External Propagators in WASP: Preliminary Report

Carmine Dodaro[1], Francesco Ricca[1], and Peter Schüller[2]

[1] Department of Mathematics and Computer Science
University of Calabria, Italy
{dodaro,ricca}@mat.unical.it
[2] Computer Engineering Department, Faculty of Engineering
Marmara University, Turkey
peter.schuller@marmara.edu.tr

Abstract. State-of-the-art ASP solvers are based on a variant of the CDCL algorithm. One of the key features of CDCL is the propagation step, whose role is to implement deterministic consequences of the input theory. It is well-known that the performance of solvers can be considerably improved on specific benchmarks by adding custom propagation functions. However, embedding a new propagator into an existing solver often requires non-trivial modifications. In this paper, we report on an extension of the ASP solver WASP that allows to provide new propagators externally, i.e. no modifications of the solver are needed. We assess our proposal on a recent application of ASP to abduction in Natural Language Understanding, where plain ASP solvers are not effective. Preliminary experiments on real-world instances show encouraging results.

Keywords: Answer Set Programming, Propagators, Natural Language Understanding

1 Introduction

Answer Set Programming (ASP) is a powerful paradigm for knowledge representation and reasoning based on the stable models semantics [1]. ASP has been applied for solving complex problems in several areas, including Artificial Intelligence [2], Bioinformatics [3], E-tourism [4] and Databases [5], to mention a few.

The success of ASP is due to the combination of its high knowledge-modeling power with robust solving technology [6, 7]. ASP systems are usually based on two modules. The first module is the grounder, which is responsible for the elimination of variables by creating a ground (or propositional) program equivalent to the input one. After the grounding process, the next module, usually called solver, computes the answer sets of the program. State-of-the-art ASP solvers perform the computation of the answer sets applying techniques introduced for SAT solving, such as CDCL backtracking search algorithm [8]. One of the key features of CDCL is the propagation step, whose role is to implement deterministic consequences of the input theory.

It turns out that many extensions of the plain ASP language such as *aggregates* [9], *acyclicity constraints* [10], and *Constraint ASP* [11] have been implemented by adding new propagation functions to the plain CDCL algorithm. Recently, Janhunen et al.

in [12] suggest that the performance of CDCL-based solvers can be considerably improved on specific benchmarks by adding custom propagation functions. However, the integration of new propagators into existing solvers often requires a deep knowledge of the internal implementation details.

In this paper, we report on an extension of the ASP solver WASP [13, 6] that makes it easier for developers to embed new external propagators in the solver. In particular, it offers multi-language support including scripting languages that require no modifications to the solver, as well as a C++ interface for performance-oriented implementations.

We assess our proposal on a recent application of ASP to abduction in Natural Language Understanding [14], where plain ASP solvers are not effective. In particular, it has been shown in [14] that the grounding of all constraints makes the subsequent solving step too hard for state-of-the-art solvers. In particular, a small set of constraints, i.e. the ones related to the transitivity condition, causes a grounding blow-up of the program that makes the usage of plain ASP not viable. For this reason, we implemented a propagator in WASP that checks whenever a transitivity violation is detected in an answer set candidate and then instantiates the constraints related to transitivity lazily, so to avoid the grounding blow-up. Preliminary results on real-world instances are encouraging: the performance of the propagator is better than approaches based on plain ASP on two out of three objective functions.

2 Preliminaries

In this section we briefly recall the Answer Set Programming (ASP) language and contemporary solving techniques.

