Overcoming the Grounding Bottleneck Due to Constraints in ASP Solving: Constraints Become Propagators

Bernardo Cuteri[1] , **Carmine Dodaro**[1] , **Francesco Ricca**[1,*] and **Peter Schüller**[2]
[1]University of Calabria, Italy
[2]TU Wien, Austria
{cuteri, dodaro, ricca}@mat.unical.it, ps@kr.tuwien.ac.at

Abstract

Answer Set Programming (ASP) is a well-known formalism for Knowledge Representation and Reasoning, successfully employed to solve many AI problems, also thanks to the availability of efficient implementations. Traditionally, ASP systems are based on the ground&solve approach, where the grounding transforms a general input program into its propositional counterpart, whose stable models are then computed by the solver using the CDCL algorithm. This approach suffers an intrinsic limitation: the grounding of one or few constraints may be unaffordable from a computational point of view; a problem known as grounding bottleneck. In this paper, we develop an innovative approach for evaluating ASP programs, where some of the constraints of the input program are not grounded but automatically translated into propagators of the CDCL algorithm that work on partial interpretations. We implemented the new approach on top of the solver WASP and carried out an experimental analysis on different benchmarks. Results show that our approach consistently outperforms state-of-the-art ASP systems by overcoming the grounding bottleneck.

1 Introduction

Answer Set Programming (ASP) [Brewka *et al.*, 2011] is a declarative formalism for knowledge representation and reasoning based on the stable model semantics [Gelfond and Lifschitz, 1991]. Efficient implementations of ASP, such as CLINGO [Gebser *et al.*, 2016] and DLV [Alviano *et al.*, 2017], are available, which made possible the development of concrete applications. In the recent years, ASP has been widely used for solving several problems in the context of artificial intelligence, such as game theory [Amendola *et al.*, 2016], natural language processing [Schüller, 2016], natural language understanding [Cuteri *et al.*, 2019b], robotics [Erdem and Patoglu, 2018], scheduling [Dodaro and Maratea, 2017], and more [Erdem *et al.*, 2016]. Therefore, the improvement of ASP systems is an interesting research topic in artificial intelligence. Traditional ASP systems are based on the ground&solve approach [Kaufmann *et al.*, 2016], in which a *grounder* module transforms the input program (containing variables) in its propositional counterpart, whose stable models are subsequently computed by the *solver* module. ASP solvers implement an extension the Conflict Driven Clause Learning (CDCL) algorithm [Kaufmann *et al.*, 2016]. Although the ASP implementations based on ground&solve are known to be effective in many contexts [Erdem *et al.*, 2016], the traditional approach has an intrinsic limitation. In particular, there are classes of programs whose evaluation is not feasible because of the combinatorial blowup of the grounding of some rules. This issue is usually referred to as *grounding bottleneck*. In many practical cases the grounding bottleneck is due to one or few constraints that model the (non) admissibility of problem solutions [Ostrowski and Schaub, 2012; Calimeri *et al.*, 2016].

In the literature, there are several attempts to solve the grounding bottleneck problem [Gebser *et al.*, 2018]. Some of these are based on language extensions that hybridize ASP with other formalisms (such as constraint programming [Ostrowski and Schaub, 2012; Balduccini and Lierler, 2017], and difference logic [Gebser *et al.*, 2016; Susman and Lierler, 2016]) that can be used to express the hard-to-ground constraints. Hybrid formalism are efficiently evaluated by coupling an ASP systems with a solver for the other theory, thus circumventing the grounding bottleneck. There are also approaches that work on plain ASP, such as *lazy grounding* techniques, that resulted in several promising systems, such as GASP [Dal Palù *et al.*, 2009], ASPERIX [Lefèvre and Nicolas, 2009] and ALPHA [Weinzierl, 2017]. The idea of lazy grounding is to instantiate a rule only when its body is satisfied. In this way, it is possible to prevent the grounding of rules which are unnecessary during the search of an answer set. Albeit lazy grounding techniques obtained good preliminary results, their performance is still not competitive with state-of-the-art systems [Gebser *et al.*, 2018]. A different approach was proposed in [Cuteri *et al.*, 2017], where problematic constraints are removed from the non-ground input program and the resulting program is provided as input to a modified version of a CDCL-based able to simulate the presence of problematic constraints. In [Cuteri *et al.*, 2017] the authors compared two alternative strategies for extending ASP solvers based on CDCL, namely *lazy instantiation*

