Conflict-Driven Answer Set Solving

Martin Gebser and **Benjamin Kaufmann** and **André Neumann** and **Torsten Schaub***
Institut für Informatik, Universität Potsdam,
Postfach 90 03 27, D–14439 Potsdam, Germany

Abstract

We introduce a new approach to computing answer sets of logic programs, based on concepts from constraint processing (CSP) and satisfiability checking (SAT). 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 for the different kinds of inferences in ASP. It also allows us to apply advanced techniques from the areas of CSP and SAT. We have implemented our approach in the new ASP solver *clasp*. Our experiments show that the approach is competitive with state-of-the-art ASP solvers.

1 Introduction

Answer set programming (ASP; [Baral, 2003]) has become an attractive tool for knowledge representation and reasoning. Although the corresponding solvers are highly optimized (cf. [Simons *et al.*, 2002; Leone *et al.*, 2006]), their performance does not match the one of state-of-the-art solvers for satisfiability checking (SAT; [Mitchell, 2005]). However, computational mechanisms of SAT and ASP solvers are not that far-off. This can, for instance, be seen on the success of SAT-based ASP solvers *assat* [Lin and Zhao, 2004] and *cmodels* [Giunchiglia *et al.*, 2006]. But despite the close relationship to SAT and, more generally, constraint processing (CSP; [Dechter, 2003]), state-of-the-art look-back techniques from these areas, like backjumping, conflict-driven learning, and restarts, are not yet established in genuine ASP solvers. In fact, recent approaches to adopt such techniques [Ward and Schlipf, 2004; Ricca *et al.*, 2006; Lin *et al.*, 2006] are rather implementation-specific and lack generality.

We address this deficiency by introducing a new computational approach to ASP solving, centered around the CSP concept of a nogood. Apart from the fact that this allows us to easily integrate solving technology from the areas of CSP and SAT, e.g., conflict-driven learning, backjumping, watched literals, etc., it also provides us with a uniform representation of inferences from logic program rules, unfounded sets, as well as nogoods learned from conflicts.

*Affiliated with the School of Computing Science at Simon Fraser University, Canada, and IIIS at Griffith University, Australia.

After establishing the formal background, we provide in Section 3 a constraint-based specification of ASP solving in terms of nogoods. Based on this uniform representation, we develop in Section 4 algorithms for ASP solving that rely on advanced CSP and SAT techniques. Notably, our solving procedure is centered around conflict-driven learning and backjumping. In Section 5, we describe our new ASP solver *clasp*, implementing our approach. We finally provide empirical results demonstrating the competitiveness of *clasp*.

2 Background

Given an alphabet $\mathcal{P}$, a (normal) *logic program* is a finite set of rules of the form $p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$ where $0 \leq m \leq n$ and $p_i \in \mathcal{P}$ is an *atom* for $0 \leq i \leq n$. A *body literal* is an atom p or its negation $not\ p$. For a rule r, let $head(r) = p_0$ be the *head* of r and $body(r) = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$ be the *body* of r. The set of atoms occurring in a logic program Π is denoted by $atom(\Pi)$. The set of bodies in Π is $body(\Pi) = \{body(r) \mid r \in \Pi\}$. For regrouping rule bodies sharing the same head p, define $body(p) = \{body(r) \mid r \in \Pi, head(r) = p\}$. In ASP, the semantics of a program Π is given by its *answer sets*, being total well-founded models of Π. For a formal introduction to ASP, we refer the reader to [Baral, 2003].

A Boolean *assignment* A over a *domain*, $dom(A)$, is a sequence $(\sigma_1, \ldots, \sigma_n)$ of *signed literals* σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$; $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ 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, $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$. We let $A \circ B$ denote the sequence obtained by concatenating assignments A and B. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access true and false propositions in A via $A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\}$ and $A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\}$.

For a canonical representation of constraints, we use the CSP concept of a nogood. In our setting, a *nogood* is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$. An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a *solution* for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.

For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is *unit-resulting* for δ wrt A, if (1) $\delta \setminus A = \{\sigma\}$

and (2) $\overline{\sigma} \notin A$. By (1), σ is the single literal from δ that is not contained in A. This implies that a violated constraint does not have a unit-resulting literal. Condition (2) makes sure that no duplicates are introduced: If A already contains $\overline{\sigma}$, then it is no longer unit-resulting. For instance, literal $\mathbf{F}q$ 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 wrt $(\mathbf{F}p, \mathbf{T}q)$. Note that our notion of a unit-resulting literal is closely related to the *unit clause rule* of DPLL (cf. [Mitchell, 2005]). For a set Δ of nogoods and an assignment A, we call *unit propagation* the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

3 Nogoods of Logic Programs

Inferences in ASP rely on atoms and program rules, which can be expressed by using atoms and bodies. For a program Π, we thus fix the domain of assignments A to $dom(A) = atom(\Pi) \cup body(\Pi)$. Such a hybrid approach may result in exponentially smaller search spaces [Gebser and Schaub, 2006]; it moreover allows for an adequate representation of nogoods, as we show in the sequel.

