Abstract Solvers for Computing Cautious Consequences of ASP programs

GIOVANNI AMENDOLA and CARMINE DODARO
University of Calabria, Italy
(*e-mail:* {amendola,dodaro}@mat.unical.it)

MARCO MARATEA
University of Genoa, Italy
(*e-mail:* marco@dibris.unige.it)

Abstract

Abstract solvers are a method to formally analyze algorithms that have been profitably used for describing, comparing and composing solving techniques in various fields such as Propositional Satisfiability (SAT), Quantified SAT, Satisfiability Modulo Theories, Answer Set Programming (ASP), and Constraint ASP.

In this paper, we design, implement and test novel abstract solutions for cautious reasoning tasks in ASP. We show how to improve the current abstract solvers for cautious reasoning in ASP with new techniques borrowed from backbone computation in SAT, in order to design new solving algorithms. By doing so, we also formally show that the algorithms for solving cautious reasoning tasks in ASP are strongly related to those for computing backbones of Boolean formulas. We implement some of the new solutions in the ASP solver wasp and show that their performance are comparable to state-of-the-art solutions on the benchmark problems from the past ASP Competitions.

KEYWORDS: Answer Set Programming, Abstract solvers, Cautious reasoning

1 Introduction

Abstract solvers are a method to formally analyse solving algorithms. In this methodology, the states of a computation are represented as nodes of a graph, the solving techniques as edges between such nodes, the solving process as a path in the graph, and formal properties of the algorithms are reduced to related graph properties. This framework enjoys some advantages w.r.t. traditional ways such as pseudo-code-based descriptions, e.g., being based on formal and well-known, yet simple, mathematical objects like graphs, which helps (*i*) comparing solving algorithms by means of comparison of their related graphs, (*ii*) mixing techniques in different algorithms in order to design novel (combination of) solving solutions, by means of mixing arcs in the related graphs, and (*iii*) stating and proving formal properties of the solving algorithms, by means of reachability within the related graphs. Abstract solvers already proved to be a useful tool for formally describing, comparing and composing solving techniques in various fields such as Propositional Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) (Nieuwenhuis et al. 2006), Quantified SAT (Brochenin and Maratea 2015b), Answer Set Programming (Lierler 2011; Lierler and Truszczynski 2011; Brochenin et al.

2014), and Constraint ASP (Lierler 2014). In ASP, such methodology led even to the development of a new ASP solver, SUP (Lierler 2011); however, abstract solvers have been so far mainly applied to ASP solvers for brave reasoning tasks where, given an input query and a knowledge base expressed in ASP, answers are witnessed by ASP solutions, i.e., stable models (Baral 2003; Eiter et al. 1997; Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991; Marek and Truszczyński 1998; Niemelä 1999).

However, in ASP, also cautious reasoning has been deeply studied in the literature: answers here must be witnessed by all stable models. This task has found a significant number of interesting applications as well, including consistent query answering (Arenas et al. 2003; Manna et al. 2013), data integration (Eiter 2005), multi-context systems (Brewka and Eiter 2007), and ontology-based reasoning (Eiter et al. 2008). Two well-known ASP solvers, i.e., DLV (Leone et al. 2006) and CLASP (Gebser et al. 2012), have been extended for computing cautious consequences of ASP programs. More recently, Alviano et al. (2014) presented a unified, high-level view of such solving procedures, and designed several algorithms for cautious reasoning in ASP, including those implemented in DLV and CLASP, borrowed from the backbone computation of Boolean formulas (Janota et al. 2015): all these techniques are implemented (and tested) on top of the ASP solver WASP (Alviano et al. 2015).

In this paper we design, implement and test novel abstract solutions for cautious reasoning tasks in ASP. We show how to improve the current abstract solvers (Brochenin and Maratea 2015a) for cautious reasoning in ASP with further techniques borrowed from backbone computation in SAT, in order to design new solving algorithms. In particular, we import a technique called "chunk", which generalizes over- and under-approximation by testing a set soft atoms simultaneously for being added in the under-approximation, and core-based algorithms, which can be considered either a solution per se, or a way for pruning the set of atoms to be considered, given that they can not guarantee completeness. By doing so, we also formally show, through a uniform treatment, that the algorithms for solving cautious reasoning tasks in ASP are strongly related to those for computing backbones of Boolean formulas. Finally, we implement some of the new solutions in the ASP solver WASP: results of a wide experimental analysis confirm that abstract solvers are a useful tool also for designing abstract solving procedures, given the performances of the related implementations are overall comparable to state-of-the-art solutions on the benchmark problems from the past ASP Competitions.

The paper is structured as follows. Section 2 introduces needed preliminaries, including a review in Section 2.3 of current algorithms for cautious reasoning trough abstract solving methodology. Section 3 shows how the algorithms for computing backbones of Boolean formulas can be imported into ASP, to design new solving algorithms. It also contains a general theorem showing the relation between backbones computation in SAT and cautious reasoning in ASP.[1] Section 4 then presents the results of the new solutions on devoted ASP benchmarks. The paper ends by discussing related work in Section 5, and by drawing conclusions in Section 6.

2 Preliminaries

In this section, we first recall basics on (ground) non-disjunctive answer set programming (ASP) and Boolean logic formulas in Conjunctive Normal Form (CNF). Then, we

---

[1] Proofs can be found in the supplementary material corresponding to this paper at the TPLP archives.

introduce the abstract solvers framework and its methodology. Finally, we recall existing abstract solvers for computing cautious consequences of ASP programs.

2.1 Boolean Formulas and Answer Set Programs

We define (ground) non-disjunctive ASP programs and CNF formulas so as to underline similarities, in order to make it easier in later sections to compare algorithms working on CNF formulas with those working on ASP programs.

Syntax. Let Σ be a propositional signature. An element $a \in \Sigma$ is called *atom* or *positive literal*. The negation of an atom a, in symbols $\neg a$, is called *negative literal*. Given a literal l, we define $|l| = a$, if $l = a$ or $l = \neg a$, for some $a \in \Sigma$. For a set of atoms $X \subseteq \Sigma$, a *literal relative to X* is a literal l such that $|l| \in X$, and $lit(X)$ is the set of all literals relative to X. We set $\bar{l} = a$, if $l = \neg a$, and $\bar{l} = \neg a$, if $l = a$. A *clause* is a finite set of literals (seen as a disjunction). A *CNF formula* is a finite set of clauses (seen as a conjunction). Given a set of literals M, we denote by M^+ the set of positive literals of M, by M^- the set of negative literals of M, and by $\overline{M}$ the set $\{\bar{l} \mid l \in M\}$. We say that M is *consistent* if it does not contain both a literal and its negation. A (non-disjunctive) *rule* is a pair (A, B), written $A \leftarrow B$, where B is a finite set of literals and A is an atom or the empty set. We may write a rule as $A \leftarrow B^+, B^-$, as an abbreviation for $A \leftarrow B^+ \cup B^-$, and $A \leftarrow l, B$ as an abbreviation for $A \leftarrow \{l\} \cup B$. A *program* is a finite set of rules. Given a set of literals M, a program Π, and a CNF formula Φ, we denote by $atoms(M)$, $atoms(\Pi)$, and $atoms(\Phi)$ the set of atoms occurring in M, Π, and Φ, respectively. It is important to emphasize here that the interpretation of negation is different in propositional formulas and in ASP programs. Indeed, in propositional formulas $\neg$ represents the classical negation, while in ASP programs it represents the *negation by default*.

