbf{T}a\}$	$\in \Delta(\{b, \mathit{not}\ a\})$	5
	$\mathbf{T}\{\mathit{not}\ a\}$	$\{\mathbf{T}b, \mathbf{F}\{\mathit{not}\ a\}\}$	$= \delta(b)$	5
	$\mathbf{F}\{\mathit{not}\ b\}$	$\{\mathbf{T}\{\mathit{not}\ b\}, \mathbf{T}b\}$	$\in \Delta(\{\mathit{not}\ b\})$	5
	$\mathbf{F}\{a\}$	$\{\mathbf{T}\{a\}, \mathbf{F}a\}$	$\in \Delta(\{a\})$	5
	$\underline{\mathbf{T}e}$	$\{\mathbf{F}e, \mathbf{T}\{b, \mathit{not}\ a\}\}$	$\in \Delta(e)$	5
	$\mathbf{T}\{e\}$	$\{\mathbf{F}\{e\}, \mathbf{T}e\}$	$= \delta(\{e\})$	5
	$\mathbf{T}\{b, d\}$	$\{\mathbf{F}\{b, d\}, \mathbf{T}b, \mathbf{T}d\}$	$= \delta(\{b, d\})$	5
	$\underline{\mathbf{T}c}$	$\{\mathbf{F}c, \mathbf{T}\{b, d\}\}$	$\in \Delta(c)$	5
	$\mathbf{T}\{b, c\}$	$\{\mathbf{F}\{b, c\}, \mathbf{T}b, \mathbf{T}c\}$	$= \delta(\{b, c\})$	5
	$\mathbf{T}\{c, d\}$	$\{\mathbf{F}\{c, d\}, \mathbf{T}c, \mathbf{T}d\}$	$= \delta(\{c, d\})$	5

Table 4: A computation of answer set $\{b, c, d, e\}$ with CDNL-ASP(Π_7).

decision level 4*. Hence,* $\lambda(d,\{d,e\})$ *is instantly asserting and returned by* CONFLICTANALYSIS *in Line 8; the smallest decision level such that, after backjumping,* $\mathbf{T}\{b,c\}$ *is unit-resulting for* $\lambda(d,\{d,e\})$ *is* 2*. The peculiarity that* CONFLICTANALYSIS *may be launched with an asserting (loop) nogood results from the "unidirectional" propagation of loop nogoods in current ASP solvers (cf. [63, 62]). (We further comment on this phenomenon in Section 4.4.)*

Given $dl = 2$ *as level to jump back to, computation proceeds by retracting all literals added to* $\mathbf{A}$ *at levels* 3 *and* 4*, while retaining the (decision) literals* $\mathbf{T}d$ *and* $\mathbf{F}\{b, not\ a\}$ *assigned at levels* 1 *and* 2*. In contrast to the previous visit of decision level* 2*, the asserting nogood* $\lambda(d,\{d,e\})$ *in* ∇ *enables the derivation of further literals by unit propagation, which results in another conflict, this time on the completion nogood* $\{\mathbf{T}\{not\ a\},\mathbf{T}a\}$*. Starting from it,* CONFLICTANALYSIS *determines the asserting nogood* $\{\mathbf{F}\{b, not\ a\},\mathbf{T}d\}$*. As a consequence,* CDNL-ASP(Π_7) *returns to decision level* 1*, where* $\mathbf{T}\{b, not\ a\}$ *is unit-resulting for* $\{\mathbf{F}\{b, not\ a\},\mathbf{T}d\}$*. A final propagation step leads to a total assignment not violating any nogood in* $\Delta_{\Pi_7} \cup \nabla$*. (Notably, nogoods in* Λ_{Π_7} *are left implicit and tested within* NOGOODPROPAGATION *via an unfounded set checking subroutine.) The true atoms of the obtained solution are underlined in Table 4; the associated answer set of Program* Π_7*,* $\{b,c,d,e\}$*, is returned as the result of* CDNL-ASP(Π_7).

After the general outline, we below detail the subroutines used in CDNL-ASP computations.

4.2 Nogood Propagation

Our subroutine for deterministically extending an assignment $\mathbf{A}$ is shown in Algorithm 2. It combines unit propagation on completion nogoods in Δ_Π and recorded nogoods in ∇ (Line 3–9) with unfounded set checking (Line 10–14). While unit propagation is always run to a fixpoint (or a conflict), sophisticated unfounded set checks are performed only if the input program Π is non-tight. In fact, when finishing the loop in Line 3–9, an assignment $\mathbf{A}$ at hand is both atom- and body-saturated for Π, so that the results in Section 3.2 serve as a basis for demand-driven unfounded set checking. In particular, if Π is tight, Theorem 3.11 tells us that all unfounded sets U are already falsified, i.e., $U \subseteq \mathbf{A}^{\mathbf{F}}$ holds, and thus unit propagation on Δ_Π is sufficient to falsify unfounded atoms.

Example 4.2. *The central idea of integrating unfounded set checking with unit propagation is to make loop nogoods from* Λ_Π *explicit in* ∇ *in order to trigger the falsification of unfounded atoms by unit propagation. To see this, consider a program* Π *containing the following rules:*

$$\begin{aligned} x &\leftarrow y, z \\ y &\leftarrow x \\ z &\leftarrow y\ . \end{aligned}$$

Let $\mathbf{A}$ *be an atom-saturated assignment such that* $U = \{x,y,z\}$ *is unfounded for* Π *wrt* $\mathbf{A}$ *and* $U \cap (\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}}) = \emptyset$*. Then, we have that* $EB_\Pi(U) \subseteq \mathbf{A}^{\mathbf{F}}$*, so that* $\mathbf{F}x$*,* $\mathbf{F}y$*, and* $\mathbf{F}z$ *are unit-resulting for* $\lambda(x,U)$*,* $\lambda(y,U)$*, and* $\lambda(z,U)$*, respectively. While neither of these literals may be unit-resulting for any completion nogood in* Δ_Π*, all of them (along with* $\mathbf{F}\{x\}$*,* $\mathbf{F}\{y\}$*, and* $\mathbf{F}\{y,z\}$*) are derived by unit propagation when given* $\Delta_\Pi \cup \{\lambda(x,U)\}$*. That is, when adding*

Algorithm 2: NogoodPropagation

Input : A decision level dl, a logic program Π, a set ∇ of nogoods, and an assignment $\mathbf{A}$.
Output: An extended assignment and set of nogoods.

```
1  $U := \emptyset$                                                        // unfounded set
2  loop
3      repeat
4          if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_\Pi \cup \nabla$ then return $(\mathbf{A}, \nabla)$           // conflict
5          $\Sigma := \{\delta \in \Delta_\Pi \cup \nabla \mid \delta \setminus \mathbf{A} = \{\overline{\sigma}\}, \sigma \notin \mathbf{A}\}$     // unit-resulting nogoods
6          if $\Sigma \neq \emptyset$ then let $\overline{\sigma} \in \delta \setminus \mathbf{A}$ for some $\delta \in \Sigma$ in
7              $dl(\sigma) := dl$
8              $\mathbf{A} := \mathbf{A} \circ \sigma$
9      until $\Sigma = \emptyset$
10     if $loop(\Pi) = \emptyset$ then return $(\mathbf{A}, \nabla)$    // no unfounded set $\emptyset \subset U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$
11     $U := U \setminus \mathbf{A}^{\mathbf{F}}$
12     if $U = \emptyset$ then $U :=$ UnfoundedSet$(\Pi, \mathbf{A})$
13     if $U = \emptyset$ then return $(\mathbf{A}, \nabla)$           // no unfounded set $\emptyset \subset U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$
14     let $p \in U$ in $\nabla := \nabla \cup \{\lambda(p, U)\}$                // record loop nogood
```

only $\lambda(x, U)$ to ∇, the whole unfounded set U is falsified by unit propagation. However, whether the addition of a single loop nogood is sufficient to falsify a whole unfounded set depends on the structure of Π. For instance, when we augment Π with $(y \leftarrow z)$, the derivation of $\mathbf{F}y$ and $\mathbf{F}z$ by unit propagation is no longer certain because (circular) supports $(y \leftarrow z)$ and $(z \leftarrow y)$ may not be eliminated by assigning x to false. We still derive $\mathbf{F}\{x\}$, i.e., rule $(y \leftarrow x)$ becomes inapplicable, so that $EB_\Pi(\{y, z\}) \subseteq (\mathbf{A} \cup \{\mathbf{F}x, \mathbf{F}\{x\}\})^{\mathbf{F}}$. This shows that $U \setminus \mathbf{A}^{\mathbf{F}} = \{x, y, z\} \setminus \{x\} = \{y, z\}$ remains as a smaller unfounded set.

The observations made in Example 4.2 motivate the strategy of Algorithm 2 to successively falsify the elements of an unfounded set U. At the start, no (non-empty) unfounded set has been determined, and so U is initialized to be empty in Line 1. Provided that unit propagation in Line 3–9 finishes without conflict and that Π is non-tight, we remove all false atoms from U in Line 11. In the first iteration of the outer loop in Line 2–14, U stays empty, and the subroutine for unfounded set detection (detailed in Section 4.3) is queried in Line 12. The crucial assumption made here is that UnfoundedSet$(\Pi, \mathbf{A})$ returns an unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ such that U is non-empty if some non-empty subset of $atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ is unfounded. Then, we have that $EB_\Pi(U) \subseteq \mathbf{A}^{\mathbf{F}}$, so that $\lambda(p, U) \setminus \mathbf{A} \subseteq \{\mathbf{T}p\}$ for every $p \in U$. Hence, if a non-empty U is returned, the addition of $\lambda(p, U)$ to ∇ for an arbitrary $p \in U$, done in Line 14, yields either a conflict or unit-resulting literal $\mathbf{F}p$ in the next iteration of the loop in Line 2–14. In the latter case, further literals may be derived and elements of U falsified upon computing the next fixpoint of unit propagation. When we afterwards reconsider the previously determined unfounded set U, the removal of false atoms in Line 11 is guaranteed to result in another (smaller) unfounded set $U \setminus \mathbf{A}^{\mathbf{F}}$. Hence, if $U \setminus \mathbf{A}^{\mathbf{F}}$ is non-empty (checked in Line 12 before computing any further unfounded set), NOGOODPROPAGATION proceeds by adding the next loop nogood to ∇, which as before yields either a conflict or a unit-resulting literal. Thus, once a non-empty unfounded set U has been detected, it is falsified element by element; only after expending all elements of U, a new unfounded set is to be computed. Overall, NOGOODPROPAGATION terminates as soon as a conflict is encountered (in Line 4) or with a fixpoint of unit propagation on $\Delta_\Pi \cup \nabla$ such that no non-empty subset of $atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ is unfounded. If Π is tight, the latter is immediately verified in Line 10. Otherwise, the UNFOUNDEDSET subroutine, queried in Line 12, could not detect any non-empty unfounded set (of non-false atoms) before finishing in Line 13.

Example 4.3. *To illustrate how* NOGOODPROPAGATION *utilizes nogoods, reconsider the computation of* CDNL-ASP(Π_7) *shown in Table 4. All implied literals, that is, the ones assigned below any (single) line at a decision level dl, are unit-resulting for nogoods in $\Delta_{\Pi_7} \cup \nabla$ and successively derived by unit propagation. In particular, at decision level 4, the implied literals σ have antecedents $\delta \in \Delta_{\Pi_7}$ such that all literals of δ except for $\overline{\sigma}$ are already contained in $\mathbf{A}$ when σ is assigned. The impact of loop nogoods in Λ_{Π_7} can be observed on the conflict encountered at decision level 4. Here, we have that $U = \{d, e\} \subseteq \mathbf{A}^{\mathbf{T}}$ is unfounded, so that $\mathbf{A}$ violates both $\lambda(d, U)$ and $\lambda(e, U)$. After detecting the unfounded set U and recording $\lambda(d, U)$ in ∇, its violation gives rise to leaving* NOGOODPROPAGATION *in Line 4 of Algorithm 2.*

In summary, our subroutine for propagation interleaves unit propagation with the recording of loop nogoods. The latter is done only if the input program is non-tight and if the falsity of unfounded atoms cannot be derived by unit propagation on other available nogoods. Clearly, our approach favors unit propagation over unfounded set computations, which can be motivated as follows. For one, unit propagation does not contribute new dynamic nogoods to ∇, so that it is more "economic" than unfounded set checking. For another, although unfounded set detection algorithms (like the one described below) are of linear time complexity, they analyze a logic program in a more global fashion than unit propagation. While the latter investigates only the rules (or nogoods) directly related to literals becoming assigned, unfounded set computations may inspect significant parts of a program (or its positive dependency graph) without eventually detecting any non-empty unfounded set. But given that unfounded set checking (wrt total assignments) is mandatory for soundness and (wrt partial assignments) also helps to detect inherent conflicts early, the respective subroutine is nonetheless an integral part of NOGOODPROPAGATION.

4.3 Unfounded Set Checking

Our unfounded set checking procedure is invoked on a non-tight program Π whenever unit propagation reaches a fixpoint without any conflict or formerly computed but yet unfalsified unfounded atoms (cf. Algorithm 2). As a matter of fact, a fixpoint of unit propagation is both atom- and body-saturated for Π. Hence, Corollary 3.12 applies and allows us to focus on unfounded sets of non-false atoms contained in non-trivial strongly connected components of the positive dependency graph of Π. To this end, for any $p \in atom(\Pi)$, let $scc(p)$ denote the set of atoms belonging to the same strongly connected component as p in $(atom(\Pi), \leq^+)$. We say that p is *cyclic*, if $\leq^+ \cap (scc(p) \times scc(p)) \neq \emptyset$ (that is, if there is some rule $r \in \Pi$ such that

$\mathit{head}(r) \in \mathit{scc}(p)$ and $\mathit{body}(r)^+ \cap \mathit{scc}(p) \neq \emptyset$), and *acyclic* otherwise. As a consequence of Proposition 3.9, we immediately get that unfounded set checking can concentrate exclusively on cyclic atoms, since only they can belong to (unfounded) loops.[7]

Beyond static information about strongly connected components, our unfounded set detection algorithm makes use of *source pointers* [119] to indicate non-circular supports of atoms. Given a program Π, the idea is to associate every (cyclic) $p \in \mathit{atom}(\Pi)$ with an element of $\mathit{body}_\Pi(p)$ (or one of the special-purpose symbols $\bot$ and $\top$), denoted by $\mathit{source}(p)$, pointing to a chain of rules witnessing that p cannot be unfounded. Hence, as long as $\mathit{source}(p)$ remains "intact", p can be excluded from unfounded set checks. In this way, source pointers enable lazy, incremental unfounded set checking relative to recent changes of an assignment. To make sure that still no unfounded set is missed, the following invariants need to be guaranteed:

1. For every cyclic $p \in \mathit{atom}(\Pi)$, we require that $\mathit{source}(p) \in \mathit{body}_\Pi(p) \cup \{\bot\}$.
2. The subgraph of $(\mathit{atom}(\Pi), \leq^+)$ containing every cyclic $p \in \mathit{atom}(\Pi)$ along with edges (q, p) for all $q \in \mathit{source}(p)^+ \cap \mathit{scc}(p)$ must be acyclic.[8]

For a program Π, we call the collection of links $\mathit{source}(p)$ for all $p \in \mathit{atom}(\Pi)$ a *source pointer configuration*. We say that a source pointer configuration is *valid*, if it satisfies the aforementioned invariants. For an appropriate initialization, we define the *initial source pointer configuration* for Π by:

$$\mathit{source}(p) = \begin{cases} \bot & \text{if } p \in \mathit{atom}(\Pi) \text{ is cyclic} \\ \top & \text{if } p \in \mathit{atom}(\Pi) \text{ is acyclic} \end{cases}$$

While $\top$ expresses that an acyclic atom p does not need to be linked to any element of $\mathit{body}_\Pi(p)$, $\bot$ indicates that a non-circular support for a cyclic atom p still needs to be determined. We assume that the initial source pointer configuration for Π, which is valid by definition, is in place upon an invocation of CDNL-ASP(Π).

Given a program Π and an assignment $\mathbf{A}$, UNFOUNDEDSET, shown in Algorithm 3, starts by collecting non-false (cyclic) atoms p whose source pointers are false ($\mathit{source}(p) \in \mathbf{A}^\mathbf{F}$) or yet undetermined ($\mathit{source}(p) = \bot$) in Line 1, as the possibility of non-circularly supporting such atoms is in question. In Line 2–5, this set is successively extended by adding atoms whose source pointers (positively) rely on it, thus providing the *scope* S for the second part of unfounded set checking. In fact, the loop in Line 6–17 aims at re-establishing source pointers for the atoms in S via rules whose bodies do not (positively) rely on S, so that these rules can provide non-circular support. Conversely, if source pointers cannot be re-established, an unfounded set is detected.

In more detail, as long as scope S is non-empty, an arbitrary atom $p \in S$ is picked in Line 6 of Algorithm 3 as starting point for the construction of a non-empty unfounded set U. If $\mathit{EB}_\Pi(U) \subseteq \mathbf{A}^\mathbf{F}$ holds in Line 9, the unfounded set U is immediately returned, so that NOGOODPROPAGATION can successively falsify its atoms by unit propagation (cf. Algorithm 2).

[7]Strongly connected components of positive dependency graphs are also exploited by unfounded set checking procedures [119, 17, 2] of native ASP solvers other than *clasp*. We further discuss relationships to them in Section 7.

[8]Recall that $\mathit{source}(p)^+ = \{p_1, \dots, p_m\}$ for a rule body $\mathit{source}(p) = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}$. For special-purpose symbols $\bot$ and $\top$, we let $\bot^+ = \top^+ = \emptyset$.

Algorithm 3: UNFOUNDEDSET

```
Input  : A logic program $\Pi$ and an assignment $\mathbf{A}$.
Output: An unfounded set of $\Pi$ wrt $\mathbf{A}$.
1  $S := \{p \in atom(\Pi) \setminus \mathbf{A^F} \mid source(p) \in \mathbf{A^F} \cup \{\bot\}\}$          // initialize scope $S$
2  repeat
3      $T := \{p \in atom(\Pi) \setminus (\mathbf{A^F} \cup S) \mid source(p)^+ \cap (scc(p) \cap S) \neq \emptyset\}$
4      $S := S \cup T$                                                    // extend scope $S$
5  until $T = \emptyset$
6  while $S \neq \emptyset$ do let $p \in S$ in                              // select starting point
7      $U := \{p\}$
8      repeat
9          if $EB_\Pi(U) \subseteq \mathbf{A^F}$ then return $U$      // unfounded set $\emptyset \subset U \subseteq atom(\Pi) \setminus \mathbf{A^F}$
10         let $\beta \in EB_\Pi(U) \setminus \mathbf{A^F}$ in
11             if $\beta^+ \cap (scc(p) \cap S) = \emptyset$ then                      // shrink U
12                 foreach $q \in U$ such that $\beta \in body_\Pi(q)$ do
13                     $source(q) := \beta$
14                     $U := U \setminus \{q\}$
15                     $S := S \setminus \{q\}$
16             else $U := U \cup (\beta^+ \cap (scc(p) \cap S))$                    // extend U
17     until $U = \emptyset$
18 return $\emptyset$                          // no unfounded set $\emptyset \subset U \subseteq atom(\Pi) \setminus \mathbf{A^F}$
```

Otherwise, some external body $\beta \in EB_\Pi(U) \setminus \mathbf{A^F}$ is selected in Line 10 for further investigation. If β^+ contains atoms in scope S that belong to the same strongly connected component of $(atom(\Pi), \leq^+)$ as the starting point p (checked in Line 11), we add them to U in Line 16, which makes β non-external wrt the extended set U. On the other hand, if such atoms do not exist in β^+, it means that β can non-circularly support all of its associated head atoms $q \in U$. Then, in Line 12–15, the source pointers of such atoms q are set to β, and the atoms q are removed from both the unfounded set U under construction and scope S. The described process continues until either U becomes empty (checked in Line 17), in which case the remaining atoms of S are investigated, or a (non-empty) unfounded set U is detected and returned in Line 9. Finally, if scope S runs empty, source pointers could be re-established for all atoms that had been contained in S, and UNFOUNDEDSET returns the empty unfounded set in Line 18.