Our approach is guided by the idea of Lin and Zhao [2004] and decomposes ASP solving into (local) inferences obtainable from the Clark completion of a program [Clark, 1978] and those obtainable from loop formulas. We begin with nogoods capturing inferences from the Clark completion.

The body of a rule is true if all its body literals are true. Conversely, some of its literals must be false if the body is false. For a body $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, the following nogood captures this:

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

Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body β, or the falsity of a contained literal. E.g. for body $\{x, not\ y\}$, we obtain $\delta(\{x, not\ y\}) = \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}$.

Additionally, a body must be false if one of its literals is false. And conversely, all contained literals must be true if the body is true. For $\beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, this is reflected by the following set of nogoods:

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

Taking again body $\{x, not\ y\}$, we obtain $\Delta(\{x, not\ y\}) = \{\ \{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\ \}$.

Nogoods induce a set of clauses, which can be used for investigating the logical contents of the underlying inferences. Given a program Π, we associate a nogood $\delta = \{\mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\}$ with the clause $\gamma(\delta) = \{\neg q_1, \ldots, \neg q_m, q_{m+1}, \ldots, q_n\}$ where $q_i = p_i$, if $p_i \in atom(\Pi)$, and $q_i = p_\beta$, if $p_i = \beta \in body(\Pi)$, for $1 \leq i \leq n$; and define $\Gamma(\Delta) = \{\gamma(\delta) \mid \delta \in \Delta\}$ for a set of nogoods Δ. For the bodies of Π, we obtain the following correspondence.

Proposition 3.1 *Let Π be a logic program.*
The set of clauses

$$\{\gamma(\delta(\beta)) \mid \beta \in body(\Pi)\} \cup \{\gamma \in \Gamma(\Delta(\beta)) \mid \beta \in body(\Pi)\}$$

is logically equivalent to the propositional theory

$$\{p_\beta \equiv p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \mid \beta \in body(\Pi), \beta = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}\}.$$

This result captures the intuition that a body should be equivalent to the conjunction of its body literals.

We now come to inferences primarily aiming at atoms. An atom p must be true if some body in $body(p)$ is true. Conversely, all elements of $body(p)$ must be false if p is false. For $body(p) = \{\beta_1, \ldots, \beta_k\}$, we get the nogoods:

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

For example, for an atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we get $\Delta(x) = \{\ \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\ \}$.

Finally, an atom p must be false if all elements of $body(p)$ are false. And conversely, some body in $body(p)$ must be true if p is true. For $body(p) = \{\beta_1, \ldots, \beta_k\}$, 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 x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$.

Dually to Proposition 3.1, we have the following for atoms.

Proposition 3.2 *Let Π be a logic program.*
The set of clauses

$$\{\gamma \in \Gamma(\Delta(p)) \mid p \in atom(\Pi)\} \cup \{\gamma(\delta(p)) \mid p \in atom(\Pi)\}$$

is logically equivalent to the propositional theory

$$\{p \equiv p_{\beta_1} \vee \cdots \vee p_{\beta_k} \mid p \in atom(\Pi), body(p) = \{\beta_1, \ldots, \beta_k\}\}.$$

Combining the last propositions yields the following result.

Theorem 3.3 *Let Π be a tight logic program and*

$$\Delta_\Pi = \{\delta(\beta) \mid \beta \in body(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in body(\Pi)\} \cup \{\delta(p) \mid p \in atom(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in atom(\Pi)\}.$$

Then, $X \subseteq atom(\Pi)$ is an answer set of Π iff $X = A^{\mathbf{T}} \cap atom(\Pi)$ for a (unique) solution A for Δ_Π.

The nogoods in Δ_Π capture the *supported models* of a program [Apt *et al.*, 1987]. Any answer set is a supported model, but the converse only holds for *tight* programs [Fages, 1994]. The mismatch on non-tight programs is caused by *loops* [Lin and Zhao, 2004], responsible for cyclic support among true atoms. Such cyclic support can be prohibited by *loop formulas*. As shown in [Lee, 2005], the answer sets of a program Π are precisely the models of Π that satisfy the loop formulas of all non-empty subsets of $atom(\Pi)$.[1] Observe that the exponential number of loops in the worst case [Lifschitz and Razborov, 2006] makes an enumeration of all loop formulas infeasible. All loop formulas can however be checked in linear time, and propagation within genuine ASP solvers makes sure that they are satisfied by a solution.

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

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

The (disjunctive) *loop formula* of U for Π, $LF_\Pi(U)$, is

$$\neg\big(\bigvee_{\beta \in EB_\Pi(U)} (\bigwedge_{p \in \beta^+} p \wedge \bigwedge_{p \in \beta^-} \neg p)\big) \rightarrow \neg\big(\bigvee_{p \in U} p\big),$$

[1] Note that a loop formula can be constructed for any set of atoms, even if this set is not a loop in the sense of [Lin and Zhao, 2004].

where $\beta^+ = \beta \cap atom(\Pi)$ and $\beta^- = \{p \mid not\ p \in \beta\}$. The loop formula of a set U of atoms enforces all elements of U to be false, if U is not *externally supported* [Lee, 2005].

