TLP **17** (5–6): 780–799, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000254 First published online 24 August 2017

*Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis**

BERNARDO CUTERI
DeMaCS, University of Calabria, Italy
(*e-mail*: cuteri@mat.unical.it)

CARMINE DODARO
DIBRIS, University of Genova, Italy
(*e-mail*: dodaro@dibris.unige.it)

FRANCESCO RICCA
DeMaCS, University of Calabria, Italy
(*e-mail*: ricca@mat.unical.it)

PETER SCHÜLLER
Faculty of Engineering, Marmara University, Turkey
Institute of Information Systems, Knowledge-based Systems Group, TU Wien, Austria
(*e-mail*: schueller.p@gmail.com)

submitted 15 July 2017; revised 20 July 2017; accepted 21 July 2017

Abstract

Answer set programming (ASP) is a well-established declarative paradigm. One of the successes of ASP is the availability of efficient systems. State-of-the-art systems are based on the ground+solve approach. In some applications, this approach is infeasible because the grounding of one or a few constraints is expensive. In this paper, we systematically compare alternative strategies to avoid the instantiation of problematic constraints, which are based on custom extensions of the solver. Results on real and synthetic benchmarks highlight some strengths and weaknesses of the different strategies.

KEYWORDS: Answer set programming, lazy grounding, propagators

1 Introduction

Answer set programming (ASP) is a declarative formalism for knowledge representation and reasoning based on stable model semantics (Gelfond and Lifschitz 1991; Brewka *et al.* 2011), for which robust and efficient implementations are available (Gebser *et al.* 2016). State-of-the-art ASP systems are usually based on

* The paper has been partially supported by the Italian Ministry for Economic Development (MISE) under project "PIUCultura – Paradigmi Innovativi per l'Utilizzo della Cultura" (no. F/020016/01-02/X27), under project "Smarter Solutions in the Big Data World (S2BDW)" (no. F/050389/01-03/X32) funded within the call "HORIZON2020" PON I&C 2014-2020, and by the Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E777.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

the 'ground+solve' approach (Kaufmann *et al.* 2016), in which a *grounder* module transforms the input program (containing variables) in an equivalent variable-free one, whose stable models are subsequently computed by the *solver* module. ASP implementations adopting this traditional approach are known to be effective for solving complex problems arising from academic and industrial applications, including product configuration (Kojo *et al.* 2003), decision support systems for space shuttle flight controllers (Nogueira *et al.* 2001), explanation of biomedical queries (Erdem and Öztok 2015), construction of phylogenetic supertrees (Koponen *et al.* 2015), data-integration (Manna *et al.* 2015), reconfiguration systems (Aschinger et al. 2011), and more. Nonetheless, there are some classes of programs (cf. Calimeri *et al.* 2016) whose evaluation is not feasible with the 'ground+solve' approach just because the grounding phase induces a combinatorial blow-up. An issue that is usually referred to as the *grounding bottleneck* of ASP.

The grounding bottleneck has been subject of several studies in recent years, and various alternative approaches to overcome it have been proposed. Some of these are based on syntactic extensions that enable the combination of ASP solvers with solvers for external theories (Ostrowski and Schaub 2012; Aziz *et al.* 2013; Balduccini and Lierler 2013; de Cat *et al.* 2015; Susman and Lierler 2016; Eiter *et al.* 2016b; Balduccini and Lierler 2017), whereas the most prominent approach working on plain ASP is *lazy grounding*, which was implemented by ASPERIX (Lefèvre and Nicolas 2009), GASP (Dal Palù *et al.* 2009), and OMIGA (Dao-Tran *et al.* 2012). Roughly, the idea of lazy grounding is to instantiate rules only when it is required during the search for a stable model (Liu *et al.* 2010). In this way, it is possible to prevent the grounding of rules that are unnecessary for the computation. Albeit lazy grounding techniques obtained promising preliminary results, they cannot yet reach the performance of state of the art systems in many benchmarks (Lefèvre and Nicolas 2009; Calimeri *et al.* 2016). One of the reasons is probably that fully fledged lazy grounding techniques could not be easily integrated within solvers based on the very efficient conflice-driven clause learning (CDCL) algorithm (Silva and Sakallah 1999; Kaufmann *et al.* 2016; Weinzierl 2017). Nonetheless, in many applications, the grounding bottleneck is merely caused by rules of a specific kind, namely constraints. For example, the following constraint has been identified as the bottleneck in programs solving a problem of natural language understanding:

$$\leftarrow eq(X,Y),\ eq(Y,Z),\ {\sim}eq(X,Z)$$

Its grounding, which features a cubic number of instances with respect to the extension of predicate *eq* in the worst case, is often not feasible for real world instances (Schüller 2016).

In this paper, we focus on the above practically relevant case of problematic constraints. In particular, we systematically compare alternative strategies that avoid the instantiation of some constraints by extending a CDCL-based ASP solver. In a nutshell, the input program is simplified by omitting problematic constraints and it is grounded; then, the resulting ground program is provided as input to a solver that is extended to emulate the presence of missing constraints. Among the strategies for extending the solver, we considered *lazy instantiation of constraints*

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

and *custom propagators*. In the first strategy, the solver searches for a stable model S of the simplified program. Then, S is returned as a solution if it satisfies also the omitted constraints, otherwise the violated instances of these constraints are lazily instantiated, and the search continues (Section 3.2). In the second strategy, the solver is extended (in possibly alternative ways) by custom *propagators*, which emulate the presence of missing constraints during the search (Section 3.3). The above-mentioned strategies can be implemented by using the API of existing CDCL-based ASP solvers (Gebser *et al.* 2016; Dodaro *et al.* 2016b). An empirical evaluation conducted on real and synthetic benchmarks (Section 4) confirms that the usage of lazy instantiation and custom propagators is effective when the grounding bottleneck is due to some constraint. The analysis of the results highlights strengths and weaknesses of the different strategies. Moreover, it shows there is not always a clear winner for a given problem, and the choice depends also on the characteristics the instances to solve. This observation suggested to investigate the applicability of algorithm selection techniques. The results are positive, in the sense that already a basic portfolio is faster than the best approach.