Semantics. An *assignment* to a set X of atoms is a total mapping from X to $\{\bot, \top\}$. We identify a consistent set M of literals with an assignment to $atoms(M)$ such that $a \in M$ iff a is mapped to $\top$, and $\neg a \in M$ iff a is mapped to $\bot$. A *classical model* of a CNF formula Φ is an assignment M to $atoms(\Phi)$ such that for each clause $C \in \Phi$, $M \cap C \neq \emptyset$. A *classical model* of a program Π is an assignment M to $atoms(\Pi)$ such that for each rule $(A, B) \in \Pi$, $A \cap M \neq \emptyset$ or $B \not\subseteq M$. We denote $M(\Phi)$ (resp. $M(\Pi)$) the set of all classical models of Φ (resp. Π). The *reduct* Π^X of a program Π w.r.t. a set of atoms X is obtained from Π by deleting each rule $A \leftarrow B^+, B^-$ such that $X \cap atoms(B^-) \neq \emptyset$ and replacing each remaining rule $A \leftarrow B^+, B^-$ with $A \leftarrow B^+$. An *answer set* (or *stable model*) of a program Π is an assignment M to $atoms(\Pi)$ such that M^+ is minimal among the M_0^+ such that M_0 is a classical model of Π^{M^+}. We denote by $AS(\Pi)$ the set of all answer sets of Π. Given a formula Φ and a program Π, we define $backbone(\Phi) = \bigcap_{M \in M(\Phi)} M^+$ and $cautious(\Pi) = \bigcap_{M \in AS(\Pi)} M^+$.

Example 1
Consider the following program $\Pi = \{a \leftarrow \neg b, \; b \leftarrow \neg a, \; c \leftarrow a, \; c \leftarrow b\}$. Π has two answer sets, namely $A_1 = \{\neg a, b, c\}$ and $A_2 = \{a, \neg b, c\}$. Hence, $A_1^+ = \{b, c\}$ and $A_2^+ = \{a, c\}$. Therefore, $cautious(\Pi) = \{b, c\} \cap \{a, c\} = \{c\}$. Now, consider the following CNF formula $\Phi = \{a \vee b, \neg a \vee c, \neg b \vee c\}$. Φ has three classical models, namely $M_1 = \{\neg a, b, c\}$, $M_2 = \{a, \neg b, c\}$, and $M_3 = \{a, b, c\}$. Hence, $M_1^+ = \{b, c\}$, $M_2^+ = \{a, c\}$, and $M_3^+ = M_3$. Therefore, $backbone(\Phi) = \{b, c\} \cap \{a, c\} \cap \{a, b, c\} = \{c\}$.

2.2 Abstract Solvers for Solving CNF Formulas and ASP Programs

Now, we introduce the abstract solvers framework and its methodology employed later on in Section 2.3 and Section 3 for computing cautious consequences of ASP programs. As we have mentioned in the introduction, abstract solvers are graphs that represent the status of the computation, and how it changes in response to an application of a technique in a search for a solution with certain properties, e.g., the satisfiability of a formula. Correspondingly, in the next paragraphs we first present the concept of a *state*, i.e., all possible paths of the computation in terms of assignments, then the *transition rules* are introduced, that showing how the state changes as a consequence of an application of a search technique if some conditions are met. The last paragraph of this subsection introduces *abstract solver graphs*, where the states are the possible nodes of the graph, while transition rules define arcs among reachable nodes.

States. Given a set of atoms X, an *action relative to X* is an element of the set $\mathscr{A}(X) = \{over, under_\emptyset\} \cup \{under_{\{a\}} \mid a \in X\}$. For a set X of atoms, a *record* relative to X is a string L from $lit(X)$ without repetitions. A record L is *consistent* if it does not contains both a literal and its negation. We may view a record as the set containing all its elements stripped from their annotations. For example, we may view $\neg ab$ as $\{\neg a, b\}$, and hence as the assignment that maps a to $\bot$ and b to $\top$. Given a set X of atoms, the set of *states relative to X*, written V_X, is the union of:

(i)   the set of *core states relative to X*, that are all $L_{O,U,A}$ such that L is a record relative to X; $O, U \in X$; and $A \in \mathscr{A}(X)$;
(ii) the set of *control states relative to X*, that are all the $Cont(O, U)$ where $O, U \in X$; and
(iii) the set of *terminal states relative to X*, that are all $Ok(W)$, where $W \in X$.

Intuitively, these states represent computation steps of the algorithms that search for assignments with certain properties, in our case being backbone or cautious consequence. The computation starts from a specific core state, called *initial state*, depending on the specific algorithm (concrete examples are given later when presenting the techniques). Other core states $L_{O,U,A}$ and the control states $Cont(O, U)$ represent all the intermediate steps of the computation, where L is the current state of the computation of a model; O is the current over-approximation of the solution; U is the current under-approximation of the solution; and A is the action currently carried out: *over* (resp. *under$_\emptyset$* or *under$_{\{a\}}$*) if over-approximation (resp. under-approximation) is being applied. Intuitively, a core state represents the computation within a call to an ASP oracle, i.e., an ASP solver, while a control state controls the computation between different calls to ASP oracles, depending on over-approximation and under-approximation. The terminal states represent the end of the computation, i.e., the termination of the algorithm.

For instance, consider the following set of atoms $X = \{a, b, c\}$. Hence, $lit(X) = \{a, b, c, \neg a, \neg b, \neg c\}$. Therefore, $\neg ab_{\{a,b\},\emptyset,over}$ is an example of core state relative to X where a is assigned to false and b to true, the over-approximation is the set $\{a, b\}$ while the under-approximation is empty, and the action executed is over. Other examples of core states are $\emptyset_{\{a\},\{b\},under_\emptyset}$ and $\neg a \neg b \neg c_{\emptyset,\emptyset,under_{\{a\}}}$. Instead, $Cont(\{a, b\}, \{a\})$,

$Cont(\{a,b,c\},\emptyset)$, $Cont(\emptyset,\emptyset)$ are examples of control states relative to X, where e.g., in the first example the over-approximation is the set $\{a,b\}$ and the under-approximation is $\{a\}$. $Ok(\{a,b,c\})$ and $Ok(\emptyset)$ are examples of terminal states relative to X, where set $\{a,b,c\}$ and $\emptyset$ are solutions.