In order to provide further intuitions, let us stress some major design principles of our unfounded set detection algorithm:

1. At each stage of the loop in Line 6–17, all atoms of U belong to $scc(p)$, where p is an atom added first to U (in Line 7). This is because further atoms, added to U in Line 16, are elements of $scc(p)$. (However, $U \subseteq scc(p)$ does not necessarily imply $p \in U$ for a (non-empty) unfounded set U returned in Line 9.)

2. At each stage of the loop in Line 6–17, we have that $U \subseteq S$, as all atoms added to U in either Line 7 or 16 belong to S. Hence, it holds that $q \in S$ whenever $source(q)$ is set to an (external) body $\beta \in body_\Pi(q)$ in Line 13, while $\beta^+ \cap (scc(p) \cap S) = \emptyset$ has been checked before (in Line 11). This makes sure that setting $source(q)$ to β does not introduce any cycle via source pointers.

3. Once detected, a (non-empty) unfounded set U is immediately returned in Line 9, and NOGOODPROPAGATION takes care of falsifying all atoms of U before checking for any further unfounded set (cf. Algorithm 2). This reduces overlaps with unit propagation on the completion nogoods in Δ_Π, as it already handles unsupported atoms, i.e., singleton unfounded sets (and bodies relying on them).

4. The source pointer of an atom q in some unfounded set U returned in Line 9 needs not and is not reset to $\bot$. (In fact, $source(q)$ is only set in Line 13 when re-establishing a potential non-circular support for q.) Rather, we admit $source(q) \in \mathbf{A^F}$ as long as $q \in \mathbf{A^F}$, derived within NOGOODPROPAGATION upon falsifying U. Thus, when $\mathbf{F}q$ becomes unassigned later on (after backjumping), $source(q)$ still allows for lazy unfounded set checking.

Example 4.4. *Let us illustrate Algorithm 3 on some invocations of* UNFOUNDEDSET$(\Pi_7, \mathbf{A})$ *made upon the computation of answer set* $\{b, c, d, e\}$ *of Program* Π_7 *described in Example 4.1. To this end, in Table 5, we indicate stages of* UNFOUNDEDSET$(\Pi_7, \mathbf{A})$ *when queried wrt fixpoints* $\mathbf{A}$ *of unit propagation at decision levels* 0, 2, *and* 4, *respectively. Beforehand, note that* $scc(c) = scc(d) = scc(e) = \{c, d, e\}$, *while* a *and* b *are acyclic. Hence, before the first invocation of* UNFOUNDEDSET$(\Pi_7, \mathbf{A})$ *at decision level* 0, *we have that* $source(a) = source(b) = \top$ *and* $source(c) = source(d) = source(e) = \bot$. *In view of Line 1 of Algorithm 3, we thus obtain scope* $S = \{c, d, e\}$. *Then, assume that* e *is picked in Line 6 and added to* U *in Line 7, and that* $\{c, d\} \in EB_{\Pi_7}(\{e\})$ *is selected in Line 10. Since* $\{c, d\} \cap (scc(e) \cap S) = \{c, d\}$, *this makes us augment* U *with both* c *and* d *in Line 16, resulting in an intermediate stage such that* $U = \{c, d, e\}$. *Further assume that* $\{b, not\ a\} \in EB_{\Pi_7}(\{c, d, e\})$ *is selected next in Line 10, for which* $\{b\} \cap (scc(e) \cap S) = \emptyset$ *holds in Line 11. Hence,* $source(e)$ *is set to* $\{b, not\ a\}$ *in Line 13, and* e *is removed from* U *and* S *in Line 14 and 15, respectively. In the same manner,* $source(d)$ *and* $source(c)$ *can in the following iterations of the loop in Line 8–17 be set to* $\{e\}$ *and* $\{b, d\}$, *respectively. Afterwards, we have that* $U = S = \emptyset$, *so that the empty unfounded set is returned (in Line 18). Given that there is no non-empty unfounded set, no literal is derived by unit propagation at decision level* 0, *as also indicated by omitting this level in Table 4.*

The invocation of UNFOUNDEDSET$(\Pi_7, (\mathbf{T}d))$ *at decision level* 1 *is not shown in Table 5, as it yields an empty scope* S. *Unlike this, with* UNFOUNDEDSET$(\Pi_7, (\mathbf{T}d, \mathbf{F}\{b, not\ a\}))$ *at decision level* 2, *we have that* $source(e) = \{b, not\ a\} \in \mathbf{A^F}$, *so that* $S = \{e\}$ *is obtained in Line 1 of Algorithm 3. In Line 2–5, we successively add* d *and* c *to* S *because* $source(d)^+ \cap S = \{e\} \cap \{e\} \neq \emptyset$ *and* $source(c)^+ \cap (S \cup \{d\}) = \{b, d\} \cap \{d, e\} \neq \emptyset$. *Afterwards, assume that* d *is added first to* U *in Line 7, and that selecting* $\{b, c\} \in EB_{\Pi_7}(\{d\})$ *in Line 10 leads to* $U = \{d\} \cup (\{b, c\} \cap (scc(d) \cap S)) = \{c, d\}$. *When investigating* $\{a\} \in EB_{\Pi_7}(\{c, d\})$ *and again* $\{b, c\} \in EB_{\Pi_7}(\{d\})$ *in the next two iterations of the loop in Line 8–17, we set* $source(c)$ *to* $\{a\}$ *and* $source(d)$ *to* $\{b, c\}$, *while obtaining* $U = \emptyset$ *and* $S = \{e\}$. *Since* $S \neq \emptyset$, *another*

dl	$source(p)$	S	U	$\beta \in EB_{\Pi_7}(U) \setminus \mathbf{A}^{\mathbf{F}}$	Line
0		$\{c,d,e\}$			1
		$\{c,d,e\}$	$\{e\}$		7
		$\{c,d,e\}$	$\{c,d,e\}$	$\{c,d\}$	16
	$source(e)$	$\{c,d\}$	$\{c,d\}$	$\{b, not\ a\}$	13
	$source(d)$	$\{c\}$	$\{c\}$	$\{e\}$	13
	$source(c)$	$\emptyset$	$\boxed{\emptyset}$	$\{b,d\}$	13
2	$\mathbf{F}\{b, not\ a\}$	$\{e\}$			1
	$\{e\}$	$\{d,e\}$			4
	$\{b,d\}$	$\{c,d,e\}$			4
		$\{c,d,e\}$	$\{d\}$		7
		$\{c,d,e\}$	$\{c,d\}$	$\{b,c\}$	16
	$source(c)$	$\{d,e\}$	$\{d\}$	$\{a\}$	13
	$source(d)$	$\{e\}$	$\emptyset$	$\{b,c\}$	13
		$\{e\}$	$\{e\}$		7
	$source(e)$	$\emptyset$	$\boxed{\emptyset}$	$\{c,d\}$	13
4	$\mathbf{F}\{b,c\}$	$\{d\}$			1
	$\{c,d\}$	$\{d,e\}$			4
		$\{d,e\}$	$\{e\}$		7
		$\{d,e\}$	$\boxed{\{d,e\}}$	$\{c,d\}$	16

Table 5: Runs of UNFOUNDEDSET($\Pi_7, \mathbf{A}$) upon a computation of answer set $\{b,c,d,e\}$.

iteration of the loop in Line 6–17 adds e to U and then removes it from U and S along with setting $source(e)$ to $\{c,d\}$. Given $U = S = \emptyset$, we again get the empty unfounded set as result.

At decision level 3, unfounded set checking is without effect because, as shown in Table 4, no rule body and, in particular, no source pointer is falsified. However, at decision level 4, we have that $source(d) = \{b,c\} \in \mathbf{A}^{\mathbf{F}}$, and thus we get $S = \{d\}$ in Line 1 of Algorithm 3. In an iteration of the loop in Line 2–5, we further add e to S because $source(e)^+ \cap S = \{c,d\} \cap \{d\} \neq \emptyset$, while c stays unaffected in view of $source(c) = \{a\} \notin \mathbf{A}^{\mathbf{F}}$. After adding e to U in Line 7, U is further extended to $\{d,e\}$ in Line 16, given that $\{c,d\} \in EB_{\Pi_7}(\{e\})$ and $\{c,d\} \cap (scc(e) \cap S) = \{d\}$. We have now obtained $U = \{d,e\}$, and it holds that $EB_{\Pi_7}(\{d,e\}) = \{\{b,c\},\{b, not\ a\}\} \subseteq \mathbf{A}^{\mathbf{F}}$. That is, the termination condition in Line 9 applies, and UNFOUNDEDSET($\Pi_7, \mathbf{A}$) *returns the (non-empty) unfounded set $\{d,e\}$.*

To conclude the example, in Table 4, we observe that adding loop nogood $\lambda(d,\{d,e\}) = \{\mathbf{T}d, \mathbf{F}\{b,c\}, \mathbf{F}\{b, not\ a\}\}$ to ∇ leads to a conflict at decision level 4. After backjumping to decision level 2, NOGOODPROPAGATION *encounters a conflict before invoking* UNFOUNDED-SET($\Pi_7, \mathbf{A}$)*. Hence,* UNFOUNDEDSET($\Pi_7, \mathbf{A}$) *is only queried again wrt the total assignment $\mathbf{A}$ derived by unit propagation after returning to decision level 1. In view of $source(c) = \{a\} \in \mathbf{A}^{\mathbf{F}}$, this final invocation (not shown in Table 5) makes us reset source pointers as follows: $source(e) = \{b, not\ a\}$, $source(d) = \{e\}$, and $source(c) = \{b,d\}$ (like at decision level 0).*

Algorithm 4: CONFLICTANALYSIS

```
Input  : A violated nogood $\delta$, a logic program $\Pi$, a set $\nabla$ of nogoods, and an assignment $\mathbf{A}$.
Output : A derived nogood and a decision level.
1 loop
2     let $\sigma \in \delta$ such that $\delta \setminus \mathbf{A}[\sigma] = \{\sigma\}$ in
3         $k := max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
4         if $k = dl(\sigma)$ then
5             let $\varepsilon \in \Delta_\Pi \cup \nabla$ such that $\varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}$ in
6                 $\delta := (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$        // resolution
7         else return $(\delta, k)$
```

As this yields only the empty unfounded set (of non-false atoms), NOGOODPROPAGATION *terminates without conflict, and* CDNL-ASP(Π_7) *returns answer set* $\{b, c, d, e\}$ *of* Π_7.

Note that a non-empty unfounded set U returned in Line 9 of Algorithm 3 is, in general, not guaranteed to be a loop in the sense of [90]. However, Theorem 3.11 tells us that U contains some loop L that is unfounded. One or several such loops L could a posteriori be extracted from U, for which purpose any of the approaches in [90, 71, 91, 2, 59] can in principle be applied.

4.4 Conflict Analysis

Finally, we turn to the subroutine for conflict analysis, whose purpose is to determine an asserting nogood, so that some literal is unit-resulting after backjumping. To this end, it resolves a violated nogood $\delta \subseteq \mathbf{A}$ against some antecedent ε of an implied literal $\sigma \in \delta$ (that is, a nogood ε such that $\varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}$) for obtaining a new violated nogood $(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$. Iterated resolution proceeds in inverse order of assigned literals, resolving first over the literal $\sigma \in \delta$ assigned last in $\mathbf{A}$, viz., $\delta \setminus \mathbf{A}[\sigma] = \{\sigma\}$, and stops as soon as δ contains exactly one literal, called *Unique Implication Point* (UIP; [97]), that has been assigned at the decision level where the conflict is encountered. The effectiveness of this approach, referred to as *First-UIP* scheme (cf. [127, 35, 96]), has in the area of SAT been demonstrated both empirically [127, 115, 29] and analytically [109]. Despite small peculiarities (discussed below Example 4.5), the First-UIP scheme can be applied unaltered in conflict-driven ASP solving. However, identifying antecedents of implied literals is less straightforward than with clauses. For instance, note that our subroutine for propagation in Algorithm 2 records a priori implicit loop nogoods from Λ_Π to make sure that every implied literal has some antecedent in $\Delta_\Pi \cup \nabla$.

Conflict resolution according to the First-UIP scheme is performed by CONFLICTANALYSIS, shown in Algorithm 4. In fact, the loop in Line 1–7 proceeds by resolving over the literal σ of the violated nogood δ assigned last in $\mathbf{A}$ (given that $\delta \setminus \mathbf{A}[\sigma] = \{\sigma\}$ is required in Line 2) until the *assertion level* [23], that is, the greatest level $dl(\rho)$ associated with literals $\rho \in \delta \setminus \{\sigma\}$, is different from and actually smaller than $dl(\sigma)$. If so, nogood δ and assertion level k (determined

δ	ε
$\left\{\boxed{\mathbf{T}\{not\ a\}}, \underline{\mathbf{T}a}\right\}$	$\left\{\underline{\mathbf{T}b}, \boxed{\mathbf{F}\{not\ a\}}\right\}$
$\left\{\boxed{\mathbf{T}a}, \underline{\mathbf{T}b}\right\}$	$\left\{\underline{\mathbf{F}\{b, not\ a\}}, \underline{\mathbf{T}b}, \boxed{\mathbf{F}a}\right\}$
$\left\{\boxed{\mathbf{T}b}, \underline{\mathbf{F}\{b, not\ a\}}\right\}$	$\left\{\underline{\mathbf{T}\{b, c\}}, \boxed{\mathbf{F}b}\right\}$
$\left\{\boxed{\mathbf{T}\{b, c\}}, \underline{\mathbf{F}\{b, not\ a\}}\right\}$	$\left\{\mathbf{T}d, \boxed{\mathbf{F}\{b, c\}}, \underline{\mathbf{F}\{b, not\ a\}}\right\}$
$\left\{\boxed{\mathbf{F}\{b, not\ a\}}, \mathbf{T}d\right\}$	

Table 6: Run of CONFLICTANALYSIS($\{\mathbf{T}\{not\ a\}, \mathbf{T}a\}, \Pi_7, \nabla, \mathbf{A}$) at decision level 2.

in Line 3) are returned in Line 7; since $\delta \subseteq \mathbf{A}$, we have that $\overline{\sigma}$ is unit-resulting for δ after backjumping to decision level k. Otherwise, if $k = dl(\sigma)$, σ is an implied literal, so that some antecedent $\varepsilon \in \Delta_\Pi \cup \nabla$ of σ can be chosen in Line 5 and used for resolution against δ in Line 6. Note that there may be several antecedents of σ in $\Delta_\Pi \cup \nabla$, and thus the choice of ε in Line 5 is, in general, non-deterministic (cf. [32]). Regarding the termination of Algorithm 4, note that a decision literal σ_d (cf. Algorithm 1) is the first literal in $\mathbf{A}$ at its (positive) level $dl(\sigma_d)$, and σ_d is also the only literal at $dl(\sigma_d)$ that is not implied. Given that CONFLICTANALYSIS is only applied to nogoods violated at decision levels beyond 0, all conflict resolution steps are well-defined and stop at latest at a decision literal σ_d. However, resolving up to σ_d can be regarded as worst case because the First-UIP scheme aims at few resolution steps to obtain a nogood that is "close" to a conflict at hand.

Example 4.5. *To illustrate Algorithm 4, let us inspect the resolution steps shown in Table 6. They are applied when resolving the violated nogood* $\{\mathbf{T}\{not\ a\}, \mathbf{T}a\}$ *against the antecedents shown in Table 4 upon analyzing the conflict encountered at decision level* 2 *in the computation of* CDNL-ASP(Π_7) *described in Example 4.1. The literal* σ *of a violated nogood* δ *assigned last in* $\mathbf{A}$ *as well as its complement* $\overline{\sigma}$ *in an antecedent* ε *of* σ *are surrounded by a box in Table 6, and further literals assigned at decision level* 2 *are underlined. The result of iterated resolution,* $\{\mathbf{F}\{b, not\ a\}, \mathbf{T}d\}$*, contains* $\mathbf{F}\{b, not\ a\}$ *as the single literal assigned at decision level* 2*, while* $\mathbf{T}d$ *has been assigned at assertion level* 1*. In this example, the first UIP* $\mathbf{F}\{b, not\ a\}$ *happens to be the decision literal at level* 2*.*

In general, a first UIP is not necessarily a decision literal, as it can for instance be observed on UIP $\mathbf{F}\{b, c\}$ in the asserting nogood $\{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\}\}$ returned by CONFLICTANALYSIS at decision level 4 in Example 4.1. Also recall that $\lambda(d, \{d, e\}) = \{\mathbf{T}d, \mathbf{F}\{b, c\}, \mathbf{F}\{b, not\ a\}\}$ served as starting point for CONFLICTANALYSIS, containing a (first) UIP without requiring any resolution step. This phenomenon is due to "unidirectional" propagation of loop nogoods, given that unfounded set checks (cf. Algorithm 3) merely identify unfounded atoms, but not rule bodies that must necessarily hold for (non-circularly) supporting some true atom. In Example 4.1, the fact that $\mathbf{T}\{b, c\}$ is required from decision level 2 on is only recognized at level 4, where assigning $\mathbf{F}\{b, c\}$ leads to a conflict. In view of this, Algorithm 3 can be understood as a checking routine guaranteeing the soundness of CDNL-ASP, while its inference capabilities do not match (full) unit propagation on loop nogoods. Similar observations have already been made in [63, 62], but more powerful yet efficient reasoning mechanisms for unfounded set handling seem to be difficult to develop; for instance, the approach suggested in [18, 19] is computationally too complex (quadratic) to be beneficial in practice.

Despite of the fact that conflict resolution in ASP can be done in the same fashion as in SAT, the input format of logic programs makes it less predetermined. For one, the completion nogoods in Δ_Π contain rule bodies as structural variables for the sake of succinct representation. For another, the number of (relevant) inherent loop nogoods in Λ_Π may be exponential [88]. Fortunately, the satisfaction of Λ_Π can be checked in linear time (e.g., via Algorithm 3), so that an explicit representation of its elements is not required. However, NOGOODPROPAGATION (cf. Algorithm 2) records loop nogoods from Λ_Π that are antecedents to make them easily accessible in CONFLICTANALYSIS.

Alternatives in the representation of constraints induced by a logic program become apparent when considering traditional ASP solvers, such as *dlv* [83] and *smodels* [119], where assignments are (logically) identified with interpretations over atoms. In order to augment *smodels* with conflict-driven learning, *smodels*$_{cc}$ [126] pursues an algorithmic approach to extract antecedents (over atoms) relative to *smodels*' inference rules. In our setting, one may restrict heuristic decisions in Line 14 of Algorithm 1 to atoms for mimicking an "atom-only" approach where truth values of rule bodies are determined by their literals. However, when CONFLICTANALYSIS remains unaltered, its asserting nogoods may still enable unit propagation to derive the falsity of bodies without (known) false body literals (or associated false head atoms), which cannot occur with atom-based approaches. To ultimately avoid such inferences, one would need to unconditionally eliminate literals over bodies from conflict nogoods by resolution against their antecedents, which is possible when heuristic decisions are restricted to atoms. This idea comes close to the learning technique of *smodels*$_{cc}$, breaking derivations relying on bodies down to their contained literals. Although such "body elimination" may enable learning on top of atom-based approaches, it still goes along with exponentially increased (best-case) complexity, independent of and thus irreparable by conflict-driven nogood learning [62].

4.5 Soundness and Completeness of CDNL-ASP Algorithm

In what follows, we elaborate upon the formal properties of the provided algorithms. Generally speaking, soundness wrt the decision problem of answer set existence is obtained from the fact that NOGOODPROPAGATION and CONFLICTANALYSIS exploit and possibly tighten available knowledge, but do not draw incorrect conclusions. In the course of this, UNFOUNDEDSET performs a sufficient amount of work to distinguish answer sets from (inadmissible) circularly supported models. The completeness of CDNL-ASP follows from the observation that its subroutines cannot loop infinitely along with the fact that conflict-driven assertions relocate variables to smaller decision levels than before, which guarantees termination (cf. [128, 115]).