To capture the effect of a loop formula induced by a set $U \subseteq atom(\Pi)$, we define the *loop nogood* of an atom $p \in U$ as

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

where $EB_\Pi(U) = \{\beta_1, \ldots, \beta_k\}$. Overall, we get the following set of loop nogoods for a program Π:

$$\Lambda_\Pi = \bigcup_{U \subseteq atom(\Pi), U \neq \emptyset} \{\lambda(p, U) \mid p \in U\} \qquad (1)$$

Observe that loop nogoods make direct use of the bodies in $EB_\Pi(U) = \{\beta_1, \ldots, \beta_k\}$, unlike loop formulas $LF_\Pi(U)$ relying on the literals in each β_i. Using bodies in loop nogoods is reasonable because unit propagation on completion nogoods makes a body false if it contains a false literal. Notably, the usage of bodies avoids a combinatorial blow-up, faced when expressing these constraints in terms of body literals. In fact, representing $\lambda(p, U)$ in terms of body literals yields about $|\beta_1 \times \cdots \times \beta_k|$ nogoods instead of a single one.

Dropping the tightness requirement, we can show that completion and loop nogoods characterize answer sets.

Theorem 3.4 *Let Π be a logic program, let Δ_Π and Λ_Π as in Theorem 3.3 and (1).*

Then, $X \subseteq atom(\Pi)$ is an answer set of Π iff $X = A^{\mathbf{T}} \cap atom(\Pi)$ for a (unique) solution A for $\Delta_\Pi \cup \Lambda_\Pi$.

The nogoods in $\Delta_\Pi \cup \Lambda_\Pi$ describe a set of constraints that must principally be checked for computing answer sets. While the size of Δ_Π is linear in $atom(\Pi) \times body(\Pi)$, the one of Λ_Π is exponential. These magnitudes apply to all existing ASP solvers, where Δ_Π is either encoded via dependency graphs (linking atoms and bodies/rules) or given through the Clark completion of Π. Loop nogoods in Λ_Π are determined only on demand by dedicated algorithms.

4 Conflict-Driven ASP Solving

Given the specification of ASP solving in terms of nogoods, we can now make use of advanced techniques from CSP and SAT for developing equally advanced ASP solving procedures. Different from SAT, where every (known) nogood is usually explicated as a clause, our algorithms work on logic programs, inducing several kinds of nogoods. In particular, the exponentially many nogoods resulting from loop formulas are implicitly given by a program, and determined only when used for propagation. The key role of the different kinds of (and partially implicit) constraints, expressed as nogoods, is to identify a reason responsible for deriving a literal by unit propagation. This makes the logical fundament of ASP solving the same as the one of CSP and SAT solving, so that we can directly apply similar reasoning strategies, without the need of a SAT conversion or proprietary designs.

To begin, we give a specification of our **nogood propagation** procedure in Algorithm 1. Propagation works on a program Π, a set ∇ of recorded nogoods, and an assignment A. First, we invoke LOCALPROPAGATION on Π and accumulated nogoods in ∇. This function adds unit-resulting literals to A, derived via nogoods either in Δ_Π or in ∇; that is, a fixpoint of unit propagation is computed. If LOCALPROPAGATION yields a violated nogood δ (line 4), then A cannot be extended to a solution. Also if Π is tight, all unfounded atoms are already falsified. In both cases, we are done with nogood propagation. Only if Π is non-tight, we check whether an *unfounded set* [van Gelder *et al.*, 1991] (accumulated in U) has to be falsified.

Algorithm 1: NOGOODPROPAGATION

Input : A program Π, a set ∇ of nogoods, and an assignment A.
Output: An extended assignment and set of nogoods.

```
1  U ← ∅                                          // set of unfounded atoms
2  loop
3      A ← LocalPropagation(Π, ∇, A)
4      if δ ⊆ A for some δ ∈ Δ_Π ∪ ∇ or Tight(Π) then
5          return (A, ∇)
6      else
7          U ← U \ A^F
8          if U = ∅ then U ← UnfoundedSet(Π, A)
9          if U = ∅ then return (A, ∇)
10         else let p ∈ U in
11             ∇ ← ∇ ∪ {λ(p, U)}
12             if Tp ∈ A then return (A, ∇)
13             else A ← A ∘ (Fp)
```

Initially, U is empty; so in line 8 we determine an unfounded set. Note that, if some non-false atom is unfounded, there always is an unfounded set not containing any false atoms. In Section 5, we describe our implementation of UNFOUNDEDSET; we here only require that an unfounded set U of non-false atoms is returned, if it exists. If so, we select in line 10 an atom p from U and add its loop nogood $\lambda(p, U)$ to ∇ (line 11).[2] If p is true, then $\lambda(p, U)$ is violated, and we return A and ∇ (line 12). Otherwise, $\mathbf{F}p$ is unit-resulting for $\lambda(p, U)$ wrt A, and we add $\mathbf{F}p$ to A (line 13). Having falsified a single element of U, we re-invoke LOCALPROPAGATION before adding any further loop nogoods. In fact, completion nogoods in Δ_Π might suffice for falsifying the residual atoms in U. For example, consider $U = \{x, y, z\}$ and rules $x \leftarrow z$, $y \leftarrow x$, $z \leftarrow y$: From $\mathbf{F}x$, we can derive $\mathbf{F}y$ and $\mathbf{F}z$. But generally, falsifying a single element does not allow for falsifying the whole set U only via completion nogoods. If we add rule $y \leftarrow z$ to the above example, then $\mathbf{F}y$ and $\mathbf{F}z$ are no longer derivable. This is reflected in line 7, where we remove false atoms from U. The shrunken set U is still unfounded, and if it is non-empty, we can immediately determine another loop nogood to falsify the next element of U. Observe that no further unfounded atoms are computed until the ones in U are expended. With changing set U, the atom p (selected in line 10) and the bodies in loop nogood $\lambda(p, U)$ change in each iteration, aiming at a firmer representation of the respective unfounded set.