2.1 Syntax and Semantics

Let $\mathcal{A}$ be a fixed, countable set of propositional atoms including $\bot$. A literal ℓ is either an atom a, or an atom preceded by the negation as failure symbol $\sim$. The complement of ℓ is denoted by $\overline{\ell}$, where $\overline{a} = \sim a$ and $\overline{\sim a} = a$ for an atom a. For a set L of literals, $\overline{L} := \{\overline{\ell} \mid \ell \in L\}$, $L^+ := L \cap \mathcal{A}$, and $L^- := \overline{L} \cap \mathcal{A}$. A program Π is a finite set of rules. A rule is an implication $a \leftarrow l_1, \ldots, l_n$, where a is an atom, and $l_1, \ldots, l_n$ are literals, $n \geq 0$. For a rule r, $H(r) = \{a\}$ is called the head of r and $B(r) = \{l_1, \ldots, l_n\}$ is called the body of r. A rule r is a fact if $B(r) = \emptyset$, and is a constraint if $H(r) = \{\bot\}$. A (partial) interpretation is a set of literals I containing $\sim\bot$. I is inconsistent if $I^+ \cap I^- \neq \emptyset$, otherwise I is consistent. I is total if $I^+ \cup I^- = \mathcal{A}$. Given an interpretation I, a literal ℓ is true if $\ell \in I$; is false if $\overline{\ell} \in I$, and is undefined otherwise. An interpretation I satisfies a rule r if $I \cap (H(r) \cup \overline{B(r)}) \neq \emptyset$. Let Π be a program, a model I of Π is a consistent and total interpretation that satisfies all rules in Π. The reduct of Π w.r.t. I is the program Π^I obtained from Π by (i) deleting all rules r having $B(r)^- \cap I \neq \emptyset$, and (ii) deleting the negative body from the remaining rules [1]. A model I of a program Π is an answer set if there is no model J of Π^I such that $J^+ \subset I^+$. A program Π is coherent if it admits answer sets, otherwise it is incoherent.

Algorithm 1: ComputeAnswerSet

```
   Input  : A program Π
   Output: An answer set for Π or ⊥
1  begin
2      I := {∼⊥};
3      (Π, I) := SimplifyProgram(Π,I);
4      I := Propagate(I);
5      if I is inconsistent then
6          I := RestoreConsistency(I);
7          if I is consistent then Π := Π ∪ CreateConstraint(I);
8          else return ⊥;
9      else if I total then
10         if CheckConsistency(I) then return I;
11         else goto 6;
12     else
13         I := RestartIfNeeded(I);        Π := DeleteConstraintsIfNeeded(Π);
14         I := I ∪ ChooseLiteral(I);
15     goto 4;
```

Function Propagate(I)

```
1 for ℓ ∈ I do I := I ∪ Propagation(ℓ);
2 I' := PostPropagation(I);
3 if I' ≠ ∅ then I := I ∪ I'; goto 1;
4 return I;
```

2.2 Answer Sets Computation

The computation of answer sets can be carried out by employing an extended version of the Conflict-Driven Clause Learning (CDCL) algorithm, introduced for SAT solving [8], and reported here as Algorithm 1. The algorithm takes as input a program Π, and produces as output an answer set if Π is consistent, $\bot$ otherwise.

The computation starts by applying polynomial simplifications to strengthen and/or remove redundant rules on the lines of methods employed by SAT solvers [8]. After the simplifications step, the backtracking search starts. First, I is extended with all the literals that can be deterministically inferred by applying some inference rule (propagation step, line 4). Three cases are possible after a propagation step is completed: (i) I is consistent but not total. In that case, an undefined literal ℓ (called branching literal) is chosen according to some heuristic criterion (line 14), and is added to I. Subsequently, a propagation step is performed that infers the consequences of this choice. (ii) I is inconsistent, thus there is a conflict, and I is analyzed. The reason of the conflict is modeled by a fresh constraint r that is added to Π (learning, line 7). Moreover, the algorithm backtracks (i.e. choices and their consequences are undone) until the consistency of I is restored (line 6). The algorithm then propagates inferences starting from the fresh constraint r. Otherwise, if the consistency of I cannot be restored, the algo-

rithm terminates returning $\bot$. Finally, in case (iii) I is total, the algorithm performs a consistency check on the interpretation I (line 10). If I is inconsistent the conflict is analyzed as in (ii). Otherwise, the algorithm terminates returning I. This check is required whenever the specific implementation of the CDCL algorithm lazily postpone some propagation inference which is required to assure the consistency of I.

3 External Propagators in WASP

One of the key features of the algorithm for computing an answer set is the function Propagate. The role of propagation is to extend the interpretation with the literals that can be deterministically inferred.