2 Answer set programming (ASP)

An ASP program Π is a finite set of rules of the form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_j, \sim b_{j+1}, \ldots, \sim b_m \tag{1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms and $n \geqslant 0$, $m \geqslant j \geqslant 0$. In particular, an *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are *terms*. Terms are alphanumeric strings and are distinguished in variables and constants. According to Prolog's convention, only variables start with an uppercase letter. A *literal* is an atom a_i (positive) or its negation $\sim a_i$ (negative), where $\sim$ denotes the *negation as failure*. Given a rule r of the form (1), the disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while $b_1, \ldots, b_j, \sim b_{j+1}, \ldots, \sim b_m$ is the *body* of r, of which $b_1, \ldots, b_j$ is the *positive body*, and $\sim b_{j+1}, \ldots, \sim b_m$ is the *negative body* of r. A rule r of the form (1) is called a *fact* if $m = 0$ and a *constraint* if $n = 0$. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Rules and programs are *positive* if they contain no negative literals, and *general* otherwise. Given a program Π, let the *Herbrand Universe* U_Π be the set of all constants appearing in Π and the *Herbrand Base* B_Π be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in Π with the constants of U_Π. Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_Π. Similarly, given a program Π, the *ground instantiation* $Ground(\Pi)$ of Π is the set $\bigcup_{r \in \Pi} Ground(r)$.

For every program Π, its stable models are defined using its ground instantiation $Ground(\Pi)$ in two steps: First, stable models of positive programs are defined, and then a reduction of general programs to positive ones is given, which is used to define stable models of general programs.

A set L of ground literals is said to be *consistent* if, for every literal $\ell \in L$, its negated literal $\sim \ell$ is not contained in L. Given a set of ground literals L, $L_{|_+} \subseteq L$

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

denotes the set of positive literals in L. An interpretation I for Π is a consistent set of ground literals over atoms in B_Π. A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its negated literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. A constraint c is said to be *violated* by an interpretation I if all literals in the body of c are true. An interpretation I is *total* if, for each atom a in B_Π, either a or $\sim a$ is in I (i.e., no atom in B_Π is undefined w.r.t. I). Otherwise, it is *partial*. A total interpretation M is a *model* for Π if, for every $r \in Ground(\Pi)$, at least one literal in the head of r is true w.r.t. M whenever all literals in the body of r are true w.r.t. M. A model X is a *stable model* for a positive program Π if any other model Y of Π is such that $X_{|_+} \subseteq Y_{|_+}$.

The *reduct* or *Gelfond–Lifschitz transform* of a general ground program Π w.r.t. an interpretation X is the positive ground program Π^X, obtained from Π by (i) deleting all rules $r \in \Pi$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules. A stable model of Π is a model X of Π such that X is a stable model of $Ground(\Pi)^X$. We denote by $SM(\Pi)$ the set of all stable models of Π, and call Π *coherent* if $SM(\Pi) \neq \emptyset$, *incoherent* otherwise.

Example 1
Consider the following program Π_1:

$$r_1 : a(1) \leftarrow \sim b(1) \qquad r_2 : b(1) \leftarrow \sim a(1) \qquad r_3 : \leftarrow a(X),\ b(X)$$
$$r_4 : c(1) \leftarrow \sim d(1) \qquad r_5 : d(1) \leftarrow \sim c(1) \qquad r_6 : \leftarrow a(X),\ \sim b(X)$$

The ground instantiation $Ground(\Pi_1)$ of the program Π_1 is the following program:

$$g_1 : a(1) \leftarrow \sim b(1) \qquad g_2 : b(1) \leftarrow \sim a(1) \qquad g_3 : \leftarrow a(1),\ b(1)$$
$$g_4 : c(1) \leftarrow \sim d(1) \qquad g_5 : d(1) \leftarrow \sim c(1) \qquad g_6 : \leftarrow a(1),\ \sim b(1)$$

Note that $M = \{\sim a(1),\ b(1),\ c(1),\ \sim d(1)\}$ is a model of $Ground(\Pi_1)$. Since $Ground(\Pi_1)^M$ comprises only the facts $b(1)$ and $c(1)$, and constraint g_3, M is a stable model of Π. ◁

Support. Given a model M for a ground program Π, we say that a ground atom $a \in M$ is *supported* with respect to M if there exists a *supporting* rule $r \in \Pi$ such that a is the only true atom w.r.t. M in the head of r, and all literals in the body of r are true w.r.t. M. If M is a stable model of a program Π, then all atoms in M are supported.

3 Solving strategies

3.1 Classical evaluation

The standard solving approach for ASP is instantiation followed by a procedure similar to CDCL for SAT with extensions specific to ASP (Kaufmann *et al.* 2016). The basic algorithm *ComputeStableModel*(Π) for finding a stable model of program Π is shown in Algorithm 1. Function Propagate combines unit propagation (as in SAT) with some additional ASP-specific propagations, which ensure that the model is stable (Kaufmann *et al.* 2016).

Given a partial interpretation I consisting of literals, and a set of rules Π, *unit propagation* infers a literal ℓ to be true if there is a rule $r \in \Pi$

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

such that r can be satisfied only by $I \cup \{\ell\}$. Given the nogood representation $C(r) = \{\sim a_1, \ldots, \sim a_n, b_1, \ldots, b_j, \sim b_{j+1}, \ldots, \sim b_m\}$ of a rule r, then the negation of a literal $\ell \in C(r)$ is unit propagated w.r.t. I and rule r iff $C(r) \setminus \{\ell\} \subseteq I$. To ensure that models are supported, unit propagation is performed on the Clark completion of Π or alternatively a *support propagator* is used (Alviano and Dodaro 2016).

