TPLP **19** (5–6): 891–907, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000255

Inconsistency Proofs for ASP: The ASP-DRUPE Format

MARIO ALVIANO
University of Calabria, Italy
(*e-mail:* mario@alviano.net)

CARMINE DODARO
University of Calabria, Italy
(*e-mail:* dodaro@mat.unical.it)

JOHANNES K. FICHTE
TU Dresden, Germany
(*e-mail:* johannes.fichte@tu-dresden.de)

MARKUS HECHER
TU Wien, Austria
(*e-mail:* hecher@dbai.tuwien.ac.at)

TOBIAS PHILIPP
secunet Security Networks AG, Germany
(*e-mail:* tobias.philipp@secunet.com)

JAKOB RATH
TU Wien, Austria
(*e-mail:* jakob.rath@tuwien.ac.at)

submitted 29 July 2019; accepted 31 July 2019

Abstract

Answer Set Programming (ASP) solvers are highly-tuned and complex procedures that implicitly solve the consistency problem, i.e., deciding whether a logic program admits an answer set. Verifying whether a claimed answer set is formally a correct answer set of the program can be decided in polynomial time for (normal) programs. However, it is far from immediate to verify whether a program that is claimed to be inconsistent, indeed does not admit any answer sets. In this paper, we address this problem and develop the new proof format ASP-DRUPE for propositional, disjunctive logic programs, including weight and choice rules. ASP-DRUPE is based on the Reverse Unit Propagation (RUP) format designed for Boolean satisfiability. We establish correctness of ASP-DRUPE and discuss how to integrate it into modern ASP solvers. Later, we provide an implementation of ASP-DRUPE into the wasp solver for normal logic programs.

KEYWORDS: Answer Set Programming, Reverse Unit Propagation proofs, inconsistency proofs

1 Introduction

Answer Set Programming (ASP) (Brewka et al. 2011) is a logic-based declarative modeling language and problem solving framework (Gebser et al. 2012) for hard computational problems and an active research area in artificial intelligence (AI) and knowledge

representation and reasoning. It has been applied both in academia (Balduccini et al. 2006; Gebser et al. 2010; Gebser et al. 2011) and industry (Gebser et al. 2011; Guziolowski et al. 2013; Ricca et al. 2012). In propositional ASP questions are encoded by atoms combined into rules and constraints which form a logic program. Solutions to the program consist of sets of atoms called answer sets; if no solutions exist then the program is *inconsistent*.

Knowledge representation languages like ASP are usually considered explainable AI, as they are based on deduction, which is an explainable procedure. For example, we can easily explain answer sets of a normal logic program in terms of program reducts and fixpoint operators (Liu et al. 2010). In this case, we may argue that answer sets are self-explanatory, and therefore ASP systems providing answer sets are explainable AI systems. However, modern ASP systems do not provide any explanation for inconsistent programs; there is no witness that can be checked or evidence of the correctness of the refutation of the input program. Hence, even if inconsistency of logic programs is anyhow explainable with mathematical rigour, ASP systems are essentially black-boxes in this case, and just report the absence of answer sets. We believe that adding inconsistency proofs in ASP systems is important to make them explainable for inconsistent programs, but also provides auditability for consistent programs. Thanks to a duality result in the literature (Pearce 1999), such inconsistency proofs for ASP can be also used as a certificate for the validity of some formulas of intuitionistic logic and other intermediate logics. A further application of these inconsistency proofs is query answering in ASP, which is usually achieved by inconsistency checks. There, the goal is to provide a witness for cautiously true answers of a given query.

Modern ASP solvers have been highly influenced by SAT solvers, which solve the Boolean satisfiability problem and are often based on conflict-driven clause learning (Silva and Sakallah 1999). Typically, ASP solvers aim for computing an answer set of a given program, and therefore solve the *consistency problem* that asks whether a given program has an answer set. This problem is on the second level of the polynomial hierarchy when allowing arbitrary propositional disjunctive programs as input and on the first level when restricting to disjunction-free programs (Truszczyński 2011). As already stated, while consistency of a program can be easily verified given such a computed answer set, verifying whether a solver correctly outputted that a program is inconsistent, is not immediate. Given that ASP solvers are also used for critical applications (Gebser et al. 2018; Haubelt et al. 2018), their correctness is of utter importance.

When looking at SAT solvers, various techniques have been developed to ensure correctness of unsatisfiability, such as clausal proof variants (Gelder 2008; Goldberg and Novikov 2003) based on clauses that have *RUP (reverse unit propagation)* and *RAT (Resolution Asymmetric Tautology)* property. These proof formats share verifiability in polynomial time in the size of the proof and input formula and can be tightly coupled with modern solving techniques. A solver outputs such a proof during solving. Thereby, the correctness of solving can be verified by a relatively simple method for every input instance. While there are variants of these proofs for various problems, such as extensions to verify the validity of quantified Boolean formulas (Heule et al. 2013; Wetzler et al. 2014) (QRAT (Heule et al. 2014) and QRAT+ (Lonsing and Egly 2018)), to our knowledge such a format is not yet available for verifying ASP solvers. One approach to certify inconsistency of a given normal program is to translate the program into a SAT formula in polynomial time (Lin and Zhao 2003; Janhunen 2006) and obtain

a proof from a SAT solver, e.g., via a RAT-based proof format, to verify that indeed the program is inconsistent. Unfortunately, this approach does not take the techniques into account that are employed by state-of-the-art ASP solvers and therefore seems to lack efficiency and scalability. Further, this is still not a suitable technique for disjunctive programs, nor to verify whether internally the ASP solver is able to *correctly* explain the obtained result.

We follow this line of research and establish the following novel results for ASP:

1. We present the proof format ASP-DRUPE based on RUP for logic programs given in SModels (Syrjänen 2000) or ASPIF (Gebser et al. 2016) (restricted to ASP without theory reasoning) input format including disjunctive programs and show its correctness.
2. We provide an algorithm for verifying that a given solving trace in the ASP-DRUPE format is indeed a valid proof for inconsistency of the input program. This algorithm works in polynomial time in the size of the given solving trace.
3. We illustrate on an abstract ASP solving algorithm how one can integrate ASP-DRUPE into state-of-the-art ASP solvers like clasp (Gebser et al. 2012) and wasp (Alviano et al. 2015).
4. We provide an implementation in a variant of wasp, where ASP-DRUPE is integrated for normal ASP. This variant of wasp is able to not only explain inconsistency for inconsistent logic programs, but also provides auditability in case of consistency for verifying whether the provided answer set was indeed correctly obtained.

Related Work. Heule et al. presented a proof format based on the RAT property (Heule et al. 2013) and subsequently a program to validate solving traces in this format (Wetzler et al. 2014). Extended resolution allows to polynomially simulate the DRAT format (Kiesl et al. 2018) and vice-versa (Wetzler et al. 2014). Many advanced techniques, such as *XOR reasoning* (Philipp and Rebola-Pardo 2016) as well as *symmetry breaking* (Heule et al. 2015) can be expressed in DRAT and efficient, verified checkers based on RAT have been developed (Cruz-Filipe et al. 2017). Further, RAT is also available for QBF (Heule et al. 2014) and has been extended to cover a more powerful redundancy property (Lonsing and Egly 2018).

2 Preliminaries

2.1 Answer Set Programming (ASP)

We follow standard definitions of propositional ASP (Brewka et al. 2011) and use rules defined by the SModels (Syrjänen 2000) or ASPIF (Gebser et al. 2016) (restricted to ASP without theory reasoning) input formats, which are widely supported by modern ASP solvers. In particular, let ℓ, m, n be non-negative integers such that $1 \leq \ell \leq m \leq n$, $a_1, \ldots, a_n$ propositional atoms, $w, w_1, \ldots, w_n$ non-negative integers. A *choice rule* is an expression of the form $\{a_1; \ldots; a_\ell\} \leftarrow a_{\ell+1}, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$, a *disjunctive rule* is of the form $a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$, and a *weight rule* is of the form $a_\ell \leftarrow w \leqslant \{a_{\ell+1} = w_{\ell+1}, \ldots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \ldots, {\sim}a_n = w_n\}$, where $\ell = 1$. A *rule* is either a disjunctive, a choice, or a weight rule. A *(disjunctive logic) program* P is a finite set of rules. For a rule r, we let $H_r := \{a_1, \ldots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \ldots, a_m\}$, $B_r^- := \{a_{m+1}, \ldots, a_n\}$, and $B_r := B_r^+ \cup \{{\sim}a \mid a \in B_r^-\}$ is a set of *literals*, i.e., an