Propagation in WASP. Propagation in WASP is implemented by a set of inferences rule, called *propagators*, taking in account the properties of ASP programs. In WASP, propagators are invoked according to their priorities. Higher priority propagators are applied by calling function $Propagation$ that includes the main inference rule called *unit* propagation. Lower priority propagators (also called post propagators) are applied later by calling function $PostPropagation$. As an example, in the implementation of WASP, $PostPropagation$ invokes the algorithm based on source pointers for *unfounded set* propagation [15].

External Propagators. The communication with external propagators follows a synchronous message passing protocol. The protocol is implemented (as customary in object-oriented languages) by means of method calls. Basically, an external propagator must be compliant with a specific interface. The methods of the interface are associated to specific events occurring during the search of an answer set. Whenever a specific point of the computation is reached the corresponding event is triggered, i.e., a method of the propagator is called. Some of the methods of the interface are allowed to return values that are subsequently interpreted by WASP. Our implementation supports propagators implemented in (i) *perl* and *python* for obtaining fast prototypes and (ii) *C++* in case better performance is needed. Note that *C++* implementations must be integrated in the WASP binary at compile time, whereas *perl* and *python* can be specified by means of text files given as parameters for WASP, thus scripting-based propagators do not require changes and recompilation of WASP. In order to simplify the description of the interface some technical details are omitted and we do not focus on a specific language. The source code and the documentation are available on the branch *plugins* at `https://github.com/alviano/wasp`.

In the following, each method of the library for specifying new propagators in WASP is described in a separate paragraph.

Method getLiterals(). This method is invoked at the beginning of the computation and it returns a list of literals L. Intuitively, literals in L are associated to the propagator, that is all changes of the truth values of literals in L will be notified to the propagator during search. Otherwise, literals that are not in L are ignored.

Method simplifyAtLevelZero(). This method is invoked before starting the search (line 3 of Algorithm 1) and returns a list of literals that have been identified to be true in all answer sets of the input program.

Method onLiteralTrue(ℓ). This method is invoked by the function $Propagation$ (line 1 of function Propagate) whenever a literal ℓ is inferred as true. The method returns a list of literals to infer as true.

Method onLiteralsTrue(L). This method is invoked by the function $PostPropagation$ (line 2 of function Propagate) and notifies that all literals in L became true. As the previous method, it returns a list of literals to infer as true.

Method getReasonForLiteral(ℓ). This method is invoked for each literal ℓ inferred as true by method *onLiteralTrue* (*onLiteralsTrue*) and returns a constraint modeling the reason for the assignment of ℓ. This reason might be used during the search if the literal ℓ is involved in a conflict.

Method onLiteralsUndefined(L). This method is invoked when some of the literals previously notified as true become again undefined (e.g. after an unroll or a restart).

Method checkAnswerSet(I). This method is invoked after an answer set candidate I is found (line 10 of Algorithm 1). The role of the method is to check whether the answer set is consistent with respect to the propagator. Therefore, it returns *true* if I is consistent, and *false* otherwise.

Method getReasonsForCheckFailure(). This method returns a list of constraints modeling the reasons for the failure triggered by the method *checkAnswerSet* (line 7 of Algorithm 1).

4 Preliminary Experiment

In this section we report on an experiment on a recently-proposed application of Answer Set Programming to abduction in Natural Language Understanding (NLU).

Case study. Abduction is a popular formalism for NLU, and we here consider a benchmark for first order Horn abduction under preference relations of cardinality minimality, coherence [16], and Weighted Abduction [17]. For example given the text "Mary lost her father. She is depressed." using appropriate background knowledge and reasoning formalism we can obtain the interpretation of the sentence that Mary is depressed *because* of the death of her father.

ASP formulations for the above NLU tasks under different objective functions (optimization criteria) were described in [14]. The prevalent evaluation strategy adopted by state of the art ASP systems, which is carried out by successively performing grounding (i.e., variable elimination) and solving (i.e., search for the answer sets of a propositional program), resulted to be not effective on large instances. This is due to the grounding blow-up caused by the following constraint which has $\mathcal{O}(n^3)$ ground instances.