Example 2

Consider the ground program Π_1 from Example 1. *ComputeStableModel*(Π_1) starts with $I = \emptyset$ and does not propagate anything in line 3. I is partial and consistent, so the algorithm continues in line 11. Assume no restart and no deletion is performed, and assume *ChooseLiteral* returns $\{a(1)\}$, i.e., $I = \{a(1)\}$. Next, *Propagate*(I) is called, which yields $I = \{a(1),\ b(1),\ \sim b(1)\}$: $\sim b(1)$ comes from unit propagation on g_3 and $b(1)$ from unit propagation on g_6. Thus, I is inconsistent and I is analyzed to compute a reason explaining the conflict, i.e., *CreateConstraint*(I) $= \{g_7\}$ with $g_7 : \leftarrow a(1)$. Intuitively, the truth of $a(1)$ leads to an inconsistent interpretation, thus $a(1)$ must be false. Then, the consistency of I is restored (line 6), i.e., $I = \emptyset$, and g_7 is added to Π_1. The algorithm again restarts at line 3 with $I = \emptyset$ and propagates $I = \{\sim a(1),\ b(1)\}$, where $\sim a(1)$ comes from unit propagation on g_7, and b from unit propagation on g_2. I is partial and consistent; therefore, lines 11 and 12 are executed. Assume again that no restart and no constraint deletion happens, and that *ChooseLiteral*(I) $= \{c(1)\}$. Therefore, the algorithm continues in line 3 with $I = \{\sim a(1),\ b(1),\ c(1)\}$. Propagation yields $I = \{\sim a(1),\ b(1),\ c(1),\ \sim d(1)\}$ because $\sim d(1)$ is support-propagated w.r.t. g_4 and I (or unit-propagated w.r.t. the completion of g_4 and I). I is total and consistent; therefore, the algorithm returns I as the first stable model. ◁

For the performance of this search procedure, several details are crucial: learning effective constraints from inconsistencies as well as heuristics for restarting, constraint deletion, and for choosing literals.

3.2 Lazy constraints

The algorithm presented in this section is reported as Algorithm 2. The algorithm takes as input a program Π and a set of constraints $C \subseteq \Pi$. Then, the constraints in C are removed from Π, obtaining the program $\mathscr{P}$. A stable model of *Ground*($\mathscr{P}$) is searched (line 3). Two cases are possible: (*i*) $\mathscr{P}$ is incoherent (line 4). Thus, the original program Π is also incoherent and the algorithm terminates returning $\bot$. (*ii*) $\mathscr{P}$ is coherent. Thus, a stable model, say I, is computed. In this case, a set of constraints $\mathscr{C} \in$ *Ground*(C) that are violated under the stable model I are extracted (line 5) and added to $\mathscr{P}$ (line 7). The process is repeated until either a stable model of $\mathscr{P}$ violating no constraints in *Ground*(C) is found or $\mathscr{P}$ is incoherent. Importantly, *Ground*(C) is never represented explicitly in the implementation of line 5.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Algorithm 1: ComputeStableModel

Input : A ground program $\mathscr{P}$
Output: A stable model for $\mathscr{P}$ or $\bot$

```
1  begin
2      I := ∅;
3      I := Propagate(I);
4      if I is inconsistent then
5          r := CreateConstraint(I) ;
6          I := RestoreConsistency(I);
7          if I is consistent then 𝒫 := 𝒫 ∪ {r};
8          else return ⊥;
9      else if I total then return I;
10     else
11         I := RestartIfNeeded(I);      𝒫 := DeleteConstraintsIfNeeded(𝒫);
12         I := I ∪ ChooseLiteral(I);
13     end
14     go to 3;
15 end
```

Function Propagate(I)

```
1 ℐ = I;
2 for ℓ ∈ ℐ do ℐ := ℐ ∪ Propagation(ℐ, ℓ) ;
3 return ℐ;
```

Example 3
Again consider program Π_1 from Example 1 and the set of constraints $C = \{r_3, r_6\}$. The algorithm computes a stable model, say $I_1 = \{a(1), \sim b(1), c(1), \sim d(1)\}$, of $\mathscr{P}_1 = Ground(\Pi_1 \setminus C)$. Thus, the ground instantiation g_6 of r_6 is violated under I_1 and therefore g_6 is added to $\mathscr{P}$. Then, a stable model of $\mathscr{P}$ is computed, say $I_2 = \{\sim a(1), b(1), c(1), \sim d(1)\}$. At this point, I_2 violates no constraint in $Ground(C)$. Thus, the algorithm terminates returning I_2. Note that all instantiations of constraint r_3 will never be violated since rules r_1 and r_2 enforce that exactly one of $a(1)$ and $b(1)$ can be true in a stable model. Thus, r_3 will never be instantiated by the algorithm.◁

An important feature of Algorithm 2 is that it requires no modifications to the search procedure implemented by the underlying ASP solver.

3.3 Constraints via propagators

In this section, constraints are replaced using the concept of *propagator*, which can set truth values of atoms during the solving process, based on truth values of other atoms. An example of a propagator is the unit propagation, detailed in Section 3.1. In contrast to the lazy instantiation of constraints that aims at adding violated

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Algorithm 2: LazyConstraintInstantiation

Input : A nonground program Π, a set of nonground constraints $C \subseteq \Pi$
Output: A stable model for Π or $\bot$

```
1 begin
2     𝒫 := Ground(Π \ C);
3     I := ComputeStableModel(𝒫);
4     if I == ⊥ then return ⊥ ;
5     𝒞 = {c | c ∈ Ground(C), c is violated};
6     if 𝒞 == ∅ then return I ;
7     𝒫 := 𝒫 ∪ 𝒞 ;
8     go to 3;
9 end
```

constraints when a stable model candidate is found, propagators usually are used to evaluate the constraints during the computation of the stable model. Given a program Π, traditional solvers usually apply propagators on the whole set of rules and constraints in $Ground(\Pi)$. An alternative strategy is to consider a variant of the program, say $\mathscr{P} = \Pi \setminus C$, where C is a set of constraints. The solver is then executed on $Ground(\mathscr{P})$ and a propagator is used to guarantee the coherence of partial interpretations with the constraints in $Ground(C)$. Constraints in C are not instantiated in practice, but their inferences are simulated by an ad-hoc procedure implemented for that purpose. This approach requires a modification of the Propagation function in Propagate, such that Propagation considers the additional set C of constraints, verifies which constraints would result in a propagation on the partial interpretation, and propagates truth values due to inferences on C in addition to unit propagation.