Transition Rules. *Transition rules* are represented with the following structure:

$$ruleName \quad S \implies S' \quad \text{if}\,\{\,conditions$$

where, (i) $ruleName$ is the name of the rule; (ii) $S \implies S'$ represents a transition from the starting state S to the arriving state S' (if the rule is applied); and (iii) *conditions* is a set of conditions for the rule to be applicable.

We also consider a special transition rule, called *Oracle*, which starts from a state $L_{O,U,A}$ and arrives to a state $L'_{O,U,A}$, if $L = \emptyset$. In symbols:

$$Oracle \quad L_{O,U,A} \implies L'_{O,U,A} \quad \text{if}\,\{\,L = \emptyset$$

Intuitively, the *Oracle* rule represents an oracle call to an ASP [resp., SAT] solver by providing as result a set of literals L' corresponding to the output of an ASP [resp., SAT] solver, i.e., L' will correspond to an answer set of a logic program [resp., a classical model of a Boolean formula], if such an answer set [resp. classical model] exists, and to an inconsistent set of literals, otherwise. Transition rules in our paper are organized into *Return* and *Control* rules. Return rules deal with the outcome of an oracle call, or the application of a given technique, depending on the status of the set of literals L returned, while Control rules start from a control state an direct the computation depending on the content of the over- and under-approximation.

Abstract Solver Graphs. Given a set of atoms X and a set of transition rules T, we define an *abstract solver graph* $G_{X,T} = \langle V_X, E_T \rangle$, where $(S, S') \in E_T$ if, and only if, a transition rule of the form $S \implies S'$ can be applied. We also denote the set of edges E_T by the set of transition rules T. We say that a state $S \in V_X$ is *reachable from* a state $S' \in V_X$, if there is a path from S' to S. Every state reachable from the initial state is called *reachable state*, and represents a possible state of a computation. Each path starting from the initial state represents the description of possible search for a certain model. We say that *no cycle is reachable* if there is no reachable state which is reachable from itself. Finally, note that transition rules T and the set X will depend from the specific input program Π, thus instead of writing $G_{X,T}$, we will write just G_Π.

2.3 Naive Abstract Solvers for Computing Cautious Consequences

In this section, we recall the abstract *over-approximation*, *under-approximation* and *mixed* strategies for computing cautious consequences of ASP programs.

Definition 1
Given a program Π [resp., a CNF formula Φ], we say that an abstract solver graph G_Π [resp., G_Φ] *solves cautious reasoning* [resp., backbone computation], if (i) G_Π [resp., G_Φ] is finite and no cycle is reachable; and (ii) the unique terminal reachable state in G_Π [resp., G_Φ] is $Ok(cautious(\Pi))$ [resp., $Ok(backbone(\Pi))$].

Return rules

$Fail_{over}$	$L_{O,U,over}$	$\Longrightarrow Cont(O,O)$	if $\{$ L is inconsistent
$Find$	$L_{O,U,A}$	$\Longrightarrow Cont(O \cap L, U)$	if $\{$ L is consistent and $L \neq \emptyset$

Control rules

$Terminal$	$Cont(O,U) \Longrightarrow Ok(O)$	if $\{$ $O = U$
$OverApprox$	$Cont(O,U) \Longrightarrow \emptyset_{O,U,over}$	if $\{$ $O \neq U$

Fig. 1. The transition rules of *ov*.

In the following, without loss of generality, we only focus on the computation of cautious consequences for an ASP program Π.

General Structure. Given a program Π, over-approximation is set to all atoms in the program, i.e., $O = atoms(\Pi)$, while the under-approximation is empty, i.e., $U = \emptyset$. Note that $U \subseteq cautious(\Pi) \subseteq O$. Iteratively either under-approximation or over-approximation are applied. When they coincide, i.e., $U = O$, the set of cautious consequences, i.e., O, has been found and the computation terminates. It means that the state $Ok(O)$ is a reachable state. Hence, the full extent of states relative to X becomes useful. The unique terminal state is $Ok(W)$, where W is the set of all cautious consequences of Π.

Over-approximation. Let $\Pi_{O,U,over} = \Pi \cup \{\leftarrow O\}$. The initial state is $\emptyset_{atoms(\Pi),\emptyset,over}$. We call *ov* the set of all the rules reported in Figure 1, that is *ov* = $\{Fail_{over}, Find,$ *Terminal*, $OverApprox\}$. Intuitively, $Fail_{over}$ means that a call to an oracle did not find an answer set, so O is the solution. If *Find* is triggered, instead, we go to a control state where O is updated according to the answer set found: then, if $O = U$ a solution is found through *Terminal*, otherwise the search is restarted ($L = \emptyset$) in an oracle state with *OverApprox*. For any Π, the graph OS_Π is $(V_{atoms(\Pi)}, \{Oracle\} \cup ov)$. Thus, in OS_Π, the oracle is called to find answer sets that reduce the over-approximation O in the *over* action, unless no answer set exists. If an answer set M is found, then $M \cap \overline{O} \neq \emptyset$, as $\Pi_{O,U,over} = \Pi \cup \{\leftarrow O\}$.

Indeed, assume by contradiction that $M \cap \overline{O} = \emptyset$, then $O \subseteq M$. Hence, M is not a model of the rule $(\emptyset, O)$, as $M \cap \emptyset = \emptyset$ and $O \subseteq M$. Therefore, M should not be a model of $\Pi_{O,U,over}$, against the assumption that M is an answer set of $\Pi_{O,U,over}$.

Under-approximation. Let $\Pi_{O,U,under_{\{a\}}} = \Pi \cup \{\leftarrow a\}$ and $\Pi_{O,U,under_\emptyset} = \Pi$. The initial state is $\emptyset_{atoms(\Pi),\emptyset,under_\emptyset}$. We call *un* the set $\{Fail_{under}, Find, Terminal, UnderApprox\}$ containing the rules presented in Figure 2 plus *Find* and *Terminal* from Figure 1. Intuitively, $Fail_{under}$ updates over- and under-approximations in case a test on the atom a failed, and leads to a control state, while *UnderApprox* restarts a new test if *Find* is not applicable. For any Π, the graph US_Π is $(V_{atoms(\Pi)}, \{Oracle\} \cup un)$. In US_Π, again, a first oracle call takes place with the action $under_\emptyset$, which provides first over-approximation, then calls with actions $under_{\{a\}}$, where the element a is the tested atom. Figure 3 shows a possible path in US_Π for the program Π of Example 1. For compactness, the syntax in which the path is presented is slighly different, with "$\Longrightarrow$" replaced by ":", and with the initial state not explicitly tagged.