To begin with, we consider crucial properties of UNFOUNDEDSET in Algorithm 3. First, we have that (positive) dependencies through source pointers are inherently acyclic.

Lemma 4.1. *Let Π be a logic program and $\mathbf{A}$ an assignment that is body-saturated for Π.*

If UNFOUNDEDSET$(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that the source pointer configuration remains valid throughout the execution of* UNFOUNDEDSET$(\Pi, \mathbf{A})$.

The above property holds because potential non-circular supports for atoms in β^+ must already be established before a source pointer can be set to a body β in Line 13 of Algorithm 3. In particular, the atoms of β^+ contained in an investigated strongly connected component of $(atom(\Pi), \leq^+)$ must not belong to scope S, comprising potentially unfounded atoms. In fact, the following result shows that all "interesting" unfounded sets, namely, unfounded loops, are part of S; conversely, atoms outside S cannot belong to an unfounded loop.

Lemma 4.2. *Let Π be a logic program and $\mathbf{A}$ an assignment that is atom-saturated for Π.*

If UNFOUNDEDSET$(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that every unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ of Π wrt $\mathbf{A}$ such that all $p \in U$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$ is contained in S whenever Line 6 of Algorithm 3 is entered.*

The previous lemmas along with Corollary 3.12 can now be combined to, essentially, establish the completeness of Algorithm 3.[9]

Theorem 4.3. *Let Π be a logic program and $\mathbf{A}$ an assignment that is both atom- and body-saturated for Π.*

If UNFOUNDEDSET$(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that* UNFOUNDEDSET$(\Pi, \mathbf{A})$ *returns an unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ of Π wrt $\mathbf{A}$, where $U = \emptyset$ iff there is no unfounded set U' of Π wrt $\mathbf{A}$ such that $U' \not\subseteq \mathbf{A^F}$.*

After considering unfounded set detection, we now turn to NOGOODPROPAGATION in Algorithm 2. The next lemma is straightforward yet helpful, as it assures the prerequisites of demand-driven unfounded set checking, mainly focusing on unfounded loops.

Lemma 4.4. *Let Π be a logic program, ∇' a set of nogoods, $dl \in \mathbb{N}$, and $\mathbf{A}'$ an assignment.*

Then, we have that $\mathbf{A}$ is both atom- and body-saturated for Π whenever Line 10 of Algorithm 2 is entered in an execution of NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$.

The following properties are essential for CONFLICTANALYSIS to be well-defined as well as the soundness and completeness of CDNL-ASP.

Lemma 4.5. *Let Π be a logic program, ∇' a set of nogoods, $dl \in \mathbb{N}$, and $\mathbf{A}'$ an assignment.*

If NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ *is invoked on a valid source pointer configuration, then we have that* NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ *returns a pair $(\mathbf{A}, \nabla)$ such that*

1. $\nabla' \subseteq \nabla \subseteq \nabla' \cup \Lambda_\Pi$;

[9]Soundness, viz., the property that any set U returned by UNFOUNDEDSET is indeed unfounded, is obvious in view of the test in Line 9 of Algorithm 3 and the fact that $\emptyset$, which can be returned in Line 18, is trivially unfounded.

2. $\mathbf{A}$ *is an assignment such that* $\mathbf{A}' \subseteq \mathbf{A}$ *and every* $\sigma \in \mathbf{A} \setminus \mathbf{A}'$ *is implied by* $\Delta_\Pi \cup \nabla$ *wrt* $\mathbf{A}$*;*

3. $\delta \subseteq \mathbf{A}$ *for some* $\delta \in \Delta_\Pi \cup \nabla$ *if* $\lambda(p, U) \subseteq \mathbf{A}$ *for some* $\lambda(p, U) \in \Lambda_\Pi$.

The first item expresses that only loop nogoods can possibly be added by NOGOODPROPAGATION, viz., in Line 14 of Algorithm 2 (provided that Π is non-tight). In view of Theorem 3.7, this makes sure that the recorded nogoods do not eliminate any answer set of Π. The second item states that any literal assigned within NOGOODPROPAGATION has some antecedent, which can (later on) be used for conflict resolution. Finally, the third item exploits Theorem 4.3 and Lemma 4.4 to establish that violations of (loop) nogoods cannot stay undetected.

Regarding CONFLICTANALYSIS in Algorithm 4, the next lemma states that its derived nogoods are asserting and entailed by the nogoods that are already given.

Lemma 4.6. *Let* Π *be a logic program,* ∇ *a set of nogoods,* $\mathbf{A}$ *an assignment such that* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ *and* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\rho) = dl(\sigma)\} \subseteq \{\sigma \in \mathbf{A} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}\}$*, and* $\delta' \subseteq \mathbf{A}$ *such that* $m = \max(\{dl(\sigma) \mid \sigma \in \delta'\} \cup \{0\}) \neq 0$.

Then, we have that CONFLICTANALYSIS$(\delta', \Pi, \nabla, \mathbf{A})$ *returns a pair* (δ, k) *such that*

1. $\delta \subseteq \mathbf{A}$*;*

2. $|\{\sigma \in \delta \mid k < dl(\sigma)\}| = 1$*;*

3. $\delta \not\subseteq \mathbf{B}$ *for any solution* $\mathbf{B}$ *for* $\Delta_\Pi \cup \nabla \cup \{\delta'\}$.

The above prerequisites regarding ∇, $\mathbf{A}$, and δ' stipulate the existence of antecedents for all but the first literal assigned at decision level $m > 0$. These conditions are guaranteed by CDNL-ASP, as it increments dl in Line 15 of Algorithm 1 before assigning a decision literal (without antecedent) in Line 17, and as conflicts are analyzed only if encountered beyond decision level 0.

After inspecting the subroutines of CDNL-ASP, important invariants of assignments and nogoods generated by CDNL-ASP can be summarized as follows.

Lemma 4.7. *Let* Π *be a logic program.*

Then, we have that the following holds whenever Line 5 of Algorithm 1 is entered in an execution of CDNL-ASP(Π)*:*

1. ∇ *is a set of nogoods such that* $\delta \not\subseteq \mathbf{B}$ *for every* $\delta \in \nabla$ *and any solution* $\mathbf{B}$ *for* $\Delta_\Pi \cup \Lambda_\Pi$*;*

2. $\mathbf{A}$ *is an assignment such that* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ *and* $\{\sigma \in \mathbf{A} \mid dl(\sigma) \leq \max(\{dl(\rho) \mid \rho \in \mathbf{A}[\sigma]\} \cup \{0\})\} \subseteq \{\sigma \in \mathbf{A} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}\}$*;*

3. $dl \in \mathbb{N}$ *is such that* $\delta \not\subseteq \{\sigma \in \mathbf{A} \mid dl(\sigma) < dl\}$ *for every* $\delta \in \Delta_\Pi \cup \Lambda_\Pi \cup \nabla$.

Given that only the (implied) literals belonging to the current assignment $\mathbf{A}$ require antecedents for the second invariant to hold, dynamic nogoods in ∇ that are not antecedents may optionally be deleted. This yields polynomial space complexity of CDNL-ASP because the number of (required) antecedents is bounded by the maximum number of assigned literals, viz., $|atom(\Pi) \cup body(\Pi)|$. In practice, nogood deletion (cf. [72, 35]) is an important technique preventing conflict-driven learning solvers from blowing up in space.

Finally, the above results allow for deriving the soundness and completeness of CDNL-ASP.

Theorem 4.8. *Let* Π *be a logic program.*

Then, we have that CDNL-ASP(Π) *terminates, and it returns an answer set of* Π *iff* Π *has some answer set.*

Soundness wrt the decision problem of answer set existence follows from the observations made above, namely, that violated loop nogoods are detected and that nogoods added by NOGOOD-PROPAGATION or derived by CONFLICTANALYSIS are entailed. The completeness of CDNL-ASP, viz., the fact that it is a decision procedure, is due to its termination. Notably, arguments for the termination of CDCL (cf. [128, 115]) also apply to CDNL-ASP, given that both search procedures make use of conflict-driven assertions to exclude repetitions of assignments.

5 The *clasp* System

Our approach to conflict-driven ASP solving is implemented in *clasp* [54, 52, 56], combining the high-level modeling capacities of ASP with state-of-the-art Boolean constraint solving techniques. The solver *clasp* is freely available as an open source package at [110] and distributed under GNU general public license.

The *clasp* system is originally designed and optimized for conflict-driven ASP solving, as described in Section 4. To this end, it features a number of sophisticated reasoning and implementation techniques, some specific to ASP and others borrowed from CDCL-based SAT solvers. Moreover, *clasp* can be used as a full-fledged SAT, MaxSAT, or PB solver, accepting propositional CNF formulas in (extended) *dimacs* format as well as PB formulas in *opb* format. The flexibility of input formats, reasoning modes (cf. Section 5.2), and tuning parameters (cf. Section 5.4) supported by *clasp* goes well beyond the margins of typical ASP or SAT solvers. This section, however, is primarily devoted to ASP solving, describing the main features of *clasp*. Albeit the theoretical considerations in Section 4 concentrated on normal logic programs, one such feature is *clasp*'s ability to treat *extended rules* [119] intrinsically (without a priori compilation), supporting choice constructs in rule heads as well as cardinality and weight constraints in rule bodies. While the nogoods stemming from normal programs, described in Section 3, can be represented by clauses, *clasp* also includes dedicated data structures for dealing with linear inequalities obtained from extended programs or PB formulas. In order to give a comprehensive overview about the functionalities provided by *clasp*, we below discuss also such features.

5.1 Interfaces and Preprocessing

For ASP solving, *clasp* reads propositional logic programs (without proper disjunctions in rule heads) in *lparse* format [120], provided by either *lparse* [120] or *gringo* [51]. Choice rules as well as cardinality and weight constraints (cf. [119, 120]) are either compiled into normal rules during parsing, configurable via option `--trans-ext`, or dealt with in an intrinsic fashion (by default; see Section 5.3 for details).

At the beginning, a logic program is subject to extensive preprocessing [55]. The idea is to simplify the program while identifying equivalences among its relevant constituents. These equivalences are then used for building a compact program representation (in terms of Boolean constraints). Logic program preprocessing is configured via option `--eq`, taking an integer value fixing the number of iterations. Once a program has been transformed into Boolean constraints, they can be subject to further preprocessing, primarily based on resolution [33]. Such *SatELite*-like preprocessing is invoked with option `--sat-prepro` and further parameters. However, care must be taken when adapting techniques from SAT because preprocessing must not eliminate variables that are relevant to unfounded set checking or that occur in extended rules and optimization statements.

A major yet internal feature of *clasp* is that it can be used in a stateful way. That is, *clasp* may keep its state, involving program representation, recorded nogoods, heuristic values, etc., and be invoked under additional (temporary) assumptions and/or by adding new atoms and rules. The corresponding interfaces are fundamental for supporting incremental ASP solving as realized in *iclingo* [46], a combination of *gringo* and *clasp* for incremental grounding and solving. Furthermore, solving under assumptions [34] is used in our parallel ASP solver *claspar* [37, 118, 50].

5.2 Reasoning Modes

Although *clasp*'s primary use case is the computation of answer sets, it also allows for computing supported models of a logic program via option `--supp-models`.[10] In addition, *clasp* provides a number of reasoning modes, determining how to proceed when a model is found.

Enumeration Solution enumeration is non-trivial in the context of backjumping and conflict-driven learning. A simple approach relies on recording solutions as nogoods and exempting them from deletion. Although *clasp* supports this via option `--solution-recording`, it is prone to blow up in space in view of an exponential number of solutions (in the worst case). Unlike this, the default enumeration algorithm of *clasp* runs in polynomial space [53]. Both enumeration approaches also allow for projecting models to a subset of atoms [57], invoked with `--project` and configured via the well-known directives `#hide` and `#show` of *lparse* and *gringo*. This option is of great practical value whenever one faces overwhelmingly many models, involving solution-irrelevant variables having proper combinatorics. For example, the program consisting of the choice rule `{a,b,c}.` has eight (obvious) answer sets. When augmented with directive `#hide c.`, still eight solutions are obtained, yet including four duplicates. Unlike this, invoking *clasp* with `--project` yields only four answer sets differing on `a` and/or `b`.

As regards implementation, it is interesting to note that *clasp* offers a dedicated interface for enumeration. This allows for abstracting from how to proceed once a model is found and thus makes the search algorithm independent of the concrete enumeration strategy. Further reasoning modes implemented via the enumeration interface admit computing the intersection or union of all answer sets of a program (via `--cautious` and `--brave`, respectively). Rather than computing the whole collection of (possibly) exponentially many answer sets, the idea is to compute a first answer set, record a constraint eliminating it from further solutions, then compute a

[10]To be more precise, option `--supp-models` disables unfounded set checking. Sometimes, the grounder or preprocessing may already eliminate some supported models such that they cannot be recovered later on.

second answer set, strengthen the constraint to represent the intersection (or union) of the first two answer sets, and to continue like this until no more answer set is obtained. This process involves computing at most as many answer sets as there are atoms in an input program. Either the cautious or the brave consequences are then given by the atoms captured by the final constraint.

Optimization As common in *lparse*-like languages, an objective function is specified via a sequence of `#minimize` or `#maximize` statements. For finding optimal solutions, *clasp* offers several options. First, the objective function can be initialized via `--opt-value`. Second, *clasp* allows for computing one or all (via `--opt-all`) optimal solutions. Such options are useful when one is interested in computing consequences belonging to all optimal solutions (in combination with `--cautious`). To this end, one starts with searching for an (arbitrary) optimal answer set and then re-launches *clasp* by bounding its search with the obtained optimum. Doing the latter with `--cautious` yields the atoms that belong to all optimal answer sets. On application problems, option `--restart-on-model`, making *clasp* restart after each (putatively optimal) solution, turned out to be effective for ameliorating convergence to an optimum. Particular strategies for lexicographic optimization [48], available in *clasp* series 2, serve the same purpose, especially on large and underconstrained multi-criteria optimization problems. Moreover, option `--opt-heu` can be used to alter sign selection (see below) towards a better objective function value. Optimization is implemented via the aforementioned enumeration interface. When a solution is found, an optimization constraint is updated with the corresponding objective function value. Then, the decision level violating the constraint is identified and retracted, or if the constraint is violated at decision level 0, search terminates. It is also worth mentioning that *clasp* propagates optimization constraints, that is, they can imply (and provide reasons for) literals upon unit propagation. Finally, when optimization is actually undesired and all solutions ought to be inspected instead, option `--opt-ignore` is available to make modifying the input (by removing optimization statements) obsolete.

5.3 Propagation and Search

Propagation in *clasp* relies on an interface called *Boolean constraint*; it is thus not limited to (clausal representations of) nogoods (cf. [35]). However, dedicated data structures are used for binary and ternary nogoods (cf. [115]), accounting for the many short nogoods stemming from Clark completion. More complex constraints are accessed via two *watch lists* for each variable, storing the Boolean constraints that need to be updated when the variable becomes true or false, respectively. While unit propagation of long nogoods is based on the well-known two-watched-literal data structure [101], a counter-based approach is used for propagating cardinality and weight constraints [47]. A literal implied by a Boolean constraint upon unit propagation stores a reference to that constraint, which in turn can be queried for an antecedent.

During unit propagation, binary nogoods are handled before ternary ones, which are in turn inspected before other Boolean constraints. As detailed in Algorithm 2, our propagation procedure is distinct in giving a clear preference to unit propagation over unfounded set computations. Unfounded set detection follows Algorithm 3 and aims at small, rather than greatest, unfounded sets. As detailed in [47], intrinsic treatment of cardinality and weight constraints augments unfounded set detection by means of source pointers, still aiming at lazy unfounded set checking. The representation of loop nogoods is controlled via option `--loops`. In the default setting, loop nogoods are generated for individual unfounded atoms, as shown in Algorithm 2. Like nogoods derived from conflicts, they are subject to unit propagation and deletion. However, when `--loops=no` is specified, loop nogoods are stored only as long as they serve as antecedents of falsified unfounded atoms.

Decision Heuristics The primary decision heuristics of *clasp* use *look-back* strategies derived from corresponding CDCL-based approaches in SAT, viz., *vsids* [101], *berkmin* [72], and *vmtf* [115]. Such heuristics privilege variables involved in recent conflicts. To this end, they maintain an activity score for each variable, increased upon conflict resolution and decayed periodically. The major difference between the approaches of *vsids* and *berkmin* lies in the scope of variables considered during decision making. While *vsids* selects a free variable that is globally most active, *berkmin* restricts the selection to variables belonging to the most recently recorded yet undispelled dynamic nogood. Although the look-back heuristics implemented in *clasp* are modeled after the corresponding CDCL-based approaches, *clasp* optionally also scores variables contained in loop nogoods. In case of *berkmin*, it may also select a free variable belonging to a recently recorded loop nogood. Finally, we note that *clasp*'s heuristic can also be based upon *look-ahead* strategies extending unit propagation by *failed-literal detection* [42]. This makes sense when running *clasp* without conflict-driven nogood learning, operating similar to *smodels*.

Once a decision variable has been selected, a sign heuristic decides about its truth value. The main criterion for look-back heuristics is to satisfy the greatest number of conflict nogoods, that is, to pick the literal that occurs in fewer of them. Initially and also for tie-breaking, *clasp* does sign selection based on the type of a variable: atoms are preferably set to false, while bodies are made true. This aims at maximizing the number of resulting implications. Another sign heuristic implemented in *clasp* is *progress saving* [107]. The idea is to remember truth values of retracted variables upon backjumping (or restarting), except for those assigned at the last decision level. These saved values are then used for sign selection. The intuition behind this strategy is that the literals assigned prior to the last decision level did not lead to a conflict and may have satisfied some subproblem. Hence, re-establishing them may help to avoid solving subproblems multiple times. Progress saving is invoked with option `--save-progress`; its computational impact, however, depends heavily on the structure of an application at hand (cf. Section 5.4).

Restart Policies The robustness of *clasp* is boosted by multiple restart strategies (cf. [74]), namely, geometric, fixed-interval, Luby-style, or a nested policy. The first two start with an initial number of conflicts after which *clasp* restarts; this threshold can then be increased after each restart. The third policy, going back to Luby, Sinclair, and Zuckerman [92], schedules restarts according to a recurrent and progressively growing sequence of numbers of conflicts, e.g., $32\ 32\ 64\ 32\ 32\ 64\ 128\ 32\ \dots$ for unit 32. In addition, the nested policy first used in *picosat* [11] is also offered by *clasp*. This policy takes three parameters, x, y, and z, and makes restarts follow a two-dimensional pattern that increases geometrically in both dimensions. The geometric restart sequence $x * y^i$ is repeated when it reaches an outer limit $z * y^j$, where i counts the number of performed restarts and j how often the outer limit was hit so far. Usually, restart strategies are based on the total number of encountered conflicts. Beyond that, *clasp* features local restarts [116]. Here, one counts the number of conflicts per decision level in order to measure the difficulty of subproblems locally. Furthermore, a bounded approach to restarting (and backjumping) is used when enumerating answer sets as described in [53]. To complement its more determined search, *clasp* also allows for initial randomized runs [35], typically with a small restart threshold, in the hope to extract putatively interesting nogoods. Finally, it is worth noting that, despite of the fact that recent SAT solvers use rather aggressive restart strategies (cf. Section 5.4), *clasp* still defaults to a more conservative geometric policy (cf. [35]) because it performs better on ASP-specific benchmarks.

Nogood Deletion To limit the number of nogoods stored simultaneously, dynamic nogoods are periodically subject to deletion. Complementing look-back heuristics, *clasp*'s nogood deletion strategy associates an activity with each recorded nogood, which is incremented whenever the nogood is used for conflict resolution. Borrowing ideas from *minisat* [35] and *berkmin* [72], the initial threshold on the number of stored nogoods is calculated from the size of an input program and increased by a certain factor upon each restart. As soon as the current threshold is exceeded, deletion is initiated and removes up to 75% of the recorded nogoods. Nogoods that are currently locked (because they serve as antecedents) or whose activities significantly exceed the average activity are exempt from deletion. However, the nogoods that are not deleted have their activities decayed in order to account for recency of usage. All in all, *clasp*'s nogood deletion strategy aims at limiting the overall number of stored nogoods, while keeping the relevant and recently recorded ones. This likewise applies to conflict and loop nogoods.