Example 4
Again consider program Π_1 from Example 1 and the set of constraints $C = \{r_3, r_6\}$. The idea is to execute Algorithm 1 on $Ground(\mathscr{P}_1)$, where $\mathscr{P}_1 = \Pi_1 \setminus C$. $ComputeStableModel(\mathscr{P}_1)$ starts with $I = \emptyset$ and does not propagate anything in line 3. I is partial and consistent, so the algorithm continues in line 11. Assume no restart and no deletion are performed, and assume *ChooseLiteral* returns $\{a(1)\}$, i.e., $I = \{a(1)\}$. Next, $Propagate(I, C)$ is called. In this case, the propagation yields $I = \{a(1), b(1), \sim b(1)\}$, where $\sim b(1)$ comes from unit propagation on g_1, while $b(1)$ comes from unit propagation on the ground instantiation g_6 of the rule r_6. Thus, I is inconsistent and I is analyzed to compute a reason that explains the conflict, i.e., $CreateConstraint(I) = \{g_7\}$ with $g_7 : \leftarrow a(1)$. Then, the algorithm continues as shown in Example 2. Note that, from this point of the computation, the ground instantiations of constraints r_3 and r_6 will never be violated again, since g_7 assure that $a(1)$ will be false in all partial interpretations under consideration. ◁

We classify constraint propagators according to the priority given to them. In particular, they are considered *eager* if propagation on non-ground constraints is executed as soon as possible, i.e., during unit propagation of already grounded

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

constraints; moreover, they are called *postponed* (or *post*) if propagation on constraints is executed after all other (unit, support, etc.) propagations.

4 Implementation and experimental analysis

4.1 Implementation

The lazy instantiation of constraints and the propagators has been implemented on top of the ASP solvers WASP (Alviano et al. 2015) and CLINGO (Gebser *et al.* 2016). The Python interface of WASP (Dodaro *et al.* 2016b) follows a synchronous message passing protocol implemented by means of method calls. Basically, a Python program implements a predetermined set of methods that are later on called by WASP whenever specific points of the computation are reached. The methods may return some values that are then interpreted by WASP. For instance, when a literal is true, the method *onLiteralTrue* of the propagator is called, whose output is a list of literals to infer as true as a consequence (see Dodaro *et al.* 20016b for further details). CLINGO 5 (Gebser *et al.* 2016) provides a Python interface where a propagator class with an interface similar to WASP can be registered.

Two important differences exist between WASP and CLINGO. First, CLINGO provides only a post-propagator interface and no possibility for realizing an eager propagator (that runs before unit propagation is finished). Second, WASP first collects nogoods added in Python and then internally applies them and handles conflicts, while CLINGO requires an explicit propagation call after each added nogood. If propagation returns a conflict then no further nogoods can be added in CLINGO, even if further nogoods were detected. After consulting the CLINGO authors, we implemented a queue for nogoods and add them in subsequent propagations if there is a conflict. This yields higher performance than abandoning these nogoods.

4.2 Description of benchmarks

In order to empirically compare the various strategies for avoiding the instantiation of constraints, we investigated several benchmarks of different nature, namely Stable Marriage, Packing, and Natural Language Understanding. All benchmarks contain one or few constraints whose grounding can be problematic.

Stable Marriage. The *Stable Marriage* problem can be described as follows: given n men and m women, where each person has a preference order over the opposite sex, marry them so that the marriage is stable. In this case, the marriage is said to be stable if there is no couple (M, W) for which both partners would rather be married with each other than their current partner. We considered the encoding used for the fourth ASP Competition. For the lazy instantiation and for the ad-hoc propagators, the following constraint has been removed from the encoding:

$$\leftarrow match(M, W1),\ manAssignsScore(M, W, Smw),\ W1 \neq W,\ manAssignsScore(M, W1, Smw1),\ Smw > Smw1,\ match(M1, W),\ womanAssignsScore(W, M, Swm),\ womanAssignsScore(W, M1, Swm1),\ Swm \geqslant Swm1.$$

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Intuitively, this constraint guarantees that the stability condition is not violated.

Packing. The *Packing* problem is related to a class of problems in which one has to pack objects together in a given container. We consider the variant of the problem submitted to the ASP Competition 2011. In that case, the problem was the packing of squares of possibly different sizes in a rectangular space and without the possibility of performing rotations. The encoding follows the typical guess-and-check structure, where positions of squares are guessed and some constraints check whether the guessed solution is a stable model. We identified two expensive sets of constraints. The first set comprises the following two constraints:

$$\leftarrow pos(I, X, Y), pos(I, X_1, Y_1), X_1 \neq X \qquad \leftarrow pos(I, X, Y), pos(I, X_1, Y_1), Y_1 \neq Y$$

which enforce that a square is not assigned to different positions. The second set comprises constraints forbidding the overlap of squares. One of these constraints is reported in the following:

$$\leftarrow pos(I_1, X_1, Y_1),\ square(I_1, D_1),\ pos(I_2, X_2, Y_2),\ square(I_2, D_2),\ I1 \neq I2$$
$$W1 = X1 + D1,\ H1{=}Y1{+}D1,\ X2 \geqslant X1,\ X2 < W1,\ Y2 \geqslant Y1,\ Y2{<}H1.$$

Other constraints are similar; thus, they are not reported.

Natural Language Understanding (NLU). The *NLU* benchmark is an application of ASP in the area of natural language understanding, in particular, the computation of optimal solutions for first-order Horn abduction problems under the following cost functions: cardinality minimality, cohesion, and weighted abduction. This problem and these objective functions have been described by Schüller (2016). In this problem, we aim to find a set of explanatory atoms that makes a set of goal atoms true with respect to a first-order Horn background theory. We here consider the acyclic version of the problem where backward reasoning over axioms is guaranteed to introduce a finite set of new terms. A specific challenge in this problem is that input terms and terms invented via backward chaining can be equivalent to other terms, i.e., the unique names assumption is partially not true. Equivalence of terms must be handled explicitly in ASP, which is done by guessing an equivalence relation. This makes the instantiation of most instances infeasible, as the number of invented terms becomes large, due to the grounding blow-up caused by the following constraint:

$$\leftarrow eq(A, B),\ eq(B, C),\ {\sim}eq(A, C).$$

4.3 Hardware and software settings

The experiments were run on a Intel Xeon CPU X3430 2.4 GHz. Time and memory were limited to 600 s and 4GB, respectively. In the following, WASP-LAZY refers to WASP implementing lazy instantiation of constraints, while WASP-EAGER and WASP-POST refer to WASP implementing eager and postponed propagators, respectively. All versions of WASP use GRINGO version 5.1.0 as grounder, whose grounding time is included in the execution time of WASP. Moreover, CLINGO LAZY and CLINGO POST

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

refer to CLINGO implementing lazy and postponed propagators, respectively. For the NLU benchmark, we always use unsat-core optimization.