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

Results. Table 1 shows preliminary experiments with the WASP solver on the Bwd-A encoding for first order Horn abduction from [14]. We show accumulated results for 50

natural language understanding instances from [16] for objective functions cardinality minimality, coherence [16], and Weighted Abduction [17].[3]

We compare two evaluation methods: *Constraint* instantiates all constraints during the initial grounding step and sends them to the solver, while *Propagator* omits a significant portion of constraints (those related to transitivity) from the initial grounding and instantiates them lazily in a new propagator whenever a transitivity violation is detected in an answer set candidate. The external propagator was implemented in python by providing functions *checkAswerSet()* and *getReasonForCheckFailure()*. Function *getLiterals()* was implemented by returning all literals whose predicate name was eq with arity 2. From a technical point of view, WASP uses internal integers to identify the literals in the program. The mapping from the symbolic representation of input atoms to the internal identifiers is done using the so called *symbol table*, which is provided as input to the propagator before all methods.

We observe that for all objective functions, there are out-of-memory conditions for 6 instances (maximum memory was 5 GB) while memory is not exhausted with propagators, and average memory usage is significantly lower with propagators (1.7 GB vs. around 150 MB). For cardinality minimality, the average time to find the optimal solution decreases sharply from 76 sec to 8 sec and we find optimal solutions for all instances. For coherence we can solve more instances optimally however the required time increases from 64 sec to 103 sec on average and 4 instances reach the timeout (600 sec). For Weighted Abduction, which represents the most complex optimization criterion, we solve fewer instances (37) compared with using pre-instantiated constraints (44 instances).

Propagators can clearly be used to trade space for time, and in some cases we decrease both space and time usage. For the complex Weighted Abduction objective functions, we can observe in the Odc column that many more invalid answer sets (2067) were rejected by the propagators compared with cardinality minimality (70) or coherence (751).

5 Related Work

The extension of CDCL solvers with propagators is at the basis of Satisfiability Modulo Theories (SMT) solvers [18]. These have been proved to be an effective extension of SAT solvers that extends the capability of a mature solving technology [8]. Similar extensions have been envisaged also for ASP [19]. Other extensions of ASP such as CASP [20] have been implemented by adding propagators to CDCL solvers [11].

The extension of WASP presented in this paper can serve as a platform for implementing such language extensions. Indeed, new propagators can be added to implement specific constraints (such as *acyclicity constraints* [10]), ASP modulo theories [19] and CASP [20], and can be also used for boosting the performance of WASP on specific benchmarks.

An extension similar to the one presented in this paper has been implemented in solvers by the Potassco group. The ASP solver CLASP [21] provides a C++ interface

[3] Encodings and WASP plugin source code is available in tag `rcra2016-wasp-prop` of repository `https://bitbucket.org/knowlp/asp-fo-abduction` .

Objective Function	Method	MO #	TO #	OPT #	T sec	M MB	Odc #
Cardinality Minimality	Constraint	6	0	44	76	1715	0
	Propagator	0	0	50	8	119	70
Coherence	Constraint	6	0	44	64	1723	0
	Propagator	0	4	46	103	131	751
Weighted Abduction	Constraint	6	0	44	66	1731	0
	Propagator	0	13	37	229	141	2067

Table 1. Experimental Results: MO/TO indicates number of instances where memory/time was exhausted, OPT the number of optimally solved instances, T/M indicates average time and memory usage on solved instances, and Odc shows number of times an answer set was invalidated and a new clause was learned, i.e., a constraint was lazily instantiated.

for post-propagation, where it is possible to invalidate an answer set candidate. The interface for defining new propagators is conceptually equivalent to the one presented in Section 3. However, at the moment CLASP does not support any external *python* (or *perl*) API to specify new propagators. A *python* library is currently supported by CLINGO [22]. First versions of the API supported by CLINGO (up to version 4) [22] have no concept of post propagators but only support the function *onModel*, which is called whenever an answer set is found. This interface does not behave well when used together with optimization, because rejecting an answer set does not prevent it to be used as a new bound. This limitation prompted the development of "workaround" Algorithm 2 [14, Section 11]. In this work we can realize the on-demand constraints without any workarounds for optimization. The version 5 of CLINGO [23] supports also a similar API to define external propagation using scripting languages.

