TPLP **20** (2): 225–248, 2020.
doi:10.1017/S1471068418000558 First published online 10 December 2018

Technical Note *The External Interface for Extending* WASP*

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

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

submitted 2 December 2017; revised 6 November 2018; accepted 06 November 2018

Abstract

Answer set programming (ASP) is a successful declarative formalism for knowledge representation and reasoning. The evaluation of ASP programs is nowadays based on the conflict-driven clause learning (CDCL) backtracking search algorithm. Recent work suggested that the performance of CDCL-based implementations can be considerably improved on specific benchmarks by extending their solving capabilities with custom heuristics and propagators. However, embedding such algorithms into existing systems requires expert knowledge of the internals of ASP implementations. The development of effective solver extensions can be made easier by providing suitable programming interfaces. In this paper, we present the interface for extending the CDCL-based ASP solver WASP. The interface is both *general*, that is, it can be used for providing either new branching heuristics or propagators, and *external*, that is, the implementation of new algorithms requires no internal modifications of WASP. Moreover, we review the applications of the interface witnessing it can be successfully used to extend WASP for solving effectively hard instances of both real-world and synthetic problems.

KEYWORDS: knowledge representation and reasoning, answer set programming, application programming interface, propagators, choice heuristics

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). ASP has been applied for solving complex problems in several areas, including artificial intelligence (Balduccini *et al.* 2001; Garro *et al.* 2006; Dodaro *et al.* 2015), bioinformatics (Erdem and Öztok 2015; Koponen *et al.* 2015), hydroinformatics (Gavanelli *et al.* 2015), databases (Marileo and Bertossi 2010; Manna *et al.* 2015;

* 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), and 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. Authors are grateful to Mario Alviano, Bernardo Cuteri, Philip Gasteiger, Nicola Leone, Benjamin Musitsch, Peter Schüller, and Kostyantyn Shchekotykhin for their suggestions.

Manna *et al.* 2013), and scheduling (Alviano *et al.* 2017; Abseher *et al.* 2016; Dodaro and Maratea 2017), to mention a few; see Erdem *et al.* (2016) for a detailed survey on ASP applications. The success of ASP is due to the combination of its high knowledge-modeling power and robust solving technology (Gebser *et al.* 2012; Maratea *et al.* 2012; Alviano *et al.* 2015; Gebser *et al.* 2015; Gebser *et al.* 2016; Lierler *et al.* 2016; Gebser *et al.* 2017; Alviano *et al.* 2017).

State-of-the-art ASP systems are usually based on the "ground+solve" approach (Kaufmann *et al.* 2016), in which a *grounder* transforms the input program (containing variables) in an equivalent variable-free one, whose stable models are subsequently computed by the *solver*. The computation of stable models is usually performed applying a variant of the conflict-driven clause learning (CDCL) backtracking search algorithm (Silva and Sakallah 1999; Zhang *et al.* 2001). Effective CDCL implementations require the combination of several features, including *choice heuristic* and *propagators*. Recently, it has been suggested that the performance of CDCL-based solvers can be considerably improved on specific benchmarks by adding domain-specific choice heuristics (Friedrich 2015) and propagators (Janhunen *et al.* 2016). However, extending in an effective way existing solvers with new algorithms is not obvious, because it requires in-depth knowledge of the internals of the implementations, which are nowadays very optimized and sophisticated.

In this paper, we provide a practical contribution in the aforementioned context, in particular we present the external programming interface of the ASP solver WASP (Dodaro *et al.* 2011; Alviano *et al.* 2013; Alviano *et al.* 2015), whose idea is to simplify the integration of custom heuristics and propagators in the solver. In particular, it offers multi-language support including *python* and *perl* languages that require no modifications to the solver, as well as *C++* for performance-oriented implementations. The interface was partially described in Dodaro *et al.* (2016) and used in the literature to embed domain-specific heuristics for two industrial problems proposed by Siemens, namely partner units and combined configuration. More recently, the interface has been also used to implement custom propagators as solution to the *grounding bottleneck* problem in three benchmarks, namely stable marriage, packing, and natural language understanding (NLU) (Cuteri *et al.* 2017). In particular, propagators were used to replace a small set of constraints causing a grounding blow-up of the program, and thus making the usage of plain "ground+solve" approach not viable.

This paper is organized as follows: in Section 2, we recall syntax and semantics of propositional ASP programs and contemporary solving techniques. In Section 3, we present the interface for extending the solving capabilities of WASP by adding new propagators and by modifying the choice heuristic. Subsequently, we show the usage of the interface by presenting three examples. In the first example, we realize new propagators as a tool for avoiding the grounding bottleneck while solving the stable marriage problem (Section 4.1). In the second example, we show how to obtain a naive solver for constraint answer set programming (CASP) (Baselice *et al.* 2005) (Section 4.2). The third one shows how the interface can be used for implementing a well-known general purpose choice heuristic (Section 4.3). Then, we review the successful applications of the interface in the literature, discussing its impact in improving the performance

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

of WASP (Section 5). Finally, after discussing related works in Section 6, we draw the conclusion.

2 Preliminaries

This section recalls syntax and semantics of propositional ASP programs and contemporary solving techniques. More detailed descriptions and a more formal account of ASP, including the features of the language employed in this paper, can be found in Baral (2003), Brewka *et al.* (2011), Gebser *et al.* (2012), Lifschitz (2016), and Janhunen and Niemelä (2016). Hereafter, we assume the reader is familiar with logic programming conventions.

2.1 ASP syntax and semantics

Syntax. 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 $n \geq 0$, $m \geq j \geq 0$. 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 divided into 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 literal ℓ, let $\overline{\ell}$ denote the complement of ℓ, that is, $\overline{a} = \sim a$ and $\overline{\sim a} = a$, for an atom a. For a set of literals S, let $\overline{S} = \{\overline{\ell} \mid \ell \in S\}$. 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 $n = 0$. Given a rule r of the form (1), $H(r)$ and $B(r)$ denote the set of atoms appearing in the head and in the body of r, respectively. 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 $\bigcup_{r \in \Pi} Ground(r)$. Given a program Π, $At(\Pi)$ denotes the set of atoms occurring in Π.

Semantics. 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, 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 $\overline{\ell}$ is not contained in L. Given a set of ground literals L, $L^+ \subseteq L$ denotes the set of positive literals in L. An interpretation I for Π is a consistent set of ground literals over

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

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 rule r is satisfied w.r.t. I if one of the atoms in the head is true w.r.t. I or one of the literals in the body is 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 M is a *stable model* for a positive program Π if $M^+ \subseteq X^+$, for each model X of Π.

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

Support. Given a model M for a ground program Π, we say that a ground atom $a \in M$ is *supported* w.r.t. 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.