All in all, our nogood propagation procedure interleaves unit propagation on completion and accumulated nogoods

[2] Given that p is unfounded, we have $\lambda(p, U) \setminus \{\mathbf{T}p\} \subseteq A$.

Algorithm 2: CDNL-ASP

Input : A program Π.
Output: An answer set of Π.

1 $A \leftarrow \emptyset$ *// assignment over $atom(\Pi) \cup body(\Pi)$*
2 $\nabla \leftarrow \emptyset$ *// set of (dynamic) nogoods*
3 $dl \leftarrow 0$ *// decision level*
4 **loop**
5 $(A, \nabla) \leftarrow$ NOGOODPROPAGATION(Π, ∇, A)
6 **if** $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ **then**
7 **if** $dl = 0$ **then return** no answer set
8 $(\delta, \sigma_{UIP}, k) \leftarrow$ CONFLICTANALYSIS$(\varepsilon, \Pi, \nabla, A)$
9 $\nabla \leftarrow \nabla \cup \{\delta\}$
10 $A \leftarrow A \setminus \{\sigma \in A \mid k < dl(\sigma)\}$
11 $dl \leftarrow k$
12 $A \leftarrow A \circ (\overline{\sigma_{UIP}})$
13 **else if** $A^{\mathbf{T}} \cup A^{\mathbf{F}} = atom(\Pi) \cup body(\Pi)$ **then**
14 **return** $A^{\mathbf{T}} \cap atom(\Pi)$
15 **else**
16 $\sigma_d \leftarrow$ SELECT(Π, ∇, A)
17 $dl \leftarrow dl + 1$
18 $A \leftarrow A \circ (\sigma_d)$

with the recording and propagation of loop nogoods. The latter is only done if the underlying program is non-tight and the falsity of unfounded atoms cannot be determined via other nogoods. Our approach favors local propagation over unfounded set computations. This is motivated by the fact that local propagation does not add any nogoods to ∇, hence, it is more economical than unfounded set falsification. We further discuss the relation between our propagation strategy and other approaches in Section 7.

Conflict-Driven Nogood Learning. Our basic algorithm for deciding whether a program has an answer set is similar to *Conflict-Driven Clause Learning* (CDCL) with *First-UIP* scheme [Mitchell, 2005]. Given a program Π, Algorithm 2 starts from an empty assignment A and an empty set ∇ of learned nogoods. Via the *decision level* dl, we count *decision literals*, i.e., the literals in A not derived by nogood propagation. The initial value of dl is 0, it is incremented before a decision literal is added to A. For a literal $\sigma \in A$, we access via $dl(\sigma)$ the decision level of σ, that is, the value dl had when σ was added to A. After encountering a conflict, the decision level is used to guide *backjumping*.

The loop of Algorithm 2 is similar to CDCL, so we here only sketch the principal steps. First, function NOGOODPROPAGATION deterministically extends A (and ∇) as described above. If this yields a conflict (line 6), function CONFLICTANALYSIS (see below) determines a conflict nogood δ to be recorded, a *unique implication point* (UIP) σ_{UIP}, and a decision level k to jump back to. Backjumping and nogood recording work as with CDCL, in particular, a conflict at decision level 0 indicates the non-existence of an answer set. If A is a solution (line 13), the atoms of Π that are true in A form an answer set of Π. Finally, if A is non-conflicting and partial, a decision literal σ_d is selected according to some heuristics (see Section 5 on further details) and added to A. Note that

Algorithm 3: CONFLICTANALYSIS

Input : A violated nogood δ, a program Π, a set ∇ of nogoods, and an assignment A.
Output: A derived nogood, a UIP, and a decision level.

1 **let** $\sigma \in \delta$ **st** $A = B \circ (\sigma) \circ B'$ **and** $\delta \setminus \{\sigma\} \subseteq B$
2 **while** $\{\rho \in \delta \mid dl(\rho) = dl(\sigma)\} \neq \{\sigma\}$ **do**
3 **let** $\varepsilon \in \Delta_\Pi \cup \nabla$ **st** $\overline{\sigma} \in \varepsilon$ **and** $\varepsilon \setminus \{\overline{\sigma}\} \subseteq B$
4 $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$
5 **let** $\sigma \in \delta$ **st** $B = C \circ (\sigma) \circ C'$ **and** $\delta \setminus \{\sigma\} \subseteq C$
6 $B \leftarrow C$
7 $k \leftarrow max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
8 **return** (δ, σ, k)

σ_d belongs to the new decision level $dl + 1$.