5.4 Fine-Tuning

Advanced Boolean constraint solving technology adds a multitude of degrees of freedom to ASP solving. Currently, *clasp* has about 40 options, half of which control the search strategy. Although considerable efforts were taken to find default parameters jointly boosting robustness and speed, the default setting still leaves room for drastic improvements on specific benchmarks by fine-tuning the parameters. The question then arises how to deal with this vast "configuration space" and how to conciliate it with the idea of declarative problem solving. Currently, there seems to be no alternative to manual fine-tuning when addressing highly demanding applications.

As rules of thumb, we usually start by investigating the following options:

`--heuristic`: Try *vsids* instead of *clasp*'s default *berkmin*-style heuristic.

`--trans-ext`: Applicable if a program contains extended rules, that is, rules including cardinality and weight constraints. Try at least the *dynamic* transformation.

`--sat-prepro`: Resolution-based preprocessing (as in *SatELite* [33]) works best on tight programs with few cardinality and weight constraints. It should almost always be used when extended rules are transformed into normal ones (via `--trans-ext`).

`--restarts`: Try aggressive restart policies, like *Luby-256* or the *nested* policy, or try disabling restarts whenever a problem is deemed to be unsatisfiable.

`--save-progress`: Progress saving typically works nicely when the average backjump length (or the #choices/#conflicts ratio) is high ($\geq$10). It usually performs best in combination with aggressive restarts.

The impact of fine-tuning can be seen on the following examples. As observed in [78], *clasp* times out on satisfiable *4-coloring* problems. However, with `--save-progress`, *clasp* solves each instance in about a second (the average backjump length is $>$60). For another example, consider the benchmark class *WeightBoundedDomSet* from the second ASP competition [28]. The default configuration of *clasp* results in timeouts (see next section), all of which vanish once aggressive restarts are used. Similar effects can be observed on application problems featuring yet different characteristics.

Although fine-tuning may greatly improve the efficiency of *clasp*, it is hard to accomplish for an unpracticed user, and after all it takes us away from the ideals of declarative problem solving. To this end, we advocate an extension of *clasp*, called *claspfolio* [49], that maps benchmark features to solver configurations (via machine learning techniques). It is part of our ongoing work to investigate how far the selection of effective parameter settings can be automated.

6 Experimental Results

We conducted experiments on NP decision problems of the second ASP competition [28], using encodings by the *Potassco* team.[11] Our comparison considers *clasp* (version 1.3.1) in its default setting as well as a setting suited better for the benchmarks in focus. The latter, denoted by *clasp*$^+$, invokes *clasp* with options `--sat-prepro` and `--trans-ext=dynamic`, using *SatELite*-like preprocessing [33] on nogoods as well as a context-dependent handling of extended rules, excluding "small" extended rules from an intrinsic treatment (cf. [47]) and rather transforming them into normal rules. For comparison, we also consider *cmodels* (version 3.79 with *minisat* 2.0), *smodels* (version 2.34 with option `-restart`), and *lp2sat* (version 1.13 with *minisat* 2.0 or *clasp* 1.3.1).[12] For *cmodels* and *lp2sat*, we below indicate the use of either *minisat* or *clasp* as underlying SAT solver by adding "[m]" or "[c]", respectively. The experiments were run sequentially under Linux on an Intel Quad-Core Xeon E5520 machine equipped with 2.27GHz processors. Every benchmark instance (in *lparse* output format [120], generated offline with *gringo*) was run three times per solver, each run restricted to 600 seconds time and 2GB RAM. A run finished when the solver found an answer set, reported unsatisfiability (no answer set), or was aborted due to time or memory exhaustion.

SAT-based solver *cmodels* converts a logic program into propositional clauses via Clark completion and delegates the search for supported models to *minisat*. Except for the treatment of

[11]See `http://dtai.cs.kuleuven.be/events/ASP-competition/SubmittedBenchmarks.shtml` for detailed descriptions of benchmark classes as well as `http://dtai.cs.kuleuven.be/events/ASP-competition/encodings.shtml` for benchmark instances and encodings.

[12]Additional results for *cmodels* with *zchaff* 2007.3.12 and *smodels* without restarts are available at [22].

extended rules, this approach is comparable to *clasp* on tight programs. In the non-tight case, *cmodels* delays (sophisticated) unfounded set checks until an assignment is total, while *clasp* and *smodels* integrate them into propagation. In fact, *smodels* is a "traditional" ASP solver using a search pattern based on systematic backtracking along with an unfounded set checking procedure computing greatest unfounded sets. Finally, *lp2sat* like *cmodels* converts a logic program into propositional clauses and delegates the search for a model to some SAT solver. On tight programs, *lp2sat*'s translation amounts to Clark completion, while level mappings [76, 103] are used to capture non-circular derivability of atoms from non-tight programs. Among the solvers accepting *lparse* output format, our experiments include the ones that were leading in the NP decision category of the second ASP competition (cf. [28]).[13] In particular, *lp2sat* has an edge on *lp2diff* [78], which applies a translation to difference logic and solvers for Satisfiability Modulo Theories (SMT; [10]) as search back-ends, on the investigated benchmarks [77].

Our experimental results are summarized in Table 7–9, giving average runtimes in seconds and numbers of aborted runs (in parentheses) for every solver on each benchmark class, with time or memory exhaustions taken as 600 seconds. More detailed benchmark results, including individual times for all instances as well as further solver configurations, are provided at [22]. While Table 7 considers all benchmarks, divided into tight and non-tight ones, Table 8 and 9 analogously report results restricted to satisfiable and unsatisfiable instances, respectively. Each table gives the number of instances per benchmark class in the column headed by "#". In addition, Table 7 provides the respective partition into satisfiable and unsatisfiable instances (in parentheses). The last column amounts to the virtual best solver, composed of the smallest runtime and the smallest number of aborts observed on each benchmark class. The rows marked with "$\varnothing(\varnothing)$" average runtimes and time or memory exhaustions over a collection of benchmark classes under consideration.[14] The following row gives the Euclidean distance (in an n-dimensional space, where n is the number of benchmark classes and a point is a column of n average runtimes) of each solver to the virtual best one on the respective collection in focus; the quadratic distance calculation scheme punishes imbalanced and rewards consistent performance more than averaging. Some benchmark classes make heavy use of extended rules, so that their different treatments, e.g., in *clasp*$^+$ and *cmodels*, have significant impact on the observed performances; such benchmark classes are marked with "*" in Table 7–9.

Considering the results on tight benchmarks in the upper part of Table 7, we note that the traditional ASP solver *smodels* is consistently outperformed by systems exploiting conflict-driven learning. For instance, *smodels* times out on all satisfiable instances of *15Puzzle*, which are rather unproblematic for the other solvers. In fact, occasional varying performances of the latter on tight programs are due to different treatments of extended rules and/or determinizations of inherent non-determinisms in *minisat* and *clasp*, respectively. Any such differences may turn out to be advantageous the one or the other way around; e.g., *clasp*$^+$ and *lp2sat*[c] have an edge on other solvers on *GraphColouring*, *clasp* in its default setting is fastest on *SchurNumbers*, while

[13]Some non-participating solvers, e.g., *assat*, *sag* [91], and *smodels*$_{cc}$, are no longer maintained and thus not incorporated here. Although original *smodels* did not participate either (but its close derivative *smodels-ie* [15] did), we still consider it for reference. At the time of running the experiments, the solver *minisat(id)* [95], supporting "inductive definitions" on top of propositional theories, did not accept *lparse* output format.

[14]We provide averages (rather than sums) for balancing diverse numbers of instances per benchmark class.

Benchmark	#	*clasp*	*clasp*$^+$	*cmodels*[m]	*smodels*	*lp2sat*[m]	*lp2sat*[c]	*virtual best*
15Puzzle	16 (16/0)	33.01 (0)	20.18 (0)	31.36 (0)	600.00 (48)	22.21 (0)	15.13 (0)	15.13 (0)
BlockedNQueens	29 (15/14)	5.09 (0)	4.91 (0)	9.04 (0)	29.37 (0)	13.19 (0)	5.22 (0)	4.91 (0)
ChannelRouting	10 (6/4)	120.13 (6)	120.14 (6)	120.58 (6)	120.90 (6)	121.34 (6)	121.08 (6)	120.13 (6)
EdgeMatching	29 (29/0)	0.23 (0)	0.41 (0)	59.32 (0)	60.32 (0)	13.05 (0)	5.58 (0)	0.23 (0)
*Fastfood**	29 (10/19)	1.17 (0)	0.90 (0)	29.22 (0)	83.93 (3)	46.85 (0)	24.95 (0)	0.90 (0)
GraphColouring	29 (9/20)	421.55 (60)	357.88 (39)	422.66 (57)	453.77 (63)	409.70 (51)	357.57 (39)	357.57 (39)
Hanoi	15 (15/0)	11.76 (0)	3.97 (0)	2.92 (0)	523.77 (39)	3.81 (0)	5.36 (0)	2.92 (0)
*HierarchicalClustering**	12 (8/4)	0.16 (0)	0.17 (0)	0.76 (0)	1.56 (0)	0.94 (0)	0.86 (0)	0.16 (0)
SchurNumbers	29 (13/16)	17.44 (0)	49.60 (0)	75.70 (0)	504.17 (72)	90.88 (6)	36.93 (0)	17.44 (0)
Solitaire	27 (22/5)	204.78 (27)	162.82 (21)	175.69 (21)	316.96 (36)	222.60 (27)	210.14 (27)	162.82 (21)
Sudoku	10 (10/0)	0.15 (0)	0.16 (0)	2.55 (0)	0.25 (0)	0.87 (0)	0.82 (0)	0.15 (0)
*WeightBoundedDomSet**	29 (29/0)	123.13 (15)	102.18 (12)	300.26 (36)	400.84 (51)	179.56 (9)	143.87 (12)	102.18 (9)
∅ (∅) *(tight)*	264 (182/82)	78.22 (9.00)	68.61 (6.50)	102.50 (10.00)	257.99 (26.50)	93.75 (8.25)	77.29 (7.00)	65.38 (6.25)
Eucl. dist. *(tight)*		81.80	32.58	227.19	991.76	141.67	70.51	0.00
*ConnectedDomSet**	21 (10/11)	40.42 (3)	36.11 (3)	7.46 (0)	183.76 (15)	13.43 (0)	13.62 (0)	7.46 (0)
*GeneralizedSlitherlink**	29 (29/0)	0.10 (0)	0.22 (0)	1.92 (0)	0.16 (0)	5.05 (0)	12.90 (0)	0.10 (0)
*GraphPartitioning**	13 (6/7)	9.27 (0)	7.98 (0)	20.19 (0)	92.10 (3)	365.18 (21)	344.39 (21)	7.98 (0)
HamiltonianPath	29 (29/0)	0.07 (0)	0.06 (0)	0.21 (0)	2.22 (0)	3.45 (0)	15.68 (0)	0.06 (0)
KnightTour	10 (10/0)	124.29 (6)	91.80 (3)	242.48 (12)	150.55 (3)	545.42 (27)	487.61 (24)	91.80 (3)
Labyrinth	29 (29/0)	123.82 (12)	82.92 (6)	142.24 (6)	594.10 (81)	282.23 (27)	534.62 (75)	82.92 (6)
MazeGeneration	29 (10/19)	91.17 (12)	89.89 (12)	90.41 (12)	293.62 (42)	125.94 (9)	85.57 (6)	85.57 (6)
Sokoban	29 (9/20)	0.73 (0)	0.80 (0)	3.39 (0)	176.01 (15)	6.11 (0)	3.99 (0)	0.73 (0)
*TravellingSalesperson**	29 (29/0)	0.05 (0)	0.06 (0)	317.82 (7)	0.22 (0)	441.68 (55)	198.34 (9)	0.05 (0)
WireRouting	23 (12/11)	42.81 (3)	36.36 (3)	175.73 (12)	448.32 (45)	460.89 (48)	459.97 (51)	36.36 (3)
∅ (∅) *(non-tight)*	241 (173/68)	43.27 (3.60)	34.62 (2.70)	100.19 (4.90)	194.11 (20.40)	224.94 (18.70)	215.67 (18.60)	31.30 (1.80)
Eucl. dist. *(non-tight)*		62.37	28.97	383.16	739.35	866.08	832.52	0.00
∅ (∅)	505 (355/150)	62.33 (6.55)	53.16 (4.77)	101.45 (7.68)	228.95 (23.73)	153.38 (13.00)	140.19 (12.27)	49.89 (4.23)
Eucl. dist.		102.86	43.59	445.45	1237.02	877.59	835.50	0.00

Table 7: Average runtimes on benchmarks of the second ASP competition.

Benchmark	#	*clasp*	*clasp*$^+$	*cmodels*[m]	*smodels*	*lp2sat*[m]	*lp2sat*[c]	*virtual best*
15Puzzle	16	33.01 (0)	20.18 (0)	31.36 (0)	600.00 (48)	22.21 (0)	15.13 (0)	15.13 (0)
BlockedNQueens	15	3.48 (0)	4.93 (0)	7.52 (0)	22.13 (0)	13.99 (0)	4.16 (0)	3.48 (0)
ChannelRouting	6	0.16 (0)	0.17 (0)	0.67 (0)	1.35 (0)	1.63 (0)	1.37 (0)	0.16 (0)
EdgeMatching	29	0.23 (0)	0.41 (0)	59.32 (0)	60.32 (0)	13.05 (0)	5.58 (0)	0.23 (0)
*Fastfood**	10	0.12 (0)	0.49 (0)	9.26 (0)	82.44 (0)	45.33 (0)	18.54 (0)	0.12 (0)
GraphColouring	9	24.99 (0)	32.66 (0)	98.64 (3)	128.80 (3)	57.93 (0)	31.27 (0)	24.99 (0)
Hanoi	15	11.76 (0)	3.97 (0)	2.92 (0)	523.77 (39)	3.81 (0)	5.36 (0)	2.92 (0)
*HierarchicalClustering**	8	0.14 (0)	0.13 (0)	1.02 (0)	1.52 (0)	1.20 (0)	1.03 (0)	0.13 (0)
SchurNumbers	13	37.25 (0)	109.13 (0)	166.77 (0)	600.00 (39)	200.49 (6)	80.40 (0)	37.25 (0)
Solitaire	22	114.96 (12)	63.46 (6)	79.25 (6)	252.63 (21)	136.82 (12)	121.54 (12)	63.46 (6)
Sudoku	10	0.15 (0)	0.16 (0)	2.55 (0)	0.25 (0)	0.87 (0)	0.82 (0)	0.15 (0)
*WeightBoundedDomSet**	29	123.13 (15)	102.18 (12)	300.26 (36)	400.84 (51)	179.56 (9)	143.87 (12)	102.18 (9)
∅ (∅) *(tight, sat)*	182	29.11 (2.25)	28.16 (1.50)	63.29 (3.75)	222.84 (16.75)	56.41 (2.25)	35.76 (2.00)	20.85 (1.25)
Eucl. dist. (tight, sat)		59.07	72.48	256.02	1037.56	203.65	85.96	0.00
*ConnectedDomSet**	10	9.28 (0)	1.74 (0)	12.06 (0)	135.10 (6)	17.57 (0)	12.36 (0)	1.74 (0)
*GeneralizedSlitherlink**	29	0.10 (0)	0.22 (0)	1.92 (0)	0.16 (0)	5.05 (0)	12.90 (0)	0.10 (0)
*GraphPartitioning**	6	0.11 (0)	0.14 (0)	4.52 (0)	0.56 (0)	114.21 (3)	118.25 (3)	0.11 (0)
HamiltonianPath	29	0.07 (0)	0.06 (0)	0.21 (0)	2.22 (0)	3.45 (0)	15.68 (0)	0.06 (0)
KnightTour	10	124.29 (6)	91.80 (3)	242.48 (12)	150.55 (3)	545.42 (27)	487.61 (24)	91.80 (3)
Labyrinth	29	123.82 (12)	82.92 (6)	142.24 (6)	594.10 (81)	282.23 (27)	534.62 (75)	82.92 (6)
MazeGeneration	10	0.07 (0)	0.08 (0)	0.08 (0)	0.15 (0)	114.32 (0)	10.92 (0)	0.07 (0)
Sokoban	9	0.63 (0)	0.78 (0)	5.40 (0)	320.54 (9)	10.77 (0)	4.34 (0)	0.63 (0)
*TravellingSalesperson**	29	0.05 (0)	0.06 (0)	317.82 (7)	0.22 (0)	441.68 (55)	198.34 (9)	0.05 (0)
WireRouting	12	74.00 (3)	62.63 (3)	134.94 (3)	407.44 (18)	513.20 (30)	519.94 (30)	62.63 (3)
∅ (∅) *(non-tight, sat)*	173	33.24 (2.10)	24.04 (1.20)	86.17 (2.80)	161.10 (11.70)	204.79 (14.20)	191.49 (14.10)	24.01 (1.20)
Eucl. dist. (non-tight, sat)		53.99	0.20	364.12	709.78	818.54	789.78	0.00
∅ (∅) *(sat)*	355	30.99 (2.18)	26.29 (1.36)	73.69 (3.32)	194.78 (14.45)	123.85 (7.68)	106.55 (7.50)	22.29 (1.23)
Eucl. dist. (sat)		80.02	72.48	445.12	1257.11	843.49	794.44	0.00

Table 8: Average runtimes on *satisfiable* benchmarks of the second ASP competition.

Benchmark	#	*clasp*	*clasp*$^+$	*cmodels*[m]	*smodels*	*lp2sat*[m]	*lp2sat*[c]	*virtual best*
BlockedNQueens	14	6.82 (0)	4.88 (0)	10.68 (0)	37.14 (0)	12.33 (0)	6.35 (0)	4.88 (0)
ChannelRouting	4	300.10 (6)	300.10 (6)	300.44 (6)	300.23 (6)	300.90 (6)	300.66 (6)	300.10 (6)
*Fastfood**	19	1.72 (0)	1.11 (0)	39.72 (0)	84.71 (3)	47.66 (0)	28.32 (0)	1.11 (0)
GraphColouring	20	600.00 (60)	504.24 (39)	568.48 (54)	600.00 (60)	567.99 (51)	504.40 (39)	504.24 (39)
*HierarchicalClustering**	4	0.21 (0)	0.24 (0)	0.24 (0)	1.63 (0)	0.42 (0)	0.50 (0)	0.21 (0)
SchurNumbers	16	1.34 (0)	1.24 (0)	1.70 (0)	426.30 (33)	1.82 (0)	1.61 (0)	1.24 (0)
Solitaire	5	600.00 (15)	600.00 (15)	600.00 (15)	600.00 (15)	600.00 (15)	600.00 (15)	600.00 (15)
∅ (∅) *(tight, unsat)*	82	215.74 (11.57)	201.69 (8.57)	217.32 (10.71)	292.86 (16.71)	218.73 (10.29)	205.98 (8.57)	201.68 (8.57)
Eucl. dist. (tight, unsat)		95.79	0.03	75.18	444.84	79.30	27.26	0.00
*ConnectedDomSet**	11	68.73 (3)	67.35 (3)	3.28 (0)	227.99 (9)	9.67 (0)	14.77 (0)	3.28 (0)
*GraphPartitioning**	7	17.12 (0)	14.70 (0)	33.62 (0)	170.56 (3)	580.29 (18)	538.22 (18)	14.70 (0)
MazeGeneration	19	139.12 (12)	137.16 (12)	137.95 (12)	448.09 (42)	132.06 (9)	124.86 (6)	124.86 (6)
Sokoban	20	0.78 (0)	0.81 (0)	2.49 (0)	110.97 (6)	4.01 (0)	3.83 (0)	0.78 (0)
WireRouting	11	8.78 (0)	7.70 (0)	220.23 (9)	492.92 (27)	403.83 (18)	394.56 (21)	7.70 (0)
∅ (∅) *(non-tight, unsat)*	68	46.91 (3.00)	45.54 (3.00)	79.51 (4.20)	290.10 (17.40)	225.97 (9.00)	215.25 (9.00)	30.27 (1.20)
Eucl. dist. (non-tight, unsat)		67.03	65.24	213.78	653.33	690.59	651.05	0.00
∅ (∅) *(unsat)*	150	145.39 (8.00)	136.63 (6.25)	159.90 (8.00)	291.71 (17.00)	221.75 (9.75)	209.84 (8.75)	130.26 (5.50)
Eucl. dist. (unsat)		116.91	65.24	226.62	790.39	695.13	651.62	0.00

Table 9: Average runtimes on *unsatisfiable* benchmarks of the second ASP competition.

lp2sat[m] yields the fewest aborts on *WeightBoundedDomSet*. On non-tight benchmarks, we observe that the problem representation overhead incurred by *lp2sat*'s translational approach is a major handicap, though *minisat* and *clasp* may react more or less sensitively (cf. *Labyrinth* and *TravellingSalesperson*). Except for *MazeGeneration*, the strategy of *cmodels* to verify candidate supported models found by *minisat* already improves on eager translation by *lp2sat*. However, integrating unfounded set checking into propagation is usually even more effective, as it can be observed when comparing *clasp* and *clasp*$^+$ to *cmodels* on *KnightTour* and *WireRouting*. Beyond dealing with (non-)tightness, the treatment of extended rules is a crucial factor on some benchmark classes. Their transformation into normal rules, as done by *cmodels* and *lp2sat*, turns out to be helpful on *ConnectedDomSet*,[15] while it drastically blows up problem representations and thus deteriorates performance on *TravellingSalesperson*.