Algorithm 1: ComputeStableModel

```
Input  : A ground program Π
Output: COHERENT if SM(Π) ≠ ∅. Otherwise, INCOHERENT
begin
  I := ∅;
  (Π, I):= SimplifyProgram(Π, I); // remove redundant rules and atoms from Π
  I := Propagate(I);           // propagate deterministic consequences of I
  if I is inconsistent then
     r := CreateConstraint(I);                                  // learning
     if B(r)=∅ then return INCOHERENT; // Π does not admit stable models
     Π := Π ∪ {r};
     I := RestoreConsistency(I, Π);         // unroll until I is consistent
  else if I is total then
     if CheckConsistency(I) then return COHERENT; // I is a stable model
     R := CreateConstraints(I);      // create constraints for the failure
     Π := Π ∪ R;
     I := RestoreConsistency(I, Π);         // unroll until I is consistent
  else
     I := RestartIfNeeded(I);                // restart of the computation
     Π := DeleteConstraintsIfNeeded(Π); // deletion of learned constraints
     I := I ∪ ChooseLiteral(I);                         // heuristic choice
  end
  goto 4;
end
```

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Function Propagate(I)

```
I := EagerPropagation(I);
if I is consistent then
    I' := PostPropagation(I);
    if I' = ∅ then return I;
    I := I ∪ I';
end
if I is inconsistent then return I;
goto 1;
```

2.2 CDCL algorithm for stable model computation

The computation of a stable model is usually carried out by employing the CDCL algorithm (Silva and Sakallah 1999; Zhang *et al.* 2001) with extensions specific to ASP (Kaufmann *et al.* 2016), reported here as Algorithm 1. The algorithm takes as input a propositional program Π and produces as output either COHERENT, if Π admits stable models, or INCOHERENT otherwise.

The computation starts by applying polynomial simplifications to strengthen and/or remove redundant rules on the lines of methods employed in Gebser *et al.* (2008) and inspired by Eén and Sörensson (2003) (see Dodaro (2015) for more details). After the simplifications step, the non-chronological backtracking search starts. First, a partial interpretation I, initially empty, 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 18) 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 (learning, line 6), computed in such a way to avoid the same conflict in the future search, for example, using the *first Unique Implication Point (UIP)* learning schema (Zhang *et al.* 2001). If the learning procedure determines that the conflict cannot be avoided, that is, the input program is incoherent, then the algorithm terminates returning INCOHERENT. Otherwise, the algorithm backtracks (i.e., choices and their consequences are undone, this is often referred to as *unroll* in the following) until the consistency of I is restored (line 9) and r is added to Π.

(iii) x I is total; the algorithm performs a consistency check on the interpretation I (line 11). If I is inconsistent, the conflict is analyzed and a set of constraints is added to Π (line 13). Otherwise, the algorithm terminates returning I. This check is required whenever the specific implementation of the CDCL algorithm lazily postpones some propagation inference which is required to assure the consistency of I.

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

Propagation. One of the key features of Algorithm 1 is the function Propagate, whose role is to extend the partial interpretation with the literals that can be deterministically inferred. In particular, a set of propagators is usually applied according to some priority sequence. Higher priority propagators are applied first (function *EagerPropagation*), while lower priority propagators are applied later (function *PostPropagation*). In the following, higher and lower priority propagators are referred to as *eager* and *post* propagators, respectively. An example of eager propagator is the *unit propagator*. That is, 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 $I \cup \{\ell\}$. Consider a rule r of the form (1), its no-good representation is $C(r) = \{\sim a_1, \ldots, \sim a_n, b_1, \ldots, b_j, \sim b_{j+1}, \ldots, \sim b_m\}$, which intuitively represents a constraint that is satisfied w.r.t. I if and only if r is satisfied w.r.t. I. Therefore, the negation of a literal $\ell \in C(r)$ is unit propagated w.r.t. I and rule r iff $C(r)\setminus\{\ell\} \subseteq I$, since this represents the only way to satisfy the rule. The choice of the propagators used for the computation of stable models depends on the implementation of CDCL, which may vary the used propagators according to specific solving strategies or to features of the input program. As an example, to ensure that models are supported in the presence of disjunctive rules with more than two atoms in the head, CLASP applies a (component-wise) *shift* technique combined with unit propagation on the Clark completion of Π (Gebser *et al.* 2013), whereas WASP employs a dedicated *support propagator* as described in Alviano and Dodaro (2016).

Heuristic choice. The heuristic criteria used for selecting the branching literal play a crucial role in the CDCL algorithm. During the recent years, several heuristic strategies have been proposed. Among them, variable state independent decaying sum (VSIDS) (Moskewicz *et al.* 2001) has been shown to be successful in solving a large number of problems. The implementation of the state-of-the-art SAT solver MINISAT (Eén and Sörensson 2003) provides a further boost in the usage of VSIDS-like heuristics. Nowadays, ASP solvers CLASP and WASP also use variants of the heuristic strategy proposed by MINISAT. This strategy keeps an *activity* value, initially set to 0, for each atom in $At(\Pi)$. When a literal $\ell = a$ or $\ell = \sim a$ is used for computing a learned constraint (e.g., during the computation of first UIP), the activity of a is incremented by a value *inc*. The value of *inc* is not static, instead it is multiplied by a constant slightly greater than 1 whenever a learned constraint is added to Π. Intuitively, this is done to give more importance to atoms which have been included in the recent learned constraints. The next branching literal is $\sim a$, where a is the undefined atom with the highest value of activity (ties are broken randomly). In the following, the default heuristic coupled with the CDCL algorithm is assumed to be the MINISAT heuristic.

3 The external interface of wasp

In this section, we describe the external interface of WASP (Alviano *et al.* 2015) for adding new propagators and heuristics. The architecture of WASP allows by design the interaction of its internal CDCL algorithm (as described in Section 2.2) with external algorithms developed according to the interface of methods described in Sections 3.1 and 3.2. During the computation of a stable model, and in particular when specific points of the computations are reached, WASP performs calls to the corresponding methods of the external interface. Therefore, propagators and heuristics are algorithms providing an implementation of specific methods of the interface. In Sections 3.1 and 3.2, we describe such methods providing their *contract*, that is, the input parameter (Parameter), the output of the method (Return), the point of CDCL when the method is called (When), the conditions that must be true when the method is called (Preconditions), and the conditions that will be true when the method has completed its task (Postconditions). In order to simplify the presentation some technical details are omitted and the general description of the interface is not committed to a specific language. Moreover, we assume that WASP takes as input a propositional program Π and creates an interpretation I initially set to $\emptyset$.

3.1 Propagators

The methods of the interface to add new propagators in WASP are reported in the following and described in what follows in separate paragraphs:

Method AttachLiterals. This method associates a set of literals to the specific propagator. The contract of the method is the following:

> **Parameter:** none.
> **Return:** a set of literals $\mathcal{L}$.
> **When:** the method is called before the initial simplifications, that is, before line 3 of Algorithm 1.
> **Preconditions:** the parsing of Π is executed and the initial simplifications are not performed.
> **Postconditions:** literals in $\mathcal{L}$ are associated to the propagator.