Our **conflict analysis** procedure determines an *asserting nogood* δ. That is, after backjumping, δ yields a unit-resulting literal, leading Algorithm 2 into a different part of the search space than traversed before. This is similar to an asserting clause, determined by conflict analysis in CDCL. In deriving δ, we follow the First-UIP scheme and stop conflict analysis at the first UIP that is found; no further UIPs are explored.

Though our conflict analysis procedure is similar to its classical CDCL counterpart, we need subtle adjustments. The reason is that unfounded set inference works in a directed way: It only falsifies unfounded atoms, but does not "protect" true atoms from becoming unfounded. For illustration, consider $\Pi = \{x \leftarrow not\ y\,; y \leftarrow not\ x\,; u \leftarrow x\,; u \leftarrow v\,; v \leftarrow u, y\}$ along with assignment $A = (\mathbf{T}u)$. Note that $\mathbf{T}u$ is a decision literal; its decision level is 1. Local propagation on Δ_Π and A yields no inferences (due to $body(u) = \{\{x\}, \{v\}\}$), and there is no unfounded set. When we extend A by decision literal $\mathbf{T}y$ at level 2, local propagation sets atom x and body $\{x\}$ to false (and v to true). But then, the set $\{u, v\}$ becomes unfounded, which makes us record the loop nogood $\delta = \lambda(u, \{u, v\}) = \{\mathbf{F}\{x\}, \mathbf{T}u\}$. Since A contains $\mathbf{F}\{x\}$ and $\mathbf{T}u$, nogood δ is violated. Also, δ contains only one literal added to A at decision level 2: $\mathbf{F}\{x\}$. Hence, $\mathbf{F}\{x\}$ is a UIP. In this example, the violated nogood δ is immediately asserting. A situation like this cannot occur in classical CDCL, where the initial violated clause always contains more than one literal from the current decision level. The difference to CDCL is caused by the directedness of unfounded set inference in ASP, which is "partial", in the sense that not all logical consequences are derived. In terms of a loop nogood $\{\mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k, \mathbf{T}p\}$, unfounded set inference can only derive $\mathbf{F}p$, but not $\mathbf{T}\beta_i$ for a body β_i ($1 \leq i \leq k$), at least as long as the loop nogood is not made explicit by recording it. For $\delta = \{\mathbf{F}\{x\}, \mathbf{T}u\}$ as above, unfounded set inference would have derived $\mathbf{F}u$ at decision level 1, if we had selected $\mathbf{F}\{x\}$ as the decision literal. However, it does not derive $\mathbf{T}\{x\}$ from assignment $(\mathbf{T}u)$, which is inferred by unit propagation once δ is available as an explicit constraint. (Undirected unfounded set inference is not yet algorithmically solved. Current algorithms only determine unfounded atoms, but not bodies that must be true according to an (implicit) loop nogood.)

Algorithm 3 shows our conflict analysis procedure. It works on an assignment A containing a violated nogood δ, either from the program Π and so in Δ_Π, or from the recorded nogoods in ∇. In line 1, we determine via σ the literal from δ added last to A. As mentioned above, σ might already be a UIP, that is, the single literal in δ from the current decision level. If σ is a UIP, we do not enter the while-loop in line 2. Otherwise, δ contains at least one literal other than σ from the current decision level. Note that, in this case, σ is not a decision literal. Hence, there is some nogood ε in Δ_Π or ∇ for which σ has been unit-resulting. Such an ε is determined in line 3, and in line 4 we resolve δ and ε into a new nogood δ. In line 5, we determine as new σ the literal from the new δ added last to A. In each iteration, σ moves closer to the front of A. Hence, we finally derive a nogood δ that contains exactly one literal σ from the current decision level; in the worst case, it is the decision literal. In line 7, we determine the decision level to jump back to as the maximum level of any literal in δ other than σ. Algorithm CONFLICTANALYSIS is very similar to the First-UIP scheme for CDCL. The difference is that conflict resolution might start from an asserting nogood.

5 The *clasp* System

Our new system *clasp* [2006] implements our approach to ASP solving. It combines the high-level modeling capacities of ASP with state-of-the-art techniques from the area of Boolean constraint solving. Unlike existing ASP solvers, *clasp* is originally designed and optimized for conflict-driven ASP solving. Rather than applying a SAT solver to a CNF conversion, *clasp* directly incorporates suitable data structures, particularly fitting backjumping and learning. This includes dedicated treatment of binary and ternary nogoods [Ryan, 2004], and watched literals for unit propagation on "long" nogoods [Moskewicz *et al.*, 2001]. Unlike *smodels*$_{cc}$ [Ward and Schlipf, 2004], which builds a material implication graph for keeping track of the multitude of inference rules found in ASP solving, *clasp* uses the more economical approach of SAT solvers: For a derived literal, it only stores a pointer to the responsible constraint in $\Delta_\Pi \cup \nabla$.

Unfounded set detection within *clasp* combines *smodels*' *source pointer* technique [Simons, 2000] with the unfounded set computation algorithm described in [Anger *et al.*, 2006]. It aims at small and "loop-encompassing", rather than greatest unfounded sets, as determined by *smodels* [Simons *et al.*, 2002] and *dlv* [Leone *et al.*, 2006]. Notably, *clasp* recognizes violated loop nogoods that are immediately asserting (cf. Section 4), so that the same nogood is not recorded twice.

