Answer Set Solving with Lazy Nogood Generation

Christian Drescher and Toby Walsh

NICTA* and the University of New South Wales

Abstract

Although Answer Set Programming (ASP) systems are highly optimised, their performance is sensitive to the size of the input and the inference it encodes. We address this deficiency by introducing a new extension to ASP solving. The idea is to integrate external propagators to represent parts of the encoding implicitly, rather than generating it a-priori. To match the state-of-the-art in conflict-driven solving, however, external propagators can make their inference explicit on demand. We demonstrate applicability in a novel Constraint Answer Set Programming system that can seamlessly integrate constraint propagation without sacrificing the advantages of conflict-driven techniques. Experiments provide evidence for computational impact.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Conflict-Driven Nogood Learning, Constraint Answer Set Programming, Constraint Propagation, Lazy Nogood Generation

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.188

1 Introduction

Developing a powerful paradigm for declarative problem solving is one of the key challenges in the area of knowledge representation and reasoning. A promising candidate is Answer Set Programming (ASP; [24, 16, 33, 43, 37, 2]) which builds on Logic Programming and Nonmonotonic Reasoning. Its success depends on two factors: efficiency of the solving capacities, and modelling convenience. Efficient ASP solvers [26, 22, 36, 32] match the state-of-the-art in conflict-driven solving [41], including conflict-driven learning, lookback-based heuristics, and backjumping. However, their performance is sensitive to the size of problem encodings which can quickly become infeasible, for instance, through the worst-case exponential number of loops in a logic program [34], or constructs that are naturally non-propositional, like constraints over finite domains. A variety of extensions to ASP have been proposed that deal with some of these issues via integration of other declarative problem solving paradigms. Recently, for example, we have witnessed the development of Constraint Answer Set Programming (CASP) that integrates Constraint Programming (CP) with ASP, supporting constraints over finite domains, and most importantly, global constraints. While this approach certainly increases modelling convenience and can drastically decrease the size of an encoding, it does not fully carry over to conflict-driven solving technology [12].

We address this problem and present a new computational extension to ASP solving, called Lazy Nogood Generation. Motivated by the success of Lazy Clause Generation [46] in Constraint Satisfaction Problem (CSP) solving, the key idea is to generate (parts of) the problem encoding on demand, only when new information can be propagated. We make several contributions to the study of Lazy Nogood Generation in ASP. First, we lay the foundations of external propagation based on a uniform characterisation of answer

* NICTA is funded by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 188–200

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

sets in terms of nogoods. This provides the underpinnings to represent conditions on the answer sets of a logic program without encoding the entire problem a-priori. However, external propagators can make parts of the encoding explicit, in particular, when they can trigger inference. As we shall see, our techniques generalise existing ones, e.g., loop formula propagation [22], and weight constraint rule propagation [21]. Second, we specify a decision procedure for ASP solving with Lazy Nogood Generation. It is centred around conflict-driven solving and integrates external propagation. Third, we demonstrate applicability. We show how to seamlessly integrate constraint propagation with our framework, resulting in a novel approach to CASP solving. Finally, we empirically evaluate a prototypical implementation and compare to the state-of-the-art in ASP and CASP solving.

2 Background

Many tasks from the declarative problem solving domain can be defined as CSP, that is a tuple (V, D, C) where V is a finite set of *constraint variables*, each $v \in V$ has an associated finite *domain* $dom(v) \in D$, and C is a set of constraints. A *constraint* c is a k-ary relation, denoted $R(c)$, on the domains of the variables in $S(c) \in V^k$. A *(constraint variable) assignment* is a function A that assigns to each variable $v \in V$ a value from $dom(v)$. For a constraint c with $S(c) = (v_1, \ldots, v_k)$ define $A(S(c)) = (A(v_1), \ldots, A(v_k))$. The constraint c *satisfied* if $A(S(c)) \in R(c)$. Otherwise, we say that c is *violated*. Let $C^A = \{c \in C \mid A(S(c)) \in R(c)\}$. An assignment A is a *solution* iff $C = C^A$. CP systems are oriented towards solving CSP and typically interleave backtracking search to explore assignments with *constraint propagation* to prune the set of values a variable can take. The effect of constraint propagation is studied in terms of *local consistency*. E.g., a binary constraint c is called *arc consistent* iff a variable in $S(c)$ is assigned any value, there exists a value in the domain for the other variable in $S(c) \setminus \{v\}$ such that c is not violated. An n-ary constraint c is called *domain consistent* iff $v \in S(c)$ is assigned any value, there exist values in the domains of all other variables in $S(c) \setminus \{v\}$ such that c is not violated. Observe that, in general, a constraint propagator that enforces domain consistency prunes more values than one that enforces arc consistency on a binary decomposition of the original constraint. CSPs can be encoded with ASP [43], which is founded on Logic Programming.