Literals in $\mathcal{L}$ are interpreted by WASP as *attached* to the propagator. That is, whenever a literal in $\mathcal{L}$ is added to or removed from the partial interpretation I, a notification is sent to the propagator using the methods described in the following (see *OnLiteralTrue* and *OnLiteralsUndefined*, respectively). Otherwise, literals which are not included in $\mathcal{L}$ are ignored. Intuitively, this method is used to limit the notifications only to a subset of literals of interest for the propagator.

Method Simplify. This method can be used to further simplify the input program. The contract of the method is the following:

> **Parameter:** none.
> **Return:** a set of literals.
> **When:** the method is called during the execution of *SimplifyProgram* (line 3 of Algorithm 1), that is, after all simplifications implemented by WASP.
> **Preconditions:** initial simplifications of the input program have been performed and I is consistent.
> **Postconditions:** literals returned by the method are added to I.

During the simplifications, a custom propagator may identify a set of literals that must be included in I. Stated differently, the propagator can return a set of literals that will be always included in all stable models of Π.

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Method OnLiteralTrue. This method can be used to implement eager propagators. The contract of the method is the following:

Parameter: a literal $\ell \in \mathcal{L}$ that has been added to I.
Return: a set of literals.
When: the method is called during the execution of *EagerPropagation* (line 1 of Propagate).
Preconditions: the interpretation I is consistent, $\ell \in (\mathcal{L} \cap I)$.
Postconditions: literals returned by the method are added to I.

A literal ℓ is added to the interpretation I. Such a literal may lead to the inference of other literals, which are returned as output by this method and that will be later on added to the interpretation I by WASP.

Method OnLiteralsTrue. This method can be used to implement post propagators. The contract of the method is the following:

Parameter: a set of literals $L \subseteq \mathcal{L}$.
Return: a set of literals.
When: the method is called during the execution of *PostPropagation* (line 3 of Propagate).
Preconditions: the interpretation I is consistent, $L \subseteq (\mathcal{L} \cap I)$.
Postconditions: literals returned by the method are added to I.

The parameter L includes the latest heuristic choice and literals added to I during the execution of *EagerPropagation*. As the previous method, it returns a set of literals which will be later on added to the interpretation I by WASP.

Method OnLiteralsUndefined. The method can be used by the propagator to keep track of modifications of I. The contract of the method is the following:

Parameter: a set of literals $L \subseteq \mathcal{L}$.
Return: none.
When: this method is called either during the execution of methods *RestoreConsistency* or *RestartIfNeeded* (lines 9 and 16 of Algorithm 1, respectively).
Preconditions: an unroll of the interpretation I has been performed, I is consistent, $L \subseteq \mathcal{L}$, $(L \cup \overline{L}) \cap I = \emptyset$.
Postconditions: none.

Literals in L have been removed from the partial interpretation I by WASP, thus they are undefined, for example, after a conflict or a restart.

Method GetReasonForLiteral. The method can be used for providing an explanation for the inference of a literal. The contract of the method is the following:

Parameter: a literal ℓ.
Return: a constraint r.
When: this method is called for each literal ℓ added to the interpretation by methods *OnLiteralTrue* and *OnLiteralsTrue*.

Preconditions: ℓ is the literal (resp. one of the literals) returned by the method *OnLiteralTrue* (resp. *OnLiteralsTrue*).
Postconditions: let r be the constraint returned by the method, the negation of ℓ is in r and all literals in r but ℓ are true w.r.t. the partial interpretation I, r is added to Π.

Whenever a literal ℓ is inferred by one of the methods *OnLiteralTrue* and *OnLiteralsTrue*, an explanation of the inference is needed to implement learning techniques properly. Such an explanation is modeled by a constraint r. More formally, let $S = \{\ell_1, \ldots, \ell_m\}$ be a set of literals such that if $S \subseteq I$ then $\ell \in I_1$ for each stable model $I_1 \supseteq I$ of Π, the constraint r is of the form $\leftarrow \ell_1, \ldots, \ell_m, \sim\ell$.

Method CheckStableModel. The method can be used to check the consistency of I w.r.t. the propagator. The contract of the method is the following:

Parameter: a set of literals representing the interpretation I.
Return: a Boolean value.
When: this method is called during the execution of *CheckConsistency* (line 11 of Algorithm 1).
Preconditions: I is total and consistent w.r.t. Π.
Postconditions: If the method returns *true*, WASP terminates its execution. If the method returns *false*, the method *GetReasonsForCheckFailure* is called.

The propagator may lazily postpone some inference that is required to assure the consistency of I; therefore, this method returns *true* if I is consistent w.r.t. the propagator, and *false* otherwise.

Method GetReasonsForCheckFailure. The method can be used for providing an explanation of the stable model check failure. The contract of the method is the following:

Parameter: none.
Return: a set of constraints.
When: this method is called whenever the method *CheckStableModel* returns *false*.
Preconditions: method *CheckStableModel* returned *false*, I is total and consistent w.r.t. Π.
Postconditions: let R be the set of constraints returned by the method, for each constraint $r \in R$ all literals in r are true w.r.t. I, R is added to Π, and the computation of WASP restarts.

The failure of *CheckStableModel* is explained in terms of constraints that are implicitly violated by the interpretation I. Indeed, the consistency check fails when the interpretation is inconsistent w.r.t. the propagator. Constraints in R are later on added to Π by WASP (line 13 of Algorithm 1).

3.2 Heuristic

The interface of WASP also allows the definition of custom heuristics that modify the default MINISAT heuristic, using the methods described in the following. Methods included in the interface were chosen by looking at modern general-purpose and domain-specific

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

heuristics. Note that an algorithm modifying the heuristic can also use the methods described in the previous section.

Method OnConflict. The method can be used to keep track that a conflict occurred during the search. The contract of the method is the following:

Parameter: none.
Return: none.
When: this method is called whenever the partial interpretation I is inconsistent (i.e., line 5 of Algorithm 1).
Preconditions: the partial interpretation I is inconsistent, that is, a conflict occurred during the search.
Postconditions: none.

The number of conflicts occurring during the search is a parameter that is often used by many look-back heuristics, as the MINISAT one. Therefore, this method provides a convenient way to keep track of the conflicts occurred.

Method OnLitInConflict. The method can be used to keep track of literals that are used during the computation of the first UIP. The contract of the method is the following:

Parameter: a literal ℓ.
Return: none.
When: this method is called during the execution of *CreateConstraint* and *CreateConstraints* (lines 6 and 12 of Algorithm 1, respectively).
Preconditions: a learned constraint r is created after a conflict and ℓ is a literal used during the computation of first UIP.
Postconditions: none.