Return rule
$Fail_{under}$ $L_{O,U,under_S} \implies Cont(O, U \cup S)$ if $\{$ L is inconsistent, and $S = \emptyset$ or $S = \{a\}$

Control rule
$UnderApprox$ $Cont(O, U) \implies \emptyset_{O,U,under_{\{a\}}}$ if $\{$ $a \in O \setminus U$

Fig. 2. The transition rules of un that are not in ov.

$$\Pi = \Pi_{\{a,b,c\},\emptyset,under_\emptyset} = \left\{ \begin{array}{l} a \leftarrow \neg b \\ b \leftarrow \neg a \\ c \leftarrow a \\ c \leftarrow b \end{array} \right\}$$

$Oracle:$	$\emptyset_{\{a,b,c\},\emptyset,under_\emptyset}$
$Find:$	$ac\neg b_{\{a,b,c\},\emptyset,under_\emptyset}$
	$Cont(\{a,c\},\emptyset)$

$\Pi_{\{a,c\},\emptyset,under_{\{c\}}} = \Pi \cup \{\leftarrow c\}$

$UnderApprox:$	$\emptyset_{\{a,c\},\emptyset,under_{\{c\}}}$
$Oracle:$	$\neg c\neg abc_{\{a,c\},\emptyset,under_{\{c\}}}$
$Fail_{under}:$	$Cont(\{a,c\},\{c\})$

$\Pi_{\{a,c\},\{c\},under_{\{a\}}} = \Pi \cup \{\leftarrow a\}$

$UnderApprox:$	$\emptyset_{\{a,c\},\{c\},under_{\{a\}}}$
$Oracle:$	$\neg abc_{\{a,c\},\{c\},under_{\{a\}}}$
$Find:$	$Cont(\{c\},\{c\})$
$Terminal:$	$Ok(\{c\})$

Fig. 3. A path in US_Π.

Mixed strategy. An abstract mixed strategy can be obtained by defining $MixS_\Pi$ as $(V_{atoms(\Pi)}, \{Oracle\} \cup un \cup ov)$. Therefore, it is possible to combine techniques described by the graph for over-approximation and those in the graph for under-approximation, by envisaging the design of new additional algorithms. Here, we have two potential initial states, i.e., $\emptyset_{atoms(\Pi),\emptyset,A}$, where $A \in \{over, under_\emptyset\}$, i.e., depending whether over-appoximation or under-approximation is first applied.

3 Advanced Abstract Solvers for Computing Cautious Consequences

In this section we import in ASP further algorithms from (Janota et al. 2015) through abstract solvers. First, we generalize the concepts of under- and over-approximation via chunks, which consider a set of atoms simultaneously. Then, we model core-based algorithms. Finally, we state a general theorem, which includes all previous results, that shows how the techniques presented can be combined to design new solving methods for finding cautious consequences of ASP programs, and states a strong analogy between algorithms for computing cautious consequences of ASP programs and those for backbones of CNF formulas.

The sets of states now include also the following: $\{chunk_N | N \subseteq atoms(\Pi)\}$, $chunk$ and $\{core_N | N \subseteq lit(atoms(\Pi))\}$.

3.1 Chunking

In (Janota et al. 2015) a more general technique for under-approximation that allows to test multiple literals at once is presented (see, Algorithm 5 in (Janota et al. 2015)). We define ch as the set $\{Fail_{chunk}, Find, Terminal, Chunk\}$ containing the rules presented in Figure 4 plus $Find$ and $Terminal$ from Figure 1. The newly introduced rules

Return rules
$Fail_{chunk}$ $L_{O,U,chunk_N}$ $\implies Cont(O \setminus N, U \cup N)$ if $\{$ L is inconsistent

Control rules
$Chunk$ $Cont(O,U)$ $\implies \emptyset_{O,U,chunk_N}$ if $\{$ $N \subseteq O \setminus U$ and $N \neq \emptyset$

Fig. 4. The transition rules of ch that are not in ov.

$$\Pi = \Pi_{\{a,b,c,d\},\emptyset,chunk} = \left\{ \begin{array}{l} a \leftarrow \neg b \\ b \leftarrow \neg a \\ c \leftarrow a \\ c \leftarrow b \\ d \leftarrow c \end{array} \right\}$$

$Oracle:$ $\emptyset_{\{a,b,c,d\},\emptyset,chunk}$
$Find:$ $ac\neg bd_{\{a,b,c,d\},\emptyset,chunk}$
$Cont(\{a,c,d\},\emptyset)$

$\Pi_{\{a,c,d\},\emptyset,chunk_{\{c,d\}}} = \Pi \cup \{\leftarrow c, d\}$

$Chunk:$ $\emptyset_{\{a,c,d\},\emptyset,chunk_{\{c,d\}}}$
$Oracle:$ $\neg abcd\neg d_{\{a,c,d\},\emptyset,chunk_{\{c,d\}}}$
$Fail_{chunk}:$ $Cont(\{a,c,d\},\{c,d\})$

$\Pi_{\{a,c,d\},\{c,d\},chunk_{\{a\}}} = \Pi \cup \{\leftarrow a\}$

$Chunk:$ $\emptyset_{\{a,c,d\},\{c,d\},chunk_{\{a\}}}$
$Oracle:$ $\neg abcd_{\{a,c,d\},\{c,d\},chunk_{\{a\}}}$
$Find:$ $Cont(\{c,d\},\{c,d\})$
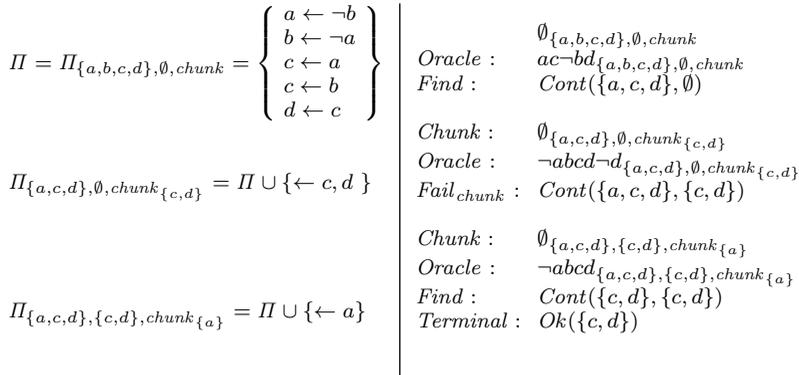$Terminal:$ $Ok(\{c,d\})$

Fig. 5. A path in CS_Π.

in Figure 4 model the new technique. In particular, $Fail_{chunk}$ updates the over- and under-approximations accordingly in case the test on the set N fails (the ASP oracle call failed, thus all literals in N must be cautious consequences), and goes to a control state. Meanwhile, $Chunk$ restarts a new ASP oracle call with a new (nonempty) set N such that $N \subseteq O \setminus U$ in case the computation must continue (cf. condition of this transition rule). For any Π, the graph CS_Π is $(V_{atoms(\Pi)}, \{Oracle\} \cup ch)$. The initial state is $\emptyset_{atoms(\Pi),\emptyset,chunk}$. We define $\Pi_{O,U,chunk_N}$ as $\Pi \cup \{\leftarrow N\}$.