---

[*] Contact Author

and *propagators*. In the lazy instantiation strategy, the solver computes a stable model of the program without problematic constraints. If this stable model satisfies also the omitted constraints, then it is also a stable model of the original program. Otherwise, the violated instances of these constraints are lazily instantiated, and the search continues. The other strategy relies on an extension of the propagation function by adding custom *propagators*, whose role is to perform the inferences of missing constraints during the search. However, both lazy instantiation and propagators were based on procedures written in a imperative language and that are specific for the problem at hand. This approach looses the declarative nature of ASP, and is a time consuming task that can be carried out only by developers expert on system APIs. Recently, Cuteri et al. [Cuteri *et al.*, 2019a] presented a strategy to translate (or compile) some non-ground constraints into a dedicated C++ procedure, which is used by the system to lazily instantiate them in an automatic way. This approach keeps declarativity of ASP and is effective when the problematic constraints are likely to be satisfied by a candidate model (i.e., whenever lazy instantiation is effective cfr. [Cuteri *et al.*, 2017]). However, a significant number of problems, especially hard combinatorial problems from ASP competitions [Calimeri *et al.*, 2016] cannot be handled efficiently by systems relying on lazy instantiation [Cuteri *et al.*, 2017].

In this paper, we push forward the idea of [Cuteri *et al.*, 2017; Cuteri *et al.*, 2019a], and we present a novel strategy for translating (compiling) non-ground constraints into dedicated C++ procedures that are used *as propagators* during the search of the CDCL algorithm. Differently from [Cuteri *et al.*, 2019a], propagators operate on partial interpretations and require radically different algorithms that are more involved than methods on total interpretations. To assess the performance of our approach, we implemented it on top of WASP [Alviano *et al.*, 2015] and conducted an experimental analysis on different benchmarks proposed in the literature. Results show that our approach outperforms state-of-the-art ASP systems in all tested scenarios.

2 Preliminaries

2.1 Answer Set Programming

An ASP program π is a finite set of rules of the form $h_1|\ldots|h_n$:- $b_1,\ldots,b_m$. where $n,m \geq 0$, $n+m \neq 0$, $h_1,\ldots,h_n$ are atoms and represent the *head* of the rule, while $b_1,\ldots,b_m$ are literals and represent the *body* of the rule. We denote by $body(r)$ the set of literals appearing in the body of r. In particular, an *atom* is an expression of the form $p(t_1,\ldots,t_k)$, where p is a predicate of arity k and $t_1,\ldots,t_k$ are *terms*. Terms are alphanumeric strings and are either variables or constants. According to Prolog conventions, only variables start with uppercase letters. A *literal* is an atom a or its negation ${\sim}a$, where $\sim$ denotes the *negation as failure*. A literal is said to be *positive* if it is an atom and *negative* if it is the negation of an atom. For an atom a, the complement is $\overline{a} = {\sim}a$, for a negated atom ${\sim}a$, the complement is $\overline{{\sim}a} = a$. For a literal l, $trm(l)$ denotes the list of terms in l, and $pred(l)$ is the name of the predicate of l. A rule is called a *constraint* if $n = 0$, and a *fact* if $n = 1$ and $m = 0$.

Algorithm 1 ComputeStableModel

```
   Input  : A ground program $\mathcal{P}$
   Output: A stable model for $\mathcal{P}$ or $\bot$
1  begin
2      $I := \emptyset$;
3      $I :=$ Propagate($I$);
4      if $I$ is inconsistent then
5          $r :=$ CreateConstraint($I$) ;
6          $I :=$ RestoreConsistency($I$);
7          if $I$ is consistent then $\mathcal{P} := \mathcal{P} \cup \{r\}$;
8          else return $\bot$;
9      else if $I$ total then return $I$;
10     else
11         $I :=$ RestartIfNeeded($I$);
12         $\mathcal{P} :=$ DeleteConstraintsIfNeeded($\mathcal{P}$);
13         $I := I \cup$ ChooseLiteral($I$);
14     goto 3;
```

Function Propagate(I)

```
1  $\mathcal{I} = I$;
2  for $\ell \in \mathcal{I}$ do $\mathcal{I} := \mathcal{I} \cup Propagation(\mathcal{I}, \ell)$ ;
3  return $\mathcal{I}$;
```