Literals used during the computation of the first UIP are usually considered good candidates as branching literals by look-back techniques (Eén and Sörensson 2003). Therefore, this method provides a convenient way to keep track of such literals.

Method OnLearningConstraint. The method can be used to keep track of constraints learned during the search. The contract of the method is the following:

Parameter: a constraint r.
Return: none.
When: this method is called whenever a new constraint r is added to the input program (lines 8 and 13 of Algorithm 1).
Preconditions: a learned constraint r is added to Π.
Postconditions: none.

Method OnRestart. The method can be used to keep track of restarts occurring during the computation. The contract of the method is the following:

Parameter: none.
Return: none.
When: this method is called during the execution of *RestartsIfNecessary* (line 16 of Algorithm 1).

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Preconditions: the solver restarted the computations from scratch and no branching choices are made.
Postconditions: none.

Method Init-MINISAT. The method can be used to initialize the MINISAT activities of some atoms. The contract of the method is the following:

Parameter: none.
Return: a set of elements of the form (a, v), where $a \in At(\Pi)$ and $v \geq 0$ is an integer.
When: this method is called by WASP after the initial simplification and before the first propagation, that is, after line 3 of Algorithm 1.
Preconditions: the simplifications are performed, no branching choices are made, and the MINISAT activities of all atoms in the program are set to 0.
Postconditions: let S be the set of elements of the form (a, v) returned by the method where $a \in At(\Pi)$ and $v \geq 0$, the MINISAT activity of the atom a is set to v for each element (a, v) in S.

The default implementation of MINISAT heuristic initializes the activity of all atoms to 0. Therefore, this method can be used to initialize the activity of atoms to a different value. Intuitively, since the MINISAT heuristic selects atoms with the highest activity value, a proper selection of initial activity values can be useful to influence the first choices made by the solver.

Method Factor-MINISAT. The method can be used to associate an amplifying factor to the MINISAT activities of some atoms. The contract of the method is the following:

Parameter: none.
Return: a set of elements of the form (a, v), where $a \in At(\Pi)$ and $v \geq 0$ is an integer.
When: this method is called by WASP after the initial simplification and before the first propagation, that is, after line 3 of Algorithm 1.
Preconditions: the simplifications are performed, no branching choices are made, and the MINISAT activities of all atoms are initialized.
Postconditions: let S be the set of elements of the form (a, v) returned by the method where $a \in At(\Pi)$ and $v \geq 0$, the MINISAT activity of the atom a is associated to the amplifying factor v for each element (a, v) in S.

The amplifying factor will be used by WASP during the computation of the heuristic choice, in particular the activity value of an atom is multiplied by its amplifying factor.

Method Sign-MINISAT. The method can be used to provide a priority on the sign of literals, that an atom will be selected with a specific polarity. The contract of the method is the following:

Parameter: none.
Return: a set of elements of the form (a, v), where $a \in At(\Pi)$ and $v \in \{pos, neg\}$.
When: this method is called by WASP after the initial simplification and before the first propagation, that is, after line 3 of Algorithm 1.

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Preconditions: the simplifications are performed, no branching choices are made, and the MINISAT activities of all atoms are initialized.
Postconditions: let S be the set of elements of the form (a, v) returned by the method where $a \in At(\Pi)$ and $v \in \{pos, neg\}$, the atom a is associated with the polarity v.

The sign of atoms allows to specify a preference of the choice polarity of atoms. In particular, let a be an atom and $v \in \{pos, neg\}$ its associated polarity, whenever the atom a is selected as branching atom by the MINISAT heuristic, the corresponding branching literal is $\sim a$ if $v = neg$, otherwise it is a.

Method SelectLiteral. The method can be used to replace the default MINISAT heuristic or to trigger a restart of the computation or an unroll. The contract of the method is the following:

Parameter: none.
Return: an element of the following form: (i) $(choice, \ell)$, where ℓ is an undefined literal w.r.t. I (ii) (MINISAT, n), where $n \geq 0$ (iii) $(unroll, \ell)$, where ℓ is a true or false literal w.r.t. I or (iv) $(restart)$.
When: this method is called by WASP whenever an heuristic choice is required, that is, during the execution of *ChooseLiteral* (line 18 of Algorithm 1).
Preconditions: the interpretation I is consistent,
Postconditions: in case (i) the undefined literal ℓ is selected as branching literal and added to I; in case (ii) the custom heuristic is disabled for the subsequent n choices switching to the MINISAT one, or the MINISAT heuristic is enabled permanently in case $n = 0$; in case (iii) all the choices are retracted until ℓ is not included anymore in I; in case (iv) WASP performs a complete restart of the computation.

3.3 Implementation

The interface described in the previous section has been implemented as an extension of WASP. Current implementation supports *perl* and *python* scripts for obtaining fast prototypes, and *C++* in case better performance is needed. It is important to emphasize that *C++* implementations must be integrated in the WASP binary at compile time, whereas *perl* and *python* scripts are specified by means of files given as parameters for WASP, thus they do not require changes and recompilation of its source code. In particular, *perl* or *python* propagators and heuristics are files that are provided to WASP by means of a command line option. Note that such files do not contain any reference to the internal data structures of WASP. The communication between the solver and the file is done by means of a message passing protocol, which is however transparent to the developer of propagators and heuristics. The source code and the documentation are available at `alviano.github.io/wasp/`.

4 Examples of usage

In this section, we provide two usage examples of the interface described in the previous section. Section 4.1 reports the implementation of two propagators for solving the *sta-*

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

ble marriage problem. Section 4.2 shows a proof-of-concept implementation of a CASP solver. Section 4.3 describes the instantiation of the interface for implementing the VSIDS heuristic as proposed by the solver CHAFF (Moskewicz *et al.* 2001).

4.1 Propagators for stable marriage

In this section, the interface of WASP is used for adding new external propagators for solving the *stable marriage* problem. The role of propagators is to replace the instantiation of some problematic constraints as described by Cuteri *et al.* (2017). 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 pair (m', w') for which both partners would rather be married with each other than their current partner. In particular, an ASP encoding of the stable marriage problem used for the fourth ASP competition is reported in Figure 1, where the two disjoint sets of men M and women W are encoded by instances of the predicates *man* and *woman*, respectively. Instances of the predicate *pref* represent preferences of men to women and of women to men. In particular, an atom of the form $\mathit{pref}(m, w, n)$ encodes the preference of man m to woman w.

The first two rules of the encoding define the search space by guessing a match of men and women, while subsequent constraints filter out matches that do not satisfy the requirements. As argued in Cuteri *et al.* (2017), the last constraint (r_7), which guarantees that the stability condition is not violated, might be problematic to be evaluated by classical solving strategies. Indeed, the instantiation of this constraint requires the generation of a huge number of ground constraints. Thus, a possible solution is to replace r_7 by means of external propagators. In particular, given the encoding reported in Figure 1, the idea is to evaluate only rules r_1–r_6, while r_7 is removed from the encoding.