Theorem 1
 Let Π be a program. Then, the graph CS_Π solves cautious reasoning.

In order to design a meaningful example of Chunk, we slightly modify our running example adding the rule $d \leftarrow c$. Figure 5 shows a possible path in CS_Π for the new defined program.

3.2 Designing New Abstract Solvers

The composition of techniques described in Section 2.3 and 3.1 can be readily applied to computing cautious consequences of a program, but actually is not included in any solver. This outlines another important feature of the abstract solvers methodology, i.e., its capability to design new solutions by means of combination of techniques implemented in different solvers.

More generally, it is possible to mix under-approximation, over-approximation, and chunking technique, and apply them for computing either cautious consequences or backbones. We next state a general theorem that subsumes all the techniques previ-

ously described, showing a strong analogy among the algorithms for computing cautious consequences and those for backbones.

Theorem 2

For any program Π, and for any set $S \subseteq \{un, ov, ch\}$ such that $S \neq \emptyset$, the graph $(V_{atoms(\Pi)}, \{Oracle_{ASP}\} \cup \bigcup_{x \in S} x)$ solves cautious reasoning, and the graph $(V_{atoms(\Pi)}, \{Oracle_{SAT}\} \cup \bigcup_{x \in S} x)$ solves backbone computation, where $Oracle_{ASP}$ and $Oracle_{SAT}$ represent an oracle call to an ASP solver and to a SAT solver, respectively.

3.3 Core-based Methods

We now model core-based algorithms from (Janota et al. 2015) in terms of abstract solvers, in particular Algorithm 6, and apply it to the computation of cautious consequences of ASP programs. First, note that $\Pi_{O,U,core_N}$ is $\Pi \cup \{\leftarrow \bar{l}|l \in N\}$, and $\emptyset_{atoms(\Pi),\emptyset,core_{\overline{atoms(\Pi)}}}$ is the initial state. Moreover, given a logic program Π, we say that a set $C \subseteq lit(atoms(\Pi))$ is *a core* of Π, if $\Pi \cup \{\leftarrow \bar{l}|l \in C\}$ is incoherent. It is important to emphasize here that this definition is in line with the one proposed by Alviano et al. (2018). In particular, unsatisfiable cores have two important properties:

- if C is an unsatisfiable core of Π then all of its supersets are also unsatisfiable cores of Π;
- an atom $p \in atoms(\Pi)$ is a cautious consequence of Π if and only if $\{\neg p\}$ is an unsatisfiable core (Proposition 4.1 of (Alviano et al. 2018)).

Moreover, in general unsatisfiable cores are not guaranteed to be minimal, albeit several strategies can be used to obtain a minimal unsatisfiable core (Lynce and Silva 2004; Alviano and Dodaro 2016; Alviano et al. 2018).

Example 2

Consider the program Π of the Example 1 and let $N = \{\neg a, \neg b, \neg c\}$. Hence, $\{\neg c\}$, $\{\neg a, \neg c\}$, $\{\neg b, \neg c\}$, $\{\neg a, \neg b\}$, and $\{\neg a, \neg b, \neg c\}$ are all cores of $\Pi_{O,U,core_N}$.

First, we consider a transition rule, called *CoreOracle*, which starts from a state $\emptyset_{O,U,core_N}$ and arrives to a state $L'_{O,U,core_N}$. In symbols:

$$CoreOracle \quad L_{O,U,core_N} \implies L'_{O,U,core_N} \quad \text{if} \{ \ L = \emptyset$$

The *CoreOracle* rule represents an oracle call to compute a set of literals L', which is an inconsistent set of literals such that the set $\widehat{L'} = \{\neg a \mid \{a, \neg a\} \subseteq L'\}$ is a core of $\Pi_{O,U,core_N}$ and a subset of N, whenever $\Pi_{O,U,core_N}$ is incoherent; and is an answer set of $\Pi_{O,U,core_N}$, otherwise. Then, we define *in* as the set of rules of Figure 6. Therefore, we consider a graph $FS_\Pi = (V_{atoms(\Pi)}, \{CoreOracle\} \cup in)$ which represents Algorithm 6 in (Janota et al. 2015). Here, we need to introduce two intermediate control states: Pre_N and $Eval$. In particular, Pre_N is reached in case of inconsistency, where N is the set of literals that may be used for the potential upcoming *core* action; while $Eval$ is reached in case of consistency. From an outermost state, of the type $Eval$, a new *core* is started with *NewSet*, whenever there is a gap between over- and under-approximation; otherwise, the *Final* control rule leads to the terminal state. $Fail^1_{pre}$ and $Fail^2_{pre}$ lead to the intermediate type of control state, Pre_N, that can either restart a *core* action with

Return rules
$$Fail^1_{pre} \quad L_{O,U,core_N} \implies Pre_{N\setminus\{l\}}(O, U \cup \{\bar{l}\}) \quad \text{if} \left\{ \; L \text{ is inconsistent and } \widehat{L} \cap N = \{l\} \right.$$
$$Fail^2_{pre} \quad L_{O,U,core_N} \implies Pre_{N\setminus\widehat{L}}(O, U) \quad \text{if} \left\{ \; L \text{ is inconsistent and } |\widehat{L} \cap N| > 1 \right.$$
$$Find_{pre} \quad L_{O,U,core_N} \implies Eval(O \cap L, U) \quad \text{if} \left\{ \; L \text{ is consistent and } L \neq \emptyset \right.$$

Control rules
$$Main \quad Pre_N(O, U) \implies Cont(O, U) \quad \text{if} \left\{ \; N = \emptyset \right.$$
$$Continue \quad Pre_N(O, U) \implies \emptyset_{O,U,core_N} \quad \text{if} \left\{ \; N \neq \emptyset \right.$$
$$NewSet \quad Eval(O, U) \implies \emptyset_{O,U,core_{\overline{O}}} \quad \text{if} \left\{ \; O \neq U \right.$$
$$Final \quad Eval(O, U) \implies Ok(O) \quad \text{if} \left\{ \; O = U \right.$$

Fig. 6. The transition rules of *in*.