Focusing on either satisfiable or unsatisfiable instances in Table 8 and 9, respectively, sheds some light on the distribution of hardness within benchmark classes, yet without exhibiting any overwhelming impact regarding relative solver performances. In fact, a look into Table 8, especially at *cmodels* and *smodels*, reveals that the satisfiable instances of *ChannelRouting*, *GraphPartitioning*, and *MazeGeneration* are rather easy. Interestingly, both *lp2sat* variants still have difficulties with the non-tight benchmarks, viz., *GraphPartitioning* and *MazeGeneration*, indicating that an eager translation of logic programs may diminish search performance. On the other hand, all solvers perform worse on the satisfiable instances of *SchurNumbers* than on the unsatisfiable ones, and the same also applies to *clasp*, *clasp*$^+$, and *lp2sat* on *WireRouting*. Looking at these two classes in Table 9, we observe that the unsatisfiable instances of *SchurNumbers* are trivial for all solvers but *smodels*, while only *clasp* and *clasp*$^+$ complete all of the unsatisfiable *WireRouting* instances. The latter suggests that such instances are not inherently hard but that lacking either conflict-driven learning (*smodels*) or native unfounded set checking (*cmodels* and *lp2sat*) renders them more difficult.

Unlike with *SchurNumbers* and *WireRouting*, some of the unsatisfiable instances of *ChannelRouting*, *GraphColouring*, *Solitaire*, and *MazeGeneration* turn out to be much harder than their satisfiable counterparts (at least for the considered solvers). Notably, *lp2sat*[c], running *clasp* as SAT solver, completes more unsatisfiable *MazeGeneration* instances than *clasp* and *clasp*$^+$ themselves, which contrasts with the behavior observed on satisfiable instances (cf. *MazeGeneration* in Table 8 and 9). A similar shift of behaviors is due to one unsatisfiable instance of *ConnectedDomSet*, which poses a problem to the intrinsic treatment of extended rules in *clasp* and *clasp*$^+$, while *cmodels* and *lp2sat* do not encounter such difficulties. In view of this, we believe that the dynamic selection among possible handlings of extended rules (intrinsic treatment and/or transformation) is interesting future work.

7 Related Work

Our approach to conflict-driven ASP solving borrows and extends state-of-the-art techniques from the area of SAT [12]. Its global search pattern is similar to CDCL with First-UIP scheme,

[15]Interestingly, a single instance of *ConnectedDomSet* is responsible for the three timeouts of *clasp* and *clasp*$^+$.

developed more than a decade ago [97, 101, 127] and nowadays quasi standard for industrial SAT solving (cf. [35, 115, 100, 29, 11, 4, 23, 96]). While traditional search procedures like Davis-Putnam-Logemann-Loveland (DPLL; [25, 24]) are polynomially equivalent to *tree-like* resolution, CDCL (with unlimited restarts) amounts to *general* resolution [109] and is thus strictly more powerful (regarding the best-case complexity of unsatisfiability proofs) than DPLL. Several investigations [62, 79, 70] show that this separation carries forward to (native) ASP solvers under standard translations between ASP and SAT, viz., Clark completion and the reduction in [102]. The *clasp* system implements the CDCL search pattern likewise for logic programs under answer set semantics as well as for propositional CNF and PB formulas. In contrast to translational approaches, as used by *minisat+* [36], it offers dedicated data structures for the internal representation of and reasoning about linear inequalities [47]. This relates *clasp* to SMT [10] techniques, applying "theory propagation" (cf. [104]) to concepts available in extended logic programs [119], viz., unfounded sets (or positive recursion) and linear inequalities (over Boolean variables).

SAT-based ASP solvers like *assat* [90], *cmodels* [71], and *sag* [91] may also exploit conflict-driven learning in the search for supported models being answer set candidates, accomplished by underlying SAT solvers. However, their integration of unfounded set checking is much more loose than in native ASP solvers. To our knowledge, the only native ASP solver other than *clasp* that implements conflict-driven learning is *smodels*$_{cc}$ [126],[16] while ASP solvers like *dlv* [83], *smodels* [119], and *nomore++* [1] perform DPLL-style search. For enabling conflict analysis, *smodels*$_{cc}$ takes an algorithmic approach, monitoring applications of *smodels*' inference rules to on-the-fly build an implication graph [97, 127, 9] as a representation of antecedents. Unlike this, *clasp* directly incorporates suitable data structures, designed to accommodate backjumping and conflict-driven learning: similar to SAT solvers, it merely stores references to antecedents upon propagation, which incurs only negligible (constant) "overhead". A prototypical extension [112] of *dlv* includes backjumping, but not learning, based on tracking *dlv*'s inference rules.

Given that answer sets are determined by atoms, native ASP solvers *dlv*, *smodels*, and *smodels*$_{cc}$ are (logically) restricted to assignments over atoms. As shown in [63, 62], this yields an exponential separation, already on tight logic programs, to solvers that in addition assign and make decisions on rule bodies. To our knowledge, *clasp* and *nomore++* are the only ASP solvers deliberately taking advantage of rule bodies in assignments. Regardless of minor technical differences, the comparison of *smodels*' and *nomore++*' inferences in [62] reveals that both are based on structural propositions for bodies, so that the restricted scope of heuristic decisions is the main trait of *smodels*' atom-oriented approach. In view of similarities to *smodels* (cf. [70]), this also applies to *dlv* and *smodels*$_{cc}$. Interestingly, CNF conversions of SAT-based ASP solvers (cf. [5]) also introduce auxiliary propositions for rule bodies to prevent an exponential blow-up. Although such auxiliary propositions can then be exploited by underlying SAT solvers, their motivation is more by need than by design. However, as there is not yet a consensus on how to represent the Boolean constraints induced by logic programs (see, e.g., the proposals in [5, 70, 86]), we have used nogoods to express conditions for (unit) propagation, thus separating semantics from syntactic representations.

[16]The solver *minisat(id)* [95] supports "inductive definitions" on top of propositional theories. Inductive definitions are closely related to logic programs, yet involve a "totality" condition not shared by the latter.

As pointed out in [63, 86], SAT-based and native ASP solvers differ in their "laziness" to apply unfounded set checks. While the former confine themselves to final tests required for soundness, in the terminology of [104] investigating SMT, native ASP solvers perform "theory propagation" via unfounded sets.[17] Notably, virtually all ASP solvers exploit strongly connected components of positive dependency graphs to limit work to necessary parts. Unlike the unfounded set checking procedures of *dlv* [17] and *smodels* [119] computing greatest unfounded sets, the ones of *clasp* and *nomore++* [2] aim at small unfounded sets and return them as soon as they are identified. The main motive for this is to reduce overlaps between (unit) propagation and unfounded set checking. Another difference between unfounded set checking approaches is that *dlv* and *nomore++* use a flag "must-be-true" to indicate (logically true) atoms whose non-circular derivability is uncertain, for which purpose *smodels* and *clasp* exploit source pointers [119]. The advantage of source pointers is that they need not be updated upon backtracking or backjumping, respectively, while "true" may have to be turned back into "must-be-true". Albeit several approaches [90, 71, 91, 2, 59] admit restricting the consideration of unfounded sets to loops, *clasp* does not guarantee that a detected non-empty unfounded set is a loop (yet it contains one), and it is an interesting open question whether a strict limitation to loops would be advantageous. We also note that the "unidirectional" unfounded set handling in native ASP solvers, not realizing full unit propagation via loop formulas, has already been recognized in [63, 62]. Unfortunately, the approach to remedy this peculiarity suggested in [18, 19] is computationally too complex (quadratic) to be beneficial in practice, and it is open whether the intended effect can be achieved by more economic techniques.

On top of its basic decision procedure, *clasp* supports various extended functionalities (cf. [56]). Particular backtracking schemes, applied after finding a solution, admit the repetition-free enumeration of answer sets [53] as well as projections of them [57] in polynomial space. Optimization strategies wrt one or multiple (lexicographically ordered) objectives are described in [48]. Furthermore, *clasp* offers advanced preprocessing on the level of logic programs [55] and also their induced constraints [33]. Techniques for the intrinsic treatment of extended rules have been presented in [47]; in particular, they include an unfounded set checking procedure extending the one in Algorithm 3. Finally, several systems implement elaborate features on top of *clasp*: disjunctive ASP solver *claspD* [31] internally couples two *clasp* engines, *clingcon* [60] embeds the *gecode* constraint library[18] into *clasp*'s propagation routine to deal with non-Boolean variables, *iclingo* [46] exploits *clasp*'s incremental interface to solve series of problems over increasing horizons [20, 34], and parallel ASP solver *claspar* [37, 118, 50] augments *clasp* with a communication module to enable message passing between distributed solver instances.

8 Summary

We have provided a uniform approach to conflict-driven ASP solving, allowing for a transparent technology transfer from (and to) neighboring areas like SAT. The idea is to view inferences

[17]The approach of *lp2diff* [78] relies on a reduction to difference logic, so that SMT solvers supporting this logic can be used to accomplish unfounded set checks and thus to compute answer sets.

[18]Available at `http://www.gecode.org`.

in ASP solving as unit propagation on nogoods, reflecting constraints from Clark completion, unfounded sets, and conflicts. We have seen that the inclusion of rule bodies in assignments allows for a natural extension of unit propagation to ASP, abolishing the pre-existing need for multiple inference rules.

In contrast to SAT, ASP induces implicit constraints given by loop nogoods. Though inherently present, these nogoods need only be expatiated when they serve as antecedents. This puts sophisticated unfounded set checks on the same logical basis as plain SAT, and we have provided a conflict-driven algorithmic framework for ASP solving, incorporating state-of-the-art SAT solving techniques. Notably, our approach favors unit propagation on explicit nogoods over unfounded set checks, testing for implicit loop nogoods that are unit or violated. In fact, many of the combinatorially constructable loop nogoods (Λ_Π) might be redundant, that is, entailed by Clark completion and/or other loop nogoods. (For tight programs, the whole set Λ_Π of loop nogoods is redundant.) In view of this, our approach makes sure that inspected loop nogoods are "1-empowering" [108] and supplement the available constraints.

We have implemented our approach in the ASP solver *clasp*, which has demonstrated its competitiveness in various settings, for instance, winning first places at the ASP, CASC, MISC, PB, and SAT contests in 2011. The *clasp* system implements state-of-the-art techniques from Boolean constraint solving without deploying or modifying any legacy SAT solver. Rather, *clasp* extends the functionalities of plain SAT solvers by unfounded set checking, intrinsic treatment of cardinality and weight constraints, and optimization. Beyond search for one answer set, *clasp* can enumerate them without falling back on solution recording; such techniques are detailed in [53] and [57]. All in all, *clasp* has become a powerful native ASP solver, whose reasoning modes [56] make it an attractive tool for knowledge representation and reasoning. The *clasp* system constitutes a central component of the *Potassco* tool suite [44] and has already been used in various applications from diverse areas like, e.g., assisted living [99], music composition [13], temporal reasoning [80], general game playing [122], hardware synthesis [75], and systems biology [64].

Acknowledgments

We are grateful to the ASP research and user community as a whole for valuable contributions to the *Potassco* project, and to the anonymous reviewers of this paper for many helpful comments.

This work was supported by the German Science Foundation (DFG) under grants SCHA 550/8-1 and -2.

A Proofs

In Appendix A.1 and A.2, we provide proofs for the formal results presented in Section 3 and 4.5, respectively.

A.1 Nogoods of Logic Programs

We begin with Lemma 3.1, establishing that any solution for Δ_Π is uniquely determined by its literals over atoms.

Lemma 3.1. *Let* Π *be a logic program and* $X \subseteq \mathit{atom}(\Pi)$.
Then, we have that

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in \mathit{atom}(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in \mathit{body}(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in \mathit{body}(\Pi), (\beta^+ \cap (\mathit{atom}(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for $\Delta_{\mathit{body}(\Pi)}$ *such that* $\mathbf{A}^\mathbf{T} \cap \mathit{atom}(\Pi) = X$.

Proof. Consider any $\beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\} \in \mathit{body}(\Pi)$, and recall that $\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\}$ and $\Delta(\beta) = \{\,\{\mathbf{T}\beta, \mathbf{F}p_1\}, \dots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \dots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\,\}$. Then, one of the following cases applies:

$\beta^+ \subseteq X$ and $\beta^- \cap X = \emptyset$**:** We have that $\{\mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\} \subseteq \mathbf{A}$ and $\{\mathbf{F}p_1, \dots, \mathbf{F}p_m, \mathbf{T}p_{m+1}, \dots, \mathbf{T}p_n\} \cap \mathbf{A} = \emptyset$. In view of the latter, $\delta \not\subseteq \mathbf{A}$ for any $\delta \in \Delta(\beta)$. Furthermore, $\mathbf{T}\beta \in \mathbf{A}$ and $\mathbf{F}\beta \notin \mathbf{A}$ make sure that $\delta(\beta) \not\subseteq \mathbf{A}$, where $\delta(\beta) \setminus \mathbf{A} = \{\mathbf{F}\beta\}$.

$(\beta^+ \cap (\mathit{atom}(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset$**:** We have that $\{\mathbf{F}p_1, \dots, \mathbf{F}p_m, \mathbf{T}p_{m+1}, \dots, \mathbf{T}p_n\} \cap \mathbf{A} \neq \emptyset$ and $\{\mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\} \not\subseteq \mathbf{A}$. The latter yields that $\delta(\beta) \not\subseteq \mathbf{A}$. Furthermore, $\mathbf{F}\beta \in \mathbf{A}$ and $\mathbf{T}\beta \notin \mathbf{A}$ make sure that $\delta \not\subseteq \mathbf{A}$ for every $\delta \in \Delta(\beta)$, where $\delta \setminus \mathbf{A} = \{\mathbf{T}\beta\}$ for some $\delta \in \Delta(\beta)$.

The above cases show that, for every $\beta \in \mathit{body}(\Pi)$, no nogood from $\{\delta(\beta)\} \cup \Delta(\beta)$ is contained in $\mathbf{A}$, so that $\mathbf{A}$ is a solution for $\Delta_{\mathit{body}(\Pi)} = \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\}$. On the other hand, for each $\beta \in \mathit{body}(\Pi)$, either $\delta(\beta) \setminus \mathbf{A} = \{\mathbf{F}\beta\}$ or $\delta \setminus \mathbf{A} = \{\mathbf{T}\beta\}$ for some $\delta \in \Delta(\beta)$. Hence, there is no solution $\mathbf{B} \neq \mathbf{A}$ for $\Delta_{\mathit{body}(\Pi)}$ such that $\mathbf{B}^\mathbf{T} \cap \mathit{atom}(\Pi) = X$. □

Theorem 3.3 on the correspondence between answer sets of a tight program Π and solutions for Δ_Π can be derived from Lemma 3.2, establishing one-to-one correspondence between supported models of Π and solutions for Δ_Π. Both results are demonstrated next.

Lemma 3.2. *Let* Π *be a logic program and* $X \subseteq \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$.
Then, we have that $(X \cap \mathit{atom}(\Pi)) \cup \{p_\beta \mid \beta \in X \cap \mathit{body}(\Pi)\}$ *is a supported model of* Π *iff* $\{\mathbf{T}v \mid v \in X\} \cup \{\mathbf{F}v \mid v \in (\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)) \setminus X\}$ *is a solution for* Δ_Π.

Proof. Let $M = (X \cap \mathit{atom}(\Pi)) \cup \{p_\beta \mid \beta \in X \cap \mathit{body}(\Pi)\}$ and $\mathbf{A} = \{\mathbf{T}v \mid v \in X\} \cup \{\mathbf{F}v \mid v \in (\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)) \setminus X\}$. Then, for any $p \in \mathit{atom}(\Pi)$ (or $\beta \in \mathit{body}(\Pi)$), we have that $p \in M$ (or $p_\beta \in M$) iff $\mathbf{T}p \in \mathbf{A}$ (or $\mathbf{T}\beta \in \mathbf{A}$), and $p \notin M$ (or $p_\beta \notin M$) iff $\mathbf{F}p \in \mathbf{A}$ (or $\mathbf{F}\beta \in \mathbf{A}$).

Considering any $\beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\} \in \mathit{body}(\Pi)$, it is not difficult to check that $\delta \subseteq \mathbf{A}$ for some $\delta \in \{\delta(\beta)\} \cup \Delta(\beta) = \{\,\{\mathbf{F}\beta, \mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\},$

$\{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\}$ iff $M \not\models (p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n)$.

Likewise, for any $p \in \mathit{atom}(\Pi)$ with $\mathit{body}_\Pi(p) = \{\beta_1, \ldots, \beta_k\}$, $\delta \subseteq \mathbf{A}$ for some $\delta \in \{\delta(p)\} \cup \Delta(p) = \{\{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}, \{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\}$ iff $M \not\models (p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k})$.

This shows that M is a supported model of Π iff $\mathbf{A}$ is a solution for Δ_Π. □

Theorem 3.3. *Let Π be a tight logic program and $X \subseteq \mathit{atom}(\Pi)$.*
Then, we have that X is an answer set of Π iff

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in \mathit{atom}(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in \mathit{body}(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in \mathit{body}(\Pi), (\beta^+ \cap (\mathit{atom}(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for Δ_Π such that $\mathbf{A^T} \cap \mathit{atom}(\Pi) = X$.

Proof. By Lemma 3.1, there is a subset of Δ_Π for which $\mathbf{A}$ is the unique solution such that $\mathbf{A^T} \cap \mathit{atom}(\Pi) = X$. Along with Lemma 3.2, we conclude that $M \cap \mathit{atom}(\Pi) = X$ for some supported model M of Π iff $\mathbf{A}$ is the unique solution for Δ_Π such that $\mathbf{A^T} \cap \mathit{atom}(\Pi) = X$. Finally, by Theorem 3.2 in [40], showing that supported models and answer sets of Π coincide if Π is tight,[19] we conclude that X is an answer set of Π iff $\mathbf{A}$ is the unique solution for Δ_Π such that $\mathbf{A^T} \cap \mathit{atom}(\Pi) = X$. □

In order to extend Theorem 3.3 to non-tight programs, we provide some properties of unfounded sets. To begin with, GRS-unfounded sets are linked to unfounded sets by the fact that, wrt a body-saturated assignment, every GRS-unfounded set is an unfounded set as well.