The primary operation mode of *clasp* is conflict-driven nogood learning. Beyond backjumping and learning, *clasp* features a number of related techniques, typically found in CDCL-based SAT solvers. *clasp* incorporates restarts, deletion of recorded conflict and loop nogoods, and decision heuristics favoring literals from conflict nogoods. All these features are configurable via command line options. The default restart and nogood deletion policies are adopted from *MiniSat* [Eén and Sörensson, 2003]; the standard heuristics is an adjustment of *BerkMin* [Goldberg and Novikov, 2002]. Although Algorithm 2 details the search for one answer set, *clasp* also allows for enumerating answer sets. This is accomplished by interleaving backjumping with (systematic) backtracking: After a solution has been found, its decision literals can only be backtracked chronologically; backjumping is restricted for not repeating already enumerated solutions. This strategy avoids the generation of nogoods excluding entire solutions, as done for instance by *smodels*$_{cc}$ and *mchaff*[3].

clasp's second major operation mode runs (systematic) backtracking without learning. This is similar to the strategy of standard ASP solvers like *smodels*, using lookahead. Both operation modes are implemented in a uniform framework, which also allows us to evaluate the efficiency of advanced SAT implementation techniques, such as watched literals, in a standard ASP solver.

6 Experiments

We conducted experiments on a variety of problem classes. Our comparison considers *clasp* (RC2) in its two major modes: (a) the standard one using backjumping and learning, and (b) the systematic backtracking mode using lookahead but no learning. We refer to these variants as *clasp*$_a$ and *clasp*$_b$. As "traditional" ASP solver, we include *smodels* (2.31). Beyond some variations, *smodels*' strategy is similar to *clasp*$_b$. We also incorporate *assat* (2.02) and *cmodels* (2.12), both using *mchaff* (spelt3), and *smodels*$_{cc}$ (1.08). Among all compared solvers, *smodels*$_{cc}$ is closest to *clasp*$_a$. SAT-based solvers *assat* and *cmodels* convert a logic program into CNF and delegate the search for a supported model to *mchaff*. For tight programs, this approach amounts to *clasp* in mode (a). In the non-tight case, *assat* and *cmodels* delay checking loop nogoods until an assignment is total, while all other solvers integrate it into their propagation.

All experiments were run on a 2.2GHz PC on Linux. We report results in seconds, taking the average of 3 runs, each restricted to 900s time and 1GB RAM. A timeout is indicated by "—". All solvers were run with their default settings except for *smodels*$_{cc}$, for which we used option "nolookahead" as recommended by the developers. The instances used in our experiments as well as extended results (e.g. for *dlv* and *nomore++*, being excluded here due to lack of space) are available at [clasp, 2006]. In brief, the instances in Table 1 and 2 are from the areas of bounded model checking (1-5;31-35), DES cryptanalysis (6-10), blocksworld planning (11-12;42-45), Hamiltonian cycles in clumpy graphs (13-20), Hamiltonian paths for the Gryzzles game (21-25), Sokoban (26-30;46-55), and machine code superoptimization (36-41). The instances numbered 1-10 and 31-41 are tight, all others are non-tight.

Table 1 gives results for computing one answer set. On the tight instances 1-10, *clasp*$_a$ performs comparable to *assat* and *cmodels*. Sometimes it is even slightly faster, showing that the low-level implementation of *clasp* is competitive with state-of-the-art SAT solvers, doing most of the work for *assat* and *cmodels*. Regarding *smodels*, we see that its systematic backtracking approach does not scale very well; the same applies to *clasp* in mode (b). Instances 11 and 12 are tight on their supported models, that is, every supported model is also an