atom or the negation thereof. We denote the sets of *atoms* occurring in a rule r or in a program P by $\mathrm{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and $\mathrm{at}(P) := \bigcup_{r \in P} \mathrm{at}(r)$. For a weight rule r, let $\mathrm{wght}(r, l)$ map literal l to its weight w_i in rule r if $l = a_i$ for $\ell + 1 \leq i \leq m$, or if $l = {\sim}a_i$ for $m+1 \leq i \leq n$, and to 0 otherwise. Further, let $\mathrm{wght}(r, L) := \Sigma_{l \in L} \mathrm{wght}(r, l)$ for a set L of literals and let $\mathrm{bnd}(r) := w$ be its *bound*. A *normal* (logic) program P is a disjunctive program P with $|H_r| \leq 1$ for every $r \in P$. The *positive dependency digraph* D_P of P is the digraph defined on the set $\{a \mid a \in H_r \cup B_r^+, r \in P\}$ of atoms, where for every rule $r \in P$ two atoms $a \in B_r^+$ and $b \in H_r$ are joined by an edge (a, b). We denote the set of all cycles (loops) in D_P by $\mathsf{loop}(P)$. A program P is called *tight*, if $\mathsf{loop}(P) = \emptyset$. While we allow programs with loops that might also involve atoms of weight rules, we consider weight rules only as a compact representation of a set of normal rules, similar to the definition of stable models in related work (Bomanson et al. 2016). In other words, we do not consider advanced semantics concerning recursive weight rules (recursive aggregates). In case of solvers with different semantics, one can restrict the input language to disregard recursive weight rules, which is also in accordance with the latest ASP-Core-2.03c standard (Calimeri et al. 2015). This restriction is motivated by a lack of consensus on the interpretation of recursive weight rules (Ferraris 2011; Faber et al. 2011; Gelfond and Zhang 2014; Pelov et al. 2007; Son and Pontelli 2007).

2.2 Solving Logic Programs

Let P be a given program, $r \in P$ be a rule, and $a \in \mathrm{at}(P)$. We define the *set* $\mathsf{IB}(r, a)$ *of induced bodies* with a in the head by the singleton $\{B_r \mid a \in H_r\}$ if r is a choice rule, by $\{B_r \cup \{{\sim}b \mid b \in H_r \setminus \{a\}\} \mid a \in H_r\}$ if r is a disjunctive rule, and by the union over $\{\{A \cap B_r^+\} \cup \{{\sim}b \mid b \in A \cap B_r^-\} \mid a \in H_r\}$ for every (subset-minimal) set L of literals such that $\mathrm{wght}(r, L) \geq \mathrm{bnd}(r)$, if r is a weight rule. This allows us to define $\mathsf{IB}(P, a) := \bigcup_{r \in P, a \in H_r} \mathsf{IB}(r, a)$, and $\mathsf{Bod}(P) := \bigcup_{a \in \mathrm{at}(P)} \mathsf{IB}(P, a)$. A *variable assignment* is either $\mathbf{T}X$ or $\mathbf{F}X$, where *variable* X is either an atom, or an induced body, or a *fresh atom* that does not occur in $\mathrm{at}(P)$. For a variable assignment l, $\bar{l}$ is the *complementary variable assignment* of l, i.e., $\bar{l} := \mathbf{F}X$ if $l = \mathbf{T}X$ and $\bar{l} := \mathbf{T}X$ if $l = \mathbf{F}X$. An *assignment* A is a set of variable assignments, where $A^{\mathbf{T}} := \{X \mid \mathbf{T}X \in A\}$, $A^{\mathbf{F}} := \{X \mid \mathbf{F}X \in A\}$, and $\overline{A} := \{\bar{l} \mid l \in A\}$ such that $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$. For a set B of literals, we define the *induced assignment* $\mathsf{IAss}(P, B) := \{\mathbf{T}a \mid a \in \mathrm{at}(P) \cap B\} \cup \{\mathbf{F}a \mid {\sim}a \in B\}$. A *nogood* δ is an assignment, which is not allowed, where $\square := \emptyset$ refers to the empty nogood. Given a set Δ of nogoods. We define the least fixpoint $\Delta_{\vdash_1}$ of *unit propagated nogoods* by the fixpoint computation $\Delta^0_{\vdash_1} := \Delta$, and $\Delta^i_{\vdash_1} := \Delta^{i-1}_{\vdash_1} \cup \{\delta \setminus \{l\} \mid \delta \in \Delta^{i-1}_{\vdash_1}, l \in \delta, \{\bar{l}\} \in \Delta^{i-1}_{\vdash_1}\}$ for $i \geq 1$. Nogood δ is a *consequence using unit propagation (UP)* of set Δ, denoted by $\Delta \vdash_1 \delta$, if $\delta \in \Delta_{\vdash_1}$. An assignment A *satisfies* a set Δ of nogoods (written $A \models \Delta$) if for every $\delta \in \Delta$, we have $\delta \not\subseteq A$. Set Γ of nogoods is a *consequence* of a set Δ of nogoods (denoted by $\Delta \models \Gamma$) if every assignment, which contains a variable assignment for all variables in $\Delta \cup \Gamma$, that satisfies Δ also satisfies Γ. The set Δ_P of *completion nogoods (Clark 1977; Gebser et al. 2012)* is defined by $\Delta_P := \Delta_P^{\mathrm{Bdef}} \cup \Delta_P^{\rightarrow} \cup \Delta_P^{\leftarrow}$, where

$$\Delta_P^{\mathrm{Bdef}} := \bigcup_{B \in \mathsf{Bod}(P)} \Big\{\{\mathbf{F}B\} \cup \mathsf{IAss}(P, B)\Big\} \cup \Big\{\{\mathbf{T}B, \bar{l}\} \mid l \in \mathsf{IAss}(P, B)\Big\}$$

$$\Delta_P^{\rightarrow} := \bigcup_{a \in \mathrm{at}(P)} \Big\{\{\mathbf{T}a\} \cup \{\mathbf{F}B \mid B \in \mathsf{IB}(P, a)\}\Big\}, \Delta_P^{\leftarrow} := \bigcup_{\substack{\text{non-choice rule } r \in P, \\ a \in H_r}} \Big\{\{\mathbf{F}a, \mathbf{T}B\} \mid B \in \mathsf{IB}(r, a)\Big\}.$$

Note that in practice, current ASP solvers do not fully compute Δ_P. Instead, these solvers partially compute Δ_P and add relevant nogoods lazily during solving (Alviano et al. 2018).

Then, if P is tight the set $A^{\mathbf{T}} \cap \mathrm{at}(P)$ is an *answer set* if and only if there is a satisfying assignment A for Δ_P (Fages 1994; Gebser et al. 2012). The set $\mathsf{EB}(P,U)$ of *external bodies* of program P and set $U \subseteq \mathrm{at}(P)$ of atoms are given by $\mathsf{EB}(P,U) := \{B \mid B \in \mathsf{IB}(P,a), B \cap U = \emptyset, a \in U\}$ (Gebser et al. 2012). We define the *loop nogood* $\lambda(a,U)$ for an atom $a \in U$ on a loop $U \in \mathsf{loop}(P)$ by $\lambda(a,U) := \{\mathbf{T}a\} \cup \{\mathbf{F}B \mid B \in \mathsf{EB}(P,U)\}$. For a logic program P, the set $A^{\mathbf{T}} \cap \mathrm{at}(P)$ is an *answer set* if and only if there is a satisfying assignment A for $\Delta_P \cup \Lambda_P$, where $\Lambda_P := \{\lambda(a,U) \mid a \in U, U \in \mathsf{loop}(P)\}$ (Lin and Zhao 2003; Faber 2005).

3 ASP-DRUPE: RUP-like Format for Proof Logging

Inspired by RUP-style unsatisfiability proofs in the field of Boolean satisfiability solving (Goldberg and Novikov 2003), we aim for a proof of inconsistency of a program. Since modern ASP solvers use *Clark's completion* (Clark 1977) to transform a program into a set of nogoods, we do so as well. Our aimed proof then has the following features:

1. *Existence of a simple verification algorithm.* In order to increase confidence in the correctness of results, the algorithm that verifies the proof has to be fairly easy to understand and to implement.
2. *Low complexity.* The proof is verifiable in polynomial time in its length and the size of the completion nogoods.
3. *Integrability into solvers that employ Conflict-Driven Nogood Learning (CDNL).* The proof can stepwise be outputted during solving with minimal impact on the solving algorithm and hence the solver.

The method works as follows: We run the solver on the set Δ_P of completion nogoods for given input program P. The solver outputs either an answer set or that P has no answer set and a proof Π. We pass P together with Π to the verifier in order to validate whether the solver's assessment is in accordance with its outputted proof.

3.1 The Proof Format for Logic Programs

The basic idea of clausal proofs for SAT is the following: One starts with the input formula in CNF (given as a set of clauses). Every step of the proof denotes an addition or deletion of a clause to/from the set of clauses. For additions, the condition is to only add clauses that are a logical consequence of the current set of clauses and that it can be checked easily, e.g., use only unit propagation.

For our format ASP-DRUPE, we consider Clark's completion Δ_P as the initial set of nogoods corresponding to the input program P. Besides addition and deletion of nogoods, we need proof steps that model how the solver excludes unfounded sets (loops).