A *(normal) logic program* P over an alphabet $\mathcal{A}$ is a finite set of *rules* r of the form $a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$ where $a_i \in \mathcal{A}$ are *atoms* for $0 \leq i \leq n$. A *default literal* is an atom a or its *default negation* $\sim a$. The atom $H(r) = a_0$ is called the *head* of r and the set of default literals $B(r) = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$ is called the *body* of r. For a set of default literals S, define $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid \sim a \in S\}$. For restricting S to atoms $\mathcal{E}$, define $S|_{\mathcal{E}} = \{a \mid a \in S^+ \cap \mathcal{E}\} \cup \{\sim a \mid a \in S^- \cap \mathcal{E}\}$. For $X \subseteq \mathcal{A}$ define *external support* for X as $ES_P(X) = \{B(r) \mid r \in P,\ H(r) \in X,\ B(r)^+ \cap X = \emptyset\}$. The set of atoms occurring in P is denoted by $At(P)$, and the set of bodies in P is $B(P) = \{B(r) \mid r \in P\}$. For regrouping rules sharing the heads in $X \subseteq \mathcal{A}$, define $P_X = \{r \in P \mid H(r) \in X\}$, and for bodies sharing the same head a, define $B(a) = \{B(r) \mid r \in P,\ H(r) = a\}$. A *logic program with externals over* $\mathcal{E}$ is a logic program P over an alphabet distinguishing regular atoms $\mathcal{A}$ and external atoms $\mathcal{E}$, such that $H(r) \in \mathcal{A}$ for each $r \in P$. Let $Y \subseteq \mathcal{E}$. For a logic program P over externals from $\mathcal{E}$ define the *pre-reduct* $P(Y) = \{H(r) \leftarrow B(r)|_{\mathcal{A}\setminus\mathcal{E}} \mid r \in P,\ B(r)^+|_{\mathcal{E}} \subseteq Y,\ B(r)^-|_{\mathcal{E}} \cap Y = \emptyset\}$. A *splitting set* for a logic program P [35] is a set $\mathcal{E} \subseteq \mathcal{A}$ if $H(r) \in \mathcal{E}$ then $B(r)^+ \cup B(r)^- \subseteq \mathcal{E}$ for each $r \in P$. Observe that, if $\mathcal{E}$ is a splitting set of P, it *splits* P into a logic program $P_{\mathcal{E}}$ over $\mathcal{E}$ and a logic program $P_{\mathcal{A}\setminus\mathcal{E}}$ with externals over $\mathcal{E}$. The semantics of a logic program P is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of P, if X is a minimal model

of the *reduct* $P^X = \{H(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$ [24]. Let $\mathcal{E}$ be a splitting set of P. The set $Z \subseteq \mathcal{A}$ is an answer set of P iff $Z = X \cup Y$ such that X is an answer set of $P_{\mathcal{E}}$ and Y is an answer set of $P_{\mathcal{A}\setminus\mathcal{E}}(Y)$ (Splitting Set Theorem, [35]). Although our semantics is propositional, modern ASP systems support *non-ground* logic programs and construct atoms in $\mathcal{A}$ from a first-order signature via a *grounding* process, systematically substituting all occurrences of first-order variables by terms, resulting in a *(ground) instantiation*.

Following [22], the answer sets of a logic program P can be characterised as Boolean assignments over $At(P) \cup B(P)$ that do not conflict with the conditions induced by the *completion* [9] and all *loop formulas* of P [30], expressed in terms of nogoods [11]. Formally, a *(Boolean) assignment* $\mathbf{A}$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of *(signed) literals* σ_i of the form $\mathbf{T}a$ or $\mathbf{F}a$ where a is in the scope of $\mathbf{A}$, e.g., $S(\mathbf{A}) = At(P) \cup B(P)$. The complement of a literal σ is denoted $\overline{\sigma}$. True and false variables in $\mathbf{A}$ are accessed via $\mathbf{A}^{\mathbf{T}}$ and $\mathbf{A}^{\mathbf{F}}$, respectively. A *nogood* represents a set $\delta = \{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a condition *conflicting* with any assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$. If $\delta \setminus \mathbf{A} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{A}$, we say that δ is *unit* and *asserts* the *unit-resulting* literal $\overline{\sigma}$. A *total* assignment, that is $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = S(\mathbf{A})$ and $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$, is a *solution* for a set of nogoods Γ if $\delta \not\subseteq \mathbf{A}$ for each $\delta \in \Gamma$.

3 Nogoods of Logic Programs with Externals

We generalise [22] and describe nogoods capturing completion and loop formulas for a logic program P with externals over $\mathcal{E}$. For $\beta = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\} \in B(P)$, define

$$\Delta_\beta = \left\{ \begin{array}{l} \{\mathbf{T}a_1, \ldots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \ldots \mathbf{F}a_n, \mathbf{F}\beta\}, \\ \{\mathbf{F}a_1, \mathbf{T}\beta\}, \ldots, \{\mathbf{F}a_m, \mathbf{T}\beta\}, \{\mathbf{T}a_{m+1}, \mathbf{T}\beta\}, \ldots, \{\mathbf{T}a_n, \mathbf{T}\beta\} \end{array} \right\}.$$

Intuitively, the nogoods in Δ_β enforce the truth of body β iff all its elements are satisfied. For an atom $a \in At(P)$ with $B(a) = \{\beta_1, \ldots, \beta_k\}$, define

$$\Delta_a = \left\{ \ \{\mathbf{T}\beta_1, \mathbf{F}a\}, \ldots, \{\mathbf{T}\beta_k, \mathbf{F}a\}, \{\mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k, \mathbf{T}a\} \ \right\}.$$