Proposition 3.4. *Let Π be a logic program, $\mathbf{A}$ an assignment, and $U \subseteq \mathit{atom}(\Pi)$.*
If $\mathbf{A}$ is body-saturated for Π, then we have that U is an unfounded set of Π wrt $\mathbf{A}$ if U is a GRS-unfounded set of Π wrt $\mathbf{A}$.

Proof. Assume that $\mathbf{A}$ is body-saturated for Π. Then, for every $\beta \in \mathit{body}(\Pi)$, $(\beta^+ \cap \mathbf{A^F}) \cup (\beta^- \cap \mathbf{A^T}) \neq \emptyset$ implies $\beta \in \mathbf{A^F}$. Hence, if U is a GRS-unfounded set of Π wrt $\mathbf{A}$, then $\mathit{EB}_\Pi(U) \subseteq \{\beta \in \mathit{body}(\Pi) \mid (\beta^+ \cap \mathbf{A^F}) \cup (\beta^- \cap \mathbf{A^T}) \neq \emptyset\}$ implies $\mathit{EB}_\Pi(U) \subseteq \mathbf{A^F}$, so that U is an unfounded set of Π wrt $\mathbf{A}$. □

Further considering the relationships between unfounded set concepts, unfounded sets and GRS-unfounded sets coincide wrt body-synchronized assignments.

Proposition 3.5. *Let Π be a logic program, $\mathbf{A}$ an assignment, and $U \subseteq \mathit{atom}(\Pi)$.*
If $\mathbf{A}$ is body-synchronized for Π, then we have that U is an unfounded set of Π wrt $\mathbf{A}$ iff U is a GRS-unfounded set of Π wrt $\mathbf{A}$.

[19]Note that the equivalences in (3) define auxiliary propositions for bodies in terms of atoms. Hence, our representation is a conservative extension of the completion of Π, originally described without propositions for bodies [21].

Proof. Assume that $\mathbf{A}$ is body-synchronized for Π. Then, $\mathbf{A}$ is body-saturated for Π according to Definition 3.3, and by Proposition 3.4, U is an unfounded set of Π wrt $\mathbf{A}$ if U is a GRS-unfounded set of Π wrt $\mathbf{A}$. It remains to show that the converse holds as well. Since $\mathbf{A}$ is body-synchronized for Π, for every $\beta \in \mathit{body}(\Pi)$, $\beta \in \mathbf{A}^{\mathbf{F}}$ implies $(\beta^+ \cap \mathbf{A}^{\mathbf{F}}) \cup (\beta^- \cap \mathbf{A}^{\mathbf{T}}) \neq \emptyset$. Hence, if U is an unfounded set of Π wrt $\mathbf{A}$, then $\mathit{EB}_\Pi(U) \subseteq \mathbf{A}^{\mathbf{F}}$ implies $\mathit{EB}_\Pi(U) \subseteq \{\beta \in \mathit{body}(\Pi) \mid (\beta^+ \cap \mathbf{A}^{\mathbf{F}}) \cup (\beta^- \cap \mathbf{A}^{\mathbf{T}}) \neq \emptyset\}$, so that U is a GRS-unfounded set of Π wrt $\mathbf{A}$. □

The following characterization of solutions for Λ_Π provides an analogy to *unfounded-free* interpretations [84], as identified wrt GRS-unfounded sets, in terms of our unfounded set notion.

Proposition 3.6. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment such that* $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$.

Then, we have that $\mathbf{A}$ *is a solution for* Λ_Π *iff* $U \subseteq \mathbf{A}^{\mathbf{F}}$ *for every unfounded set* U *of* Π *wrt* $\mathbf{A}$.

Proof. We have that $\mathbf{A}$ is not a solution for Λ_Π iff $\lambda(p, U) \subseteq \mathbf{A}$ for some $\emptyset \subset U \subseteq \mathit{atom}(\Pi)$ and $p \in U$ iff $p \in U \cap \mathbf{A}^{\mathbf{T}}$ and $\mathit{EB}_\Pi(U) \subseteq \mathbf{A}^{\mathbf{F}}$ for some $U \subseteq \mathit{atom}(\Pi)$ iff $U \not\subseteq \mathbf{A}^{\mathbf{F}}$ for some unfounded set U of Π wrt $\mathbf{A}$. □

We are now ready to extend Theorem 3.3 to non-tight programs.

Theorem 3.7. *Let* Π *be a logic program and* $X \subseteq \mathit{atom}(\Pi)$.

Then, we have that X *is an answer set of* Π *iff*

$$\begin{array}{rcl}\mathbf{A} & = & \{\mathbf{T}p \mid p \in X\} \cup \{\mathbf{F}p \mid p \in \mathit{atom}(\Pi) \setminus X\} \\ & \cup & \{\mathbf{T}\beta \mid \beta \in \mathit{body}(\Pi), \beta^+ \subseteq X, \beta^- \cap X = \emptyset\} \\ & \cup & \{\mathbf{F}\beta \mid \beta \in \mathit{body}(\Pi), (\beta^+ \cap (\mathit{atom}(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\}\end{array}$$

is the unique solution for $\Delta_\Pi \cup \Lambda_\Pi$ *such that* $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$.

Proof. By Lemma 3.1, there is a subset of Δ_Π for which $\mathbf{A}$ is the unique solution such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$. Along with Lemma 3.2, we conclude that $M \cap \mathit{atom}(\Pi) = X$ for some supported model M of Π iff $\mathbf{A}$ is the unique solution for Δ_Π such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$. For any supported model M of Π, it is clear that $M \cap \mathit{atom}(\Pi)$ is a model of Π, that is, $\mathit{head}(r) \in M$, $\mathit{body}(r)^+ \not\subseteq M$, or $\mathit{body}(r)^- \cap M \neq \emptyset$ holds for every $r \in \Pi$. Hence, if $\mathbf{A}$ is the unique solution for Δ_Π such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$, then $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi)$ is a model of Π. In addition, we have that $\mathbf{A}$ is body-synchronized for Π according to Definition 3.3 because $\mathbf{A}^{\mathbf{F}} \cap \mathit{body}(\Pi) = \{\beta \in \mathit{body}(\Pi) \mid (\beta^+ \cap (\mathit{atom}(\Pi) \setminus X)) \cup (\beta^- \cap X) \neq \emptyset\} = \{\beta \in \mathit{body}(\Pi) \mid (\beta^+ \cap \mathbf{A}^{\mathbf{F}}) \cup (\beta^- \cap \mathbf{A}^{\mathbf{T}}) \neq \emptyset\}$. We use these properties to show the implications of the statement:

- $\Rightarrow$: Assume that X is an answer set of Π. Then, by Corollary 1 in [93], we have that $M \cap \mathit{atom}(\Pi) = X$ for some supported model M of Π. (See Footnote 19 for remarks on the role of auxiliary propositions for bodies in the Clark completion of Π.) That is, $\mathbf{A}$ is the unique solution for Δ_Π such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$. Furthermore, by Theorem 4.6 in [84], we have that $U \cap X = \emptyset$ holds for every GRS-unfounded set U of Π wrt X. Since $\mathbf{A}$ is body-synchronized for Π, by Proposition 3.5, we conclude that $U \cap \mathbf{A}^{\mathbf{T}} = \emptyset$ and $U \subseteq \mathbf{A}^{\mathbf{F}}$ hold for every unfounded set U of Π wrt $\mathbf{A}$. Hence, by Proposition 3.6, $\mathbf{A}$ is a solution for Λ_Π and the unique solution for $\Delta_\Pi \cup \Lambda_\Pi$ such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(\Pi) = X$.

$\Leftarrow$: Assume that $\mathbf{A}$ is the unique solution for $\Delta_\Pi \cup \Lambda_\Pi$ such that $\mathbf{A^T} \cap atom(\Pi) = X$. Then, by Proposition 3.6, we have that $U \cap \mathbf{A^T} = \emptyset$ holds for every unfounded set U of Π wrt $\mathbf{A}$. Since $\mathbf{A}$ is body-synchronized for Π, Proposition 3.5 yields that $U \cap X = \emptyset$ holds for every GRS-unfounded set U of Π wrt X. Along with the fact that $\mathbf{A^T} \cap atom(\Pi) = X$ is a model of Π, by Theorem 4.6 in [84], we conclude that X is an answer set of Π.

We have thus shown that both implications of the statement hold. $\square$

In what follows, we show further crucial properties of unfounded sets. To begin with, false atoms of an unfounded set may be removed, while still maintaining unfoundedness when the assignment at hand is body-saturated.

Proposition 3.8. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* U *an unfounded set of* Π *wrt* $\mathbf{A}$.

If $\mathbf{A}$ *is body-saturated for* Π*, then we have that* $U \setminus \mathbf{A^F}$ *is an unfounded set of* Π *wrt* $\mathbf{A}$.

Proof. Assume that $\mathbf{A}$ is body-saturated for Π. Then, for every $\beta \in EB_\Pi(U \setminus \mathbf{A^F}) \setminus EB_\Pi(U)$, the fact that $\beta^+ \cap (U \cap \mathbf{A^F}) \neq \emptyset$ implies $\beta \in \mathbf{A^F}$. Along with $EB_\Pi(U) \subseteq \mathbf{A^F}$, we conclude that $EB_\Pi(U \setminus \mathbf{A^F}) \subseteq \mathbf{A^F}$, so that $U \setminus \mathbf{A^F}$ is an unfounded set of Π wrt $\mathbf{A}$. $\square$

The following auxiliary result shows that, wrt an atom-saturated assignment, any non-empty unfounded set of non-false atoms that is not a loop contains in turn a non-empty proper subset that is unfounded.

Lemma A.1. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ *a non-empty unfounded set of* Π *wrt* $\mathbf{A}$.

If $\mathbf{A}$ *is atom-saturated for* Π *and if* $U \notin loop(\Pi)$*, then there is some non-empty unfounded set* $U' \subset U$ *of* Π *wrt* $\mathbf{A}$.

Proof. Assume that $\mathbf{A}$ is atom-saturated for Π and that $U \notin loop(\Pi)$. Then, for every $p \in U$, the prerequisite that $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ implies $body_\Pi(p) \not\subseteq \mathbf{A^F}$, while $EB_\Pi(U) \subseteq \mathbf{A^F}$ yields $body_\Pi(p) \cap EB_\Pi(U) \subseteq \mathbf{A^F}$. That is, for every $p \in U$, there is some $\beta \in body_\Pi(p) \setminus \mathbf{A^F}$, and $\beta^+ \cap U \neq \emptyset$ holds for each $\beta \in body_\Pi(p) \setminus \mathbf{A^F}$. Hence, every atom of U has some predecessor belonging to U in $(atom(\Pi), \leq^+)$. However, since $U \notin loop(\Pi)$, we have that the subgraph of $(atom(\Pi), \leq^+)$ induced by U is not strongly connected. Along with the fact that U is finite, we conclude that there is some strongly connected component of $(U, \{(p, head(r)) \mid r \in \Pi, head(r) \in U, p \in body(r)^+ \cap U\})$ such that its vertices C do not reach atoms in $U \setminus C$.[20] The latter means that $\beta^+ \cap C = \emptyset$ holds for every $\beta \in EB_\Pi(U \setminus C)$, so that $EB_\Pi(U \setminus C) \subseteq EB_\Pi(U)$. Since $\emptyset \subset C \subset U$ and $EB_\Pi(U) \subseteq \mathbf{A^F}$, this shows that $U' = U \setminus C$ is a non-empty unfounded set of Π wrt $\mathbf{A}$ such that $U' \subset U$. $\square$

The previous lemma allows us to conclude that, wrt an atom-saturated assignment, every non-empty unfounded set of non-false atoms must contain an unfounded loop.

[20]Note that the "condensation" of $(U, \{(p, head(r)) \mid r \in \Pi, head(r) \in U, p \in body(r)^+ \cap U\})$, obtained by contracting each strongly connected component to a single vertex, is a directed acyclic graph (cf. [111]).

Proposition 3.9. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ *a non-empty unfounded set of* Π *wrt* $\mathbf{A}$*.*

If $\mathbf{A}$ *is atom-saturated for* Π*, then there is some unfounded set* $L \subseteq U$ *of* Π *wrt* $\mathbf{A}$ *such that* $L \in \mathit{loop}(\Pi)$*.*

Proof. Assume that $\mathbf{A}$ is atom-saturated for Π. Then, since $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ is a non-empty unfounded set of Π wrt $\mathbf{A}$, there is some non-empty unfounded set $L \subseteq U$ of Π wrt $\mathbf{A}$ such that $\emptyset$ and L are all unfounded sets of Π wrt $\mathbf{A}$ contained in L. For each such $L \subseteq U$, by Lemma A.1, we conclude that $L \in \mathit{loop}(\Pi)$. □

Corollary 3.10. *Let* Π *be a logic program,* $\mathbf{A}$ *an assignment, and* $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ *a non-empty unfounded set of* Π *wrt* $\mathbf{A}$*.*

If $\mathbf{A}$ *is atom-saturated for* Π*, then there is some non-empty unfounded set* $U' \subseteq U$ *of* Π *wrt* $\mathbf{A}$ *such that all* $p \in U'$ *belong to the same non-trivial strongly connected component of* $(atom(\Pi), \leq^+)$*.*

Proof. This result follows immediately from Proposition 3.9, since all atoms of some $L \in \mathit{loop}(\Pi)$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$, which must be non-trivial by the definition of a loop. □

Finally, we combine Proposition 3.8 and 3.9 to show that, wrt an assignment that is both atom- and body-saturated, any unfounded set that includes non-false atoms must contain an unfounded loop of non-false atoms.

Theorem 3.11. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment.*

If $\mathbf{A}$ *is both atom- and body-saturated for* Π *and if there is some unfounded set* U *of* Π *wrt* $\mathbf{A}$ *such that* $U \not\subseteq \mathbf{A^F}$*, then there is some unfounded set* $L \subseteq U \setminus \mathbf{A^F}$ *of* Π *wrt* $\mathbf{A}$ *such that* $L \in \mathit{loop}(\Pi)$*.*

Proof. Assume that $\mathbf{A}$ is both atom- and body-saturated for Π and that there is some unfounded set U of Π wrt $\mathbf{A}$ such that $U \not\subseteq \mathbf{A^F}$. Then, by Proposition 3.8, we have that $U \setminus \mathbf{A^F}$ is a non-empty unfounded set of Π wrt $\mathbf{A}$. Furthermore, by Proposition 3.9, there is some unfounded set $L \subseteq U \setminus \mathbf{A^F}$ of Π wrt $\mathbf{A}$ such that $L \in \mathit{loop}(\Pi)$. □

Corollary 3.12. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment.*

If $\mathbf{A}$ *is both atom- and body-saturated for* Π *and if there is some unfounded set* U *of* Π *wrt* $\mathbf{A}$ *such that* $U \not\subseteq \mathbf{A^F}$*, then there is some non-empty unfounded set* $U' \subseteq U \setminus \mathbf{A^F}$ *of* Π *wrt* $\mathbf{A}$ *such that all* $p \in U'$ *belong to the same non-trivial strongly connected component of* $(atom(\Pi), \leq^+)$*.*

Proof. This result follows immediately from Theorem 3.11, since all atoms of some $L \in \mathit{loop}(\Pi)$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$, which must be non-trivial by the definition of a loop. □

A.2 Soundness and Completeness of CDNL-ASP Algorithm

We begin with showing fundamental properties of UNFOUNDEDSET in Algorithm 3, where Lemma 4.1 and 4.2 establish invariants that are crucial for its soundness and completeness, stated in Theorem 4.3.

Lemma 4.1. *Let* Π *be a logic program and* $\mathbf{A}$ *an assignment that is body-saturated for* Π.

If UNFOUNDEDSET$(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that the source pointer configuration remains valid throughout the execution of* UNFOUNDEDSET$(\Pi, \mathbf{A})$.

Proof. Assume that UNFOUNDEDSET$(\Pi, \mathbf{A})$ is invoked on a valid source pointer configuration. Then, an invalid source pointer configuration could in principle be obtained only in Line 13 of Algorithm 3, where $\mathit{source}(q)$ is set for some (cyclic) $q \in \mathit{atom}(\Pi)$. However, by induction on executions of Line 13, we show that the source pointer configuration remains valid:

Base case**:** Since the given source pointer configuration is valid and $\mathbf{A}$ is body-saturated for Π, after finishing the loop in Line 2–5 of Algorithm 3, we have that $\mathit{source}(p) \in \mathit{body}_\Pi(p) \setminus \mathbf{A^F}$ and $\mathit{source}(p)^+ \cap (\mathbf{A^F} \cup (\mathit{scc}(p) \cap S)) = \emptyset$ hold for every cyclic $p \in \mathit{atom}(\Pi) \setminus (\mathbf{A^F} \cup S)$. For the atoms C of any non-trivial strongly connected component of $(\mathit{atom}(\Pi), \leq^+)$, this implies that $\bigcup_{p \in C \setminus (\mathbf{A^F} \cup S)} (\mathit{source}(p)^+ \cap C) \subseteq C \setminus (\mathbf{A^F} \cup S)$. In words, the source pointers of atoms in C that are neither false in $\mathbf{A}$ nor in scope S do not contain any atom of C that is false in $\mathbf{A}$ or in scope S.

Induction step**:** Let $q \in U$ be any cyclic atom such that the condition in Line 12 of Algorithm 3 applies to q, and let $C = \mathit{scc}(q)$. Then, in view of the choice of some $p \in S$ in Line 6 along with Line 7 and 14–16, manipulating the contents of U and S, respectively, we have that $U \subseteq C \cap S$, which yields that $q \in C \cap S$. Furthermore, assume that the source pointer configuration is valid and that $\bigcup_{p \in C \setminus (\mathbf{A^F} \cup S)} (\mathit{source}(p)^+ \cap C) \subseteq C \setminus (\mathbf{A^F} \cup S)$ holds before setting $\mathit{source}(q)$ to some $\beta \in \mathit{body}_\Pi(q)$ in Line 13. In terms of the subgraph of $(\mathit{atom}(\Pi), \leq^+)$ containing every cyclic $p \in \mathit{atom}(\Pi)$ along with edges (p', p) for all $p' \in \mathit{source}(p)^+ \cap \mathit{scc}(p)$, $\bigcup_{p \in C \setminus (\mathbf{A^F} \cup S)} (\mathit{source}(p)^+ \cap C) \subseteq C \setminus (\mathbf{A^F} \cup S)$ means that it does not contain any edge from an atom in $C \cap (\mathbf{A^F} \cup S)$ to an atom in $C \setminus (\mathbf{A^F} \cup S)$. For β, since $\mathbf{A}$ is body-saturated for Π, the condition $\beta \in \mathit{EB}_\Pi(U) \setminus \mathbf{A^F}$ in Line 10 makes sure that $\beta^+ \cap \mathbf{A^F} = \emptyset$, and $\beta^+ \cap (C \cap S) = \emptyset$ is verified in Line 11. Hence, we have that $\beta^+ \cap C \subseteq C \setminus (\mathbf{A^F} \cup S)$, so that, for all edges (p, q) from atoms $p \in \beta^+ \cap C$ to q, it holds that $p \in C \setminus (\mathbf{A^F} \cup S)$. As we have seen above that $q \in C \cap S$ does not reach atoms in $C \setminus (\mathbf{A^F} \cup S)$, we conclude that the subgraph of $(\mathit{atom}(\Pi), \leq^+)$ containing every cyclic $p \in \mathit{atom}(\Pi)$ along with edges (p', p) for all $p' \in \mathit{source}(p)^+ \cap \mathit{scc}(p)$ remains acyclic after setting $\mathit{source}(q)$ to β in Line 13. This shows that the source pointer configuration obtained by executing Line 13 is in turn valid. Finally, we have that the induction hypothesis still holds for $S \setminus \{q\}$ constructed in Line 15, that is, $\bigcup_{p \in C \setminus (\mathbf{A^F} \cup (S \setminus \{q\}))} (\mathit{source}(p)^+ \cap C) = \left(\bigcup_{p \in C \setminus (\mathbf{A^F} \cup S)} (\mathit{source}(p)^+ \cap C)\right) \cup (\beta^+ \cap C) \subseteq C \setminus (\mathbf{A^F} \cup S) \subseteq C \setminus (\mathbf{A^F} \cup (S \setminus \{q\}))$.