Example 1
Consider program $P = \{a \leftarrow b,d;\ b \leftarrow a,d;\ a \leftarrow c;\ b \leftarrow c,d;\ c \leftarrow \sim d;\ d \leftarrow \sim c;\ e \leftarrow c, \sim e;\ e \leftarrow \sim a, \sim e\}$, which is inconsistent. P contains only the positive loop $L = \{a,b\}$, whose external support is given by the set $\{a \leftarrow c;\ b \leftarrow c,d\}$ of rules, and thus $\mathsf{EB}(P,L) =$

$\{\{c\},\{c,d\}\}$. Set L induces two possible loop nogoods, $\lambda(a,L)=\{\mathbf{T}a,\mathbf{F}\{c\},\mathbf{F}\{c,d\}\}$ and $\lambda(b,L)=\{\mathbf{T}b,\mathbf{F}\{c\},\mathbf{F}\{c,d\}\}$.

We describe the proof format ASP-DRUPE for logic programs and adapt the RUP property (Goldberg and Novikov 2003) to nogoods as follows.

Definition 1 (*nogood RUP*)
Let Δ be a set of nogoods. Then, a nogood δ is *RUP* (reverse unit propagable) *for* Δ if $\Delta\cup\{\{\bar{l}\}\mid l\in\delta\}\vdash_1\square$, i.e., we can derive $\square$ using only unit propagation.

A *proof step* is a triple (t,δ,a), where $t\in\{\mathsf{a},\mathsf{c},\mathsf{s},\mathsf{e},\mathsf{d},\mathsf{l}\}$ denotes the *type* of the step, δ is an assignment, and a is an atom or ϵ. The type $t\in\{\mathsf{a},\mathsf{c},\mathsf{s},\mathsf{e},\mathsf{d},\mathsf{l}\}$ indicates whether the step is an addition (a), a completion rule addition (c), a completion support addition (s), an extension (e), a deletion (d), or a loop addition (l). A *proof sequence* for a logic program P is a finite sequence $\Pi:=\langle\sigma_1,\ldots,\sigma_n\rangle$ of proof steps. Initially, a proof sequence gets associated with a set $\nabla_0(\Pi):=\Delta_P^{\mathrm{Bdef}}$ of nogoods. Then, each proof step σ_i for $1\leq i\leq n$ subsequently transforms $\nabla_{i-1}(\Pi)$ into the induced set $\nabla_i(\Pi)$ of nogoods, formally defined below. An *ASP-DRUPE derivation* is a proof sequence that allows for RUP-like rules for ASP and includes both deletion and extension. In an ASP-DRUPE derivation each step $\sigma=(t,\delta,a)$ has to satisfy a condition depending on its type as follows:

1. An *addition* $\sigma=(\mathsf{a},\delta,\epsilon)$ inserts a nogood δ that is RUP for $\nabla_{i-1}(\Pi)$.
2. A *completion rule addition* $\sigma=(\mathsf{c},\delta,\epsilon)$ inserts a nogood $\delta\in\Delta_P^{\leftarrow}$.
3. A *completion support addition* $\sigma=(\mathsf{s},\{\mathbf{F}B_1,\ldots,\mathbf{F}B_k\},a)$ inserts a nogood $\{\mathbf{T}a,\mathbf{F}B_1,\ldots,\mathbf{F}B_k\}\in\Delta_P^{\rightarrow}$ if $\{B_1,\ldots,B_k\}=\mathsf{IB}(P,a)$.
4. An *extension* $\sigma=(\mathsf{e},\delta,a)$ introduces a definition that renders nogood δ equivalent to a fresh atom a, i.e., a does not appear in $\bigcup_{j=0}^{i-1}\nabla_j(\Pi)\cup\Delta_P$. Formally, this rule represents the set $\mathsf{ext}(a,\delta):=\big\{\delta\cup\{\mathbf{F}a\}\big\}\cup\big\{\{\mathbf{T}a,\bar{l}\}\mid l\in\delta\big\}$ of *extension nogoods*.
5. A *deletion* $\sigma=(\mathsf{d},\delta,\epsilon)$ represents the deletion of δ from $\nabla_{i-1}(\Pi)$.
6. A *loop addition*[1] $\sigma=(\mathsf{l},\{\mathbf{T}a_1,\ldots,\mathbf{T}a_k\},a_1)$ inserts a loop nogood $\lambda(a_1,L)$ for a loop $L=\{a_1,\ldots,a_k\}\in\mathsf{loop}(P)$.

Given an ASP-DRUPE derivation $\Pi=\langle\sigma_1,\ldots,\sigma_n\rangle$, we define the *set* $\nabla_i(\Pi)$ *of nogoods induced by step* i as the result of applying proof steps $\langle\sigma_1,\ldots,\sigma_i\rangle$ to the initial completion body definitions Δ_P^{Bdef} for $0\leq i\leq n$. For our inductive definition in the following, we use multiset semantics for additions and deletions of nogoods, and write $\uplus$ for the multiset sum.

$$\nabla_0(\Pi):=\Delta_P^{\mathrm{Bdef}}$$

$$\nabla_i(\Pi):=\begin{cases}\nabla_{i-1}(\Pi)\uplus\{\delta\}, & \text{if } \sigma_i=(\mathsf{a},\delta,\epsilon),\\ \nabla_{i-1}(\Pi)\uplus\{\delta\}, & \text{if } \sigma_i=(\mathsf{c},\delta,\epsilon),\\ \nabla_{i-1}(\Pi)\uplus\{\delta\cup\{\mathbf{T}a\}\}, & \text{if } \sigma_i=(\mathsf{s},\delta,a),\\ \nabla_{i-1}(\Pi)\uplus\mathsf{ext}(a,\delta), & \text{if } \sigma_i=(\mathsf{e},\delta,a),\\ \nabla_{i-1}(\Pi)\setminus\{\delta\}, & \text{if } \sigma_i=(\mathsf{d},\delta,\epsilon),\\ \nabla_{i-1}(\Pi)\uplus\{\lambda(a_1,\{a_1,\ldots,a_k\})\}, & \text{if } \sigma_i=(\mathsf{l},\{\mathbf{T}a_1,\ldots,\mathbf{T}a_k\},a_1).\end{cases}$$

[1] There could be an exponential number of external bodies involving weight rules. However, both clasp and wasp treat weight rules differently (Alviano et al. 2015). Alternatively, one could easily modify the loop addition type to list also involved external bodies (as in the completion support addition type), which we did not for the sake of readability.

Then, we say that an ASP-DRUPE derivation Π is an *ASP-DRUPE proof* for the inconsistency of P if it actually derives inconsistency for P, formally, $\Box \in \nabla_n(\Pi)$. Note that Δ_P^{Bdef} might be exponential in the input program size, in the worst case. However, there is no need to materialize the set Δ_P^{Bdef}, as, intuitively, this set of body definitions only ensures that every induced body has a reserved auxiliary atom that can be used to "address" the body in a compact way. In an actual implementation of a solver that uses ASP-DRUPE, one needs to specify these used auxiliary atoms anyway, cf. Section 5, where implementational specifications of ASP-DRUPE are described.

Example 2
Consider program P from Example 1 and loop $L = \{a, b\}$, which induces loop nogood $\lambda(a, L) = \{\mathbf{T}a, \mathbf{F}\{c\}, \mathbf{F}\{c, d\}\}$. Then, the proof sequence $\Pi = \langle \sigma_1, \ldots, \sigma_{18} \rangle$ is an ASP-DRUPE proof for the inconsistency of P with