Let $\Delta_P^{\mathcal{E}} = \bigcup_{\beta \in B(P)} \Delta_\beta \cup \bigcup_{a \in At(P)\setminus\mathcal{E}} \Delta_a$. The solutions for $\Delta_P^{\emptyset}$ correspond to the models of the completion of P [22]. To capture the effect of loop formulas induced by a set $L \subseteq At(P) \setminus \mathcal{E}$, for $a \in L$ define $\lambda(a, L) = \{\{\mathbf{T}a\} \cup \{\mathbf{F}\beta \mid \beta \in ES_P(L)\}\}$. The set of loop nogoods is $\Lambda_P^{\mathcal{E}} = \bigcup_{L \subseteq At(P)\setminus\mathcal{E}, L \neq \emptyset} \{\lambda(a, L) \mid a \in L\}$. Let P be a logic program and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution for $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ such that $\mathbf{A}^{\mathbf{T}} \cap At(P) = X$ [22]. We combine this result with the Splitting Set Theorem [35].

▶ **Proposition 1.** *Let P be a logic program, $\mathcal{E}$ a splitting set for P, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta_{P_{\mathcal{E}}}^{\emptyset} \cup \Lambda_{P_{\mathcal{E}}}^{\emptyset} \cup \Delta_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}} \cup \Lambda_{P_{\mathcal{A}\setminus\mathcal{E}}}^{\mathcal{E}}$ such that $\mathbf{A}^{\mathbf{T}} \cap (At(P_{\mathcal{E}}) \cup At(P_{\mathcal{A}\setminus\mathcal{E}})) = X$.*

An efficient algorithm for computing solutions to $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ is Conflict-Driven Nogood Learning (CDNL, [22]). It combines search and propagation by recursively assigning the value of a proposition and performing unit-propagation to determine its consequences [41].

4 Lazy Nogood Generation

Instead of generating all nogoods $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ a-priori, referred to as *eager* encoding, we introduce external propagators to generate nogoods on demand, i.e., only when they are able to propagate new information. We call this technique *Lazy Nogood Generation*, generalising an approach to encoding constraints over finite domains into sets of clauses by executing constraint propagation during SAT search and recording the propagation in terms of clauses (Lazy Clause Generation; [46]). Formally, an *external propagator* for a set of

nogoods Γ is a function π that maps a Boolean assignment to a subset of Γ such that for each total assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \subseteq \mathbf{A}$ for some $\delta' \in \pi(\mathbf{A})$. In other words, an external propagator generates a conflicting nogood from Γ iff some nogood in Γ is conflicting with the total assignment. We call an external propagator *conflict-optimal*, if this condition holds for each (partial) assignment. Notice that, even for a conflict-optimal external propagator, unit-propagation on Γ can infer more unit-resulting literals than unit-propagation on lazily generated nogoods. To close this gap, we define inference-optimal external propagators. An external propagator π for a set of nogoods Γ is *inference-optimal* if π is conflict-optimal and for each non-conflicting assignment $\mathbf{A}$ if $\delta \setminus \mathbf{A} = \{\sigma\}$ such that $\overline{\sigma} \notin \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \setminus \mathbf{A} = \{\sigma\}$ for some $\delta' \in \pi(\mathbf{A})$. The correspondence between external propagation and the set of nogoods it represents can be formalised as follows.

▶ **Proposition 2.** *Let Δ be a set of nogoods, and π be an external propagator for $\Gamma \subseteq \Delta$. Then, the assignment $\mathbf{A}$ is a solution of Δ iff $\mathbf{A}$ is a solution of $(\Delta \setminus \Gamma) \cup \pi(\mathbf{A})$.*

One of the advantages of Lazy Nogood Generation over eager encodings is space efficiency. For instance, the worst-case exponential number of loops in a logic program P makes an eager encoding of the conditions induced by $\Lambda^{\emptyset}_{P}$ infeasible [34]. Non-optimal external propagation, however, can check whether an *unfounded set* [50] has to be falsified in linear time [7], and determines nogoods in $\Lambda^{\emptyset}_{P}$ on demand via directed unfounded set inference [22]. To reflect Lazy Nogood Generation also on the language level of ASP, we make use of *splitting* [35] for outsourcing conditions over $\mathcal{E} \subseteq \mathcal{A}$ into $P_{\mathcal{E}}$. Instead of making $P_{\mathcal{E}}$ explicit, however, a set of external propagators Π can be provided that precisely represent the conditions induced by $P_{\mathcal{E}}$. We will write $At(\Pi)$ to access $\mathcal{E}$. The previous propositions yield the following result.

▶ **Theorem 3.** *Let P be a logic program, $\mathcal{E}$ a splitting set for P, Π a set of external propagators for $\Delta^{\emptyset}_{P_{\mathcal{E}}} \cup \Lambda^{\emptyset}_{P_{\mathcal{E}}}$, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta^{\mathcal{E}}_{P_{\mathcal{A}\setminus\mathcal{E}}} \cup \Lambda^{\mathcal{E}}_{P_{\mathcal{A}\setminus\mathcal{E}}} \cup \bigcup_{\pi\in\Pi} \pi(\mathbf{A})$ s.t. $\mathbf{A}^{\mathbf{T}} \cap (At(P_{\mathcal{E}}) \cup At(\Pi)) = X$.*

External propagation provides a form of modularity that allows programmers to select encodings which propagate better, but were previously avoided for space-related reasons. E.g., in [12] we describe eager encodings that simulate constraint propagators for the ALL-DIFFERENT constraint which achieve arc, bound, or range consistency. A constraint propagator that can achieve domain consistency exists [48] but it cannot be simulated efficiently [6]. Because of the fact that external propagators generate nogoods only on demand, however, we can implicitly represent encodings via Lazy Nogood Generation that are otherwise infeasible.