An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Rules are *safe*, that is each variable occurs in a positive literal of the body. 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. $\mathcal{B}$ denotes $B_\pi \cup \overline{B_\pi}$. Given a rule r, $gnd(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_π. For a program π, the *ground instantiation* $gnd(\pi)$ of π is the set $\bigcup_{r\in\pi} gnd(r)$. Stable models of a program π are defined using its ground instantiation $gnd(\pi)$. An interpretation I for π is a set of literals. I is total if $\forall a \in B_\pi$, either $a \in I$ or ${\sim}a \in I$ and $l \in I \implies \overline{l} \notin I$. Given an interpretation I, I^+ denotes the set of positive literals in I and I^- denotes the set of negative literals in I. A ground literal l is *true* w.r.t. I if $l \in I$, otherwise it is false. A total interpretation I is a *model* for π if, for every $r \in gnd(\pi)$, at least one atom in the head of r is true w.r.t. I whenever all literals in the body of r are true w.r.t. I. The *reduct* of a ground program π w.r.t. a model I is the ground program π^I, obtained from π by (i) deleting all rules $r \in \pi$ whose negative body is false w.r.t. I and (ii) deleting the negative body from the remaining rules. An interpretation I is a *stable model* of a program π if I is a model of π, and there is no J such that J is a model of π^I and $J^+ \subset I^+$. A program π is *coherent* if it admits at least one stable model, *incoherent* otherwise.

2.2 Classical CDCL 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 $ComputeStableModel(\Pi)$ for finding a stable model of program Π is shown in Algorithm 2. The Function 1 combines unit propagation (as in SAT) with some additional ASP-specific propagations, which ensures the model is stable (cf. [Kaufmann *et al.*, 2016]). The algorithm calls for each additional propagator a procedure called Propagation in Function 1. which takes as input a true literal, and the interpretation, and returns as output the extended interpretation with literals that are inferred and their reason.

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 $r \in \Pi$ 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 $C(r) \setminus \{\ell\} \subseteq I$. In the following we refer to $C(r) \setminus \{\ell\}$ as the *reason* for the inference of ℓ.

Pairs (ℓ, reason of ℓ) are stored during the execution of propagation, and will be used to perform conflict resolution, and more specifically during the UIP computation [Alviano *et al.*, 2015]. We refer to the list of such pairs as the *implication list*. 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 1 *Consider the following ground program* Π_1

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

$ComputeStableModel(\Pi_1)$ *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* $\{a\}$*, i.e.,* $I = \{a\}$*. Next,* $Propagate(I)$ *is called which yields* $I = \{a, b, \sim b\}$ *where* $\sim b$ *comes from unit propagation on* g_5 *and* b *from unit propagation on* g_6*. I is inconsistent and an analysis yields the reason of the conflict, i.e.,* $CreateConstraint(I) = \{g_7\}$ *with* $g_7 : \leftarrow\ a$*. Intuitively, the truth of* a *leads to an inconsistent interpretation, thus* a *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* $I = \{\sim a, b\}$*, where* $\sim a$ *comes from unit propagation on* g_7*, and* b *from unit propagation on* g_2*. I is partial and consistent, therefore lines 11–13 are executed. Assume again that no restart and no constraint deletion happens, and that* $ChooseLiteral(I) = \{c\}$*. Therefore, the algorithm continues in line 3 with* $I = \{\sim a, b, c\}$*. Propagation yields* $I = \{\sim a, b, c, \sim d\}$ *because* $\sim d$ *is support-propagated w.r.t.* g_3 *and* I *(or unit-propagated w.r.t. the completion of* g_3 *and* I*). I is total and consistent, therefore the algorithm returns* I *as the first stable model.*

3 Constraints as Propagators

In this section we present our strategy for evaluating constraints with propagators that are automatically generated by a compilation-based approach. In the following, we assume w.l.o.g. that the bodies of constraints never contain two literals with the same predicate name. Algorithms 3-4 present the pseudo-code of the compiler generating propagators from constraints. To ease readability, we write in red the code that is produced by the compiler, and in black the code (e.g., variables and references) that are in the scope of the compiler. Thus, red lines, written inside the symbols « and », represent the code that is printed by the compiler, and black references in gray lines denote the fact that the compiler is printing the value of such variables. For example, «case "pred(C)"» is equivalent to the C++ instruction `printf("case \"%s\"",pred(C))`. Algorithm 3 takes as input a non-ground ASP constraint C and prints the code of the propagator for C. It starts by declaring in the propagator code an empty implication list (line 2) which will be in charge of accumulating the result of the propagation of a literal l (the literal in input to the propagator), which we call *starting* literal. Depending on the predicate name of l, the propagator code must evaluate one of the $|body(C)|$ possible join orders. To do so, it writes a switch on the predicate name of l, writing one case for each literal in the constraint. In each case, it prints a dedicated code that is able to propagate l calling algorithm 4. The evaluation is written as a nested join loop, which is a cascade of `for` and `if` blocks.