4.4 Discussion of results

Stable Marriage. Concerning Stable Marriage, we executed the 30 instances selected for the Fourth ASP Competition. CLINGO and WASP executed on the full encoding are able to solve 29 out of the 30 instances with an average running time of 50 and 29 s, respectively. On the same instances, ad-hoc propagators cannot reach the same performance. Indeed, WASP-LAZY and WASP-POST perform the worst solving 0 and 5 instances, respectively, whereas WASP-EAGER is much better with 17 solved instances. The same performance is obtained by CLINGO-LAZY and CLINGO-POST which can solve 0 and 17 instances in the allotted time, respectively. The poor performance of the lazy instantiation can be explained by looking at the specific nature of the instances. Indeed, each instance contains a randomly generated set of preferences of men for women (resp. women for men). By looking at the instances, we observed that each man (resp. woman) has a clear, often total, preference order over each woman (resp. man). This specific case represents a limitation for employing the lazy instantiation. Indeed, WASP and CLINGO executed on the encoding without the stability constraint perform naive choices until a stable model candidate is found. Then, each candidate contains several violations of the stability condition and many constraints are added. However, those constraints are not helpful since they only invalidate the current stable model candidate. In general, for instances where the program without the stability condition is under-constrained, many stable model candidates need to be invalidated before an actual solution is found (intuitively, given a program Π and a set of constraints $C \subseteq \Pi$, $|SM(\Pi \setminus C)| \gg |SM(\Pi)|$).

In order to further analyze this behavior empirically, we have conducted an additional experiment on the same problem. In particular, we randomly generated instances where each man (resp. woman) gives the same preference to each woman (resp. man), so basically the stability condition is never violated. Then, we consider a percentage k of preferences, i.e., each man (resp. woman) gives the same preference to all the women (resp. men) but to $k\%$ of them a lower preference is given. In this way, instances with small values of k should be easily solved by lazy instantiation, whereas instances with high values of k should be hard. For each considered percentage k, we executed 10 randomly generated instances. Results are reported in Table 1, where the number of solved instances and the average running time are shown for each tested approach. Concerning WASP, as observed before, for instances where the value of k is small (up to 50%) the lazy approach can solve all the instances with an average running time of about 5 s. On the other hand, for high values of k the advantages of the lazy approach disappear, as observed for the competition instances. Interestingly, the eager propagator obtained the best performance overall. For the tested instances, it seems to benefit of a smaller program and generation of the inferences does not slow down the performance as observed for competition instances. Concerning CLINGO, the lazy approach is the best performing one for

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Table 1. *Stable Marriage: Number of solved instances and average running time (in seconds)*

	WASP		WASP-LAZY		WASP-EAGER		WASP-POST		CLINGO		CLINGO-LAZY		CLINGO-POST	
Pref. (%)	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t
0	10	4.1	10	4.7	10	4.6	10	4.7	10	10.6	10	4.2	10	4.2
5	9	16.2	10	4.7	10	4.3	10	4.9	10	23.0	10	4.6	10	4.4
10	10	19.2	10	4.7	10	4.3	10	4.6	10	34.6	10	6.4	10	8.2
15	9	24.3	10	4.7	10	4.4	10	4.8	10	42.9	10	9.6	10	17.5
20	8	35.2	10	4.8	10	4.6	10	5.2	10	48.9	10	16.5	10	24.7
25	10	34.8	10	4.8	10	5.4	10	6.0	10	53.9	10	22.2	10	42.8
30	6	97.0	10	5.0	10	7.7	10	7.6	10	59.5	10	32.2	10	92.1
35	10	42.1	10	5.0	10	8.2	10	10.0	10	65.8	10	62.4	10	115.9
40	9	51.3	10	5.2	10	7.6	10	9.2	10	68.4	10	81.8	10	117.5
45	10	113.4	10	5.4	10	10.8	10	12.0	10	71.0	10	97.7	10	140.8
50	6	74.6	10	5.1	10	22.4	10	20.3	10	72.0	10	153.6	10	143.4
55	9	44.5	8	5.9	10	39.4	10	23.6	10	72.9	10	193.8	10	166.5
60	9	70.9	10	7.7	10	23.8	10	25.0	10	74.6	10	241.1	10	181.6
65	7	99.3	10	11.4	10	64.7	10	54.2	10	74.7	10	295.6	10	209.8
70	9	89.3	5	25.5	10	121.8	10	101.8	10	75.0	10	361.1	10	235.3
75	8	77.0	0	–	10	184.0	10	146.7	10	75.1	6	472.1	10	311.0
80	7	85.5	0	–	10	248.6	8	274.7	10	76.3	0	–	10	434.3
85	4	259.5	0	–	10	232.3	1	337.2	10	82.3	0	–	7	569.7
90	9	79.2	0	–	5	449.4	0	–	10	251.1	0	–	1	577.7
95	10	46.3	0	–	0	–	0	–	6	273.6	0	–	3	580.8
100	8	67.6	1	81.2	10	133.3	10	153.6	10	74.1	6	493.3	10	323.9

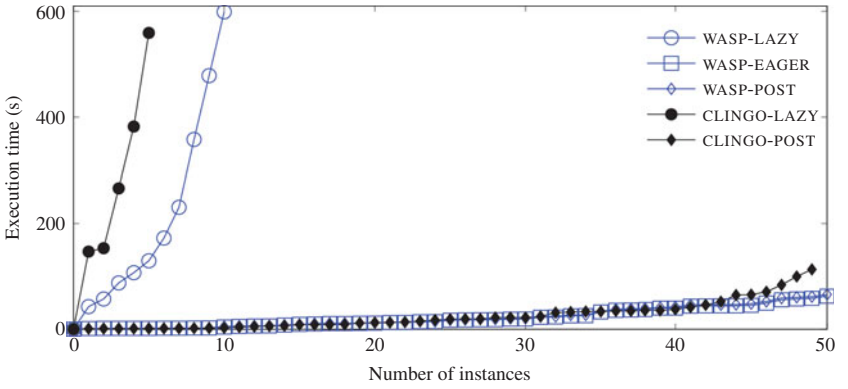


Fig. 1. Packing: Comparison of LAZY and PROPAGATORS approaches on 50 instances.

instances where the value of k is up to 35%. As shown for WASP, the performance of the lazy approach are worse for high values of k.