5 Conflict-Driven Nogood Learning with Lazy Nogood Generation

We develop a decision procedure for answer set solving with Lazy Nogood Generation based on CDNL [22]. It is centred around *conflict analysis* according to the *First-UIP* scheme [41]. That is, a conflicting nogood is iteratively resolved against other nogoods until a conflicting nogood that contains a *unique implication point* is obtained. This guides backjumping. Recording the resolved nogood enables conflict-driven learning, which can further prune the search space. For controlling the set of recorded nogoods, deletion strategies can be applied (cf. [42]). In contrast to CDNL we will integrate external propagators that perform Lazy Nogood Generation in order to represent conditions on the answer sets of a logic program that are not encoded eagerly. Much like their eager counterpart, lazily generated nogoods can contribute to conflict analysis and lookback-based search heuristics. This can improve propagation. Different to eagerly encoded nogoods, however, the amount of lazily generated nogoods can be controlled via deletion.

Input : A logic program P with external propagators Π.
Output : An answer set of P if one exists.

```
1  A ← ∅                                                    // Boolean assignment
2  ∇ ← ∅                                                    // set of recorded nogoods
3  dl ← 0                                                   // decision level
4  loop
5      (A, ∇) ← Propagation(P, Π, ∇, A)
6      if δ ⊆ A for some δ ∈ Δ_P^{At(Π)} ∪ ∇ then
7          if dl = 0 then return no answer set
8          (ε, k) ← ConflictAnalysis(δ, P, ∇, A)
9          ∇ ← ∇ ∪ {ε}
10         A ← A \ {σ ∈ A | k < dl(σ)}
11         dl ← k
12     else if A^T ∪ A^F = At(P) ∪ B(P) ∪ At(Π) then
13         return A^T ∩ (At(P) ∪ At(Π))
14     else
15         σ_d ← Select(P, Π, ∇, A)
16         A ← A ∘ (σ_d)
17         dl ← dl + 1
```

Algorithm 1 CDNL-LNG.

5.1 Main Algorithm

Algorithm 1 specifies our main procedure, CDNL-LNG. It takes a logic program P with external propagators Π, and starts with an empty assignment $\mathbf{A}$ and an empty set ∇ that will store recorded nogoods, including lazily generated nogoods. The *decision level* dl is initialised with 0. Its purpose is counting *decision literals* in the assignment. We use $dl(\sigma)$ to access the decision level of literal σ. The following loop is very similar to CDNL. First, Propagation (Line 5) extends $\mathbf{A}$ and ∇, as described in the next section. If this encounters a conflict (Line 6), the ConflictAnalysis procedure generates a conflicting nogood ε by exploiting interdependencies between nogoods in $\Delta_P^{At(\Pi)} \cup \nabla$ through conflict resolution, and determines a decision level k to continue search at. Then, ε is added to the set of recorded nogoods ∇ in Line 9. This can prune the search space and lead to faster propagation. Lines 10–11 account for backjumping to level k. Thereafter ε is unit and triggers inference in the next round of propagation. If ConflictAnalysis, however, yields a conflict at level 0, no answer set exists (Line 7). Furthermore, we distinguish the cases of a complete assignment (Lines 12–13) and a partial one (Lines 14–17). In case of a complete assignment, the atoms in $\mathbf{A}^\mathbf{T}$ correspond to an answer set of P. In the other case, $\mathbf{A}$ is partial and no nogood is conflicting. Then, a decision literal σ_d is selected by some heuristic, added to $\mathbf{A}$, and the decision level is incremented. While ConflictAnalysis and Select are similar to the ones in CDNL, we extend Propagation to accommodate Lazy Nogood Generation.

5.2 Propagation

A specification of our Propagation procedure is shown in Algorithm 2. It works on a logic program P with external propagators Π, a set of recorded nogoods ∇, and an assignment $\mathbf{A}$. Propagation interleaves unit-propagation on nogoods $\Delta_P^{At(\Pi)}$ and recorded nogoods ∇ including lazily generated nogoods from external propagators. We start with

Input : A logic program P with external propagators Π, recorded nogoods ∇, Boolean assignment $\mathbf{A}$.
Output : An extended assignment and set of recorded nogoods.

```
1  loop
2      repeat                                                    // unit-propagation
3          if δ ⊆ A for some δ ∈ Δ_P^{At(Π)} ∪ ∇ then return (A, ∇)
4          Σ ← {δ ∈ Δ_P^{At(Π)} ∪ ∇ | δ\A = {σ}, σ̄ ∉ A}
5          if Σ ≠ ∅ then let σ ∈ δ\A for some δ ∈ Σ in
6              A ← A ∘ (σ̄)
7      until Σ = ∅
8      foreach π ∈ Π do
9          Σ ← π(A)                                              // external propagation
10         if Σ ≠ ∅ then break
11     if Σ = ∅ then
12         Σ ← LoopFormulaPropagation(P, A)                      // loop formula propagation
13     if Σ = ∅ then return (A, ∇)
14     ∇ ← ∇ ∪ Σ
```