Algorithm 2 CompilePropagateConstraint

```
   Input  : A constraint $C$
   Output: Prints the propagator for $C$.
1  begin
2      «$I_l = \emptyset$»
3      «switch pred(l) »
4      forall the $c \in body(C)$ do
5          «   case "pred(c)":»
6          CompilePropagateConstraintWithStarter(c,C)
7          «   break»
8      «return $I_l$»
```

We now describe what is happening in algorithm 4, which receives a constraint C and a literal $c \in C$. First it builds a substitution σ that will be used and updated in the whole nested join. Initially, σ is set to the empty substitution ϵ (line 1). Then, from line 2 to 4 the compiler writes the code that adds to σ a mapping from the variables in c (known at compilation time, thus black in the algorithm) to the constants in l (ground literal known at execution time). Recall that $trm(x)$ returns the list of terms of a literal x. The square notation, commonly used in C++ programming, denotes the access of a list to a specific (one-based) index (e.g. if the list is $trm(c) = [X, 2, Y]$, then $trm(c)[1]$ is X). At line 5, the algorithm reorders the body of C in a new list B where negative literals are always at the end of B, and c is not in B.

In the propagator code, u will be the complement of the literal that is unit propagated, and it is initialized to $\perp$ to denote that it is not known at the beginning. The printing of the code of the nested join loop starts at line 8. Essentially, at each iteration i we either print a `for` loop if $B[i]$ is positive and an `if` statement if it is negative. Each subsequent `for` or `if` is nested inside the previous one. When $B[i]$ is positive, the propagator will first collect in the set T_i the true literals that match (read it as *has a substitution to*) $\sigma(B[i])$, i.e. the current body literal B_i to which the building substitution σ is applied (line 10). Such true literals are joining the current substitution, and in case of propagation, will build the reason of the propagation. Moreover, if the undefined literal is still $\perp$ (line 12) it means the join must also collect the undefined literals that match $\sigma(B[i])$, line 11. At line 14 the propagator iterates with a variable b_i the union of the true literals and eventually undefined literals just collected. When b_i is undefined, u becomes b_i (lines 15 - 16). At this point, σ is extended with the variables of the current body literal $B[i]$, by mapping them to the constants in b_i (line 17), similarly to what was done before with c and l. On the other hand, when $B[i]$ is negative (line 20) the compiler iterates over a negative literal of the constraint. In such case, all variables have a mapping in σ (because of safety condition of ASP and negative literals are at the end of B). The propagator will determine the ground literal $b_i = \sigma(B[i])$. At this point, the join can continue either if b_i is true, or the undefined u is still $\perp$ and b_i is undefined (line 22). In the second case, u will be set to be equal to b_i. Once the compiler completes the writing of the cascade of nested `for` and `if` blocks, and it is in the most nested block, it can write the code that collects the successful match of the constraint as an additional pair in the implication list of the propagator (lines 25-29). The literal l is always part of the reason R (line 25), and the propagator will also add to R all the true literals in the join (all b_i, beside $b_i = u$). The complement of u which is a propagated literal, together with its reason R are added to the implication list (29). Finally, from line 30 to line 34, the compiler writes the code that rolls back sigma to its previous state (at the end of each block), and eventually sets u back to $\perp$ in case it is equal to b_i and closes the parenthesis of the nested `for` and `if` blocks. In rough terms, the compiler produces that code of a procedure that is able to find an instantiation of the constraint with a single undefined literal to be unit propagated.