Algorithm 2: LazyStableMarriage

Global variables: Π, C;

Method: CheckStableModel(I)
  $C := \emptyset$;
  $\mathit{Matches} := \{a \mid a \in \mathit{At}(\Pi) \cap I,\ a \text{ is of the form } \mathit{match}(m, w)\}$;
  for $\mathit{match}(m, w_1) \in \mathit{Matches}$ **do**
    for $\mathit{match}(m_1, w) \in \mathit{Matches}$ **and** $m \neq m_1$ **do**
      //$\mathit{pref}(m, w_1, sm_1), \mathit{pref}(m, w, sm), \mathit{pref}(w, m_1, sw_1), \mathit{pref}(w, m, sw) \in \mathit{At}(\Pi)$
      if $sm > sm_1$ **and** $sw \geq sw_1$ **then** $C := C \cup \{\leftarrow \mathit{match}(m, w_1), \mathit{match}(m_1, w)\}$;
    end
  end
  return $C = \emptyset$; `//no constraints to add: model is stable and return` *`true`*

Method: GetReasonForCheckFailure()
  return C; `//` C `is modified by CheckStableModel`

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Algorithm 3: EagerStableMarriage

Global variables: Π, ℓ_1;

Method: AttachLiterals()
  return $\{a \mid a \in At(\Pi),\ a\ is\ of\ the\ form\ match(m, w_1)\}$;

Method: OnLiteralTrue(ℓ)
  $\ell_1 := \ell$; $L := \emptyset$; $Matches := \{a \mid a \in At(\Pi),\ a\ is\ of\ the\ form\ match(m_1, w), m \neq m_1\}$;
  `//` ℓ `is of the form` $match(m, w_1)$
  for $match(m_1, w) \in Matches$ **do**
    `//` $pref(m, w_1, sm_1), pref(m, w, sm), pref(w, m_1, sw_1), pref(w, m, sw) \in At(\Pi)$
    `//` $pref(m_1, w_1, sm_2), pref(m_1, w, sm_3), pref(w_1, m, sw_2), pref(w_1, m_1, sw_3) \in At(\Pi)$
    if $sm > sm_1$ **and** $sw \geq sw_1$ **or** $sm_2 > sm_3$ **and** $sw_3 \geq sw_2$ **then**
      $L := L \cup \{\sim match(m_1, w)\}$;
    end
  end
  return L;

Method: GetReasonForLiteral(ℓ)
  `// let` ℓ `be of the form` $\sim match(m_1, w)$
  return $\leftarrow match(m_1, w), \ell_1$; `//` ℓ_1 `is modified by OnLiteralTrue`

Then, stable models of the resulting program violating r_7 are filtered out by means of *ad hoc* implementations. Two strategies were proposed and analyzed in Cuteri *et al.* (2017). The first one was called *lazy* and basically lazily instantiates r_7 whenever it is violated by a stable model candidate. The second strategy was called *eager*. In this case, the idea is to simulate the unit propagation over r_7 during the computation of the stable model candidate. Both strategies can be implemented in WASP using the interface described in Section 3.

Lazy. Lazy approach is reported in Algorithm 2. This strategy uses methods *CheckStableModel* and *GetReasonForCheckFailure*. Whenever a stable model candidate is found, WASP calls the method *CheckStableModel*, which checks whether the stability condition modeled by constraint r_7 is violated. If there is no violation, then the algorithm terminates returning *true*, thus witnessing the stability of the model candidate. Otherwise, it produces a set of violated constraints which are later on added to WASP using the method *GetReasonForCheckFailure*.

Eager. Eager approach is reported in Algorithm 3. In contrast to the previous strategy that aims at adding violated constraints when a stable model candidate is found, this one evaluates the constraints during the computation of the stable model. Thus, constraint r_7 is never instantiated in practice but its inference is simulated by an *ad hoc* procedure

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\%\ \textit{Guess matches}} \\
r_1: & match(M,W) & \leftarrow & man(M),\ woman(W),\ {\sim}nmatch(M,W) \\
r_2: & nmatch(M,W) & \leftarrow & man(M),\ woman(W),\ {\sim}match(M,W) \\
\\
\multicolumn{4}{l}{\%\ \textit{No polygamy}} \\
r_3: & & \leftarrow & match(M_1,W),\ match(M_2,W),\ M_1 \neq M_2 \\
r_4: & & \leftarrow & match(M,W_1),\ match(M,W_2),\ W_1 \neq W_2 \\
\\
\multicolumn{4}{l}{\%\ \textit{No singles}} \\
r_5: & married(M) & \leftarrow & match(M,W) \\
r_6: & & \leftarrow & man(M),\ {\sim}married(M) \\
\\
\multicolumn{4}{l}{\%\ \textit{Strong stability condition}} \\
r_7: & & \leftarrow & match(M,W_1),\ match(M_1,W),\ W_1 \neq W, \\
 & & & pref(M,W_1,SM_1),\ pref(M,W,SM),\ SM > SM_1, \\
 & & & pref(W,M_1,SW_1),\ pref(W,M,SW),\ SW \geq SW_1
\end{array}
$$

Fig. 1. ASP encoding of the stable marriage problem.

implemented for that purpose. In particular, this strategy takes advantage of methods *AttachLiterals*, *OnLiteralTrue*, and *GetReasonForLiteral*. Method *AttachLiterals* returns a list of all literals of the form $match(m, w)$, thus whenever an atom of this kind is propagated as true the method *OnLiteralTrue* is called. The role of *OnLiteralTrue* is to simulate unit propagation inferences over constraint r_7.

4.2 Implementation of a CASP solver

In this section, we sketch how the interface of WASP can be used for implementing a naive CASP solver. Our example is based on a fragment of the EZ language, as described in Balduccini and Lierler (2017) and Susman and Lierler (2016), and is inspired by the solving approach of CLINGCON (Ostrowski and Schaub 2012).

An EZ program P is a triple $(\Pi, \mathcal{B}, \gamma)$, where Π is ASP program, $\mathcal{B}$ is a set of *constraints* of a Constraint Satisfaction Problem (CSP) (e.g., linear constraints) (Rossi *et al.* 2006), and γ is an injective function from the set of irregular atoms $C \subseteq atoms(\Pi)$ to $\mathcal{B}$. An answer set of P is an answer set X of Π such that $\bigcup_{p \in X \cap C}\{\gamma(p)\}$ has a solution (Susman and Lierler 2016).