Algorithm 2 Propagation.

unit-propagation (Lines 2–7), resulting either in a conflict, i.e., some nogood is conflicting (Line 3), or in a fixpoint possibly extending $\mathbf{A}$ with unit-resulting literals. If there is no conflict, Propagation performs external propagation following some priority (Lines 8–10). Based on $\mathbf{A}$, each propagator may encode inference in a set of lazily generated nogoods Σ which is added to the set of recorded nogoods ∇ at the end of the loop in Line 14. The LoopFormulaPropagation procedure (Line 12; [22]) works similarly to ensure that no loop formula is violated, i.e., no loop nogood in $\Lambda_P^{At(\Pi)}$ is conflicting. This only has an effect if the logic program is non-tight [18]. Note that external propagation is interleaved by unit-propagation in order to assign unit-resulting literals immediately and detect conflicts early. Our algorithm also favours external propagation over loop formula propagation, motivated by the fact that external propagators can affect the assignment to atoms in $At(\Pi)$, possibly falsifying external support for a loop in P.

6 Constraint Answer Set Solving via Lazy Nogood Generation

One difficult task for ASP solving with Lazy Nogood Generation remains, i.e., to design efficient external propagators. A research area that is largely concerned with efficient propagation is CP. We here follow the idea from [46] and apply CP techniques to generate lazy nogoods representing constraints over finite domains. To reflect this on the language level, we make use of CASP, a paradigm that naturally merges CP and ASP.

CASP abstracts from non-propositional constraints by incorporating *constraint atoms* into logic programs. We access the constraint atom associated to a constraint c via the function $At(c)$. A *constraint logic program* is a tuple $\mathbb{P} = (V, D, C, P)$, where V, D, C are the same as in the definition of a CSP, and P is a logic program with externals over *constraint atoms* $\mathcal{C} = \{At(c) \mid c \in C\}$. A fundamental difference to traditional CP is that, in CASP, each constraint c is reified via $At(c)$. Its truth value is determined by the conditions induced by P and an assignment A to the variables in $S(c)$. The set of constraint atoms $\mathcal{C}^A = \{At(c) \mid$

Input : A Boolean assignment $\mathbf{A}$.
Output : A set of lazily generated nogoods.

1 $\nabla \leftarrow \emptyset$ *// set of lazily generated nogoods*
2 **if** $\mathbf{T}val(v,i) \in \mathbf{A}$ for some $i \in dom(v)$ **then**
3 $\quad \nabla \leftarrow \{\{\mathbf{T}val(v,i),\ \mathbf{T}val(v,j)\} \mid j \in dom(v) \backslash \{i\},\ \mathbf{F}val(v,j) \notin \mathbf{A}\}$
4 **if** $\mathbf{T}val(v,i) \notin \mathbf{A}$ for some $i \in dom(v) \wedge \forall j \in dom(v) \backslash \{i\}\ \mathbf{F}val(v,j) \in \mathbf{A}$ **then**
5 $\quad \nabla \leftarrow \{\{\mathbf{F}val(v,i) \mid i \in dom(v)\}\}$
6 **return** ∇

Algorithm 3 An external propagator for the value encoding Γ_v.

$c \in C^A\}$ correspond to the constraints satisfied by A. Let $\mathbb{P}$ be a constraint logic program and A an assignment. The pair (X, A) is a *constraint answer set* of $\mathbb{P}$ iff X is an answer set of $P(\mathcal{C}^A)$ (cf. [23]). Given that assignments A and their effect on each constraint can be represented in a logic program [43], the task of computing constraint answer sets can be reduced to the one of computing answer sets of P with external propagators for generating assignments A and capturing the inference of constraint propagation.

To begin with, CASP solving via Lazy Nogood Generation requires a propositional representation of assignments to constraint variables. In the *value encoding*, an atom $val(v,i)$, representing $v = i$, is introduced for each variable $v \in V$ and value $i \in dom(v)$. Intuitively, the atom $val(v,i)$ is true if v takes the value i, and false if v takes a value different from i (cf. [51]). To insure that an assignment $\mathbf{A}$ represents a consistent set of possible values for v, we encode the conditions that v must not take two values, i.e., $\{\mathbf{T}val(v,i), \mathbf{T}val(v,j)\} \not\subseteq \mathbf{A}$ for all $i, j \in dom(v),\ i \neq j$, and that v must take at least one value, i.e., $\mathbf{F}val(v,i) \notin \mathbf{A}$ for some $i \in dom(v)$, in the set of nogoods $\Gamma_v = \{\{\mathbf{T}val(v,i), \mathbf{T}val(v,j)\} \mid i, j \in dom(v),\ i \neq j\} \cup \{\{\mathbf{F}val(v,i) \mid i \in dom(v)\}\}$ [12]. We employ external propagators to represent the nogoods in Γ_v. Algorithm 3 provides a specification of an inference-optimal external propagator for this task. It takes a Boolean assignment $\mathbf{A}$ and returns a set of lazily generated nogoods, initialised in Line 1, that are unit or conflicting. Lines 2–3 insure that if v is assigned a value i then all other values are removed from its domain, while Lines 4–5 deal with the condition that there is at least one value that can be assigned to v. This procedure can be made very efficient, e.g., by using *watched literals* [42]. Another representation for constraint variables is the *bound encoding*, where an atom is introduced for each variable $v \in V$ and value $i \in dom(v)$ to represent that v is bounded by i, i.e., $v \leq i$ (cf. [49]). Similar to the value encoding, we can define nogoods that insure a consistent Boolean assignment [12]. A combination of value and bound encoding is also possible.