Algorithm 3 CompilePropagateConstraintWithStarter

Input : A constraint C, a literal $c \in C$
Output: Prints an algorithm that is able to perform the unit propagation of C starting from a ground literal whose predicate is the same of c

```
1  «$\sigma = \epsilon$»
2  forall the $k = 1, \dots, |trm(c)|$ do
3      if trm(c)[k] is variable then
4          «$\sigma = \sigma \cup \{trm(c)[k] \mapsto trm(l)[k]\}$»
5  $B = computeBodyOrdering(C, c)$
6  «$u := \perp$»
7  forall the $i = 1, \dots, |B|$ do
8      «$\sigma_i = \sigma$»
9      if $B[i]$ is positive then
10         «$T_i = \{t \in I^+ \mid match(\sigma(B[i]), t)\}$»
11         «$U_i = \emptyset$»
12         «if $u = \perp$»
13         «   $U_i = \{p \in (\mathcal{B} \setminus I)^+ \mid match(\sigma(B[i]), p)\}$»
14         «for $b_i \in (T_i \cup U_i)$ { »
15         «   if $b_i \in U_i$»
16         «       $u = b_i$»
17         forall the $k = 1, \dots, |trm(B[i])|$ do
18             if trm(B[i])[k] is variable then
19                 «$\sigma = \sigma \cup \{trm(B[i])[k] \mapsto trm(b_i)[k]\}$»
20     else
21         «$b_i = \sigma(B[i])$»
22         «if $b_i \in I \vee (u = \perp \wedge b_i \in (\mathcal{B} \setminus I))\{$»
23         «   if $u = \perp \wedge b_i \in (\mathcal{B} \setminus I)$»
24         «       $u = b_i$»
25 «                                   $R = \{l\}$»
26 forall the $i = 1, \dots, |B|$ do
27     «                               $R = R \cup \{b_i\}$»
28 «                                   $R = R \setminus \{u\}$»
29 «                                   $I_l = I_l \cup (\overline{u}, R)$»
30 forall the $i = |B|, \dots, 1$ do
31     «   $\sigma = \sigma_i$»
32     «   if $u = b_i$»
33     «       $u = \perp$»
34     «$\}$»
```

3.1 Compilation Example

To better understand what the compiler actually prints, in algorithm 5 we provide an example of a generated propagator for a small constraint. The compiled constraint C in the example is `b(X), c(X,Y), ∼ d(X,1)`. The input of the propagator is an interpretation I and a starting literal l, and the output is the implication list containing literals to be propagated with their reasons.

First the propagator switches on the predicate name of the starting literal l (line 2). There is a case for each predicate name appearing in C, i.e., "b", "c" and "d". For the sake of readability we present the single case for predicate name "b" (lines 4-36). The propagator creates the substitution σ, where the variable X is mapped to the first (and only) constant of l (lines 4 and 5). Then the literal u is initialized to $\perp$ and σ is stored in σ_1 so that it can be rolled back at each iteration of the next loop. Subsequently, the propagator collects all the true atoms of $c(X, Y)$, where X is first replaced by the constant in l by the application of σ, as well as all the undefined atoms of $c(X, Y)$ analogously. Then, the propagator iterates over true and undefined atoms at line 12, continuing the join. For each b_1 in $T_1 \cup U_1$, the propagator checks whether b_1 is undefined and eventually updates u, if no undefined has been found up to now. Then σ is extended by the mappings of X and Y respectively to the first and second constants in b_1 (lines 16-17). At this point, in line 18, the propagator handles the negative literal $\sim d(X, 1)$, which becomes a ground literal b_2 after the substitution with σ; and the propagator checks whether b_2 is the first undefined (and updates u), or b_2 is true and a complete substitution is found. In these cases the propagator execution would reach the innermost code (from line 24) where it adds a propagation to the output implication list. In particular, if $u = b_1$, $\overline{b_1}$ is the propagated literal and the reason is $R \cup \{b_2\}$, otherwise ($u = b_2$), $\overline{b_2}$ is the propagated literal and the reason is $R \cup \{b_1\}$. At lines 30 to 32 and 33 to

36, the propagator restores the state of σ and eventually the undefined literal to $\perp$ and closes a nested code block.

3.2 Implementation

The implementation follows the execution presented in pseudo-code in algorithms 3 and 4. The compiler has been implemented in C++, and its output is also C++ code compliant to the WASP propagator interface, and is loaded in the ASP solver as a C++ dynamic library. Several optimization have been implemented, some of them discussed in the following.