$$\Pi_{\{a,b,c\},\emptyset,core_{\{\neg a,\neg b,\neg c\}}} = \left\{ \begin{array}{l} a \leftarrow \neg b \\ b \leftarrow \neg a \\ c \leftarrow a \\ c \leftarrow b \end{array} \right\} \cup \left\{ \begin{array}{l} \leftarrow a \\ \leftarrow b \\ \leftarrow c \end{array} \right\}$$

$$\begin{array}{ll} & \emptyset_{\{a,b,c\},\emptyset,core_{\{\neg a,\neg b,\neg c\}}} \\ CoreOracle: & c\neg c_{\{a,b,c\},\emptyset,core_{\{\neg a,\neg b,\neg c\}}} \\ Fail^1_{pre}: & Pre_{\{\neg a,\neg b\}}(\{a,b,c\},\{c\}) \\ Continue: & \emptyset_{\{a,b,c\},\{c\},core_{\{\neg a,\neg b\}}} \end{array}$$

$$\Pi_{\{a,b,c\},\{c\},core_{\{\neg a,\neg b\}}} = \left\{ \begin{array}{l} a \leftarrow \neg b \\ b \leftarrow \neg a \\ c \leftarrow a \\ c \leftarrow b \end{array} \right\} \cup \left\{ \begin{array}{l} \leftarrow a \\ \leftarrow b \end{array} \right\}$$

$$\begin{array}{ll} CoreOracle: & ab\neg a\neg b_{\{a,b,c\},\{c\},core_{\{\neg a,\neg b\}}} \\ Fail^2_{pre}: & Pre_\emptyset(\{a,b,c\},\{c\}) \\ Main: & Cont(\{a,b,c\},\{c\}) \end{array}$$

Fig. 7. A path in FS_Π.

Continue, or continue with the *Main* rule. Figure 7 shows a possible path in FS_Π for the program Π of Example 1.

Theorem 3

Let Π be a program, and let O and U be two set of atoms. Then, (i) the only reachable terminal states are either $Cont(O, U)$ or $Ok(O)$; (ii) if $Ok(O)$ is reachable in FS_Π, then FS_Π solves cautious reasoning; (iii) if $Cont(O, U)$ is reachable in FS_Π, then $U \subseteq cautious(\Pi) \subseteq O$.

Chunking and core-based methods can be combined using our methodology to abstract Algorithm 7 from (Janota et al. 2015). Such a combination will be employed in the experiments.

4 Experimental Analysis

The abstract solvers reported in this paper have been used for implementing several algorithms in the ASP solver WASP (Alviano et al. 2015; Alviano et al. 2019), resulting in the following new versions of WASP:

- WASP-CHUNK-2, i.e., WASP running the algorithm based on chunking, with the size of the chunk set to 2;
- WASP-CHUNK-20%, i.e., WASP running the algorithm based on chunking, with the size of the chunk set to the 20% of the initial number of candidates, where the initial set of candidates is the whole set of atoms;
- WASP-CB, i.e., WASP running the algorithm based on cores.

- wasp-cb-2, i.e., wasp running the algorithm based on cores and chunking, with the size of the chunk set to 2;
- wasp-cb-20%, i.e., wasp running the algorithm based on cores and chunking, with the size of the chunk set to the 20% of the initial number of candidates.

Benchmark selection. The performance of these versions of wasp was measured on the benchmarks considered in (Alviano et al. 2018). In particular, (Alviano et al. 2018) includes *(i)* all the 193 instances from the latest ASP Competitions (Calimeri et al. 2014; 2016; Gebser et al. 2017) involving non-ground queries; *(ii)* 115 instances of ASP Competitions classified as *easy*, that is, those for which a stable model is found within 20 seconds of computation by mainstream ASP systems; and *(iii)* instances from abstract argumentation frameworks submitted to the 2nd International Competition on Computational Models of Argumentation. In this paper, instances from *(iv)* are not included since they are trivial for all tested solvers (Alviano et al. 2018).

Compared approaches. As a reference to the state of the art, we used clasp v. 3.3.3 (Gebser et al. 2012), which implements algorithm or (i.e., *over-approximation*), and the best performing algorithms implemented by wasp (Alviano et al. 2014; Alviano et al. 2018), namely or (i.e., *over-approximation*), ict (i.e., *under-approximation*), opt, and cm.

Algorithm opt was presented in (Alviano et al. 2018). The idea is as follows. Given a set of objective atoms A, the branching heuristic of the solver is forced to select $\neg p$ for $p \in A$, before any other unassigned literal. In this way, the search is driven to falsify as many atoms in A as possible. When all atoms in A are assigned, standard answer set search procedure is applied without further modifications to the branching heuristic. Therefore, whenever an answer set is found, it is guaranteed to be minimal with respect to the set of objective atoms (Di Rosa et al. 2010). When the current assignment to atoms in A cannot be extended to an answer set, then the assignment of some atom in A is flipped, and hence the procedure is repeated with a different assignment for the objective atoms. For cautious reasoning, A is initialized to the set of all candidates and updated whenever an answer set is found.

Algorithm cm was also presented in (Alviano et al. 2018) and is based on the property that an atom is a cautious consequence of a given program if and only if the negation of the atom is an unsatisfiable core. Hence, the algorithm searches for an answer set falsifying all candidates, with the aim of eliminating all remaining candidates at once. As soon as no such an answer set exists, the returned unsatisfiable core is either minimized to a singleton or used to discard candidates.

Note that all tested algorithms take advantage of the incremental interface of clasp and wasp, which is based on the concept of *assumptions literals*. The incremental interface allows the solver to reuse part of the computation among different calls, e.g., learned constraints and heuristic parameters.

The dlv solver is not considered here as its performance on cautious reasoning has been shown in earlier work to be dominated by the other approaches considered in (Alviano et al. 2014).
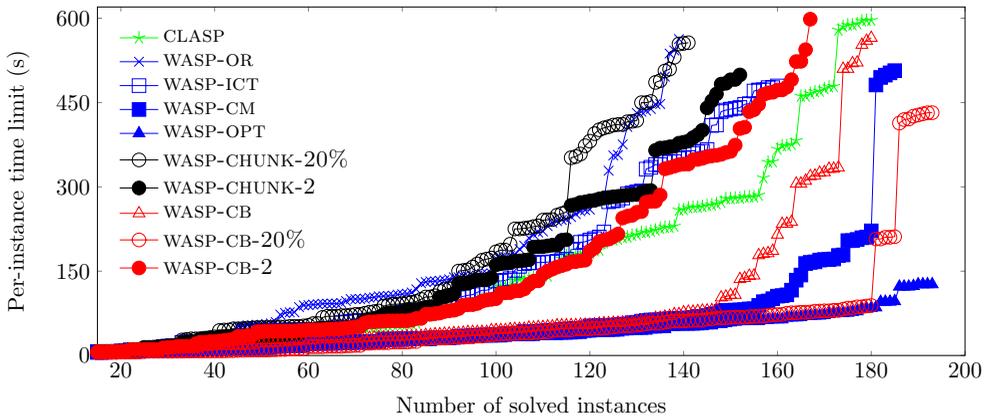
Fig. 8. Benchmark *(i)*: Performance comparison on non-ground queries in ASP Competitions.