Packing. Concerning Packing problem, we considered all 50 instances submitted to the Third ASP Competition. Interestingly, when all constraints are considered none of the instances can be instantiated within the time limit. Thus, CLINGO and WASP do not even start the computation of a stable model. The grounding time substantially decreases when the two sets of expensive constraints described in Section 4.2 are removed from the encoding. Indeed, in this case, the grounding time on the tested instances is 5 s on average, with a peak of 16 s. Results of the lazy constraint instantiation and of constraint propagators on the resulting program are reported in the cactus plot of Figure 1. The graph highlights that WASP-EAGER, WASP-POST, and CLINGO-POST basically obtained the same performance. Indeed, the first two solve all the tested instances with an average running time of 22 and 23 s, respectively, while CLINGO-POST solves 49 out of 50 instances with an average running time of 25 s. Both WASP-POST and CLINGO-POST outperform their lazy counterparts. Indeed, WASP-LAZY solves 10 instances, with an average running time of 226 s, while CLINGO-LAZY solves 5 instances, with an average running time of 301 s. As already observed on the Stable Marriage instances, lazy instantiation cannot compete with constraint propagators. In this experiment, we observed that WASP and CLINGO perform naive choices on the encoding without the expensive constraints; thus, each candidate stable model contains several violations of constraints, leading to inefficient search in harder instances.

Natural Language Understanding (NLU). Concerning NLU, we considered all 50 instances and all 3 objective functions used by Schüller (2016). Results are reported in Table 2. As a general observation, all the tested instances are solved by WASP-LAZY and WASP-POST, no matter the objective function. Moreover, WASP-LAZY is on average faster than all other alternatives for both the objective functions cardinality and weighted abduction. The good performance of lazy instantiation is related to the small number of failing stable model checks performed. Indeed, only 2, 16, and

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Table 2. *NLU Benchmark: Number of solved instances and average running time (in seconds)*

	WASP		WASP-LAZY		WASP-EAGER		WASP-POST		CLINGO		CLINGO-LAZY		CLINGO-POST	
Obj. func.	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t	sol.	avg t
Card.	43	39.7	50	2.3	50	4.3	50	3.3	41	30.7	50	4.5	50	1.5
Coh.	43	40.1	50	18.5	50	8.8	50	6.3	41	30.7	49	24.6	49	15.8
W. Abd.	43	49.3	50	26.6	49	66.1	50	62.6	41	33.9	48	31.9	50	24.0

64 invalidations are on average required for cardinality, coherence, and weighted abduction, respectively. The number of propagation calls is much higher for WASP-EAGER than for WASP-POST (approximately WASP-EAGER performs three times more propagation calls than WASP-POST). However, the number of propagated literals that are not immediately rolled back because of a conflict is very similar; hence, it is clear that WASP-EAGER performs a lot of unnecessary propagations in this benchmark and WASP-POST should be preferred. Note that this is not generally the case for other benchmarks. Concerning CLINGO, 45, 248, and 321, stable model candidates are invalidated with CLINGO-LAZY, respectively, and a similar amount (26, 589, and 700, respectively) with CLINGO-POST. This shows that CLINGO tends to produce more stable models that violate lazy constraints. These violations are detected earlier with CLINGO-POST, therefore it outperforms CLINGO-LAZY in all objectives. None of the CLINGO propagators is able to solve all instances with all objectives, whereas WASP-POST solves all of them within 600 s. In particular, for objective functions cardinality and coherence, WASP is always slightly faster and uses slightly more memory than CLINGO. For weighted abduction, CLINGO-POST is most efficient with WASP-LAZY in second place. Nevertheless, using CLINGO or WASP with a LAZY or POST propagator will always be an advantage over using the pure ASP encoding where the constraints are instantiated prior to solving. Hence, the choice of the method for instantiating constraints is more important than the choice of the solver.

Discussion. We empirically investigated whether lazy instantiation or propagators can be a valid option for enhancing the traditional 'ground+solve' approach. When the full grounding is infeasible, both lazy instantiation and propagators can overcome this limitation, even though they exhibit different behaviors depending on the features of the problem and of the instances. This is particularly evident in Packing, where no instance can be grounded within the time limit. Since propagators are activated during the search, while lazy instantiation intervenes only when a total interpretation is computed, propagators are preferable when the problematic constraint is important to lead the search toward a solution (as overlap constraints in Packing). On the other hand, a high number of unnecessary propagations can make propagators inefficient and even slower than the lazy approach. In these cases, we observed that post propagators are better than eager propagators as remarked by the results on the objective function 'weighted abduction' in the NLU benchmark. The experiment on Stable Marriage highlights that lazy instantiation is effective when a few constraints are instantiated during the search. This is the case when: (i) it is very likely that a stable model of the simplified (i.e., without problematic constraints) input program also satisfies the lazy constraints; or (ii) the solver heuristics is such that one of the first candidate total interpretations also satisfies the lazy constraints. This is also confirmed in the NLU benchmark where the instances often have the above characteristics, and the propagator is better only when the constraints generated by the lazy approach do not fit the working memory. Moreover, from case (ii), we conjecture that the lazy approach can be effective in combination with domain-specific heuristics (Gebser *et al.* 2013; Dodaro *et al.* 2016a).

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

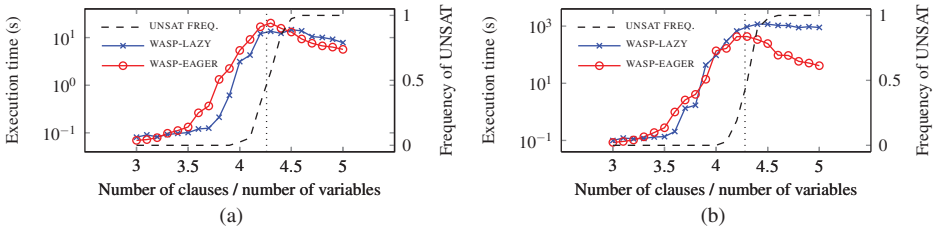


Fig. 2. 3-SAT experiments. Red and blue lines correspond to eager propagators and lazy instantiation respectively. The dashed black line represents the percentage of UNSAT instances, while the vertical dotted line evidences the phase transition point (frequency is about 0.5 at $R = 4.26$). (a) Results with 220 variables. (b) Results with 280 variables.