Nested loops are made efficient by using proper indexes on terms, for which we use hash-maps to access matching ground literals. This is possible since the order of evaluation is fixed and the predicates extensions can be indexed statically (index terms are known at compilation time). For example, if we evaluate `:- a(X,Y), b(Y,Z)` with a as starter, we can benefit from indexing the extension of the predicate `b` on the first term (variable `Y`). By using a predicate-wise split of the interpretation and indexes, we highly optimize the propagator code related to interpretation access (e.g. lines 10, 13 and 22 of algorithm4). Moreover, even though, in the compiler pseudo-code, we assumed that there is no repetition of predicates names, we explicitly handle such cases in the implementation: basically, we detect when the same undefined literal is used more than once in the same nested join. The implementation also supports built-in arithmetic comparisons, and features a heuristic to employ a smart body ordering (line 5 of Algorithm 3). At the moment aggregates cannot be compiled. The latest release is available at https://github.com/wasp-eager/wasp-eager together with the experiments benchmarks.

4 Experiments

In order to empirically assess the impact of the proposed technique, we considered several benchmarks used in previous editions of the ASP Competitions [Calimeri *et al.*, 2016] or proposed in the literature, namely Incremental Scheduling, Natural Language Processing (NLP) using three different objective functions (cardinality, coherence and weighted abduction), Packing, and Partner Units. All benchmarks contain at least one constraint whose grounding is expensive from a computational point of view. We used as reference for the state-of-the-art the traditional ASP system CLINGO [Gebser *et al.*, 2016] and the lazy-grounding based ASP system ALPHA [Weinzierl, 2017]. It is important to observe that some of the encodings include features that are not currently supported by ALPHA, e.g., weak constraints. For such benchmarks, the performance of ALPHA is not reported. Moreover, we included in the comparison the solver WASP [Alviano *et al.*, 2015] and the version of WASP implementing the lazy propagator as described in [Cuteri *et al.*, 2019a] referred to as WASP-LAZY. In the following, our implementation is referred to as WASP-EAGER. All versions of WASP use CLINGO as grounder (executed with the option `-output=smodels`). WASP-LAZY and WASP-EAGER compile as lazy and eager propagators, respectively, *all* constraints that do not contain aggregates in the encodings, and therefore such constraints are not grounded in advance. Note that compilation is done only once for each benchmark and its running time is negligible (less than 2 seconds). The experiments were run on an Intel Xeon CPU E7-8880 v4 @ 2.20GHz, time and memory were limited to 2 hours and 5 GB, respectively.

An overview of the obtained results is given in Table 1: we report the number of solved instances, N/A indicates that the

Algorithm 4 Ex. of compiling :- b(X), c(X,Y), ~ d(X,1)

```
    Input  : An interpretation $I$ and a literal $l$ to propagate
    Output: An implication list
 1  $I_l = \emptyset$
 2  switch pred(l) do
 3      case "b"
 4          $\sigma = \epsilon$
 5          $\sigma = \sigma \cup \{X \mapsto trm(l)[1]\}$
 6          $u = \perp$
 7          $\sigma_1 = \sigma$
 8          $T_1 = \{t \in I^+ \mid match(\sigma(c(X,Y),t))\}$
 9          $U_1 = \emptyset$
10          if $u = \perp$ then
11              $U_1 = \{p \in (\mathcal{B}\setminus I)^+ \mid match(\sigma(c(X,Y),p)\}$
12          forall the $b_1 \in T_1 \cup U_1$ do
13              {
14              if $b_1 \in U_1$ then
15                  $u = b_1$
16              $\sigma = \sigma \cup \{X \mapsto trm(b_1)[1]\}$
17              $\sigma = \sigma \cup \{Y \mapsto trm(b_1)[2]\}$
18              $b_2 = \sigma(\sim d(X,1))$;
19              $\sigma_2 = \sigma$
20              if $b_2 \in I \vee (u = \perp \wedge b_2 \in (\mathcal{B} \setminus I))$ then
21                  {
22                  if $u = \perp \wedge b_2 \in (\mathcal{B} \setminus I)$ then
23                      $u = b_2$;
24                  $R = \{l\}$;
25                  $R = R \cup \{b_1\}$;
26                  $R = R \cup \{b_2\}$;
27                  $R = R \setminus \{u\}$;
28                  $I_l = I_l \cup \{(\overline{u}, R)\}$;
29                  $\sigma = \sigma_2$;
30                  if $u = b_2$ then
31                      $u = \perp$
32                  }
33              $\sigma = \sigma_1$
34              if $u = b_1$ then
35                  $u = \perp$
36              }
37      case "c"
38          ...;      // analogous to previous case
39          break
40      case "d"
41          ...;      // analogous to previous case
42          break
43      return $I_l$
```