Hardware configurations and limits. The experiments were run on computing nodes with Intel Xeon 2.4-GHz processors and 16 GB of memory. Time and memory limits were set to 600 seconds and 15 GB, respectively.

4.1 Results

Concerning benchmark *(i)*, results are shown in the cactus plot of Figure 8, where for each algorithm the number of solved instances in a given time is reported, producing an aggregated view of its overall performance. As a first observation, WASP cannot reach the performance of CLASP on the execution of algorithm OR, and indeed CLASP solved 41 instances more than WASP-OR. However, such a huge gap is completely filled by WASP-CB-20%, which actually solves 13 instances more than CLASP. Indeed, WASP-CB-20% is able to solve all instances with an average running time of 56 seconds, and is comparable to the best performing algorithm, namely WASP-OPT, which solves all instances with an average running time of 36 seconds. Notably, even a small size of the chunk may have a huge impact on the performance of the algorithms. Indeed, WASP-CB outperforms WASP-CB-2, solving 13 instances more. Finally, we observe that WASP-CHUNK-20% and WASP-CHUNK-2 are not competitive with algorithms based on cores.

Concerning benchmark *(ii)*, results are shown in the cactus plot of Figure 9. It is possible to observe that CLASP is the best performing solver on this benchmark, solving 53 instances overall. If we focus on WASP, the best performance is obtained by WASP-CHUNK-2, WASP-OR, WASP-CM, and WASP-CHUNK-20% which are able to solve 41, 41, 41, and 40 instances, respectively. Moreover, WASP-CB cannot reach the same performance on this benchmark, solving only 25 instances. We observe that the poor performance depends on the first calls to the oracle, since they are expensive in terms of solving time. This negative effect is mitigated by chunking since WASP-CB-20% and WASP-CB-2 solve 37 and 39 instances, respectively.

Finally, detailed results of benchmarks *(i)* and *(ii)* are shown in Table 1, where we report the 5 algorithms solving the largest number of instances. In particular, for each algorithm we report the number of solved instances and the cumulative solving time (for

Table 1. *Numbers of solved instances and cumulative running time (in seconds; each timeout adds 600 seconds) on instances from benchmarks (i) and (ii).*

|		clasp		wasp-cm		wasp-opt		wasp-cb		wasp-cb-20%	
Benchmark	#	sol.	sum t	sol.	sum t	sol.	sum t	sol.	sum t	sol.	sum t
CQA-Q3	40	40	4354	40	1313	40	1276	40	1291	40	1303
CQA-Q6	40	40	8505	40	2149	40	1956	40	3544	40	1849
CQA-Q7	40	40	8929	40	1741	40	1681	40	1735	40	1724
MCSQ	73	60	12701	65	11007	73	1995	60	15757	73	5924
GracefulGraphs	1	1	51	1	45	1	32	1	44	1	57
GraphCol	1	0	600	0	600	0	600	0	600	0	600
IncrSched	6	5	857	2	2692	1	3016	1	3006	1	3004
KnightTour	2	2	62	0	1200	0	1200	0	1200	0	1200
Labyrinth	32	6	18377	0	19200	0	19200	0	19200	1	18912
NoMystery	2	1	1091	1	694	0	1200	1	706	1	721
PPM	15	15	264	15	81	15	76	15	113	15	76
QualSpatReas	18	18	1019	17	4537	7	7406	7	7083	14	5707
Sokoban	36	3	20529	3	20665	1	21102	1	21023	2	20918
VisitAll	2	2	80	2	408	1	757	2	348	2	396
Total	**308**	**233**	**78584**	**226**	**66931**	**219**	**62097**	**208**	**76252**	**230**	**62993**
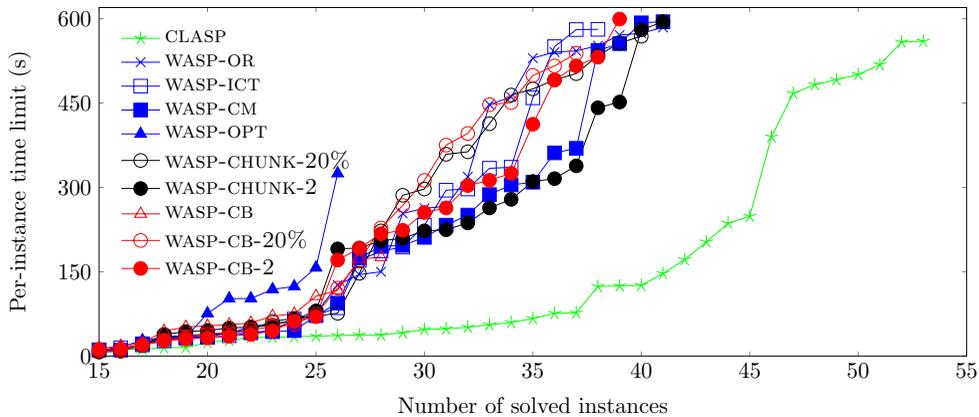
Fig. 9. Benchmark *(ii)*: Performance comparison on computation of cautious consequences for *easy* instances of ASP Competitions.

each timeout we added 600 seconds). We also observe that WASP-CB-20% is comparable with CLASP solving only 3 instances less.

5 Related Work

Abstract solvers methodology for describing solving procedures have been introduced for the DPLL procedure with learning of SAT solving and for certain extensions implemented in SMT solvers (Nieuwenhuis et al. 2006). In ASP, Lierler (2008) introduced and compared the abstract solvers for SMODELS and CMODELS on non-disjunctive programs, then in (Lierler 2011) the framework has been extended by introducing transition rules that

capture backjumping and learning techniques. Lierler and Truszczynski (2011) presented a unifying perspective based on completion of solvers for non-disjunctive answer set solving. Brochenin et al. (2014) presented abstract solvers for disjunctive answer set solvers CMODELS, GNT and DLV implementing plain backtracking, and Lierler (2014) defined abstract frameworks for Constraint ASP solvers.

All these papers describe ASP procedures for computing (one) stable models in abstract solvers methodology. In our paper we have, instead, focused on the description of ASP procedures for cautious reasoning tasks, possibly employing some of the solutions presented in related papers as ASP oracle calls. Our paper significantly extends the short technical communication (Brochenin and Maratea 2015a) by (*i*) designing more advanced solving techniques, like chunking and core-based algorithms, that lead to new solving solutions, (*ii*) implementing and testing such new solutions, (*iii*) adding further examples and a detailed related work, and (*iv*) formally stating a strong analogy between backbones computation in SAT and cautious reasoning in ASP.