We see atoms from the value and bound encoding as *primitive constraints*, as all constraints can be decomposed into nogoods over them, e.g., by describing changes in the variables' domains inferred by constraint propagation. This way, constraint propagators can be encoded eagerly or lazily. Transforming a constraint propagator into an external propagator is straightforward: Rather than applying domain changes directly, the constraint propagator has to be made encoding its inferences in form of nogoods over primitive constraints [46].

▶ **Example 4.** An external propagator for encoding the reified ALL-DIFFERENT constraint c is specified in Algorithm 4. Provided with a Boolean assignment $\mathbf{A}$, it starts with an empty set of lazily generated nogoods, followed by a distinction into two cases. First, if the constraint is to be satisfied, i.e., $\mathbf{T}At(c) \in \mathbf{A}$, then for each variable in the scope of the constraint that has a value assigned, a nogood is generated that asserts the removal of this value from the domain of all other variables in the scope of the constraint (Lines 2–3). On the other hand,

Input : A Boolean assignment $\mathbf{A}$.
Output : A set of lazily generated nogoods.

```
1  ∇ ← ∅                                                    // set of lazily generated nogoods
2  if TAt(c) ∈ A then foreach v ∈ S(c) s.t. Tval(v,i) ∈ A for some i ∈ dom(v) do
3      ∇ ← ∇ ∪ {{TAt(c), Tval(v,i), Tval(w,i)} | w ∈ S(c)\{v},
                                                  i ∈ dom(w), Fval(w,i) ∉ A}
4  else
5      foreach v ∈ S(c) s.t. Tval(v,i) ∈ A for some i ∈ dom(v) do
6          if w ∈ S(c)\{v} s.t. Tval(w,i) ∈ A then
7              if FAt(c) ∉ A then
8                  ∇ ← {{TAt(c), Tval(v,i), Tval(w,i)}}
9              return ∇
10     if ∀v ∈ S(c) ∃i ∈ dom(v) s.t. Tval(v,i) ∈ A then
11         ∇ ← {{FAt(c)} ∪ {Tval(v,i) | v ∈ S(c), i ∈ dom(v), Tval(v,i) ∈ A}}
12 return ∇
```

Algorithm 4 An external propagator for encoding the reified ALL-DIFFERENT constraint c.

if the constraint is not set to be satisfied, the algorithm checks whether two variables in the scope of the constraint have the same value assigned (Lines 5–9). If so, the ALL-DIFFERENT constraint is violated and a nogood asserting that the constraint atom is set to false will be returned (unless $\mathbf{F}At(c) \in \mathbf{A}$, in which case the constraint atom is already false). If, however, no such two variables can be found and all variables in the scope of the constraint have a value assigned, then the ALL-DIFFERENT condition is satisfied and a nogood is generated that asserts the truth of the constraint atom (Lines 10–11). Observe that this propagator enforces arc consistency on the binary decomposition of the reified ALL-DIFFERENT constraint if $At(c)$ is true, but propagates weakly if $At(c)$ is false. However, propagators that achieve higher levels of local consistency are also possible [48].

While constraint propagators encode their inference into unit or conflicting nogoods, unit-propagation processes this information within the next iteration. Unit-propagation, constraint propagation, and loop formula propagation are repeated until a fixpoint is reached or a conflict is encountered. By generating a conflicting nogood, e.g., a constraint propagator can yield that the underlying constraint is violated.

7 Experiments

We have implemented our approach with Lazy Nogood Generation for constraint variables, the ALL-DIFFERENT and integer LINEAR constraints within a new version of our prototypical CASP system *inca* [54] which is based on the latest development version of *clingo* (3.0.92; [53]). The default setting uses an ALL-DIFFERENT propagator that enforces arc consistency, while *inca*^{DC} enforces domain consistency, representing an infeasible encoding. To compare with the state-of-the-art, we include *clingcon* (2.0.0-beta; [53]) in our analysis. It also extends *clingo*, but integrates the CP solver *gecode* (3.7.1;[52]). Similar to our approach, *clingcon* is based on CDNL and abstracts from the constraints via constraint atoms, but it employs *gecode* to check the existence of a constraint variable assignment that does not violate any constraint (according to the assignment to constraint atoms). In turn, the CP solver can yield a conflict or propagate constraint atoms by generating nogoods over constraint atoms that

Table 1 Average time in seconds over completed runs on Quasigroup, Graceful Graph, Packing, and Numbrix benchmarks. Number of completed runs are given in parenthesis.