Problem	#	ALPHA	CLINGO	WASP	WASP-LAZY	WASP-EAGER
Incr. Sched.	50	N/A	33	29	2	**36**
NLP (card)	50	N/A	42	44	**50**	**50**
NLP (coh)	50	N/A	41	44	**50**	**50**
NLP (wa)	50	N/A	42	44	**50**	**50**
Packing	50	0	0	0	0	**35**
Part. Units	47	0	5	0	0	**10**
Total	297	0	160	158	152	**231**

Table 1: Number of solved instances for each benchmark.

solver does not support the encoding of the benchmark. As a first comment, it is possible to observe that WASP-EAGER outperforms all other tested systems, solving 71, 73, and 79 more instances than CLINGO, WASP, and WASP-LAZY, respectively. Interestingly, WASP-EAGER outperforms the traditional systems CLINGO and WASP in all benchmarks. In particular, we observe that CLASP and WASP often exceed the memory limit in many of the unsolved instances due to the large number of constraints produced during the grounding step, whereas WASP-EAGER does not instantiate such constraints. This is particularly evident in Packing, where no instance can be grounded within the allotted memory limit and therefore the solving step of CLINGO and WASP does not even start. Concerning WASP-LAZY, we observe that it is competitive with WASP-EAGER only on NLP as shown also in [Cuteri *et al.*, 2017]. This behavior can be explained by the fact that WASP-LAZY lazily instantiates the problematic constraints whenever they are violated by a stable model candidate. In NLP, WASP-LAZY performs a small number of failing stable model checks (on average less than 100 checks are required). Instead, in other problems, each stable model contains many violations of constraints, leading the solver to inefficient search in harder instances. Finally, we report that ALPHA solves no instance within the allotted time and memory. However, ALPHA exceeds the time limit in 22 out of 47 Packing instances, where as CLINGO and WASP exceed the memory limit for all unsolved instances. This confirms that lazy grounding systems, like ALPHA, can use less memory than ground&solve systems, even if in this benchmark ALPHA is not competitive with WASP-EAGER.

Concerning solving times, we show aggregated results in the cactus plot of Figure 1. We recall that in a cactus plot a line is reported for each combination of tested system; where instances are ordered by solving time and a point (i, j) in the graph represents that the i-th instance is solved in j seconds. It is possible to observe that WASP-EAGER can solve the majority of the instances (217 out of 231) within 1200 seconds and it in general scales better than all other tested systems, solving more instances with lower running times.

5 Related Work

Traditional ASP systems, like CLINGO [Gebser *et al.*, 2016] and DLV [Alviano *et al.*, 2017], are based on the ground&solve approach. Grounding is performed using semi-naïve database evaluation techniques [Ullman, 1988] for avoiding duplicate work during grounding, whereas

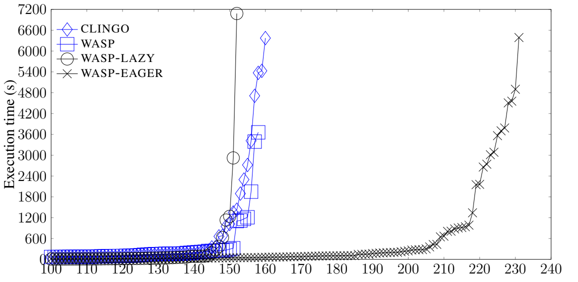

Figure 1: Comparison of all tested systems.

solvers are nowadays based on the CDCL algorithm [Kaufmann *et al.*, 2016]. These approaches intrinsically suffer from the grounding bottleneck. The only alternative to the traditional approach working on plain ASP is lazy grounding, as implemented by GASP [Dal Palù *et al.*, 2009], ASPERIX [Lefevre *et al.*, 2017], and ALPHA [Weinzierl, 2017]. In lazy grounding a rule is instantiated only when its body is satisfied in the current assignment of the search process.

Our approach extends traditional systems to avoid the grounding bottleneck due to constraints, and can be viewed as a hybrid solution between traditional and lazy grounding systems. Indeed, given a non-ground input program Π and a set of constraints $C \subset \Pi$, it uses the ground&solve approach on $\Pi \setminus C$, whereas constraints in C are not instantiated but simulated by propagators automatically integrated in the CDCL. An important difference with lazy grounding is that in our approach the problematic constrains are never instantiated.