Finally, we conducted an additional experiment, where we do not oppose our approaches with the ground+solve one as in the previous cases, but it only aims at comparing the lazy propagation versus propagators in a controlled setting. In particular, we considered a synthetic benchmark based on the well-known 3-SAT problem that is interesting for our study, since it allows us to control both the hardness of the instances and the probability that an interpretation satisfies the constraint. Indeed, we generated the instances uniformly at random in a range centered on the phase transition (Achlioptas 2009). We used a straightforward ASP encoding where we guess an interpretation and we check by a single (non-ground) constraint whether this satisfies all clauses. The results are summarized in Figure 2 where we present two representative runs on formulas with 220 and 280 Boolean variables, respectively. Since eager and post propagators behave very similarly we only show comparisons between eager propagator and a lazy instantiation.

Expectedly, execution times follow the easy-hard-easy pattern (Achlioptas 2009), centered on the phase transition, while varying the ratio R of clauses over variables. Initially, the problem is very easy and both approaches are equally fast. Then, there is an interval in which the lazy approach is preferable, and finally the eager approach becomes definitely better than the lazy. Note that, on formulas with 220 variables (see Figure 2(a)), the lazy approach is preferable also on the hardest instances, instead with 280 variables (see Figure 2(b)), the eager approach becomes more convenient before the phase transition. To explain this phenomenon, we observe that the lazy approach can be exemplified by assuming that the solver freely guesses a model and then the lazy instantiator checks it, until every clause is satisfied by an assignment or no model can be found. The probability that a random model satisfies all clauses is $(\frac{7}{8})^k$, where k is the number of clauses; thus, fewer tries are needed on average to converge to a solution if the formula has fewer clauses. This intuitively explains why, as the number of variables increases, the eager approach becomes more convenient at smaller and smaller values of R. It is worth pointing out that this simplified model does not fully capture the behavior of lazy instantiation that is more efficient in practice, since the implementation learns from previous failures (by instantiating violated constraints).

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Table 3. *Applicability of a portfolio approach: experimental results*

		WASP-based				CLINGO-based			
Problem	#instances	Prec	Recall	F-meas	Perf. gain	Prec	Recall	F-meas	Perf. gain
Stable Marriage	500	0.66	0.67	0.63	26.7%	0.74	0.75	0.74	13.6%
NLU	150	0.88	0.92	0.90	38.3%	0.84	0.84	0.84	10.0%

4.5 On the applicability of techniques for automatic algorithm selection

The analysis conducted up to now shows that there is not always a clear winner among the strategies for realizing constraints, since the best solving method depends on characteristics of the encoding and the instance at hand. In similar scenarios, portfolio approaches that automatically choose one out of a set of possible methods have proven to be very effective in increasing system performance, since they combine the strengths of the available methods. Therefore, we investigated whether algorithm selection techniques can improve performance in our context.

We apply basic algorithm selection based on classification with machine learning: We extract some natural features from each instance, and train a C4.5 (Quinlan 1993) classifier to predict the best solving method (i.e., the one that required the least amount of time) among all the available ones (including the plain solver). We limit our analysis to Stable Marriage and NLU, because in these domains, none of the available methods is clearly superior. As features forStable Marriage, we used the number of persons and the percentage or preferences, and for NLU, we used the number of facts and the number of distinct constants and (instance-specific) predicates. We create portfolios for both WASP-based and CLINGO-based implementations.

Table 3 shows the results of our evaluation using 10-fold cross-validation (i.e., we split the set of instances into 10 partitions and use each partition as test set while training on the remaining partitions). For each problem, we report (weighted average) precision, recall, and F-measure of the prediction, as well as the average performance gain of the portfolio (i.e., by gain, we mean the difference in percentage between the sum of the execution times measured for the portfolio and for its best method). We observe that the classifier is able to choose the best algorithm in many cases, and the choice is almost ideal in NLU (F-measure of 0.9 for WASP and 0.84 for CLASP). The portfolios are always faster (in terms of execution times) than the corresponding best method for the respective problem. The performance gain peaks to 38% for the WASP-based, and is less pronounced for the CLINGO-based (peak at 13.6%). This is expected since CLINGO features a basic solver that is more competitive with propagator-based solutions in these domains.

Summarizing, these results confirm that already the application of basic portfolio techniques is a viable option for improving the performance when propagators are available.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

5 Related work

The grounding bottleneck in ASP has been subject of various studies. The most prominent grounding-less approach that works on plain ASP is *lazy grounding*, which was implemented by ASPERIX (Lefèvre and Nicolas 2009), GASP (Dal Palù *et al.* 2009), and OMIGA (Dao-Tran *et al.* 2012). Differently from our approach that is focused on constraints, these solvers perform lazy instantiation for *all* the rules of a program, and do not perform (conflict) clause learning. Weinzierl (2017) recently investigated learning of non-ground clauses.

Lazy instantiation of constraints was topic of several works on *integrating ASP with other formalisms*. These include CASP (Baselice et al. 2005; Ostrowski and Schaub 2012; Balduccini and Lierler 2017), ASPMT (Susman and Lierler 2016), BFASP (Aziz *et al.* 2013), and HEX (Eiter *et al.* 2016a). Differently from our approach, these approaches are based on syntactic extensions that enable the combination of ASP solvers with solvers for external theories. HEX facilitates the integration of generic computation oracles as literals in ASP rule bodies, and allows these computations not only to return true or false, but also to inject constraints into the search. This gave rise to the 'on-demand constraint' usage pattern of external atoms (Eiter *et al.* 2016b) that roughly corresponds with the LAZY propagators in this work. HEX also permits a declarative specification of properties of external computations (Redl 2016), e.g., antimonotonicity with respect to some part of the model. Such specifications automatically generate additional lazy constraints. Integration of ASP with continuous motion planning in robotics, based on HEX, was investigated by Erdem *et al.* (2016): Adding motion constraints in a POST propagator was found to be significantly faster than checking only complete stable model candidates (LAZY). For integrating *CModels with BProlog* (Balduccini and Lierler 2013), it was shown that using BProlog similar as a POST propagator (clearbox) performs better than using it as a LAZY propagator (black-box).

De Cat *et al.* (2015) provide a theory and implementation for *lazy model expansion* within the FO(ID) formalism that is based on *justifications* that prevent instantiation of certain constraints under assumptions. These assumptions are relative to a model candidate and can be revised from encountered conflicts, leading to a partially lazy instantiation of these constraints.

We finally observe that lazy constraints can be seen as a simplified form of lazy clause generation that was originally introduced in Constraint Programming (Feydy and Stuckey 2009).