We have thus shown that a valid source pointer configuration cannot be invalidated when invoking UNFOUNDEDSET$(\Pi, \mathbf{A})$ with an assignment $\mathbf{A}$ that is body-saturated for Π. □

Lemma 4.2. *Let Π be a logic program and $\mathbf{A}$ an assignment that is atom-saturated for Π.*

If $\textsc{UnfoundedSet}(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that every unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ of Π wrt $\mathbf{A}$ such that all $p \in U$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$ is contained in S whenever Line 6 of Algorithm 3 is entered.*

Proof. Assume that $\textsc{UnfoundedSet}(\Pi, \mathbf{A})$ is invoked on a valid source pointer configuration. Then, let $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ be any unfounded set of Π wrt $\mathbf{A}$ such that all $p \in U$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$. Since $U \cap \mathbf{A^F} = \emptyset$ and $\mathbf{A}$ is atom-saturated for Π, we have that $body_\Pi(p) \not\subseteq \mathbf{A^F}$ for every $p \in U$, while $EB_\Pi(U) \subseteq \mathbf{A^F}$ implies that $\beta^+ \cap U \neq \emptyset$ holds for each $\beta \in body_\Pi(p) \setminus \mathbf{A^F}$. That is, all $p \in U$ are cyclic, and $source(p) \in body_\Pi(p) \cup \{\bot\}$ holds because the given source pointer configuration is valid. By induction on executions of the test in Line 6 of Algorithm 3, we show that $U \not\subseteq S$ is impossible whenever Line 6 is entered:

Base case: For the sake of contradiction, assume that $U \not\subseteq S$ after finishing the loop in Line 2–5 of Algorithm 3. Then, in view of Line 1, for each $p \in U \setminus S$, we have that $source(p) \notin \mathbf{A^F} \cup \{\bot\}$, which further implies that $source(p) \in body_\Pi(p) \setminus \mathbf{A^F}$ and $source(p)^+ \cap U \neq \emptyset$. Moreover, the condition $source(p)^+ \cap (scc(p) \cap S) \neq \emptyset$ in Line 3 does not apply to $source(p)$, which yields that $source(p)^+ \cap (U \cap S) = \emptyset$ and $source(p)^+ \cap (U \setminus S) \neq \emptyset$. Since $U \setminus S$ is finite and each atom of $U \setminus S$ has some predecessor belonging to $U \setminus S$ in the subgraph of $(atom(\Pi), \leq^+)$ containing every cyclic $p \in atom(\Pi)$ along with edges (q, p) for all $q \in source(p)^+ \cap scc(p)$, we conclude that this subgraph cannot be acyclic, which is a contradiction to the assumption that $\textsc{UnfoundedSet}(\Pi, \mathbf{A})$ is invoked on a valid source pointer configuration.

Induction step: For the sake of contradiction, assume that $U \subseteq S$ at the beginning of an iteration of the loop in Line 6–17 of Algorithm 3, but $U \not\subseteq S$ when Line 6 is re-entered after finishing the iteration. In this iteration, the elements of $U \setminus S$ must have (successively) been removed from S in Line 15. In particular, some $q \in U \setminus S$ has been removed from S before any other atom of U. To achieve this, the condition in Line 11 must have applied to some $\beta \in body_\Pi(q) \setminus \mathbf{A^F}$, which yields that $\beta^+ \cap (scc(q) \cap U') = \emptyset$ for some superset U' of U. Since $U \subseteq scc(q)$, this implies that $\beta^+ \cap U = \emptyset$, which is a contradiction to the assumption that U is an unfounded set of Π wrt $\mathbf{A}$.

We have thus shown that, if $\textsc{UnfoundedSet}(\Pi, \mathbf{A})$ is invoked on a valid source pointer configuration with an assignment $\mathbf{A}$ that is atom-saturated for Π, every unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{A^F}$ of Π wrt $\mathbf{A}$ such that all $p \in U$ belong to the same strongly connected component of $(atom(\Pi), \leq^+)$ must be contained in S whenever Line 6 of Algorithm 3 is entered. If any such unfounded set U is non-empty, this invariant excludes the termination of Algorithm 3 by returning $\emptyset$ in Line 18. □

Theorem 4.3. *Let Π be a logic program and $\mathbf{A}$ an assignment that is both atom- and body-saturated for Π.*

If UNFOUNDEDSET$(\Pi, \mathbf{A})$ *is invoked on a valid source pointer configuration, then we have that* UNFOUNDEDSET$(\Pi, \mathbf{A})$ *returns an unfounded set* $U \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$ *of* Π *wrt* $\mathbf{A}$*, where* $U = \emptyset$ *iff there is no unfounded set* U' *of* Π *wrt* $\mathbf{A}$ *such that* $U' \not\subseteq \mathbf{A}^{\mathbf{F}}$.

Proof. Assume that UNFOUNDEDSET$(\Pi, \mathbf{A})$ is invoked on a valid source pointer configuration. Then, in view of the condition $EB_{\Pi}(U) \subseteq \mathbf{A}^{\mathbf{F}}$ in Line 9 of Algorithm 3 and the fact that $\emptyset$, which can be returned in Line 18, is a (trivial) unfounded set of Π wrt $\mathbf{A}$, we have that UNFOUNDEDSET$(\Pi, \mathbf{A})$ can only return an unfounded set of Π wrt $\mathbf{A}$. By Corollary 3.12, the existence of some non-empty unfounded set U' of Π wrt $\mathbf{A}$ such that $U' \not\subseteq \mathbf{A}^{\mathbf{F}}$ implies that there is a non-empty unfounded set $U \subseteq U' \setminus \mathbf{A}^{\mathbf{F}}$ such that all $p \in U$ belong to the same strongly connected component of $(atom(\Pi), \leq^{+})$. Furthermore, by Lemma 4.2, any such unfounded set U of Π wrt $\mathbf{A}$ is contained in scope S whenever Line 6 is entered. This shows that UNFOUNDEDSET$(\Pi, \mathbf{A})$ cannot return $\emptyset$ in Line 18 if there is some non-empty unfounded set U' of Π wrt $\mathbf{A}$ such that $U' \not\subseteq \mathbf{A}^{\mathbf{F}}$.

It only remains to show that UNFOUNDEDSET$(\Pi, \mathbf{A})$ terminates. To this end, note that scope S is increasing over iterations of the loop in Line 2–5 of Algorithm 3, and strictly decreasing over iterations of the loop in Line 6–17. For U handled in the loop in Line 8–17, we observe that it is strictly increasing when U is extended in Line 16, and strictly decreasing when an element q is removed from U in Line 14, where q cannot be added back later on because it is also removed from S in Line 15. Since $atom(\Pi)$ is finite and $U \subseteq S \subseteq atom(\Pi) \setminus \mathbf{A}^{\mathbf{F}}$, we conclude that none of the loops in Algorithm 3 can be iterated infinitely. Rather, any atom can be added to and removed from S and U, respectively, at most once, which yields that the time complexity of UNFOUNDEDSET$(\Pi, \mathbf{A})$ is linear in the size of Π. □

Next, we show properties of NOGOODPROPAGATION in Algorithm 2. Lemma 4.4 essentially establishes the applicability of Theorem 4.3 whenever an unfounded set check is initiated in Line 12 of Algorithm 2, and Lemma 4.5 provides properties crucial for the soundness and completeness of conflict-driven ASP solving.

Lemma 4.4. *Let* Π *be a logic program,* ∇' *a set of nogoods,* $dl \in \mathbb{N}$*, and* $\mathbf{A}'$ *an assignment.*

Then, we have that $\mathbf{A}$ *is both atom- and body-saturated for* Π *whenever Line 10 of Algorithm 2 is entered in an execution of* NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$.

Proof. For the sake of contradiction, assume that Line 10 of Algorithm 2 is entered in an execution of NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$, while the current assignment $\mathbf{A}$ is not atom-saturated or not body-saturated for Π. Then, some of the following cases applies:

$body_{\Pi}(p) \subseteq \mathbf{A}^{\mathbf{F}}$ **but** $\mathbf{F}p \notin \mathbf{A}$ **for some** $p \in atom(\Pi)$**:** The nogood $\delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\}$, where $body_{\Pi}(p) = \{\beta_1, \dots, \beta_k\}$, is such that $\delta(p) \setminus \mathbf{A} \subseteq \{\mathbf{T}p\}$. In view of the condition in Line 4 of Algorithm 2, tested in the previous iteration of the loop in Line 3–9, we have that $\mathbf{T}p \notin \mathbf{A}$ and $\delta(p) \setminus \mathbf{A} = \{\mathbf{T}p\}$. But this implies that $\mathbf{F}p$ is unit-resulting for $\delta(p)$ wrt $\mathbf{A}$, so that the condition $\Sigma = \emptyset$ cannot hold in Line 9, which contradicts that Line 10 is entered with $\mathbf{A}$ being the current assignment.

$(\beta^+ \cap \mathbf{A}^\mathbf{F}) \cup (\beta^- \cap \mathbf{A}^\mathbf{T}) \neq \emptyset$ but $\mathbf{F}\beta \notin \mathbf{A}$ for some $\beta \in \mathit{body}(\Pi)$: Some nogood $\delta = \{\mathbf{T}\beta, \rho\} \in \Delta(\beta)$ is such that $\delta \setminus \mathbf{A} \subseteq \{\mathbf{T}\beta\}$. In view of the condition in Line 4 of Algorithm 2, tested in the previous iteration of the loop in Line 3–9, we have that $\mathbf{T}\beta \notin \mathbf{A}$ and $\delta \setminus \mathbf{A} = \{\mathbf{T}\beta\}$. But this implies that $\mathbf{F}\beta$ is unit-resulting for δ wrt $\mathbf{A}$, so that the condition $\Sigma = \emptyset$ cannot hold in Line 9, which contradicts that Line 10 is entered with $\mathbf{A}$ being the current assignment.

Since each of the above cases yields a contradiction, we conclude that $\mathbf{A}$ is both atom- and body-saturated for Π whenever Line 10 of Algorithm 2 is entered. □

Lemma 4.5. *Let Π be a logic program, ∇' a set of nogoods, $dl \in \mathbb{N}$, and $\mathbf{A}'$ an assignment.*

If NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ *is invoked on a valid source pointer configuration, then we have that* NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ *returns a pair $(\mathbf{A}, \nabla)$ such that*

1. *$\nabla' \subseteq \nabla \subseteq \nabla' \cup \Lambda_\Pi$;*
2. *$\mathbf{A}$ is an assignment such that $\mathbf{A}' \subseteq \mathbf{A}$ and every $\sigma \in \mathbf{A} \setminus \mathbf{A}'$ is implied by $\Delta_\Pi \cup \nabla$ wrt $\mathbf{A}$;*
3. *$\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_\Pi \cup \nabla$ if $\lambda(p, U) \subseteq \mathbf{A}$ for some $\lambda(p, U) \in \Lambda_\Pi$.*

Proof. Assume that NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ is invoked on a valid source pointer configuration. Then, we begin with showing that the items of the statement hold if NOGOOD-PROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ returns a pair $(\mathbf{A}, \nabla)$:

1. Since ∇' can be augmented only with elements of Λ_Π in Line 14 of Algorithm 2, we have that $\nabla' \subseteq \nabla \subseteq \nabla' \cup \Lambda_\Pi$.
2. In view of Line 5 of Algorithm 2, for each literal σ added to an assignment $\mathbf{B}$ such that $\mathbf{A}' \subseteq \mathbf{B} \subset \mathbf{A}$, we have that $\{\sigma, \overline{\sigma}\} \cap \mathbf{B} = \emptyset$ and that there is an antecedent $\delta \in \Delta_\Pi \cup \nabla$ of σ wrt $\mathbf{B}$, so that σ is implied by $\Delta_\Pi \cup \nabla$ wrt $\mathbf{A}$.
3. For the sake of contradiction, assume that $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta_\Pi \cup \nabla$ and $\lambda(p, U) \subseteq \mathbf{A}$ for some $\lambda(p, U) \in \Lambda_\Pi$. Then, U is an unfounded set of Π wrt $\mathbf{A}$ such that $U \not\subseteq \mathbf{A}^\mathbf{F}$. Furthermore, $(\mathbf{A}, \nabla)$ must be returned in Line 10 or 13 of Algorithm 2, and Lemma 4.4 tells us that $\mathbf{A}$ is both atom- and body-saturated for Π. By Theorem 3.11, we conclude that some $L \in \mathit{loop}(\Pi)$ is unfounded for Π wrt $\mathbf{A}$, so that Π is not tight. Hence, $(\mathbf{A}, \nabla)$ must be returned in Line 13 after obtaining $\emptyset$ as the result of UNFOUNDEDSET$(\Pi, \mathbf{A})$ in Line 12. However, by Lemma 4.4 and 4.1, we conclude that Theorem 4.3 is applicable, which contradicts that $\emptyset$ is obtained as the result of UNFOUNDEDSET$(\Pi, \mathbf{A})$ in Line 12.

It remains to show that NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ terminates. To this end, note that an assignment $\mathbf{B}$ such that $\mathbf{A}' \subseteq \mathbf{B}$ is increasing over iterations of the loop in Line 3–9 of Algorithm 2, as shown in the proof of the second item. Furthermore, by Theorem 4.3 (along with Lemma 4.4 and 4.1), any invocation of UNFOUNDEDSET$(\Pi, \mathbf{B})$ in Line 12 terminates with an unfounded set $U \subseteq atom(\Pi) \setminus \mathbf{B}^\mathbf{F}$ of Π wrt $\mathbf{B}$. Hence, we have that $U = \emptyset$ or $\lambda(p, U) \setminus \mathbf{B} \subseteq \{\mathbf{T}p\}$ for each $p \in U$; by Lemma 4.4 and Proposition 3.8, the same applies to $U \setminus \mathbf{B}^\mathbf{F}$ determined in

Line 11. Thus, any execution of Line 11–12 is followed by the termination of Algorithm 2 in Line 13, or in view of Line 14, by the termination in Line 4 or the addition of a literal $\mathbf{F}p$ (for $p \in \mathit{atom}(\Pi) \setminus (\mathbf{B^T} \cup \mathbf{B^F})$) to $\mathbf{B}$ in Line 8 in the next iteration of the loop in Line 2–14. Since $\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$ is finite, there cannot be infinitely many literals added to $\mathbf{A}'$ over iterations of the loops in Line 2–14 and 3–9, respectively, so that NOGOODPROPAGATION$(dl, \Pi, \nabla', \mathbf{A}')$ terminates by returning a pair $(\mathbf{A}, \nabla)$. □

The following lemma expresses that CONFLICTANALYSIS in Algorithm 4 returns an asserting nogood when given a nogood violated at a decision level greater than 0.

Lemma 4.6. *Let* Π *be a logic program,* ∇ *a set of nogoods,* $\mathbf{A}$ *an assignment such that* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ *and* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\rho) = dl(\sigma)\} \subseteq \{\sigma \in \mathbf{A} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}\}$*, and* $\delta' \subseteq \mathbf{A}$ *such that* $m = \max(\{dl(\sigma) \mid \sigma \in \delta'\} \cup \{0\}) \neq 0$.

Then, we have that CONFLICTANALYSIS$(\delta', \Pi, \nabla, \mathbf{A})$ *returns a pair* (δ, k) *such that*

1. $\delta \subseteq \mathbf{A}$*;*
2. $|\{\sigma \in \delta \mid k < dl(\sigma)\}| = 1$*;*
3. $\delta \not\subseteq \mathbf{B}$ *for any solution* $\mathbf{B}$ *for* $\Delta_\Pi \cup \nabla \cup \{\delta'\}$.

Proof. Given that $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$, every $\varepsilon' \subseteq \mathbf{A}$ such that $\max(\{dl(\rho) \mid \rho \in \varepsilon'\} \cup \{0\}) = m$ contains a literal σ such that $\varepsilon' \setminus \mathbf{A}[\sigma] = \{\sigma\}$ and $dl(\sigma) = m$. (Such literals σ are determined in Line 2 of Algorithm 4.) Then, by induction on iterations of the loop in Line 1–7 of Algorithm 4, we show that the items of the statement hold if CONFLICTANALYSIS$(\delta', \Pi, \nabla, \mathbf{A})$ returns a pair (δ, k):

Base case**:** Let $\delta \subseteq \mathbf{A}$ be some set of literals such that $\max(\{dl(\rho) \mid \rho \in \delta\} \cup \{0\}) = m$ and $\delta \not\subseteq \mathbf{B}$ for any solution $\mathbf{B}$ for $\Delta_\Pi \cup \nabla \cup \{\delta'\}$. For the literal $\sigma \in \delta$ determined in Line 2 of Algorithm 4, if the test in Line 4 yields that $k = \max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) \neq m$, then $|\{\rho \in \delta \mid k < dl(\rho)\}| = |\{\sigma\}| = 1$, and (δ, k) is returned in Line 7.

Induction step**:** Let $\varepsilon' \subseteq \mathbf{A}$ be some set of literals such that $\max(\{dl(\rho) \mid \rho \in \varepsilon'\} \cup \{0\}) = m$ and $\varepsilon' \not\subseteq \mathbf{B}$ for any solution $\mathbf{B}$ for $\Delta_\Pi \cup \nabla \cup \{\delta'\}$. For the literal $\sigma \in \varepsilon'$ determined in Line 2 of Algorithm 4, if the test in Line 4 yields that $\max(\{dl(\rho) \mid \rho \in \varepsilon' \setminus \{\sigma\}\} \cup \{0\}) = m$, there is some $\rho \in \mathbf{A}[\sigma]$ such that $dl(\rho) = dl(\sigma) = m$. Given that $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\rho) = dl(\sigma)\} \subseteq \{\sigma \in \mathbf{A} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}\}$, $\Delta_\Pi \cup \nabla$ contains an antecedent of σ wrt $\mathbf{A}$, and some such ε is selected in Line 5. Since $\varepsilon' \setminus \{\sigma\} \subseteq \mathbf{A}[\sigma] \subseteq \mathbf{A}$ and $\varepsilon \setminus \{\overline{\sigma}\} \subseteq \mathbf{A}[\sigma] \subseteq \mathbf{A}$, for $(\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ constructed in Line 6, it holds that $(\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \subseteq \mathbf{A}[\sigma] \subseteq \mathbf{A}$. Furthermore, $\max(\{dl(\rho) \mid \rho \in \varepsilon' \setminus \{\sigma\}\} \cup \{0\}) = m$ implies that $\max(\{dl(\rho) \mid \rho \in (\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})\} \cup \{0\}) = m$. Finally, since any solution $\mathbf{B}$ for $\Delta_\Pi \cup \nabla \cup \{\delta'\}$ contains either σ or $\overline{\sigma}$, while $\varepsilon' \not\subseteq \mathbf{B}$ and $\varepsilon \not\subseteq \mathbf{B}$, we have that $(\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \not\subseteq \mathbf{B}$.

It remains to show that CONFLICTANALYSIS$(\delta', \Pi, \nabla, \mathbf{A})$ terminates, i.e., that the base case of the induction eventually applies. To this end, note that, in the induction step above, we have that

$(\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) \subseteq \mathbf{A}[\sigma] \subseteq \mathbf{A}$, i.e., all literals $\rho \in (\varepsilon' \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ precede σ in $\mathbf{A}$. From this and the fact $\mathbf{A}$ does not include duplicate literals, we conclude that there cannot be infinitely many applications of the induction step over iterations of the loop in Line 1–7 of Algorithm 4, so that CONFLICTANALYSIS$(\delta', \Pi, \nabla, \mathbf{A})$ terminates by returning a pair (δ, k). □

We now turn to CDNL-ASP in Algorithm 1 for deciding whether a logic program has an answer set, where Lemma 4.7 establishes invariants that are crucial for the main soundness and completeness result stated in Theorem 4.8.