[3] http://www.princeton.edu/∼chaff/

nr	benchmark	*assat*	*cmodels*	$smodels_{cc}$	$clasp_a$	$clasp_b$	*smodels*
1	dp_10.formula1-i-O2-b12	0.54	11.17	9.33	0.51	299.15	228.93
2	dp_8.fsa-D-i-O2-b8	0.21	0.09	0.16	0.15	1.1	0.17
3	elevator_4-D-s-O2-b10	0.83	1.48	16.62	1.15	168	78.96
4	mmgt_3.fsa-D-i-O2-b10	3.28	5.65	45.5	3.21	66.85	331.89
5	mmgt_4.fsa-D-s-O2-b8	0.49	0.98	3.37	0.3	343.72	142.24
6	des-r3-p6-t1	1.82	1.79	1.28	0.93	—	821.1
7	des-r3-p7-t1	2	2.01	1.9	1.04	—	—
8	des-r3-p8-t1	2.28	2.69	3.54	1.26	—	—
9	des-r3-p9-t1	2.2	2.64	1.82	1.44	—	280.34
10	des-r3-p10-t1	3.3	2.96	2.4	1.62	—	—
11	p3_time8	0.91	0.99	17.43	2.81	15.63	30.87
12	p3_time9	1.03	1.18	23.75	3.2	21.38	18.23
13	clumpyhc12_12_08	4.3	3.73	0.29	0.2	—	53.69
14	clumpyhc12_12_10	3.66	4.48	0.26	0.22	—	0.74
15	clumpyhc14_14_08	10.73	8.86	1.59	0.25	—	—
16	clumpyhc14_14_10	27.08	43.18	0.95	0.32	—	—
17	clumpyhc16_16_08	203.05	25.87	6.23	0.46	—	—
18	clumpyhc16_16_10	23.99	18.99	1.42	1.62	—	—
19	clumpyhc18_18_08	—	—	3.13	23.73	—	—
20	clumpyhc18_18_10	—	—	1.61	1.23	—	—
21	gryzzles.6	2.16	117.69	0.23	0.11	23.02	15.46
22	gryzzles.32	2.85	3.63	0.25	0.14	106.24	11.09
23	gryzzles.36	145.74	19.31	0.72	0.55	—	—
24	gryzzles.41	3.57	2.23	0.18	0.11	10.05	25.75
25	gryzzles.47	7.76	46.87	1.23	0.33	—	—
26	yorick.51.n11.len11	36.53	104.98	36.26	6.17	107.02	211.44
27	yoshio.11.n15.len15	—	330.93	64.1	13.34	—	—
28	yoshio.16.n13.len13	84.57	111.11	52.74	8.04	126.18	303.94
29	yoshio.33.n12.len12	15.4	16	9.32	9.06	—	—
30	yoshio.50.n10.len10	22.21	13.57	14.78	1.05	2.69	94.52

Table 1: Experiments computing *one* answer set.

answer set. As unfounded set checks produce unnecessary overhead here, *assat* and *cmodels* are a bit faster than $clasp_a$. Looking at the Hamiltonian problems in 13-25, we see that $smodels_{cc}$ and $clasp_a$ scale best. They outperform *assat* and *cmodels* by some orders of magnitude; on two clumpy graphs, *assat* and *cmodels* even time out (viz. 19 and 20). Both *smodels* and $clasp_b$ are ineffective on Hamiltonian problems and time out on most of the instances. On Sokoban problems in 26-30, $clasp_a$ outperforms the other solvers. Only *cmodels* and $smodels_{cc}$ never time out, but they are much slower.

Table 2 shows results for computing all answer sets, or for determining that no answer sets exist (0 #sol). Given that *assat* cannot enumerate answer sets, we only include it on unsatisfiable programs. On satisfiable instances 31-35, we see that $clasp_a$ is relatively fast enumerating all answer sets. The superoptimization instances in 36-41 are easily determined unsatisfiable by *assat*, *cmodels*, and $clasp_a$. Both *smodels* and $clasp_b$ show a clear exponential behavior and finally time out. Surprisingly, $smodels_{cc}$ scales worst, that is, several orders of magnitude behind other learning solvers and timing out even before non-learning ones. We conjecture that this is because $smodels_{cc}$ does, differently from other learning solvers, not include rule bodies in conflict nogoods. Looking at the satisfiable instances among 42-55, we see that $clasp_a$ is faster enumerating all answer sets than any other solver we tested. Unsatisfiable blocksworld problems 42 and 45 are most effectively solved by *assat* and *cmodels*. (Note that instances 42-45 are tight on their supported models.) Like with computing one answer set on (satisfiable) Sokoban problems in 26-30, $clasp_a$ is fastest on both satisfiable and unsatisfiable problems in 46-55.

nr	benchmark	#sol	*assat*	*cmodels*	$smodels_{cc}$	$clasp_a$	$clasp_b$	*smodels*
31	dp_10.formula1-i-O2-b12	12600	n/a	22.38	179.74	76.21	—	—
32	dp_8.fsa-D-i-O2-b8	40320	n/a	39.8	78	13.06	40.12	16.53
33	elevator_4-D-s-O2-b10	6240	n/a	14.32	40.48	6.53	171.42	97.58
34	mmgt_3.fsa-D-i-O2-b10	288	n/a	97.26	199.43	28.1	98.77	395.99
35	mmgt_4.fsa-D-s-O2-b8	1344	n/a	28.67	62.15	12.56	417.76	248
36	test12	0	0.09	0.09	0.57	0.08	2.01	0.9
37	test14	0	0.12	0.1	10.96	0.1	10.8	4.21
38	test16	0	0.16	0.12	200.4	0.14	53.56	18.56
39	test18	0	0.19	0.15	—	0.18	260.38	86.57
40	test20	0	0.23	0.18	—	0.23	—	407.35
41	test22	0	0.29	0.21	—	0.27	—	—
42	p3_time7	0	0.75	0.92	12.23	2.29	10.61	14.06
43	p3_time8	28	n/a	16.54	18.43	2.93	25.15	53.38
44	p3_time9	3374	n/a	—	56.69	14.75	163.68	417.61
45	p4_time5	0	1.74	1.62	4.41	8.03	8.46	4.1
46	yorick.51.n11.len10	0	22.81	27.08	11.48	2.08	410.46	836.24
47	yorick.51.n11.len11	512	n/a	—	41.01	11.12	—	—
48	yoshio.11.n15.len14	0	185.47	162.14	38.18	12.11	—	—
49	yoshio.11.n15.len15	512	n/a	—	87.95	19.57	—	—
50	yoshio.16.n13.len12	0	57.94	38.18	30.04	3.82	—	882.44
51	yoshio.16.n13.len13	32	n/a	—	50.81	8.51	—	—
52	yoshio.33.n12.len11	0	16.21	23.69	33.43	8.81	860.09	688.21
53	yoshio.33.n12.len12	114176	n/a	—	—	622.93	—	—
54	yoshio.50.n10.len9	0	33.98	23.25	8.66	2.13	42.38	27.99
55	yoshio.50.n10.len10	384	n/a	—	17.29	4.82	172.11	94.57

Table 2: Experiments computing *all* answer sets.