6 Conclusion and Ongoing Work

In this paper, we preliminary report on a new library for embedding new external propagators into the ASP solver WASP. Our proposal has been assessed on a recent application of ASP to abduction in Natural Language Understanding [14] showing encouraging preliminary results. Our current prototype implementation only checks when a full answer set candidate has been found, while most violated constraints could also be detected based on a partial interpretations. Thus, we are implementing a propagator that can take more advantage from the interface of WASP by working on partial interpretations (this will require to use *onLiteralTrue()* or *onLiteralsTrue()* functions). We also plan to experiment with the optimal frequency of propagation, which is known to play a role in similar implementations for robotics planning. Moreover, our current prototype is able to learn only one single constraint per invalidated answer set, however one answer set might contain several violations of not instantiated constraints (the CLINGO-based engine in [14] can learn constraints for all violations and performs better than the current prototype). Adding all these at once might improve the performance of the solver to find an optimal solution.

Acknowledgements

We thank the anonymous reviewers for their detailed and useful comments. This work has been supported by The Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E777 and by MISE under project "PIUCultura", N. F/020016/01-02/X27.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4) (1991) 365–386
2. Erdem, E., Patoglu, V., Saribatur, Z.G., Schüller, P., Uras, T.: Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. TPLP **13**(4-5) (2013) 831–846
3. Erdem, E., Öztok, U.: Generating explanations for biomedical queries. TPLP **15**(1) (2015) 35–78
4. Dodaro, C., Nardi, B., Leone, N., Ricca, F.: Allotment problem in travel industry: A solution based on ASP. In: RR. Volume 9209 of LNCS., Springer (2015)
5. Manna, M., Ricca, F., Terracina, G.: Taming primary key violations to query large inconsistent data via ASP. TPLP **15**(4-5) (2015) 696–710
6. Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. Volume 9345 of LNCS., Springer (2015) 40–54
7. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: LPNMR. Volume 6645 of LNCS., Springer (2011) 345–351
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
9. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1) (2011) 278–298
10. Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B., Schaub, T.: Answer set programming modulo acyclicity. In: LPNMR. Volume 9345 of LNCS., Springer (2015) 143–150
11. Ostrowski, M., Schaub, T.: ASP modulo CSP: the clingcon system. TPLP **12**(4-5) (2012) 485–503
12. Janhunen, T., Tasharrofi, S., Ternovska, E.: SAT-to-SAT: Declarative Extension of SAT Solvers with New Propagators. In: AAAI, AAAI Press (2016) 978–984
13. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: LPNMR. Volume 8148 of LNCS., Springer (2013) 54–66
14. Schüller, P.: Modeling Variations of First-Order Horn Abduction in Answer Set Programming. Fundamenta Informaticae (2016) To appear, arXiv:1512.08899 [cs.AI].
15. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1-2) (2002) 181–234
16. Ng, H.T., Mooney, R.J.: Abductive Plan Recognition and Diagnosis: A Comprehensive Empirical Evaluation. In: Knowledge Representation and Reasoning. (1992) 499–508
17. Hobbs, J.R., Stickel, M., Martin, P., Edwards, D.: Interpretation as Abduction. Artif. Intell. **63**(1-2) (1993) 69–142
18. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). J. ACM **53**(6) (2006) 937–977
19. Bartholomew, M., Lee, J.: Functional stable model semantics and answer set programming modulo theories. In: IJCAI, IJCAI/AAAI (2013) 718–724

20. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: ICLP. Volume 3668 of LNCS., Springer (2005) 52–66
21. Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., Schaub, T.: Progress in clasp Series 3. In: LPNMR. Volume 9345. (2015) 368–383
22. Gebser, M., Kaminski, R., Obermeier, P., Schaub, T.: Ricochet robots reloaded: A case-study in multi-shot ASP solving. In: Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation. Volume 9060 of LNCS., Springer (2015) 17–32
23. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: ICLP (Technical Comm.). LIPIcs. (2016) To appear.