We implement the semantics by using the propagator reported in Algorithm 4, which basically checks whether the CSP associated to the current candidate answer set has solution. If this is the case, the solution is printed, otherwise a reason for the failure of the check is a single constraint having in the body all irregular atoms that are true (see function GetReasonForCheckFailure). For completeness, the role of function AttachLiterals in Algorithm 4 is simply to intercept irregular atoms. In our example, all irregular atoms are expected to be of the form $required(\cdot)$, and the specification of domain variables uses the special atoms $cspdomain(\cdot)$ and $cspvar(\cdot)$. This format is compliant with the output of the preprocessor of EZSMT (Susman and Lierler 2016). We refer the reader to Balduccini and Lierler (2017) and Susman and Lierler (2016) for more details.

Algorithm 4: A naive CASP solver

```
Global variables: Π, C, R;

Method: AttachLiterals()
    C := {a | a ∈ At(Π), a is of the form required(c)};
    C := C ∪ {a | a ∈ At(Π), a is of the form cspdomain(d)};
    C := C ∪ {a | a ∈ At(Π), a is of the form cspvar(x, n, m)};
    return ∅;

Method: CheckStableModel(I)
    T := {a | a ∈ C ∩ I};        // identify true irregular and special atoms
    𝒞 := Atoms2Constraints(T);                              // implements γ
    (res, S) := ConstraintSolver(𝒞);                 // call an external solver
    if res then
        Print(S);                                         // print the solution
    else
        R := T;                                      // trivial reason of failure
    end
    return res; // no constraints to add: model is stable and return true

Method: GetReasonForCheckFailure()
    return ← l_1, ..., l_n;   // R = {l_1, ..., l_n} is modified by CheckStableModel
```

We remark that the goal of this example is to show the applicability of the interface. Therefore, the simple CASP-solving strategy presented here is not expected to be efficient, since it does not include all sophisticated techniques proposed by state-of-the-art CASP solvers that would complicate the description.

4.3 Implementation of VSIDS heuristic

In this section, the interface of WASP is used for implementing the general-purpose VSIDS heuristic. VSIDS was proposed in the SAT solver CHAFF (Moskewicz *et al.* 2001) and it inspired several modern heuristics, including the MINISAT heuristic. The basic idea is to store atoms in a list and to associate a numerical score to each atom, initially set to 0. After learning a constraint, the score of the atoms included in its body is incremented (called *bumping*). The score of each atom is halved every 256 conflicts (called *rescoring*) and the list of atoms is sorted descending according to their new score. Whenever a choice is required the first undefined atom in the list is selected. An implementation of the VSIDS heuristic as described in Biere and Fröhlich (2015) using the interface of WASP is reported in Algorithm 5. Method *AttachLiterals* is used to initialize the scores of atoms, while *OnLiteralsTrue* and *OnUnrollLiteral* are used to store and update the partial interpretation I. Method *OnLearningConstraint* is used to bump the scores of atoms appearing in the constraint, while method *OnConflict* is used to count the number of conflicts and to periodically update the scores of atoms. Finally, *SelectLiteral* is used when a choice is needed.

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

Algorithm 5: VSIDS

Global variables: Π, $I := \emptyset$, $scores := []$, $conflicts := 0$;

Method: AttachLiterals()
  for $a \in At(\Pi)$ **do** $scores[a] := 0$; `// init scores of atoms`
  return $\{a \mid a \in At(\Pi)\} \cup \{\sim a \mid a \in At(\Pi)\}$;

Method: OnLiteralsTrue(L)
  $I := I \cup L$;
  return $\emptyset$; `// No inferences done here`

Method: OnUnrollLiterals(L)
  $I := I \setminus L$;
  return;

Method: OnLearningConstraint(r)
  for $a \in B(r)$ **do** $scores[a] := scores[a] + 1$; `// atoms bumping`
  return;

Method: OnConflict()
  $conflicts := conflicts + 1$;
  if $conflicts = 256$ **then**
    $conflicts := 0$;
    for $a \in At(\Pi)$ **do** $scores[a] := scores[a] \div 2$; `// atoms rescoring`
    sort($scores, descending$); `// sort scores in descending order`
  end
  return;

Method: SelectLiteral()
  $a := GetFirstUndefined(scores, I)$; `// select the first undefined atom`
  return $\sim a$;

Note that modern implementations of VSIDS also bump the scores of atoms when they are used for the computation of a learned constraint. In this case, method *OnLitInConflict* of the interface can be used.

5 Successful applications

The interface of WASP described in Section 3 has been already used in the literature to enhance the standard solving capabilities of WASP (Cuteri *et al.* 2017; Dodaro *et al.* 2016; Dodaro *et al.* 2016). In the following, such applications are reviewed discussing how the interface was used and its impact on the performance of WASP.

5.1 Propagators

The interface of WASP was used in Cuteri *et al.* (2017) and Dodaro *et al.* (2016) for avoiding the instantiation of some problematic constraints as shown in Section 4.1.

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

In particular, three propagators were proposed called *lazy*, *eager*, and *post*. The first two propagators use the same methods as described in Section 4.1. In particular, methods *CheckStableModel* and *GetReasonForCheckFailure* were used for implementing the lazy propagator, while *OnLiteralTrue* and *GetReasonForLiteral* were used for implementing the eager propagator. In addition, the post propagator was implemented using *OnLiteralsTrue* and *GetReasonForLiteral*. Those propagators were evaluated on three benchmarks, namely stable marriage, packing, and NLU. In the following, we review the main results obtained in Cuteri *et al.* (2017).

Stable marriage. The experiment was executed on randomly generated instances. Basically, instances were generated as follows: each man (resp. woman) gives the same preference to each woman (resp. man), so that the stability condition is never violated. Then, a value percentage k of preferences was also considered, that is, each man (resp. woman) gives the same preference to all the women (resp. men) but a lower preference is given to k% of them. For each considered percentage k, 10 randomly generated instances were considered. Results show that for instances where the value of k is small (up to 50%) the lazy approach was better than the eager approach and than standard solving strategy of WASP. On the other hand, for high values of k the advantages of the lazy approach disappear and the eager propagator obtained the best performance overall. Interestingly, WASP without external propagators was better than its counterparts only when the value of k was 95%. Thus, the usage of external propagators was beneficial for solving this problem.

Packing. Concerning this problem, propagators were evaluated on 50 instances submitted to the third ASP competition. Interestingly, without the usage of external propagators none of the instances can be instantiated by the state-of-the-art grounder GRINGO, thus standard solving strategies were not feasible for this kind of problem. On the other hand, performance of propagators was much better. Indeed, lazy approach solved 20% of the instances, while eager and post propagators were able to solve *all* the considered instances.

Natural language understanding. Concerning NLU, propagators were evaluated on 50 instances using the objective functions proposed in Schüller (2016), namely *Cardinality*, *Coherence*, and *Weighted Abduction*. The performance of propagators was dependent on the specific objective function considered, for example, the lazy approach was slightly faster than other propagators on *Cardinality* and *Weighted Abduction*, while for *Coherence* post propagator was the best alternative. However, all the propagators outperform the standard solving strategy, thus showing also in this case a clear advantage of using custom propagators.