benchmark class	*clingo*	*clingcon*	*clingcon*^{DC}	*inca*	*inca*^{DC}
Quasigroup Completion (200)	106.6 (93)	34.4 (9)	4.6 (200)	86.2 (171)	24.7 (200)
Quasigroup Existence (21)	25.7 (18)	61.4 (10)	88.2 (11)	60.3 (20)	26.6 (20)
Graceful Graphs (10)	3.0 (9)	15.7 (4)	31.3 (7)	5.2 (6)	12.6 (10)
Packing (50)	104.1 (1)	33.1 (50)	33.1 (50)	24.6 (50)	24.6 (50)
Numbrix (12)	10.4 (12)	17.4 (12)	51.3 (12)	1.3 (12)	5.2 (12)
weighted, penalised time	228.3	267.0	124.6	103.1	24.2

occur in the constraint logic program. This constitutes a very limited form of Lazy Nogood Generation. We have set *clingcon* to generate nogoods by looking at dependency between constraints according to the *irreducibly inconsistent set construction* method in "forward" mode, when we noticed that this option significantly improves the performance of *clingcon*. Furthermore, the setting *clingcon*^{DC} uses domain consistency propagation. Our experiments also consider eager encodings for a comparison with the state-of-the-art in ASP solving, given through *clingo*. We conducted experiments on Quasigroup Completion ($n = 40$), Quasigroup Existence (QG1-4: $n = 7 \dots 9$; QG5: $n = 12 \dots 14$; QG6-7: $n = 10 \dots 12$) and Graceful Graphs benchmarks that stem from [12], Packing benchmarks from [8] and Numbrix [45] puzzles. Experiments were run on a Linux PC, where each run was limited to 600 sec CPU time on a 2.00 GHz core and 2 GB RAM. A summary of our results is provided in Table 1.

Although more benchmark classes are needed for a meaningful comparison, we can draw a few interesting conclusions. First, execution time can improve when CP constructs are treated by external propagation rather than encoding them eagerly. The latter can lead to huge encodings, in particular, when large domains are involved. In fact, the encoding of the Packing problem that was given in the system track of the competition quickly reaches the memory limit of 2 GB in 49 over 50 instances, while the CASP systems *clingcon* and *inca* solve every instance within a reasonable amount of space and time. Second, the advantage of generating nogoods to describe the inferences of constraint propagators is that CDNL can exploit constraint interdependencies for directing search, and most importantly conflict analysis. The fact that *clingcon* does not encode CP constructs into nogoods, by design, is likely to be the reason for its limited success in our experiments, where *clingcon* is particularly ineffective on Quasigroup problems. Third, experiments show that our approach, represented through *inca*, combines the best of both worlds: It can avoid huge encodings via abstraction to external propagation while retaining the ability to make the encoding explicit. It outperforms the state-of-the-art in CASP solving on individual benchmark classes, and is more robust over all benchmark instances. On most benchmarks, a dedicated treatment of infeasible ALL-DIFFERENT encodings via external propagation has further improved performance.

8 Related Work

Related work on the integration of ASP with other declarative problem solving paradigms is plentiful, and roughly falls into one of three categories: translation-based approaches, modular approaches, and integrated approaches. In translation-based approaches, all parts of an (extended) ASP model are eagerly encoded into a single language for which highly efficient off-the-shelf solvers are available. Niemelä [43] provides a simple mapping of constraints into ASP given by allowed or forbidden combinations of values. We have demonstrated efficiency in [12], describing what type of local consistency the unit-propagation of an ASP

solver achieves on value, bounds, and range encodings. Specialised encodings for GRAMMAR and related constraints are presented in [14]. There is also a substantial body of work on encoding constraints into SAT [51, 25, 4, 15, 49, 5] which can be translated into ASP [43]. Similarly, (extended) ASP models can be translated, e.g., into SAT [27], SAT with inductive definitions [38], and difference logic [28]. In a modular approach, theory-specific solvers interact in order to compute solutions. Baselice et al. [3] and Mellarkod and Gelfond [39] combine systems for solving ASP and CP that do not ground constraint variables. Instead, constraint variables are handled in a CP solver. Dal Palú et al. [10] employ a CP system for intermediate grounding and the computation of answer sets. The approach taken by Balduccini [1] consists of writing logic programs whose answer sets encode a desired CSP, which is, in turn, solved by a CP system. Järvisalo et al. [29] obtain the overall semantics from the ones of individual modules, including CP modules. While above modular approaches see ASP and CP solvers as blackboxes, Mellarkod et al. [40] integrate a CP solver into the decision engine of a backtracking-based ASP solver. Gebser et al. [23] integrate constraint atoms with conflict-driven techniques by extending the conflict analysis of an ASP solver. An implementation of their approach is given through the CASP system *clingcon*. The abstraction from the inference performed by constraint propagation, however, limits the exploitation of constraint interdependencies. ASP solving via Lazy Nogood Generation was first outlined in [13], and falls into the category of integrated approaches. The related work closest to this paper is Lazy Clause Generation [46], a SAT-based approach to CSP solving where lazy clause generators encode the inference of *propagation rules* into clauses. However, our approach is fundamentally more general then Lazy Clause Generation, where the truth value of each constraint atom is known a-priori and every nogood is represented by a clause. Nogoods can also be represented by other ASP constructs, such as cardinality rules, weight constraint rules [44], and aggregation [47, 19]. Gebser et al. [20] show that constraint variables can be conveniently expressed by means of cardinality rules. Elkabani et al. [17] provide a generic framework which provides an elegant treatment of such extensions to ASP, employing constraint propagators for their handling, though, without support for conflict-driven techniques. A thorough approach to integrating propagators for weight constraint rules within a conflict-driven framework is presented in [21].