Lemma 4.7. *Let* Π *be a logic program.*

Then, we have that the following holds whenever Line 5 of Algorithm 1 is entered in an execution of CDNL-ASP(Π)*:*

1. ∇ *is a set of nogoods such that* $\delta \not\subseteq \mathbf{B}$ *for every* $\delta \in \nabla$ *and any solution* $\mathbf{B}$ *for* $\Delta_\Pi \cup \Lambda_\Pi$*;*
2. $\mathbf{A}$ *is an assignment such that* $\{\sigma \in \mathbf{A} \mid \rho \in \mathbf{A}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ *and* $\{\sigma \in \mathbf{A} \mid dl(\sigma) \leq \max(\{dl(\rho) \mid \rho \in \mathbf{A}[\sigma]\} \cup \{0\})\} \subseteq \{\sigma \in \mathbf{A} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}\}$*;*
3. $dl \in \mathbb{N}$ *is such that* $\delta \not\subseteq \{\sigma \in \mathbf{A} \mid dl(\sigma) < dl\}$ *for every* $\delta \in \Delta_\Pi \cup \Lambda_\Pi \cup \nabla$.

Proof. By induction on iterations of the loop in Line 4–17 of Algorithm 1, we show that the items of the statement hold whenever Line 5 is entered in an execution of CDNL-ASP(Π):

Base case**:** Before the first iteration, in view of Line 1–3 of Algorithm 1, we have that $\mathbf{A} = \nabla = \emptyset$ and $dl = 0$, for which the items of the statement trivially hold, and also that $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}\} \cup \{0\}) \leq dl$.

Induction step**:** At the beginning of an iteration of the loop in Line 4–17 of Algorithm 1, let ∇', $\mathbf{A}'$, and dl' be such that the items (*1*, *2*, and *3*) of the statement are satisfied wrt them, and assume that (*a*) $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}'\} \cup \{0\}) \leq dl'$.[21] Then, by Lemma 4.5 (along with Lemma 4.1 and 4.4) and in view of Line 7–8 of Algorithm 2, we have that NOGOODPROPAGATION$(dl', \Pi, \nabla', \mathbf{A}')$ invoked in Line 5 returns a pair $(\mathbf{A}, \nabla)$ such that the items of the statement still hold for ∇, $\mathbf{A}$, and dl', respectively. In addition, we have that $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}\} \cup \{0\}) \leq dl'$. Afterwards, one of the following cases applies:

$\varepsilon \subseteq \mathbf{A}$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$**:** If the condition in Line 7 of Algorithm 1 applies, CDNL-ASP(Π) immediately terminates by returning "no answer set". Otherwise, by Lemma 4.6, CONFLICTANALYSIS$(\varepsilon, \Pi, \nabla, \mathbf{A})$ returns a pair (δ, k) such that $\delta \subseteq \mathbf{A}$, $|\{\sigma \in \delta \mid k < dl(\sigma)\}| = 1$, and $\delta \not\subseteq \mathbf{B}$ for any solution $\mathbf{B}$ for $\Delta_\Pi \cup \nabla$. Since any solution $\mathbf{B}$ for $\Delta_\Pi \cup \Lambda_\Pi$ is a solution for $\Delta_\Pi \cup \nabla$ as well, it follows that $\delta \not\subseteq \mathbf{B}$, so that (*1*) $\mathbf{B}$ is a solution for $\Delta_\Pi \cup (\nabla \cup \{\delta\})$, where $\nabla \cup \{\delta\}$ is constructed in Line 9. Furthermore, $\mathbf{A}_k = \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$ constructed in Line 10 is such that (*2*)

[21] We below indicate derivations of our induction hypotheses by (*1*), (*2*), and (*3*), standing for the first, the second, and the third item of the statement, respectively, as well as by (*a*), expressing that $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}\} \cup \{0\}) \leq dl$ holds for an assignment $\mathbf{A}$ and the current value of dl.

$\{\sigma \in \mathbf{A}_k \mid \rho \in \mathbf{A}_k[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ and $\{\sigma \in \mathbf{A}_k \mid dl(\sigma) \leq \max(\{dl(\rho) \mid \rho \in \mathbf{A}_k[\sigma]\} \cup \{0\})\} \subseteq \{\sigma \in \mathbf{A}_k \mid \varepsilon \in \Delta_\Pi \cup (\nabla \cup \{\delta\}), \varepsilon \setminus \mathbf{A}_k[\sigma] = \{\overline{\sigma}\}\}$. Since $0 \leq k$ holds in view of Line 3 of Algorithm 4, (*a*) $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}_k\} \cup \{0\}) \leq dl$, where dl is set to k in Line 8. Finally, $\delta \subseteq \mathbf{A}$ and $|\{\sigma \in \delta \mid k < dl(\sigma)\}| = 1$ yield that $dl < dl'$ and $\delta \not\subseteq \mathbf{A}_k$, so that (*3*) $dl \in \mathbb{N}$ is such that $\delta' \not\subseteq \{\sigma \in \mathbf{A}_k \mid dl(\sigma) < dl\}$ for every $\delta' \in \Delta_\Pi \cup \Lambda_\Pi \cup (\nabla \cup \{\delta\})$. That is, the induction hypotheses still apply wrt $\nabla \cup \{\delta\}$, $\mathbf{A}_k$, and dl.

$\varepsilon \not\subseteq \mathbf{A}$ for all $\varepsilon \in \Delta_\Pi \cup \nabla$**:** If the condition in Line 11 of Algorithm 1 applies, CDNL-ASP(Π) terminates in Line 12 by returning $\mathbf{A}^\mathbf{T} \cap atom(\Pi)$. Otherwise, some decision literal $\sigma_d = \mathbf{T}v$ or $\sigma_d = \mathbf{F}v$ such that $v \in (atom(\Pi) \cup body(\Pi)) \setminus (\mathbf{A}^\mathbf{T} \cup \mathbf{A}^\mathbf{F})$, as required in Section 4.1, is returned by SELECT($\Pi, \nabla, \mathbf{A}$) in Line 14. Let $dl = dl'+1$, as set in Line 15. Since $dl(\sigma_d)$ is set to dl in Line 16, we have that $\mathbf{A}_{\sigma_d} = \mathbf{A} \circ \sigma_d$ constructed in Line 17 is such that (*a*) $\max(\{dl(\sigma) \mid \sigma \in \mathbf{A}_{\sigma_d}\} \cup \{0\}) = dl$ and (*2*) $\{\sigma \in \mathbf{A}_{\sigma_d} \mid \rho \in \mathbf{A}_{\sigma_d}[\sigma], dl(\sigma) < dl(\rho)\} = \emptyset$ and $\{\sigma \in \mathbf{A}_{\sigma_d} \mid dl(\sigma) \leq \max(\{dl(\rho) \mid \rho \in \mathbf{A}_{\sigma_d}[\sigma]\} \cup \{0\})\} \subseteq \{\sigma \in \mathbf{A}_{\sigma_d} \mid \varepsilon \in \Delta_\Pi \cup \nabla, \varepsilon \setminus \mathbf{A}_{\sigma_d}[\sigma] = \{\overline{\sigma}\}\}$. Finally, we note that (*1*) ∇ is not altered, and by the third item in the statement of Lemma 4.5, we have that (*3*) $dl \in \mathbb{N}$ is such that $\delta \not\subseteq \{\sigma \in \mathbf{A}_{\sigma_d} \mid dl(\sigma) < dl\}$ for every $\delta \in \Delta_\Pi \cup \Lambda_\Pi \cup \nabla$. That is, the induction hypotheses still apply wrt ∇, $\mathbf{A}_{\sigma_d}$, and dl.

We have thus shown that the items of the statement hold whenever Line 5 of Algorithm 1 is entered. □

Theorem 4.8. *Let* Π *be a logic program.*

Then, we have that CDNL-ASP(Π) *terminates, and it returns an answer set of* Π *iff* Π *has some answer set.*

Proof. If $\mathbf{A}^\mathbf{T} \cap atom(\Pi)$ is returned in Line 12 of Algorithm 1, then the test in Line 6 and the third item in the statement of Lemma 4.5 establish that $\mathbf{A}$ is a solution for $\Delta_\Pi \cup \Lambda_\Pi$. Furthermore, by Lemma 3.1, we have that there is no solution $\mathbf{B} \neq \mathbf{A}$ for $\Delta_\Pi \cup \Lambda_\Pi$ such that $\mathbf{B}^\mathbf{T} \cap atom(\Pi) = \mathbf{A}^\mathbf{T} \cap atom(\Pi)$. Hence, by Theorem 3.7, we conclude that $\mathbf{A}^\mathbf{T} \cap atom(\Pi)$ is an answer set of Π. On the other hand, if CDNL-ASP(Π) returns “no answer set” in Line 7, in view of the third item in the statement of Lemma 4.7, we have that $\max(\{dl(\sigma) \mid \sigma \in \varepsilon\} \cup \{0\}) = 0$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ such that $\varepsilon \subseteq \mathbf{A}$. Then, by the second item in the statement of Lemma 4.7, for every $\sigma \in \varepsilon$, there is some antecedent of σ wrt $\mathbf{A}$ in $\Delta_\Pi \cup \nabla$, so that there cannot be any solution for $\Delta_\Pi \cup \nabla$. Along with the first item in the statement of Lemma 4.7, it follows that there is no solution for $\Delta_\Pi \cup \Lambda_\Pi$. Hence, by Theorem 3.7, we conclude that Π has no answer set.

It remains to show that CDNL-ASP(Π) terminates. In view of the second and the third item in the statement of Lemma 4.7 along with the condition in Line 7 of Algorithm 1, we have that Lemma 4.6 applies whenever CONFLICTANALYSIS($\varepsilon, \Pi, \nabla, \mathbf{A}$) is invoked in Line 8. Hence, it returns a pair (δ, k) such that some literal ρ is unit-resulting for δ wrt $\mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$. As a consequence, ρ will be added to $\mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$ in Line 5 in the next iteration of the loop in Line 4–17. That is, after every backjump in Line 10, some element of $atom(\Pi) \cup body(\Pi)$ is assigned at a smaller (non-negative) decision level than before.

Since $atom(\Pi) \cup body(\Pi)$ is finite, this implies that CDNL-ASP(Π) admits only finitely many backjumps.[22] Along with the fact that $\mathbf{A}$ is strictly extended in Line 17, so that either a backjump or termination in Line 12 is inevitable within a linear number of iterations of the loop in Line 4–17, we conclude that CDNL-ASP(Π) eventually terminates in Line 7 or 12 of Algorithm 1. $\square$

References

[1] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The `nomore++` approach to answer set solving. In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005.

[2] C. Anger, M. Gebser, and T. Schaub. Approaching the core of unfounded sets. In J. Dix and A. Hunter, editors, *Proceedings of the Eleventh International Workshop on Nonmonotonic Reasoning (NMR'06)*, number IFI-06-04 in Institute for Informatics, Clausthal University of Technology, Technical Report Series, pages 58–66, 2006.

[3] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann Publishers, 1987.

[4] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In Boutilier [14], pages 399–404.

[5] Y. Babovich and V. Lifschitz. Computing answer sets using program completion. 2003. `http://www.cs.utexas.edu/users/tag/cmodels/cmodels-1.ps`.

[6] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[7] C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[8] C. Baral, G. Greco, N. Leone, and G. Terracina, editors. *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[22]See, e.g., [128, 115] for detailed arguments for the fact that the search pattern combining backjumping with conflict-driven assertions is complete for (UN)SAT. In a nutshell, such arguments work by ranking assignments according to the numbers of variables assigned per decision level and by verifying that the sequence of assignments generated during search is strictly monotonic. Since the total number of variables is finite, every such sequence must be finite as well (yet its length depends on heuristics). Note that this does not necessitate keeping all recorded conflict (or loop) nogoods. Rather, only the antecedents of assigned literals are ultimately needed (for conflict resolution), and their number is bounded by the number of variables.

[9] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[10] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability* [12], chapter 26, pages 825–885.

[11] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[12] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[13] G. Boenn, M. Brain, M. De Vos, and J. Fitch. Automatic composition of melodic and harmonic music by answer set programming. In Garcia de la Banda and Pontelli [43], pages 160–174.

[14] C. Boutilier, editor. *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*. AAAI Press/The MIT Press, 2009.

[15] M. Brain and M. De Vos. The significance of memory costs in answer set solver implementation. *Journal of Logic and Computation*, 19(4):615–641, 2009.

[16] G. Brewka and J. Lang, editors. *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*. AAAI Press, 2008.

[17] F. Calimeri, W. Faber, G. Pfeifer, and N. Leone. Pruning operators for disjunctive logic programming systems. *Fundamenta Informaticae*, 71(2-3):183–214, 2006.

[18] X. Chen, J. Ji, and F. Lin. Computing loops with at most one external support rule. In Brewka and Lang [16], pages 401–410.

[19] X. Chen, J. Ji, and F. Lin. Computing loops with at most one external support rule for disjunctive logic programs. In Erdem et al. [38], pages 130–144.

[20] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In P. Baumgartner and C. Fermüller, editors, *Proceedings of the Workshop on Model Computation — Principles, Algorithms, Applications (MODEL'03)*, 2003.

[21] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

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

[23] A. Darwiche and K. Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability* [12], chapter 3, pages 99–130.

[24] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[25] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[26] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[27] J. Delgrande and W. Faber, editors. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2011.

[28] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. The second answer set programming competition. In Erdem et al. [38], pages 637–654.

[29] N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In Marques-Silva and Sakallah [98], pages 287–293.

[30] W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[31] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In Brewka and Lang [16], pages 422–432.

[32] C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub. Heuristics in conflict resolution. In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.

[33] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2005.

[34] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[35] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518, 2003.

[36] N. Eén and N. Sörensson. Translating Pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[37] E. Ellguth, M. Gebser, M. Gusowski, R. Kaminski, B. Kaufmann, S. Liske, T. Schaub, L. Schneidenbach, and B. Schnor. A simple distributed conflict-driven answer set solver. In Erdem et al. [38], pages 490–495.

[38] E. Erdem, F. Lin, and T. Schaub, editors. *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2009.

[39] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

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

[41] P. Ferraris. Answer sets for propositional theories. In Baral et al. [8], pages 119–131.

[42] J. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[43] M. Garcia de la Banda and E. Pontelli, editors. *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[44] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[45] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at [110].

[46] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In Garcia de la Banda and Pontelli [43], pages 190–205.

[47] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In Hill and Warren [73], pages 250–264.

[48] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-criteria optimization in answer set programming. In J. Gallagher and M. Gelfond, editors, *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–10. Dagstuhl Publishing, 2011.

[49] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In Delgrande and Faber [27], pages 352–357.

[50] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor. Cluster-based ASP solving with *claspar*. In Delgrande and Faber [27], pages 364–369.

[51] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In Delgrande and Faber [27], pages 345–351.

[52] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Baral et al. [7], pages 260–265.

[53] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In Baral et al. [7], pages 136–148.

[54] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In Veloso [125], pages 386–392.

[55] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

[56] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In Erdem et al. [38], pages 509–514.

[57] M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems. In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

[58] M. Gebser, J. Lee, and Y. Lierler. Elementary sets for logic programs. In Gil and Mooney [69], pages 244–249.

[59] M. Gebser, J. Lee, and Y. Lierler. Head-elementary-set-free logic programs. In Baral et al. [7], pages 149–161.

[60] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In Hill and Warren [73], pages 235–249.

[61] M. Gebser and T. Schaub. Loops: Relevant or redundant? In Baral et al. [8], pages 53–65.

[62] M. Gebser and T. Schaub. Characterizing ASP inferences by unit propagation. In E. Giunchiglia, V. Marek, D. Mitchell, and E. Ternovska, editors, *Proceedings of the First International Workshop on Search and Logic: Answer Set Programming and SAT (LaSh'06)*, pages 41–56, 2006.

[63] M. Gebser and T. Schaub. Tableau calculi for answer set programming. In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

[64] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011.

[65] M. Gelfond. Answer sets. In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier, 2008.

[66] M. Gelfond and N. Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[67] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP'88)*, pages 1070–1080. The MIT Press, 1988.

[68] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[69] Y. Gil and R. Mooney, editors. *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 2006.

[70] E. Giunchiglia, N. Leone, and M. Maratea. On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):169–204, 2008.

[71] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[72] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*, pages 142–149. IEEE Press, 2002.

[73] P. Hill and D. Warren, editors. *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[74] J. Huang. The effect of restarts on the efficiency of clause learning. In Veloso [125], pages 2318–2323.

[75] H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub. Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. *Journal of Reconfigurable Computing*, 2009. `http://www.hindawi.com/journals/ijrc/2009/863630.html`.

[76] T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.

[77] T. Janhunen and I. Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In M. Balduccini and T. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, pages 111–130. Springer-Verlag, 2011.

[78] T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In Erdem et al. [38], pages 142–154.

[79] M. Järvisalo and E. Oikarinen. Extended ASP tableaux and rule redundancy in normal logic programs. *Theory and Practice of Logic Programming*, 8(5-6):691–716, 2008.

[80] T. Kim, J. Lee, and R. Palla. Circumscriptive event calculus as answer set programming. In Boutilier [14], pages 823–829.

[81] H. Kleine Büning and X. Zhao, editors. *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[82] J. Lee. A model-theoretic counterpart of loop formulas. In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

[83] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[84] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112, 1997.

[85] C. Li and F. Manyà. MaxSAT. In *Handbook of Satisfiability* [12], chapter 19, pages 613–631.

[86] Y. Lierler. Abstract answer set solvers with learning. *Theory and Practice of Logic Programming*, 11(2-3):135–169, 2011.

[87] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[88] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

[89] V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

[90] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[91] Z. Lin, Y. Zhang, and H. Hernandez. Fast SAT-based answer set solver. In Gil and Mooney [69], pages 92–97.

[92] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[93] V. Marek and V. Subrahmanian. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theoretical Computer Science*, 103(2):365–386, 1992.

[94] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[95] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In Kleine Büning and Zhao [81], pages 211–224.

[96] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability* [12], chapter 4, pages 131–153.

[97] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[98] J. Marques-Silva and K. Sakallah, editors. *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[99] A. Mileo, D. Merico, and R. Bisiani. A logic programming approach to home monitoring for risk prevention in assisted living. In Garcia de la Banda and Pontelli [43], pages 145–159.

[100] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

[101] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[102] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[103] I. Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):313–329, 2008.

[104] 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.

[105] S. Perri, F. Scarcello, G. Catalano, and N. Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):195–228, 2007.

[106] D. Pearce. A new logical characterisation of stable models and answer sets. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Proceedings of the Sixth Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*, volume 1216 of *Lecture Notes in Computer Science*, pages 57–70. Springer-Verlag, 1996.

[107] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In Marques-Silva and Sakallah [98], pages 294–299.

[108] K. Pipatsrisawat and A. Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In D. Fox and C. Gomes, editors, *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 1481–1484. AAAI Press, 2008.

[109] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[110] Potassco. `http://potassco.sourceforge.net`.

[111] P. Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10:76–94, 1970.

[112] F. Ricca, W. Faber, and N. Leone. A backjumping technique for disjunctive logic programming. *AI Communications*, 19(2):155–172, 2006.

[113] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[114] O. Roussel and V. Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability* [12], chapter 22, pages 695–733.

[115] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[116] V. Ryvchin and O. Strichman. Local restarts. In Kleine Büning and Zhao [81], pages 271–276.

[117] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51:64–86, 1995.

[118] L. Schneidenbach, B. Schnor, M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Experiences running a parallel answer set solver on Blue Gene. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Proceedings of the Sixteenth European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI'09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 64–72. Springer-Verlag, 2009.

[119] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[120] T. Syrjänen. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[121] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[122] M. Thielscher. Answer set programming for single-player games in general game playing. In Hill and Warren [73], pages 327–341.

[123] G. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic*, part 2, pages 115–125. Consultants Bureau, 1970.

[124] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[125] M. Veloso, editor. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. AAAI Press/The MIT Press, 2007.

[126] J. Ward and J. Schlipf. Answer set programming with clause learning. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *Lecture Notes in Artificial Intelligence*, pages 302–313. Springer-Verlag, 2004.

[127] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.

[128] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Sixth Conference on Design, Automation and Test in Europe (DATE'03)*, pages 10880–10885. IEEE Press, 2003.