5.2 Heuristics

In Dodaro *et al.* (2016), the interface of WASP was used for integrating domain-specific heuristics for two industrial problems proposed by Siemens, namely partner units and combined configuration. In the following, we review the main results obtained by WASP in Dodaro *et al.* (2016).

Partner units. Concerning partner units problem, three different heuristics were proposed. All of them take advantage of *AttachLiterals* to initialize their internal data

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

structures and *OnConflict* to update them during the search. In addition, methods *OnLiteralTrue* (*OnLiteralsTrue*) and *OnUnrollLiteral* are used to synchronize the internal state of the heuristics with the partial interpretation of the CDCL algorithm. Moreover, the heuristics are able to recognize that the current partial assignment cannot be completed to a valid solution. Thus, in this case, a restart is performed returning *(restart)* after the subsequent call to the method *SelectLiteral*. The performance of WASP employing such heuristics was empirically evaluated on 36 instances provided by Siemens. The results reported in Dodaro *et al.* (2016) show that WASP equipped with custom heuristics outperforms state-of-the-art approaches, including CLASP, CLASPFOLIO, and MEASP.

Combined configurations. Concerning combined configurations problems, a set of six heuristics were analyzed. All of them share a similar skeleton, and as for the partner units problem, methods *AttachLiterals* and *OnConflict* are used to initialize and update the internal structures and strategies, respectively. Methods *OnLiteralTrue* (*OnLiteralsTrue*) and *OnUnrollLiteral* are used to synchronize the internal state of the heuristics with the partial interpretation of the CDCL algorithm. An interesting aspect of some the strategies proposed is the interoperability with the default MINISAT heuristic implemented in WASP. Indeed, one strategy switches to the MINISAT heuristic when some conditions are satisfied. This is implemented by returning (MINISAT, 0) when the method *SelectLiteral* is called. While, another strategy alternates the custom heuristic with the MINISAT one. In particular, method *SelectLiteral* returns (MINISAT, 1) every 1 s. The experiment considered 36 instances provided by Siemens. The results in Dodaro *et al.* (2016) show that WASP equipped with the heuristic implementing the alternating strategy solves all the tested instances outperforming state-of-the-art approaches, including CLASP, CLASPFOLIO, and MEASP.

6 Related work

Propagators. The extension of CDCL solvers with propagators is at the basis of Satisfiability Modulo Theories (SMT) solvers (Nieuwenhuis *et al.* 2006). Indeed, external theories are usually implemented by means of propagators on top of state-of-the-art SAT solvers. Similar extensions have been envisaged also for ASP (Bartholomew and Lee 2013). Other extensions of ASP such as CASP (Baselice *et al.* 2005) or aggregates (Alviano *et al.* 2018) have been implemented by adding propagators to CDCL solvers (Ostrowski and Schaub 2012).

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* (Bomanson *et al.* 2015; Bomanson *et al.* 2016)), ASP modulo theories (Bartholomew and Lee 2013; Bartholomew and Lee 2014), and CASP (Baselice *et al.* 2005), 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 of Potassco project (Gebser *et al.* 2011). The ASP solver CLASP (Gebser *et al.* 2015) provides a C++ interface for post-propagation, where it is possible to invalidate a stable model candidate. The interface for defining new propagators is conceptually equivalent

https://doi.org/10.1017/S1471068418000558 Published online by Cambridge University Press

to the one presented in Section 3.1. 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 (Gebser *et al.* 2014). First versions of the API supported by CLINGO (up to version 4) (Gebser *et al.* 2014) have no concept of post propagators but only support a function similar to *CheckStableModel*, which is called whenever a stable model is found. The version 5 of CLINGO (Gebser *et al.* 2016) supports also a similar API to define external propagation using scripting languages. Several important differences exist between the interface of WASP and the one of CLINGO. In particular, CLINGO provides only a post-propagator interface and no possibility for realizing an eager propagator (that runs before unit propagation is finished). Moreover, WASP first collects constraints added in *python* and then internally applies them and handles conflicts, while CLINGO requires an explicit propagation call after each added constraint. Finally, CLINGO offers the possibility to control both grounding and solving via *python*, while WASP only works on propositional programs.

Heuristics. A declarative approach to define domain-specific heuristics in ASP is presented in Gebser *et al.* (2013). The suggested HCLASP framework extends the GRINGO language with a special `_heuristic` predicate. Atoms over this predicate allow one to influence the choices made by the default heuristic of CLASP. In fact, a user can provide initial weights, importance factors, sign selection, and decision levels for atoms involved in non-deterministic decisions. The first three elements can be also specified in WASP using methods *Init*MINISAT, *Factor*MINISAT, and *Sign*MINISAT, respectively, while levels are not supported by the interface of WASP. Moreover, HCLASP supports the definition of dynamic heuristics by considering only those atoms over `_heuristic` predicate that are true in the current interpretation. In contrast to static heuristics, where heuristic decisions are encoded as facts, dynamic ones comprise normal rules with a `_heuristic` atom in the head. As argued in Gebser *et al.* (2015), the grounding of programs comprising definitions of dynamic heuristics can be expensive as it impacts on grounding speed and size. The reason is that a grounder needs to output a rule for every possible heuristic decision. Since version 5 of CLINGO, the `_heuristic` predicate has been replaced by the directive `#heuristic`, whose working principles are similar. In Balduccini (2011), a technique which allows learning of domain-specific heuristics in DPLL-based solvers is presented. The basic idea is to analyze off-line the behavior of the solver on representative instances from the domain to learn and use a heuristic in later runs. The interface of WASP presented in this paper could be considered for porting the ideas of Balduccini (2011) in a CDCL solver. As far as we know, there are no other ASP solvers that support *python* (or *perl*) implementations to specify new heuristics.

7 Conclusion

In this paper, we presented the external interface of the ASP solver WASP conceived for easing the burden of extending its solving capabilities by means of new propagators and choice heuristics. The implementation of the interface supports both rapid development by means of scripting languages, where no modifications to the solver is required, and performance-oriented development in *C++*. Successful applications of the interface in the literature witness that the usage of the interface for developing domain-specific propagators and heuristics can be very effective for solving real-world problems and in general for speeding up the performance of WASP.

References

ABSEHER, M., GEBSER, M., MUSLIU, N., SCHAUB, T. AND WOLTRAN, S. 2016. Shift design with answer set programming. *Fundamenta Informaticae 147*, 1, 1–25.

ALVIANO, M., CALIMERI, F., DODARO, C., FUSCÀ, D., LEONE, N., PERRI, S., RICCA, F., VELTRI, P. AND ZANGARI, J. 2017. The ASP system DLV2. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 10377. Springer, 215–221.

ALVIANO, M. AND DODARO, C. 2016. Completion of disjunctive logic programs. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI Press, 886–892.