Overall, we notice a huge gap between learning and non-learning solvers: The latter frequently time out, while the best among the former solve the same instances within seconds (cf. Table 1). The short run-times of learning solvers do sometimes not permit a reliable comparison between them. We however see a clear distinction between different concepts: Problems that are intractable for systematic backtracking methods are often easily dealt with using backjumping and learning. Note that all our benchmarks are structured to some extent. This is useful for learning solvers, as structure can be explicated via learned constraints. The picture might be different on unstructured random problems as, e.g., reported in the SAT literature.

7 Discussion

We have provided a uniform approach to ASP solving, allowing for a transparent technology transfer from CSP and SAT. The idea is to view ASP inferences as unit propagation on nogoods, reflecting constraints from program rules, unfounded sets, and conflicts. We have seen that SAT translations are unnecessary for applying techniques found in SAT solvers.

In contrast to SAT, ASP induces further implicit constraints given by loop nogoods. Though inherently present, these nogoods need only be explicated when used for propagation and conflict analysis. Thus, sophisticated unfounded set checks still work on the logical fundament of CSP and SAT. Based on this perception, we have provided a conflict-driven algorithm for ASP solving, using state-of-the-art SAT solving techniques. Notably, our approach favors local propagation on explicit nogoods over unfounded set checks, which explicate inherent loop nogoods that give rise to unit propagation. In fact, many of the combinatorially constructable loop nogoods might be redundant, that is, entailed by completion and other loop nogoods. (For tight programs, the loop nogoods of all non-singletons are redundant.) In this respect, our ap-

proach guarantees that only non-redundant loop nogoods are used for propagation and, particularly, for conflict analysis.

We have implemented our approach in the *clasp* system. Our empirical results show that *clasp* is competitive with existing ASP solvers. The *clasp* system directly incorporates state-of-the-art techniques from Boolean constraint solving, avoiding a SAT translation as it is done by *assat*, *cmodels*, and *sag* [Lin *et al.*, 2006]. Also, *clasp* records loop nogoods only when ultimately needed for unit propagation; this is different from *assat* and *sag*, which determine loop formulas for all "terminating" loops. Unlike genuine ASP solvers *smodels* and *dlv*, *clasp* does not determine greatest unfounded sets. Rather, it applies local propagation directly after an unfounded set has been found. Different from *dlv* with backjumping [Ricca *et al.*, 2006] and $smodels_{cc}$, the inclusion of rule bodies in nogoods allows for a straightforward extension of unit propagation to ASP, abolishing the need for multiple inference rules. Notably, *clasp* can enumerate answer sets of a program without explicitly prohibiting already computed solutions by nogoods, as done by *cmodels* and $smodels_{cc}$.

Acknowledgments

The authors are grateful to Wolfgang Faber, Yuliya Lierler, and Ilkka Niemelä for helpful comments on previous drafts of this paper.

References

[Anger *et al.*, 2006] C. Anger, M. Gebser, and T. Schaub. Approaching the core of unfounded sets. In *Proceedings of the 11th International Workshop on Nonmonotonic Reasoning*, pages 58–66. Clausthal University of Technology, 2006.

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

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

[Clark, 1978] K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[clasp, 2006] `http://www.cs.uni-potsdam.de/clasp`.

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

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[Fages, 1994] 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.

[Gebser and Schaub, 2006] M. Gebser and T. Schaub. Tableau calculi for answer set programming. In *Proceedings of the 22nd International Conference on Logic Programming*, pages 11–25. Springer, 2006.

[Giunchiglia *et al.*, 2006] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 2006. To appear.

[Goldberg and Novikov, 2002] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the 5th Conference on Design, Automation and Test in Europe*, pages 142–149, 2002.

[Lee, 2005] J. Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 503–508. Professional Book Center, 2005.

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

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

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

[Lin *et al.*, 2006] Z. Lin, Y. Zhang, and H. Hernandez. Fast SAT-based answer set solver. In *Proceedings of the 21st National Conference on Artificial Intelligence*. AAAI Press/The MIT Press, 2006.

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

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation*, pages 530–535, 2001.

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

[Ryan, 2004] L. Ryan. Efficient algorithms for clause-learning SAT solvers. MSc, Sim. Fraser University, 2004.

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

[Simons, 2000] P. Simons. *Extending and Implementing the Stable Model Semantics*. Dissertation, Helsinki UT, 2000.

[van Gelder *et al.*, 1991] 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.

[Ward and Schlipf, 2004] J. Ward and J. Schlipf. Answer set programming with clause learning. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 302–313. Springer, 2004.