$$
\begin{aligned}
\sigma_1 &= (\mathsf{s}, \{\mathbf{F}\{b, d\}, \mathbf{F}\{c\}\}, a) & \nabla_1(\Pi) &= \Delta_P^{\mathrm{Bdef}} \uplus \{\{\mathbf{T}a, \mathbf{F}\{b, d\}, \mathbf{F}\{c\}\}\}\\
\sigma_2 &= (\mathsf{c}, \{\mathbf{F}c, \mathbf{T}\{\sim d\}\}, \epsilon) & \nabla_2(\Pi) &= \nabla_1(\Pi) \uplus \{\{\mathbf{F}c, \mathbf{T}\{\sim d\}\}\}\\
\sigma_3 &= (\mathsf{c}, \{\mathbf{F}d, \mathbf{T}\{\sim c\}\}, \epsilon) & \nabla_3(\Pi) &= \nabla_2(\Pi) \uplus \{\{\mathbf{F}d, \mathbf{T}\{\sim c\}\}\}\\
\sigma_4 &= (\mathsf{c}, \{\mathbf{F}b, \mathbf{T}\{a, d\}\}, \epsilon) & \nabla_4(\Pi) &= \nabla_3(\Pi) \uplus \{\{\mathbf{F}b, \mathbf{T}\{a, d\}\}\}\\
\sigma_5 &= (\mathsf{s}, \{\mathbf{F}\{c, \sim e\}, \mathbf{F}\{\sim a, \sim e\}\}, e) & \nabla_5(\Pi) &= \nabla_4(\Pi) \uplus \{\{\mathbf{T}e, \mathbf{F}\{c, \sim e\}, \mathbf{F}\{\sim a, \sim e\}\}\}\\
\sigma_6 &= (\mathsf{l}, \{\mathbf{T}a, \mathbf{T}b\}, a) & \nabla_6(\Pi) &= \nabla_5(\Pi) \uplus \{\{\mathbf{T}a, \mathbf{F}\{c\}, \mathbf{F}\{c, d\}\}\}\\
\sigma_7 &= (\mathsf{a}, \{\mathbf{F}\{c\}, \mathbf{T}a\}, \epsilon) & \nabla_7(\Pi) &= \nabla_6(\Pi) \uplus \{\{\mathbf{F}\{c\}, \mathbf{T}a\}\}\\
\sigma_8 &= (\mathsf{s}, \{\mathbf{F}\{\sim d\}\}, c) & \nabla_8(\Pi) &= \nabla_7(\Pi) \uplus \{\{\mathbf{T}c, \mathbf{F}\{\sim d\}\}\}\\
\sigma_9 &= (\mathsf{s}, \{\mathbf{F}\{\sim c\}\}, d) & \nabla_9(\Pi) &= \nabla_8(\Pi) \uplus \{\{\mathbf{T}d, \mathbf{F}\{\sim c\}\}\}\\
\sigma_{10} &= (\mathsf{s}, \{\mathbf{F}\{a, d\}, \mathbf{F}\{c, d\}\}, b) & \nabla_{10}(\Pi) &= \nabla_9(\Pi) \uplus \{\{\mathbf{T}b, \mathbf{F}\{a, d\}, \mathbf{F}\{c, d\}\}\}\\
\sigma_{11} &= (\mathsf{a}, \{\mathbf{T}e, \mathbf{F}\{\sim a, \sim e\}\}, \epsilon) & \nabla_{11}(\Pi) &= \nabla_{10}(\Pi) \uplus \{\{\mathbf{T}e, \mathbf{F}\{\sim a, \sim e\}\}\}\\
\sigma_{12} &= (\mathsf{c}, \{\mathbf{F}e, \mathbf{T}\{c, \sim e\}\}, \epsilon) & \nabla_{12}(\Pi) &= \nabla_{11}(\Pi) \uplus \{\{\mathbf{F}e, \mathbf{T}\{c, \sim e\}\}\}\\
\sigma_{13} &= (\mathsf{a}, \{\mathbf{T}a\}, \epsilon) & \nabla_{13}(\Pi) &= \nabla_{12}(\Pi) \uplus \{\{\mathbf{T}a\}\}\\
\sigma_{14} &= (\mathsf{c}, \{\mathbf{F}a, \mathbf{T}\{b, d\}\}, \epsilon) & \nabla_{14}(\Pi) &= \nabla_{13}(\Pi) \uplus \{\{\mathbf{F}a, \mathbf{T}\{b, d\}\}\}\\
\sigma_{15} &= (\mathsf{c}, \{\mathbf{F}a, \mathbf{T}\{c\}\}, \epsilon) & \nabla_{15}(\Pi) &= \nabla_{14}(\Pi) \uplus \{\{\mathbf{F}a, \mathbf{T}\{c\}\}\}\\
\sigma_{16} &= (\mathsf{a}, \{\mathbf{T}e\}, \epsilon) & \nabla_{16}(\Pi) &= \nabla_{15}(\Pi) \uplus \{\{\mathbf{T}e\}\}\\
\sigma_{17} &= (\mathsf{c}, \{\mathbf{F}e, \mathbf{T}\{\sim a, \sim e\}\}, \epsilon) & \nabla_{17}(\Pi) &= \nabla_{16}(\Pi) \uplus \{\{\mathbf{F}e, \mathbf{T}\{\sim a, \sim e\}\}\}\\
\sigma_{18} &= (\mathsf{a}, \Box, \epsilon) & \nabla_{18}(\Pi) &= \nabla_{17}(\Pi) \uplus \{\Box\}.
\end{aligned}
$$

We show that the proof step σ_7 is correct, i.e., that $\{\mathbf{F}\{c\}, \mathbf{T}a\}$ is RUP for $\nabla_6(\Pi)$. To this end, we need to derive $\Box$ from $\nabla_6(\Pi) \cup \{\{\mathbf{T}\{c\}\}, \{\mathbf{F}a\}\}$ by unit propagation. With the nogood $\{\mathbf{F}\{c\}, \mathbf{T}c\} \in \Delta_P^{\mathrm{Bdef}}$, we derive the unit nogood $\{\mathbf{T}c\}$. With $\{\mathbf{T}\{c, d\}, \mathbf{F}c\} \in \Delta_P^{\mathrm{Bdef}}$ we now get $\{\mathbf{T}\{c, d\}\}$. With these unit nogoods, $\lambda(a, L)$ reduces to $\Box$.

3.2 Correctness of ASP-DRUPE

Next, we establish soundness and completeness of the ASP-DRUPE format.

Lemma 1 (*Invariants*)
Let P be a logic program and $\Pi = \langle \sigma_1, \ldots, \sigma_n \rangle$ be a finite ASP-DRUPE derivation for program P. Moreover, let D_i be the accumulated set of nogoods introduced by the

extension rules in σ_k for all $k \in \{1, \ldots, i\}$. Then, the following holds: $\Delta_P \cup \Lambda_P \cup D_i \models \nabla_i(\Pi)$ for all $i \in \{0, \ldots, n\}$.

Proof (Sketch).
We proceed by induction over the length n of the derivation. For the base case, we have $\nabla_0(\Pi) = \Delta_P^{\text{Bdef}}$. Hence, $\nabla_0(\Pi) \subseteq \Delta_P \cup \Lambda_P \cup D_0$ and the claim holds trivially. For the induction step, we assume that the statement holds for length i and consider step σ_{i+1}. It remains to do a case distinction for the type:

1. Deletion with $\sigma_{i+1} = (\mathsf{d}, \delta, \epsilon)$: Immediately, we have $\nabla_i(\Pi) \models \nabla_{i+1}(\Pi)$. Thus, transitivity of $\models$ and the induction hypothesis establishes this case.
2. Addition with $\sigma_{i+1} = (\mathsf{a}, \delta, \epsilon)$: Since δ is RUP for $\nabla_i(\Pi)$, we know that $\{\delta\}$ is a logical consequence of Δ_i. The remaining steps to draw the conclusion are similar to the deletion step case.
3. Completion Rule Addition with $\sigma_{i+1} = (\mathsf{c}, \delta, \epsilon)$: Since the resulting nogood $\delta \in \Delta_P^{\leftarrow}$ is contained in Δ_P, the result follows.
4. Completion Support Addition with $\sigma_{i+1} = (\mathsf{s}, \delta, a)$: Since the resulting nogood $\{\mathbf{T}a\} \cup \delta$ is contained in $\Delta_P^{\rightarrow}$, the result follows.
5. Extension with $\sigma_{i+1} = (\mathsf{e}, \delta, a)$: According to the induction hypothesis we have $\Delta_P \cup \Lambda_P \cup D_i \models \nabla_i(\Pi)$. Then, $D_i \subset D_{i+1}$, since a is a fresh variable and $\mathsf{ext}(a, \delta) \in D_{i+1} \setminus D_i$. As $\models$ is monotone, and $D_i \subset D_{i+1}$, we know that $\Delta_P \cup \Lambda_P \cup D_{i+1} \models \nabla_i(\Pi)$. It then follows that $\Delta_P \cup \Lambda_P \cup D_{i+1} \models \nabla_{i+1}(\Pi)$.
6. Loop addition with $\sigma_{i+1} = (\mathsf{l}, L, a_1)$: By definition nogood $\delta := \lambda(a_1, L)$ is already contained in $\Delta_P \cup \Lambda_P \cup D_i$, which immediately establishes this case. □

Theorem 1 (*Soundness and Completeness*)
Let P be a logic program. Then, P is inconsistent if and only if there is an ASP-DRUPE proof for P.

Proof (Sketch).
Let P be a logic program. "⇐": Assume there is an ASP-DRUPE proof of P. By definition, there is a finite sequence of proof steps $\langle \sigma_i, \ldots, \sigma_n \rangle$ such that $\square \in \nabla_n(\Pi)$ and $\nabla_n(\Pi)$ is inconsistent. From Lemma 1, we obtain that $\Delta_P \cup \Lambda_P \cup D_n$ is inconsistent. As D consists of extension nogoods with disjoint variables, we know that $\Delta_P \cup \Lambda_P$ is inconsistent. We conclude from an earlier result (Gebser et al. 2012, Theorem 5.4) that P is inconsistent. "⇒": Suppose P is inconsistent. According to earlier work (Gebser et al. 2012, Theorem 5.4), we know that $\Delta_P \cup \Lambda_P$ is inconsistent. RUP is complete (Gelder 2008; Goldberg and Novikov 2003), which means that for every propositional, unsatisfiable formula F there is a RUP proof for F. Hence, we can construct an ASP-DRUPE proof for P as follows: (i) Output all completion rule additions for $\Delta_P^{\leftarrow}$ and completion support additions for $\Delta_P^{\rightarrow}$. (ii) Generate loop addition steps for all loops $L \in \mathsf{loop}(P)$. (iii) Transform $\Delta_P \cup \Lambda_P$ into a propositional formula $F = \overline{\Delta_P \cup \Lambda_P}$ by inverting all nogoods. (iv) Construct and use a RUP proof for F. Then, output addition rules accordingly, where again all clauses need to be inverted to obtain addition proof steps using nogoods. □

Listing 1: ASP-DRUPE-Checker

Input : A logic program P and an ASP-DRUPE derivation $\Pi = \langle \sigma_1, \ldots, \sigma_n \rangle$.
Output : *Success* if Π proves that P has no answer set, *Error* otherwise.