Our strategy is similar in principle with the lazy approach presented in [Cuteri *et al.*, 2019a]. However, constraints in C are not compiled into propagators but they are instantiated when they are violated by a stable model of $\Pi \setminus C$. Moreover, the lazy approach works only on total interpretations, whereas our propagator-based approach works also on partial interpretations. This has a huge impact on the performance as shown in [Cuteri *et al.*, 2017] and in our experiments. Notably, our approach needs no language extension as CASP [Balduccini and Lierler, 2017; Ostrowski and Schaub, 2012], ASPMT [Bartholomew and Lee, 2014] and DLVHEX [Redl, 2016].

6 Conclusion

In this paper, we presented a novel approach for the automatic compilation of constraints into propagators and we implemented it on top of the ASP solver WASP. The performance of our tool has been empirically validated on benchmarks whose evaluation was not feasible due to the combinatorial blowup of the grounding of some constraints. Results show that our solution outperforms state-of-the-art systems on benchmarks modeling AI problems. Concerning future work, we plan to extend our implementation for supporting also other ASP constructs, such as aggregates.

Acknowledgments

This work was partially supported by MISE under PON MAP4ID Prog. n. F/190138/01-03/X44.

References

[Alviano and Dodaro, 2016] Mario Alviano and Carmine Dodaro. Completion of disjunctive logic programs. In *IJCAI*, pages 886–892. IJCAI/AAAI Press, 2016.

[Alviano *et al.*, 2015] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015.

[Alviano *et al.*, 2017] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017.

[Amendola *et al.*, 2016] Giovanni Amendola, Gianluigi Greco, Nicola Leone, and Pierfrancesco Veltri. Modeling and reasoning about NTU games via answer set programming. In *IJCAI*, pages 38–45. IJCAI/AAAI Press, 2016.

[Balduccini and Lierler, 2017] Marcello Balduccini and Yuliya Lierler. Constraint answer set solver EZCSP and why integration schemas matter. *TPLP*, 17(4):462–515, 2017.

[Bartholomew and Lee, 2014] Michael Bartholomew and Joohyung Lee. System aspmt2smt: Computing ASPMT theories by SMT solvers. In *JELIA*, volume 8761 of *Lecture Notes in Computer Science*, pages 529–542. Springer, 2014.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[Calimeri *et al.*, 2016] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.

[Cuteri *et al.*, 2017] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP*, 17(5-6):780–799, 2017.

[Cuteri *et al.*, 2019a] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Partial compilation of ASP programs. *TPLP*, 19(5-6):857–873, 2019.

[Cuteri *et al.*, 2019b] Bernardo Cuteri, Kristian Reale, and Francesco Ricca. A logic-based question answering system for cultural heritage. In *JELIA*, volume 11468 of *Lecture Notes in Computer Science*, pages 526–541. Springer, 2019.

[Dal Palù *et al.*, 2009] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[Dodaro and Maratea, 2017] Carmine Dodaro and Marco Maratea. Nurse scheduling via answer set programming. In *LPNMR*, volume 10377 of *LNCS*, pages 301–307. Springer, 2017.

[Erdem and Patoglu, 2018] Esra Erdem and Volkan Patoglu. Applications of ASP in robotics. *KI*, 32(2-3):143–149, 2018.

[Erdem *et al.*, 2016] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[Gebser *et al.*, 2016] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP TCs*, volume 52 of *OASICS*, pages 2:1–2:15, 2016.

[Gebser *et al.*, 2018] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, pages 5450–5456. ijcai.org, 2018.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[Kaufmann *et al.*, 2016] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[Lefèvre and Nicolas, 2009] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver: Asperix. In *LPNMR*, volume 5753 of *LNCS*, pages 522–527. Springer, 2009.

[Lefevre *et al.*, 2017] Claire Lefevre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming*, 17(3):266–310, 2017.

[Ostrowski and Schaub, 2012] Max Ostrowski and Torsten Schaub. ASP modulo CSP: the clingcon system. *TPLP*, 12(4-5):485–503, 2012.

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

[Schüller, 2016] Peter Schüller. Modeling variations of first-order horn abduction in answer set programming. *Fundam. Inform.*, 149(1-2):159–207, 2016.

[Susman and Lierler, 2016] Benjamin Susman and Yuliya Lierler. SMT-based constraint answer set solver EZSMT (system description). In *ICLP TCs*, volume 52 of *OASICS*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[Ullman, 1988] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.

[Weinzierl, 2017] Antonius Weinzierl. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *LPNMR*, volume 10377 of *LNCS*, pages 191–204, 2017.