ALVIANO, M., DODARO, C., FABER, W., LEONE, N. AND RICCA, F. 2013. WASP: A native ASP solver based on constraint learning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148. Springer, 54–66.

ALVIANO, M., DODARO, C., LEONE, N. AND RICCA, F. 2015. Advances in WASP. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

ALVIANO, M., DODARO, C. AND MARATEA, M. 2017. An advanced answer set programming encoding for nurse scheduling. In *AI*IA*. Lecture Notes in Computer Science, vol. 10640. Springer, 468–482.

ALVIANO, M., DODARO, C. AND MARATEA, M. 2018. Shared aggregate sets in answer set programming. *Theory and Practice of Logic Programming 18*, 3–4, 301–318.

BALDUCCINI, M. 2011. Learning and using domain-specific heuristics in ASP solvers. *AI Communications 24*, 2, 147–164.

BALDUCCINI, M., GELFOND, M., WATSON, R. AND NOGUEIRA, M. 2001. The USA-advisor: A case study in answer set planning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 2173. Springer, 439–442.

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.

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

BARTHOLOMEW, M. AND LEE, J. 2013. Functional stable model semantics and answer set programming modulo theories. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI, 718–724.

BARTHOLOMEW, M. AND LEE, J. 2014. System aspmt2smt: Computing ASPMT theories by SMT solvers. In *European Conference on Logics in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 8761. Springer, 529–542.

BASELICE, S., BONATTI, P. A. AND GELFOND, M. 2005. Towards an integration of answer set and constraint solving. In *International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 3668. Springer, 52–66.

BIERE, A. AND FRÖHLICH, A. 2015. Evaluating CDCL variable scoring schemes. In *SAT*. Lecture Notes in Computer Science, vol. 9340. Springer, 405–422.

BOMANSON, J., GEBSER, M., JANHUNEN, T., KAUFMANN, B. AND SCHAUB, T. 2015. Answer set programming modulo acyclicity. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 143–150.

Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B. and Schaub, T. 2016. Answer set programming modulo acyclicity. *Fundamenta Informaticae 147*, 1, 63–91.

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

Cuteri, B., Dodaro, C., Ricca, F. and Schüller, P. 2017. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory and Practice of Logic Programming 17*, 5–6, 780–799.

Dodaro, C. 2015. *Computational Tasks in Answer Set Programming: Algorithms and Implementations*. Ph.D. dissertation, University of Calabria.

Dodaro, C., Alviano, M., Faber, W., Leone, N., Ricca, F. and Sirianni, M. 2011. The birth of a WASP: Preliminary report on a new ASP solver. In *Italian Conference on Computational Logic*. CEUR Workshop Proceedings, vol. 810. CEUR-WS.org, 99–113.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F. and Schekotihin, K. 2016. 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., Leone, N., Nardi, B. and Ricca, F. 2015. Allotment problem in travel industry: A solution based on ASP. In *International Conference on Web Reasoning and Rule Systems*. Lecture Notes in Computer Science, vol. 9209. Springer, 77–92.

Dodaro, C. and Maratea, M. 2017. Nurse scheduling via answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 10377. Springer, 301–307.

Dodaro, C., Ricca, F. and Schüller, P. 2016. External propagators in WASP: Preliminary report. In *International RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*. CEUR Workshop Proceedings, vol. 1745. CEUR-WS.org, 1–9.

Eén, N. and Sörensson, N. 2003. An extensible sat-solver. In *International Conference on Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science, vol. 2919. Springer, 502–518.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37*, 3, 53–68.

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

Friedrich, G. 2015. Industrial success stories of ASP and CP: What's still open? Joint invited talk at ICLP and CP 2015. http://booleconferences.ucc.ie/iclp2015speakers.

Garro, A., Palopoli, L. and Ricca, F. 2006. Exploiting agents in e-learning and skills management context. *AI Communications 19*, 2, 137–154.

Gavanelli, M., Nonato, M. and Peano, A. 2015. An ASP approach for the valves positioning optimization in a water distribution system. *Journal of Logic and Computation 25*, 6, 1351–1369.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Wanko, P. 2016. Theory solving made easy with clingo 5. In *International Conference on Logic Programming (Technical Communications)*. OpenAccess Series in Informatics, vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2:1–2:15.

Gebser, M., Kaminski, R., Kaufmann, B., Romero, J. and Schaub, T. 2015. Progress in clasp Series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 368–383.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2014. Clingo = ASP + control: Preliminary report. CoRR abs/1405.3694.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T. and Schneider, M. T. 2011. Potassco: The potsdam answer set solving collection. *AI Communications 24*, 2, 107–124.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2008. Advanced preprocessing for answer set solving. In *European Conference on Artificial Intelligence*. Frontiers in Artificial Intelligence and Applications, vol. 178. IOS Press, 15–19.

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

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

Gebser, M., Kaufmann, B. and Schaub, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *International Joint Conference on Artificial Intelligence*. IJCAI/AAAI.

Gebser, M., Maratea, M. and Ricca, F. 2015. The design of the sixth answer set programming competition. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 531–544.

Gebser, M., Maratea, M. and Ricca, F. 2016. What's Hot in the Answer Set Programming Competition. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 4327–4329.

Gebser, M., Maratea, M. and Ricca, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

Gebser, M., Ryabokon, A. and Schenner, G. 2015. Combining heuristics for configuration problems using answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 9345. Springer, 384–397.

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

Janhunen, T. and Niemelä, I. 2016. The answer set programming paradigm. *AI Magazine 37*, 3, 13–24.

Janhunen, T., Tasharrofi, S. and Ternovska, E. 2016. SAT-to-SAT: Declarative extension of SAT solvers with new propagators. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 978–984.

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

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.

Lierler, Y., Maratea, M. and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine 37*, 3, 45–52.

Lifschitz, V. 2016. Answer sets and the language of answer set programming. *AI Magazine 37*, 3, 7–12.

Manna, M., Ricca, F. and Terracina, G. 2013. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory and Practice of Logic Programming 13*, 2, 227–252.

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.

Maratea, M., Pulina, L. and Ricca, F. 2012. The multi-engine ASP solver ME-ASP. In *European Conference on Logics in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 7519. Springer, 484–487.

Marileo, M. C. and Bertossi, L. E. 2010. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data & Knowledge Engineering 69*, 6, 545–572.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*. ACM, 530–535.

Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53*, 6, 937–977.

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

Rossi, F., van Beek, P. and Walsh, T., Eds. 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, vol. 2. Elsevier.

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 *International Conference on Logic Programming (Technical Communications)*. OpenAccess Series in Informatics, vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 1:1–1:15.

Zhang, L., Madigan, C. F., Moskewicz, M. W. and Malik, S. 2001. Efficient conflict driven learning in Boolean satisfiability solver. In *IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society, 279–285.