$\nabla \coloneqq \Delta_P^{\mathrm{Bdef}}$
for $i = 1, \ldots, n$ **do**
  if $\sigma_i = (\mathsf{a}, \delta, \epsilon)$ **and** $\nabla \vdash_1 \delta$ **then**
    $\nabla \coloneqq \nabla \uplus \{\delta\}$
  else if $\sigma_i = (\mathsf{c}, \delta, \epsilon)$ **and** $\delta \in \Delta_P^{\leftarrow}$ **then**
    $\nabla \coloneqq \nabla \uplus \{\delta\}$
  else if $\sigma_i = (\mathsf{s}, \delta, a)$ **and** $\delta = \{\mathbf{F}B \mid B \in \mathsf{IB}(P, a)\}$ **then**
    $\nabla \coloneqq \nabla \uplus \{\delta \cup \{\mathbf{T}a\}\}$
  else if $\sigma_i = (\mathsf{e}, \delta, a)$ **and** a is a fresh atom w.r.t. $\bigcup_{j=0}^{i-1} \nabla_j(\Pi) \cup \Delta_P$ **then**
    $\nabla \coloneqq \nabla \uplus \mathsf{ext}(a, \delta)$
  else if $\sigma_i = (\mathsf{d}, \delta, \epsilon)$ **then**
    $\nabla \coloneqq \nabla \setminus \{\delta\}$
  else if $\sigma_i = (\mathsf{l}, \{\mathbf{T}a_1, \ldots, \mathbf{T}a_k\}, a_1)$ **then**
    $U \coloneqq \{a_1, \ldots, a_k\}$
    if $U \in \mathsf{loop}(P)$ **then** $\nabla \coloneqq \nabla \uplus \{\lambda(a_1, U)\}$ **else return** *Error*
  else
    return *Error*
if $\square \in \nabla$ **then return** *Success* **else return** *Error*

Note that in the only-if direction of the proof, one can also use RAT (Wetzler et al. 2014) proofs without deletion information and afterwards translate RAT steps into extended resolution steps (Kiesl et al. 2018).

Listing 1 presents the ASP-DRUPE checker, that decides whether a given ASP-DRUPE proof is correct. The input to the checker is both the original program P and the proof Π. To check the proof, we encode P into nogoods Δ_P and then check each statement $\sigma \in \Pi$ sequentially.

Lemma 2
For a given logic program P and an ASP-DRUPE derivation Π, the ASP-DRUPE-Checker runs in time at most $|\Pi|^{\mathcal{O}(1)}$.

Corollary 1
Given a logic program P and an ASP-DRUPE derivation Π. Then, the ASP-DRUPE-Checker is correct, i.e., it outputs *Success* if and only if Π is an ASP-DRUPE proof for the inconsistency of P.

3.3 Extension to Optimization

Next, we briefly mention how to verify cost optimization. To this end, an *optimization rule* is an expression of the form $\leftsquigarrow l_1[w_1]$, where l_1 is a literal. Intuitively this indicates that if an assignment satisfies $\mathsf{IAss}(P, \{l_1\})$, then this results in costs w_1. Overall, one aims to minimize the total costs, i.e., the goal is to deliver an answer set of minimal total costs. Therefore, if one wants to verify whether a given answer set candidate is indeed an answer set of minimal costs, we foresee the following extension to ASP-DRUPE, where such a proof consists of the following two parts. (i) An answer set that shows a solution

Listing 2: CDNL-ASP-DRUPE: CDNL-ASP (Gebser et al. 2012, page 93) extended by proof logging

Input : A logic program P.
Output : An answer set of P or an ASP-DRUPE proof Π certifying that P has no answer set.

1 $A \coloneqq \emptyset$, $\nabla \coloneqq \emptyset$, $dl \coloneqq 0$, $\Pi \coloneqq \emptyset$
2 **loop**
3 $(A, \nabla, \Pi) \coloneqq \text{NogoodPropagation}(P, \nabla, A, \Pi)$ // deterministic cons.
4 **if** $\varepsilon \subseteq A$ **for some** $\varepsilon \in \Delta_P \cup \nabla$ **then** // conflict
5 **if** $\max(\{dlevel(\sigma) \mid \sigma \in \varepsilon\} \cup \{0\}) = 0$ **then**
6 **return** (INCONSISTENT, $\Pi \cdot \langle(\mathsf{a}, \square, \epsilon)\rangle$)
7 $(\delta, dl) \coloneqq \text{ConflictAnalysis}(\varepsilon, P, \nabla, A)$ // δ is RUP for $\Delta_P \cup \nabla$
8 $\nabla \coloneqq \nabla \cup \{\delta\}$ // add conflict nogood
9 $\Pi \coloneqq \Pi \cdot \langle(\mathsf{a}, \delta, \epsilon)\rangle$ // record nogood addition in proof
10 $A \coloneqq A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$ // backjumping
11 **else if** $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \text{at}(P) \cup \text{Bod}(P)$ **then** // answer set found
12 **return** (CONSISTENT, $A^{\mathbf{T}} \cap \text{at}(P)$)
13 **else**
14 $\sigma_d \coloneqq \text{Select}(P, \nabla, A)$ // decision
15 $dl \coloneqq dl + 1$
16 $dlevel(\sigma_d) \coloneqq dl$
17 $A \coloneqq A \cup \{\sigma_d\}$

with costs c exists. (ii) An ASP-DRUPE proof that shows that the program restricted to costs $c - 1$ is inconsistent. Note that for disjunctive programs already the first part also needed to contain a second proof showing that indeed there cannot be an unfounded set for the provided answer set. Further, it is not immediate, how this extends to unsatisfiable cores. Hence, so far it only applies to progression based approaches.

4 Integrating ASP-DRUPE Proofs into a Solver

In the following, we describe the CDNL-ASP algorithm for logic programs P that we use as a basis for our theoretical model. Afterwards, we describe how *proof logging* can be integrated. In other words, during the run of an ASP solver, we immediately print the corresponding ASP-DRUPE rules that are needed later for verification in case the ASP solver concludes that the program is inconsistent. A typical CDNL-based ASP solver (cf., Listing 2) relies on unit propagation, since this is a rather simple and efficient way of concluding consequences. Thereby it keeps a set ∇ of nogoods, a current assignment A, and a decision level dl. In a loop it applies *NogoodPropagation* (Gebser et al. 2012, page 101) consisting of unit propagation and loop propagation (using *UnfoundedSet* (Gebser et al. 2012, page 104)) whenever suited. Then, if there is some nogood that is not satisfied, either the program is inconsistent (at decision level 0) or *ConflictAnalysis* (Gebser et al. 2012, page 108) triggers backtracking to an earlier decision level, followed by the learning of a conflict nogood δ. Otherwise, if all nogoods in $\Delta_P \cup \nabla$ are satisfied and all the variables are assigned, an answer set is found, and otherwise some free variable is selected (*Select*).

Listings 2 and 3 contain a prototypical CDNL-based ASP solver that is extended by proof logging, where the changes for proof logging are highlighted in red. We use the

Listing 3: NogoodPropagation (Gebser et al. 2012, page 101) adapted for proof logging