Alternative computation models that aim at limiting the need for preliminary grounding but do not integrate ASP with other declarative paradigms have also been proposed (e.g., [31]).

9 Conclusion

We presented a comprehensive extension for ASP solving to address the scalability and efficiency of ASP, called Lazy Nogood Generation. Founded on a nogood-based characterisation of external propagation, our techniques allow for representing encodings that are otherwise infeasible. However, external propagators can make parts of the encoding explicit whenever it triggers inference. We presented key algorithms that are centred around conflict-driven learning, and seamlessly applied our techniques to CASP solving by employing constraint propagation. Experiments show that our prototypical implementation is competitive with the state-of-the-art. We expect further significant computational impact given the empirical evidence provided by Lazy Clause Generation [46]. Moreover, Lazy Nogood Generation generalises Lazy Clause Generation, as every nogood can be syntactically represented by a clause, but other ASP constructs are also possible. Future work considers the exploitation of ASP constructs like aggregation and loops. Many questions on modelling and solving CASP also remain open, concerning encoding optimisations and further language extensions.

References

1 M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *25th International Conference on Logic Programming (ICLP'09) Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, 2009.

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

3 S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *21st International Conference on Logic Programming (ICLP'05)*, pages 52–66. Springer, 2005.

4 C. Bessière, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 299–314. Springer, 2003.

5 C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 419–424, 2009.

6 C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 412–418, 2009.

7 F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning operators for answer set programming systems. In *9th International Workshop on Non-Monotonic Reasoning (NMR'02)*, pages 200–209, 2002.

8 F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 388–403. Springer, 2011.

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

10 A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *25th International Conference on Logic Programming (ICLP'09)*, pages 115–129. Springer, 2009.

11 R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

12 C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming, 26th International Conference on Logic Programming (ICLP'10) Special Issue*, 10(4-6):465–480, 2010.

13 C. Drescher and T. Walsh. Conflict-driven constraint answer set solving with lazy nogood generation. In *25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1772–1773. AAAI Press, 2011.

14 C. Drescher and T. Walsh. Modelling grammar constraints with answer set programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28–39. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.

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

16 T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

17 I. Elkabani, E. Pontelli, and T. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *20th International Conference on Logic Programming (ICLP'04)*, pages 73–89. Springer, 2004.

18 E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.

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

20 M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *23rd Workshop on (Constraint) Logic Programming (WLP'09)*, 2009.

21 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *25th International Conference on Logic Programming (ICLP'09)*, pages 250–264. Springer, 2009.

22 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/MIT Press, 2007.

23 M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *25th International Conference on Logic Programming (ICLP'09)*, pages 235–249. Springer, 2009.

24 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pages 1070–1080. MIT Press, 1988.

25 I. P. Gent. Arc consistency in SAT. In *15th Eureopean Conference on Artificial Intelligence (ECAI'02)*, pages 121–125. IOS Press, 2002.

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

27 T. Janhunen and I. Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 111–130. Springer, 2011.

28 T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 142–154. Springer, 2009.

29 M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 155–169. Springer, 2009.

30 J. Lee. A model-theoretic counterpart of loop formulas. In *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

31 C. Lefèvre and P. Nicolas. The first version of a new ASP solver : ASPeRiX. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 522–527. Springer, 2009.

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

33 V. Lifschitz. Answer set planning. In *16th International Conference on Logic Programming (ICLP'99)*, pages 23–37. MIT Press, 1999.

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

35 V. Lifschitz and H. Turner. Splitting a logic program. In *11th International Conference on Logic Programming (ICLP'94)*, pages 23–37. MIT Press, 1994.

36 M. Maratea, F. Ricca, W. Faber, and N. Leone. Look-back techniques and heuristics in DLV: Implementation, evaluation, and comparison to QBF solvers. *Algorithms*, 63(1-3):70–89, 2008.

37 V. M. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-year perspective*, pages 375–398. Springer, 1999.

38 M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, pages 211–224. Springer, 2008.

39 V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In *9th International Symposium Proceedings on Functional and Logic Programming (FLOPS'08)*, pages 15–31. Springer, 2008.

40 V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

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

42 M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001.

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

44 I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, pages 317–331. Springer, 1999.

45 `http://www.parade.com/askmarilyn/numbrix/`.

46 O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

47 N. Pelov. *Semantics of logic programs with aggregates*. PhD thesis, Department of Computer Science, K.U. Leuven, Belgium, 2004.

48 J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *12th AAAI Conference on Artificial Intelligence (AAAI'94)*, pages 362–367. AAAI Press, 1994.

49 N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

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

51 T. Walsh. SAT v CSP. In *6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 441–456. Springer, 2000.

52 `http://www.gecode.org`.

53 `http://potassco.sourceforge.net`.

54 `http://potassco.sourceforge.net/labs.html`.