6 Conclusion

In this paper, we compared several solutions for addressing the problem of the grounding bottleneck focusing on the practically-relevant case of problematic constraints without resorting to any language extension. The considered approach can be seen as a natural extension of the "ground+solve" paradigm, adopted by state of the art ASP systems, where some constraints are replaced either by lazy instantiators or propagators. The solutions fit CDCL-based solving

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

strategies, and can be implemented using APIs provided by state of the art solvers.

Experiments conducted on both real-world and synthetic benchmarks clearly outline that all the approaches can solve instances that are out of reach of state of the art solvers because of the grounding blowup. Lazy instantiation is the easiest to implement, and it is the best choice when the problematic constraints have a high probability to be satisfied. Otherwise, eager and post propagators perform better, with the latter being slightly more efficient when the constraint is activated more often during propagation. Our empirical analysis shows that there is not always a clear winner for a given problem, thus we investigated the applicability of algorithm selection techniques. We observed that a basic portfolio can improve on the best strategy also on these cases. As far as future work is concerned, we will study what are the conditions under which an entire subprogram (and not just some constraints) can be replaced by a propagator. Another line of research might be to investigate the impact of applying rule decomposition techniques before handling the constraints (Bichler *et al.* 2016).

References

Achlioptas, D. 2009. Random satisfiability. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Amsterdam/Washington, Vol. 185. IOS, 245–270.

Alviano, M. and Dodaro, C. 2016. Completion of disjunctive logic programs. In *Proc. of International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 886–892.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *Proc. of International Conference on Logic Programming and Non-monotonic Reasoning*. Lecture Notes in Computer Science, Vol. 9345. Springer, 40–54.

Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A. and Thorstensen, E. 2011. Optimization methods for the partner units problem. In *Proc. of International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, 4–19.

Aziz, R. A., Chu, G. and Stuckey, P. J. 2013. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming 13,* 4-5, 517–532.

Balduccini, M. and Lierler, Y. 2013. Integration schemas for constraint answer set programming: A case study. *Theory and Practice of Logic Programming 13,* 4-5 Online-Supplement.

Balduccini, M. and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory and Practice of Logic Programming 17*, 4, 462–515.

Baselice, S., Bonatti, P. A. and Gelfond, M. 2005. A preliminary report on integrating of answer set and constraint solving. In *Proc. of CEUR Workshop on Answer Set Programming*, vol. 142.

Bichler, M., Morak, M. and Woltran, S. 2016. The power of non-ground rules in answer set programming. *Theory and Practice of Logic Programming 16,* 5-6, 552–569.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Calimeri, F., Gebser, M., Maratea, M. and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence 231*, 151–181.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Dal Palù, A., Dovier, A., Pontelli, E. and Rossi, G. 2009. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3, 297–322.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G. and Weinzierl, A. 2012. Omiga: An open minded grounding on-the-fly answer set solver. In *Proc. of European Conference on Logics in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 7519. Springer, 480–483.

de Cat, B., Denecker, M., Bruynooghe, M. and Stuckey, P. J. 2015. Lazy model expansion: Interleaving grounding with search. *Journal of Artificial Intelligence Research 52*, 235–286.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F. and Schekotihin, K. 2016a. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *Theory and Practice of Logic Programming 16,* 5–6, 653–669.

Dodaro, C., Ricca, F. and Schüller, P. 2016b. External propagators in WASP: preliminary report. In *Proc. of RCRA*. CEUR Workshop Proceedings, vol. 1745. CEUR-WS.org, 1–9.

Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C. and Schüller, P. 2016a. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming 16,* 4, 418–464.

Eiter, T., Redl, C. and Schüller, P. 2016b. Problem solving using the HEX family. In *Computational Models of Rationality, Essays Dedicated to Gabriele Kern-Isberner on the Occasion of her 60th Birthday*. College Publications, 150–174.

Erdem, E. and Öztok, U. 2015. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming 15,* 1, 35–78.

Erdem, E., Patoglu, V. and Schüller, P. 2016. A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks. *AI Communications 29,* 2, 319–349.

Feydy, T. and Stuckey, P. J. 2009. Lazy clause generation reengineered. In *Proc. of Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 5732. Springer, 352–366.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Wanko, P. 2016. Theory solving made easy with clingo 5. In *Proc. of Technical Communications of International Conference on Logic Programming*. OASICS, vol. 52, 2:1–2:15.

Gebser, M., Kaufmann, B., Romero, J., Otero, R., Schaub, T. and Wanko, P. 2013. Domain-specific heuristics in answer set programming. In *Proc. of Conference on Artificial Intelligence*. AAAI.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput. 9,* 3/4, 365–386.

Kaufmann, B., Leone, N., Perri, S. and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Mag. 37,* 3, 25–32.

Kojo, T., Männistö, T. and Soininen, T. 2003. Towards intelligent support for managing evolution of configurable software product families. In *Proc. of System Configuration Management*. Lecture Notes in Computer Science, vol. 2649. Springer, 86–101.

Koponen, L., Oikarinen, E., Janhunen, T. and Säilä, L. 2015. Optimizing phylogenetic supertrees using answer set programming. *Theory and Practice of Logic Programming 15,* 4-5, 604–619.

Lefèvre, C. and Nicolas, P. 2009. The first version of a new ASP solver: Asperix. In *Proc. of Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 5753. Springer, 522–527.

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

Manna, M., Ricca, F. and Terracina, G. 2015. Taming primary key violations to query large inconsistent data via ASP. *Theory and Practice of Logic Programming 15,* 4-5, 696–710.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. 2001. An A-prolog decision support system for the space shuttle. In: *PADL*. 169–183.

Ostrowski, M. and Schaub, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming 12,* 4–5, 485–503.

Quinlan, J. 1993. *C4.5: Programs for Empirical Learning*. Morgan Kaufmann.

Redl, C. 2016. The dlvhex system for knowledge representation: Recent advances (system description). *TPLP 16,* 5-6, 866–883.

Schüller, P. 2016. Modeling variations of first-order horn abduction in answer set programming. *Fundamenta Informaticae 149,* 1–2, 159–207.

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

Susman, B. and Lierler, Y. 2016. SMT-based constraint answer set solver EZSMT (system description). In *Proc. of Technical Communications of International Conference on Logic Programming*. OASICS, vol. 52. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 1:1–1:15.

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR*. Lecture Notes in Computer Science, vol. 10377, 191–204.

https://doi.org/10.1017/S1471068417000254 Published online by Cambridge University Press