Input : A logic program P, a set ∇ of nogoods, an assignment A, and an ASP-DRUPE derivation Π.
Output : An extended assignment, a set of nogoods, and an ASP-DRUPE derivation (possibly empty).

```
1  U := ∅
2  loop
3      repeat
4          if δ ⊆ A for some δ ∈ Δ_P ∪ ∇ then                                  // conflict
5              Π := Π · ⟨σ | σ = (c, δ, ϵ), δ ∈ Δ_P^←, σ ∉ Π⟩·   // record confl. completion
               nogood ⟨σ | σ = (s, δ \ {Ta}, a), δ ∈ Δ_P^→, Ta ∈ δ, σ ∉ Π⟩
6              return (A, ∇, Π)
7          Σ := {δ ∈ Δ_P ∪ ∇ | δ \ A = {σ̄}, σ ∉ A}                  // unit-resulting nogoods
8          for δ ∈ Σ do                          // record unit completion nogoods in proof
9              Π := Π · ⟨σ | σ = (c, δ, ϵ), δ ∈ Δ_P^←, σ ∉ Π⟩·
                       ⟨σ | σ = (s, δ \ {Ta}, a), δ ∈ Δ_P^→, Ta ∈ δ, σ ∉ Π⟩
10         if Σ ≠ ∅ then
11             let σ̄ ∈ δ for some δ ∈ Σ in
12                 dlevel(σ) := max({0} ∪ {dlevel(ρ) | ρ ∈ δ \ {σ̄}})
13                 A := A ∪ {σ}
14     until Σ = ∅
15     if loop(P) = ∅ then return (A, ∇, Π)
16     U := U \ A^F
17     if U = ∅ then U := UnfoundedSet(P, A)
18     if U = ∅ then return (A, ∇, Π)                                  // no unfounded set
19     let a_0 ∈ U in
20         ∇ := ∇ ∪ {{Ta_0} ∪ {FB | B ∈ EB(P, U)}}                      // add loop nogood
21         Π := Π · ⟨(l, {TX | X ∈ U}, a_0)⟩                          // record loop in proof
```

element operator ($\in$) to determine whether an element is in a sequence, and denote the concatenation of two proofs by the $\cdot$ operator as follows: $\langle \sigma_1, \ldots, \sigma_i \rangle \cdot \langle \sigma_{i+1}, \ldots, \sigma_n \rangle := \langle \sigma_1, \ldots, \sigma_i, \sigma_{i+1}, \ldots, \sigma_n \rangle$. The idea is to start with an empty ASP-DRUPE derivation. Whenever a new nogood, or loop nogood is learned and added to ∇ accordingly, this results in an added addition or loop addition proof step, respectively. Note that in Listing 3 we add completion rule addition steps and completion support addition steps, whenever unit propagation (or conflicts) involve rules in $\Delta_P^{\leftarrow}$ or $\Delta_P^{\rightarrow}$, respectively. In particular, Lines 5 and 9 take care of adding involved parts of the completion to the proof (if needed) accordingly. At the end, when the ASP solver concludes inconsistency, the proof is returned including the empty nogood as last nogood. Note that advanced techniques (see, e.g., (Gebser et al. 2012)) like *forgetting* of learned clauses and *restarting* of the ASP solver can also be implemented using deletion rules with ASP-DRUPE. As it turns out, *preprocessing* in ASP is less sophisticated as for SAT. In the literature, CDNL-based ASP solvers often use preprocessing techniques (Gebser et al. 2008) similar to SAT solvers, i.e., SatElite-like (Eén and Biere 2005) operations as variable and nogood elimination. For simple preprocessing operations restricted to variable and nogood elimination ASP-DRUPE suffices. Note that if Clark's completion is exponential in the program size due to weight rules, also propagators (Alviano et al. 2018) are supported. For details we refer to the implementation in Section 4.1.

Example 3 (*CDNL-ASP-DRUPE*)
We continue the previous Example 2 and indicate a possible CDNL-ASP-DRUPE run on P that leads to the exemplary ASP-DRUPE proof given above. We use the notation $\mathbf{T}X@dl$ ($\mathbf{F}X@dl$) to indicate that X was assigned true (false) at decision level dl.

1. Initially, nothing can be propagated.
2. After the decision $\mathbf{T}a@1$, unit propagation derives only $\mathbf{F}\{\sim a, \sim e\}@1$.
3. After the second decision $\mathbf{F}\{c\}@2$, we eventually derive $\mathbf{T}a@2$ and $\mathbf{T}b@2$ by unit propagation. Thus we discover the unfounded set $U = \{a, b\} \in \mathsf{loop}(P)$, and add the loop nogood $\lambda(a, U)$.
4. The loop nogood immediately leads to a conflict, and conflict analysis learns the nogood $\{\mathbf{F}\{c\}, \mathbf{T}a\}$.
5. We backtrack to decision level 1, and after propagation, make the decision $\mathbf{T}e@2$. We arrive at another conflict, and learn $\{\mathbf{T}e, \mathbf{F}\{\sim a, \sim e\}\}$.
6. After backtracking, a conflict appears at decision level 1, and we learn $\{\mathbf{T}a\}$.
7. We backtrack to decision level 0, and decide on $\mathbf{T}e@1$. After arriving at a conflict almost immediately, we learn $\{\mathbf{T}e\}$.
8. We backtrack once more, and finally arrive at a conflict at decision level 0, returning INCONSISTENT along with an ASP-DRUPE proof.

4.1 Implementation of ASP-DRUPE in wasp solver

We provide an implementation of ASP-DRUPE within the wasp (Alviano et al. 2015) solver that is available on github[2]. The solver prints a proof for inconsistency in the file `proof.log` if the solver gets passed the program options `--disable-simplifications --proof-log=proof.log`. Actually wasp prints an ASP-DRUPE derivation also in the positive case of consistency. This derivation can still be used to verify whether the nogoods learned by the solver are correct. Currently, proof logging is restricted to normal programs and we do not yet support recursive weight rules due to discrepancy among different semantics as discussed in the preliminaries. Moreover, we had to introduce a normalized form because of several (in-processing) simplifications that would otherwise require major refactoring to isolate. Just to mention one of these simplifications, a rule of the form $a \leftarrow \ell_1, \ldots, \ell_n, \sim a$ is replaced by the integrity constraint $\leftarrow \ell_1, \ldots, \ell_n, \sim a$. While these simplifications are required to achieve efficient computation, they alter the completion of the input program. Therefore, wasp cannot log in the proof the auxiliary atoms required to keep the completion compact. The problem is circumvented by introducing a normalized form such that the completion can be compactly computed without introducing any auxiliary atoms.

A program P is in *short-body normalized form* if for each atom $a \in \mathrm{at}(P)$ either $|\mathsf{IB}(P, a)| \leq 1$, or for any body $B \in \mathsf{IB}(P, a)$, we have $|B| \leq 1$. Any normal program can be transformed into short-body normalized form in linear time by introducing a linear number of auxiliary atoms (in the program size). This normalized form allows us to get rid of auxiliary variables for bodies $B \in \mathsf{Bod}(P)$, i.e., we can set $\Delta_P^{\mathrm{Bdef}} = \emptyset$, and replace $\mathbf{T}B$ in $\Delta_P^{\leftarrow}$ by $\mathsf{IAss}(P, B)$, and $\mathbf{F}B$ in $\Delta_P^{\rightarrow}$ by $\overline{\mathsf{IAss}(P, B)}$. For simplification and

[2] The repository can be found at https://github.com/alviano/wasp/tree/unsatproof.

increased readability of a compact resulting proof log, we further do not use completion rule addition nor completion support addition types. Instead, we assume that the checker is aware of the completion from the beginning. In this respect, we have to observe that for weight rules, the completion might be exponential in the program size. Therefore, we require that the checker is equipped with a propagator (Alviano et al. 2018) for drawing conclusions by unit propagation on parts of the completion associated with weight rules. We provide an implementation of such a checker tool as well[3].

5 ASP-DRUPE Implementational Specifications

Next, we discuss the specific format description of ASP-DRUPE that we think can be commonly used in ASP solvers. To this end, we assume a program P and a set $\{\hat{B} \mid B \in \mathtt{Bod}(P)\}$ of fresh atoms, where we have one fresh atom for each induced body in P. Further, let $\mathsf{vm} : (\mathrm{at}(P) \cup \{\hat{B} \mid B \in \mathtt{Bod}(P)\}) \to \mathbb{N}$ be an injective mapping of atoms $a \in \mathrm{at}(P) \cup \{\hat{B} \mid B \in \mathtt{Bod}(P)\}$ to a unique positive integer $n \geq 1$ such that $\mathsf{vm}(a) := n$, and $a = \mathsf{vm}^{-1}(n)$. Note that for atoms $a \in \mathrm{at}(P)$ this can be (partly) already provided by the input format. However, for technical reasons, we assume such a mapping also for atom $\hat{B}$, where $B \in \mathtt{Bod}(P)$, as these integers will then correspond to fresh atoms. We define in the following an SModels-like (Syrjänen 2000) output format of strings for a given program P, which is ready for the checker to parse. Actually, the output format is "line-based", i.e., it is even ASPIF-like (Gebser et al. 2016). However, ASP-DRUPE still supports ASP only, and not ASP solving with theory reasoning. To this end, let the *truth value mapping* tv map a variable assignment l to an integer different from 0, where a positive integer represents an atom and a negative integer a negated atom.

$$\mathsf{tv}(l) := \begin{cases} \mathsf{vm}(X), & \text{if } l = \mathbf{T}Y \text{ and } X = Y \text{ is an atom or } X = \hat{Y} \text{ for } Y \in \mathtt{Bod}(P), \\ -\mathsf{vm}(X), & \text{if } l = \mathbf{F}Y \text{ and } X = Y \text{ is an atom or } X = \hat{Y} \text{ for } Y \in \mathtt{Bod}(P). \end{cases}$$

Then, the *ASP-DRUPE output format* is a sequence $\zeta = \langle s_1, \ldots, s_j \rangle$ of strings, where each element in the sequence corresponds to one rule in an ASP-DRUPE derivation and is terminated by character "0". Each part of an element in the sequence is separated by a white space (␣). We indicate other strings constants by $'string'$. Then, element s_i of the sequence ζ for $1 \leq i \leq j$ is of the following form.

- A *body definition string* is of the form $'\mathsf{b}'␣b_1␣n_1␣\ldots␣n_k␣'0'$ such that b_1, n_1, …, $n_k \in \mathbb{N}$. Further, we require that $\{\mathsf{tv}^{-1}(n_1), \ldots, \mathsf{tv}^{-1}(n_k)\} = \mathsf{IAss}(P, B)$ for some $B \in \mathtt{Bod}(P)$. Finally, for s_i the string corresponds to the proof step $(\mathsf{e}, \{\mathbf{T}B\}, \hat{B})$, where $\hat{B} = \mathsf{vm}^{-1}(b_1)$. The technical purpose of s_i is to specify the fresh body variable $\hat{B}$ for a body $B \in \mathtt{Bod}(P)$.
- An *addition string* is of the form $'\mathsf{a}'␣n_1␣\ldots␣n_k␣'0'$ such that $n_1, \ldots, n_k \in \mathbb{N}$, which corresponds to proof step $(\mathsf{a}, \{\mathsf{tv}^{-1}(n_1), \ldots, \mathsf{tv}^{-1}(n_k)\}, \epsilon)$.
- A *completion rule addition string* is of the form $'\mathsf{c}'␣b_1␣n_1␣\ldots␣n_k␣'0'$ such that $b_1, n_1, \ldots, n_k \in \mathbb{N}$ and $\hat{B} = \mathsf{vm}^{-1}(b_1)$, which relates to proof step $(\mathsf{c}, \{\mathbf{F}\mathsf{vm}^{-1}(n_1), \ldots, \mathbf{F}\mathsf{vm}^{-1}(n_k), \mathbf{T}B\}, \epsilon)$.

[3] Both the checker tool and a tool for bringing normal logic programs in short-body normalized form can be found at https://github.com/alviano/python/tree/master/asp-proof.

```
b 6 3 0
b 7 -3 0
b 8 -4 0
b 9 1 4 0
b 10 2 4 0
b 11 3 4 0
b 12 -1 -5 0
b 13 3 -5 0
```

```
s 1 10 6 0
c 8 3 0
c 7 4 0
c 9 2 0
s 5 13 12 0
l 1 2 0
a -6 1 0
s 3 8 0
s 4 7 0
```

```
s 2 9 11 0
a 5 -12 0
c 13 5 0
a 1 0
c 10 1 0
c 6 1 0
a 5 0
c 12 5 0
a 0
```

Fig. 1. ASP-DRUPE output format $\mathsf{smod}(\Pi)$ of the proof Π of Example 2.

- A *completion support addition string* is of the form $'\mathsf{s}' \sqcup n_1 \sqcup b_1 \sqcup \ldots \sqcup b_k \sqcup '0'$ such that $n_0, b_1, \ldots, b_k \in \mathbb{N}$ and $\hat{B}_i = \mathsf{vm}^{-1}(b_i)$ for $i \in \{1, \ldots, k\}$, which corresponds to proof step $(\mathsf{s}, \{\mathbf{F}B_1, \ldots, \mathbf{F}B_k\}, \mathsf{vm}^{-1}(n_0))$.
- An *extension string* is of the form $'\mathsf{e}' \sqcup n_0 \sqcup n_1 \sqcup \ldots \sqcup n_k \sqcup '0'$ such that $n_0, n_1, \ldots, n_k \in \mathbb{N}$, which corresponds to proof step $(\mathsf{e}, \{\mathsf{tv}^{-1}(n_1), \ldots, \mathsf{tv}^{-1}(n_k)\}, \mathsf{vm}^{-1}(n_0))$.
- A *deletion string* is of the form $'\mathsf{d}' \sqcup n_1 \sqcup \ldots \sqcup n_k \sqcup '0'$ such that $n_1, \ldots, n_k \in \mathbb{N}$, which corresponds to proof step $(\mathsf{d}, \{\mathsf{tv}^{-1}(n_1), \ldots, \mathsf{tv}^{-1}(n_k)\}, \epsilon)$.
- A *loop addition string* is of the form $'\mathsf{l}' \sqcup n_1 \sqcup \ldots \sqcup n_k \sqcup '0'$ such that $n_1, \ldots, n_k \in \mathbb{N}$, which corresponds to proof step $(\mathsf{l}, \{\mathbf{T}\mathsf{vm}^{-1}(n_1), \ldots, \mathbf{T}\mathsf{vm}^{-1}(n_k)\}, \mathsf{vm}^{-1}(n_1))$.
- An *unfounded set addition string* is of the form $'\mathsf{u}' \sqcup k \sqcup n_1 \sqcup \ldots \sqcup n_k \sqcup o_1 \sqcup \ldots \sqcup o_m \sqcup '0'$ such that $n_1, \ldots, n_k, o_1, \ldots, o_m \in \mathbb{N}$, which then corresponds to proof step $(\mathsf{u}, \{\mathbf{T}\mathsf{vm}^{-1}(n_1), \ldots, \mathbf{T}\mathsf{vm}^{-1}(n_k)\}, \epsilon, \{\mathsf{tv}^{-1}(o_1), \ldots, \mathsf{tv}^{-1}(o_m)\})$.

Next, we define how to obtain the ASP-DRUPE output format $\zeta = \mathsf{smod}(\Pi)$ of a given ASP-DRUPE derivation $\Pi = \langle \sigma_1, \ldots, \sigma_n \rangle$. To this end, we define $s = \mathsf{smod}(\sigma_i)$, which transforms a proof step σ_i into a string s for $1 \leq i \leq n$, by slight abuse of notation.

$$\mathsf{smod}(\sigma_i) := \begin{cases} '\mathsf{a}' \sqcup \mathsf{tv}(\mathsf{l}_1) \sqcup \ldots \sqcup \mathsf{tv}(\mathsf{l}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{a}, \{l_1, \ldots, l_k\}, \epsilon), \\ '\mathsf{c}' \sqcup \mathsf{vm}(\hat{\mathsf{B}}) \sqcup \mathsf{vm}(\mathsf{a}_1) \sqcup \ldots \sqcup \mathsf{vm}(\mathsf{l}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{c}, \{\mathbf{F}a_1, \ldots, \mathbf{F}a_k, \mathbf{T}B\}, \epsilon), \\ '\mathsf{s}' \sqcup \mathsf{vm}(\mathsf{a}) \sqcup \mathsf{vm}(\mathsf{B}_1) \sqcup \ldots \sqcup \mathsf{vm}(\mathsf{B}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{s}, \{\mathbf{F}B_1, \ldots, \mathbf{F}B_k\}, a), \\ '\mathsf{e}' \sqcup \mathsf{vm}(\mathsf{a}) \sqcup \mathsf{tv}(\mathsf{l}_1) \sqcup \ldots \sqcup \mathsf{tv}(\mathsf{l}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{e}, \{l_1, \ldots, l_k\}, a), \\ '\mathsf{d}' \sqcup \mathsf{tv}(\mathsf{l}_1) \sqcup \ldots \sqcup \mathsf{tv}(\mathsf{l}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{d}, \{l_1, \ldots, l_k\}, \epsilon), \\ '\mathsf{l}' \sqcup \mathsf{vm}(\mathsf{a}_1) \sqcup \ldots \sqcup \mathsf{vm}(\mathsf{a}_k) \sqcup '0', & \text{if } \sigma_i = (\mathsf{l}, \{\mathbf{T}a_1, \ldots, \mathbf{T}a_k\}, a_1), \\ '\mathsf{u}' \sqcup \mathsf{k} \sqcup \mathsf{vm}(\mathsf{a}_1) \sqcup \ldots \sqcup \mathsf{vm}(\mathsf{a}_k), & \text{if } \sigma_i = (\mathsf{u}, \{\mathbf{T}a_1, \ldots, \mathbf{T}a_k\}, \epsilon, \\ \quad \mathsf{tv}(\mathsf{l}_1) \sqcup \ldots \sqcup \mathsf{tv}(\mathsf{l}_m) \sqcup '0', & \qquad \{l_1, \ldots, l_m\}). \end{cases}$$

Since fresh body atoms require introduction using extension proof steps in advance, we assume $\mathsf{Bod}(P) = \{B_1, \ldots, B_q\}$, where $B_i = \{l_{i,1}, \ldots l_{i,|B_i|}\}$ for $1 \leq i \leq q$. Finally, let $\mathsf{smod}(\Pi) := \langle '\mathsf{b}' \sqcup \mathsf{vm}(\hat{B}_1) \sqcup \mathsf{tv}(l_{1,1}) \sqcup \ldots \sqcup \mathsf{tv}(l_{1,|B_1|}) \sqcup '0' \rangle \cdot \ldots \cdot \langle '\mathsf{b}' \sqcup \mathsf{vm}(\hat{B}_q) \sqcup \mathsf{tv}(l_{q,1}) \sqcup \ldots \sqcup \mathsf{tv}(l_{q,|B_q|}) \sqcup '0' \rangle \cdot \mathsf{smod}(\sigma_1) \cdot \ldots \cdot \mathsf{smod}(\sigma_n)$. As a simplification, one can leave out additional, unused body definition strings.

Example 4

Consider the ASP-DRUPE proof Π for inconsistency of P from Example 2 and assume the dictionary in the program input assigns $\mathsf{vm}(a) = 1$, $\mathsf{vm}(b) = 2$, $\mathsf{vm}(c) = 3$, $\mathsf{vm}(d) = 4$,

and $\mathsf{vm}(e) = 5$. We extend this to the necessary bodies: $\mathsf{vm}(\{c\}) = 6$, $\mathsf{vm}(\{\sim c\}) = 7$, $\mathsf{vm}(\{\sim d\}) = 8$, $\mathsf{vm}(\{a, d\}) = 9$, $\mathsf{vm}(\{b, d\}) = 10$, $\mathsf{vm}(\{c, d\}) = 11$, $\mathsf{vm}(\{\sim a, \sim e\}) = 12$, and $\mathsf{vm}(\{c, \sim e\}) = 13$. Figure 1 corresponds to $\mathsf{smod}(\Pi)$. Note that we use body definitions for required body variables.

6 Conclusion & Future Work

ASP solvers are highly-tuned decision procedures that are widely applied in academia and industry. In this paper, we considered how to ensure that if an ASP solver outputs that a program has no answer set then the solver is indeed right. Similar to unsat certificates in SAT solvers, we propose an approach that augments the inconsistency answer of an ASP solver with a certificate of inconsistency. This approach allows the use of unverified, efficient ASP solvers while guaranteeing that a particular run of an ASP solver has been correct.

To this end, we developed a new proof format called ASP-DRUPE. It allows several types of proof steps: *(RUP) addition* that models nogood learning, *completion rule addition* and *completion support addition* for adding completion rules on demand, *deletion* that models nogood forgetting, *extension* that allows to infer new definitions and *loop addition* that adds nogoods to forbid assignments that do not correspond to answer sets. ASP-DRUPE supports formula simplification methods that can be obtained by learning entailed nogoods, nogood deletion as well as extended resolution. We established that ASP-DRUPE is sound and complete for logic programs and can be used effectively, i.e., a program P is inconsistent if and only if a ASP-DRUPE proof of P exists and that we can check an ASP-DRUPE proof in polynomial time of the proof length. Further, we demonstrated how to augment CDNL-based solvers with proof logging. It is in our interest for future work to continue this line of research. Potential next steps include the study of theory reasoning towards covering the full ASPIF intermediate format.

Finally, we would like to say a few words about RAT-style proofs. The combination of nogood deletion, loop, and RAT addition results in an inconsistent proof system in which we can infer a conflict although a non-tight program is consistent. This stems from the situation that clauses that are RAT with respect to ∇ are not necessarily RAT with respect to $\Delta_P \cup \Lambda_P$. Although it was recently shown that extended resolution simulates DRAT (Kiesl et al. 2018), we are unaware whether ASP-DRUPE can be extended to RAT such that each rule application can be checked in polynomial time, which we believe is an interesting question for future work.

Acknowledgments

Mario Alviano and Carmine Dodaro have been partially supported by MIUR under project "Declarative Reasoning over Streams" (CUP H24I17000080001) – PRIN 2017, by MISE under project "S2BDW" (F/050389/01-03/X32) – "Horizon2020" PON I&C2014-20, by Regione Calabria under project "DLV LargeScale" (CUP J28C17000220006) – POR Calabria 2014-20, and by GNCS-INdAM. Markus Hecher has been supported by Austrian Science Fund (FWF) Grant Y698. Johannes K. Fichte has been funded by German Science Fund (DFG) Grant HO 1294/11-1.

References

Alviano, M., Dodaro, C., Leone, N., and Ricca, F. 2015. Advances in WASP. In *LPNMR 2015*. LNCS, vol. 9345. Springer, 40–54.

Alviano, M., Dodaro, C., and Maratea, M. 2018. Shared aggregate sets in answer set programming. *TPLP 18,* 3-4, 301–318.

Balduccini, M., Gelfond, M., and Nogueira, M. 2006. Answer set based design of knowledge systems. *Ann. Math. Artif. Intell. 47,* 1-2, 183–219.

Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B., and Schaub, T. 2016. Answer set programming modulo acyclicity. *Fundam. Inform. 147,* 1, 63–91.

Brewka, G., Eiter, T., and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. 2015. ASP-core-2 input language format.

Clark, K. L. 1977. Negation as failure. In *Symposium on Logic and Data Bases 1977*. Advances in Data Base Theory. Plemum Press, 293–322.

Cruz-Filipe, L., Heule, M. J. H., Jr., W. A. H., Kaufmann, M., and Schneider-Kamp, P. 2017. Efficient certified RAT verification. In *CADE 2017*. LNCS, vol. 10395. Springer, 220–236.

Eén, N. and Biere, A. 2005. Effective preprocessing in SAT through variable and clause elimination. In *SAT 2005*. LNCS, vol. 3569. Springer, 61–75.

Faber, W. 2005. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *LPNMR 2005*. LNCS, vol. 3662. Springer, 40–52.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell. 175,* 1, 278–298.

Fages, F. 1994. Consistency of clark's completion and existence of stable models. *Meth. of Logic in CS 1,* 1, 51–60.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log. 12,* 4, 25:1–25:40.

Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., and Veber, P. 2010. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *KR 2010*. The AAAI Press, 497–507.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. 2016. Theory solving made easy with clingo 5. In *ICLP (Tech. C.)*. OASICS, vol. 52. Dagstuhl Pub., 2:1–2:15.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2011. Multi-Criteria Optimization in Answer Set Programming. In *ICLP 2011*. LIPIcs, vol. 11. Dagstuhl Pub., 1–10.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2008. Advanced preprocessing for answer set solving. In *ECAI 2008*. Frontiers in Artificial Intelligence and Applications, vol. 178. IOS Press, 15–19.

Gebser, M., Obermeier, P., Ratsch-Heitmann, M., Runge, M., and Schaub, T. 2018. Routing driverless transport vehicles in car assembly with answer set programming. *CoRR abs/1804.10437*.

Gebser, M., Schaub, T., Thiele, S., and Veber, P. 2011. Detecting inconsistencies in large biological networks with answer set programming. *TPLP 11,* 2-3, 323–360.

Gelder, A. V. 2008. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *TPLP 14,* 4-5, 587–601.

Goldberg, E. I. and Novikov, Y. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*. IEEE Computer Society, 10886–10891.

Guziolowski, C., Videla, S., Eduati, F., Thiele, S., Cokelaer, T., Siegel, A., and Saez-Rodriguez, J. 2013. Exhaustively characterizing feasible logic models of a signaling network using answer set programming. *Bioinformatics 29,* 18, 2320–2326. Erratum see Bioinformatics 30, 13, 1942.

Haubelt, C., Neubauer, K., Schaub, T., and Wanko, P. 2018. Design space exploration with answer set programming. *KI - Künstliche Intelligenz 32,* 2, 205–206.

Heule, M., Hunt Jr., W. A., and Wetzler, N. 2013. Verifying refutations with extended resolution. In *CADE 2013*. LNCS, vol. 7898. Springer, 345–359.

Heule, M., Hunt Jr., W. A., and Wetzler, N. 2015. Expressing symmetry breaking in DRAT proofs. In *CADE 2015*. LNCS, vol. 9195. Springer, 591–606.

Heule, M., Seidl, M., and Biere, A. 2014. A unified proof system for QBF preprocessing. In *IJCAR*. LNCS, vol. 8562. Springer, 91–106.

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

Kiesl, B., Rebola-Pardo, A., and Heule, M. J. H. 2018. Extended resolution simulates DRAT. In *IJCAR 2018*. LNCS, vol. 10900. Springer, 516–531.

Lin, F. and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *IJCAI'03*. Morgan Kaufmann, 853–858.

Liu, L., Pontelli, E., Son, T. C., and Truszczynski, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artif. Intell. 174,* 3-4, 295–315.

Lonsing, F. and Egly, U. 2018. QRAT+: generalizing QRAT by a more powerful QBF redundancy property. In *IJCAR*. LNCS, vol. 10900. Springer, 161–177.

Pearce, D. 1999. Stable inference as intuitionistic validity. *J. Log. Program. 38,* 1, 79–91.

Pelov, N., Denecker, M., and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

Philipp, T. and Rebola-Pardo, A. 2016. DRAT proofs for XOR reasoning. In *JELIA 2016*. LNCS, vol. 10021. Springer, 415–429.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *TPLP 12*, 361–381.

Silva, J. P. M. and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers 48,* 5, 506–521.

Son, T. C. and Pontelli, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming 7,* 3, 355–375.

Syrjänen, T. 2000. Lparse 1.0 User's Manual.

Truszczyński, M. 2011. Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *TPLP 11*, 881–904.

Wetzler, N., Heule, M., and Hunt Jr., W. A. 2014. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT 2014*. LNCS, vol. 8561. Springer, 422–429.