As far as the application of abstract solvers methodology outside ASP is concerned, the first application has been already mentioned and is related to the seminal paper (Nieuwenhuis et al. 2006), where SMT problems with certain logics via a lazy approach (Sebastiani 2007) are considered. Then, abstract solvers have been presented for the satisfiability of Quantified Boolean Formulas by Brochenin and Maratea (2015b), and for solving certain reasoning taks in Abstract Argumentation under preferred semantics (Brochenin et al. 2018). Finally, in another number of papers, starting from a developed concept of modularity in answer set solving (Lierler and Truszczynski 2013), abstract modeling of solvers for multi-logic systems are presented (Lierler and Truszczynski 2014; Lierler and Truszczynski 2015; Lierler and Truszczynski 2016).

Another added, general, value of our paper is in its practical part, i.e., an implementation of new solutions designed through abstract solvers. In fact, while nowadays abstract solvers methodology has been widely used, often in the mentioned papers the presented results have rarely led to implementations, with the exception of (Nieuwenhuis et al. 2006), where the related BARCELOGIC implementation won the SMT Competition 2005 on same logics, and (Lierler 2011), where a proposed combination of SMODELS and CMODELS techniques has been implemented in the solver SUP, that reached positive results at the ASP Competition 2011 and, more recently, (Brochenin et al. 2018), where the new designed solution, obtained as a modification of the CEGARTIX solver, performed often better than the basic CERGATIX solver on preferred semantics, that was among the best solvers in the first ICCMA competition.

Finally, very recently improved algorithms for computing cautious consequences of ASP programs have been presented in (Alviano et al. 2018): such algorithms could be also modeled through abstract solvers and combined with the ones presented in this paper.

6 Conclusion

In this paper we modeled through abstract solvers advanced techniques for solving cautious reasoning tasks in ASP. Such advanced techniques have been borrowed from the computation of backbones of propositional formulas. We have then designed new solving procedures, and implemented them in WASP, that already included algorithms of

([Alviano et al. 2014](#); [Alviano et al. 2018](#)). Experiments on devoted benchmarks have shown positive results for the new proposed solutions. At the same time, our work has formally stated, through an uniform treatment, a strong analogy among the algorithms for computing backbones of propositional formulas and those for computing cautious consequences of ASP programs. Finally, we remark that algorithms presented in this paper are independent with respect to the underlying solving strategies, and can be complemented with existing heuristics and optimization techniques ([Giunchiglia et al. 2002](#); [Giunchiglia et al. 2003](#); [Giunchiglia et al. 2008](#)).

References

ALVIANO, M., AMENDOLA, G., DODARO, C., LEONE, N., MARATEA, M., AND RICCA, F. 2019. Evaluation of disjunctive programs in WASP. In *Proc. of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019)*, M. Balduccini, Y. Lierler, and S.Woltran, Eds. Lecture Notes in Computer Science, vol. 11481. Springer, 241–255.

ALVIANO, M. AND DODARO, C. 2016. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming 16,* 5-6, 533–551.

ALVIANO, M., DODARO, C., JÄRVISALO, M., MARATEA, M., AND PREVITI, A. 2018. Cautious reasoning in ASP via minimal models and unsatisfiable cores. *Theory and Practice of Logic Programming 18,* 3-4, 319–336.

ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. 2015. Advances in WASP. In *Proceedings of the 13th International Conference of Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

ALVIANO, M., DODARO, C., AND RICCA, F. 2014. Anytime computation of cautious consequences in answer set programming. *Theory and Practice of Logic Programming 14,* 4-5, 755–770.

ARENAS, M., BERTOSSI, L. E., AND CHOMICKI, J. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming 3,* 4-5, 393–424.

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

BREWKA, G. AND EITER, T. 2007. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of National conference on Artificial Intelligence (AAAI 2007)*, AAAI Press, 385–390.

BROCHENIN, R., LIERLER, Y., AND MARATEA, M. 2014. Abstract disjunctive answer set solvers. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, 165–170.

BROCHENIN, R., LINSBICHLER, T., MARATEA, M., WALLNER, J. P., AND WOLTRAN, S. 2018. Abstract solvers for Dung's argumentation frameworks. *Argument & Computation 9,* 1, 41–72.

BROCHENIN, R. AND MARATEA, M. 2015a. Abstract answer set solvers for cautious reasoning. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*, M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, Eds. CEUR Workshop Proceedings. vol. 1433. CEUR-WS.org.

BROCHENIN, R. AND MARATEA, M. 2015b. Abstract solvers for quantified boolean formulas and their applications. In *Proc. of AI*IA 2015: Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence*, M. Gavanelli, E. Lamma, and F. Riguzzi, Eds. Lecture Notes in Computer Science, vol. 9336. Springer, 205–217.

CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence 231*, 151–181.

CALIMERI, F., IANNI, G., AND RICCA, F. 2014. The third open answer set programming competition. *Theory and Practice of Logic Programming 14,* 1, 117–135.

DI ROSA, E., GIUNCHIGLIA, E., AND MARATEA, M. 2010. Solving satisfiability problems with preferences. *Constraints 15,* 4, 485–515.

EITER, T. 2005. Data integration and answer set programming. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer, 13–25.

EITER, T., GOTTLOB, G., AND MANNILA, H. 1997. Disjunctive Datalog. *ACM Transactions on Database Systems 22,* 3 (Sept.), 364–418.

EITER, T., IANNI, G., LUKASIEWICZ, T., SCHINDLAUER, R., AND TOMPITS, H. 2008. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence 172,* 12-13, 1495–1539.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

GEBSER, M., MARATEA, M., AND RICCA, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

GELFOND, M. AND LIFSCHITZ, V. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*. MIT Press, Cambridge, Mass., pp. 1070–1080. .

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9*, 365–385.

GIUNCHIGLIA, E., LEONE, N., AND MARATEA, M. 2008. On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 169–204.

GIUNCHIGLIA, E., MARATEA, M., AND TACCHELLA, A. 2002. Dependent and independent variables in propositional satisfiability. In *Proc. of the European Conference on Logics in Artificial Intelligence (JELIA 2002)*, S. Flesca, S. Greco, N. Leone, and G. Ianni, Eds. Lecture Notes, vol. 2424. Springer, 296–307.

GIUNCHIGLIA, E., MARATEA, M., AND TACCHELLA, A. 2003. (in)effectiveness of look-ahead techniques in a modern SAT solver. In *Proc. of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, F. Rossi, Ed. Lecture Notes in Computer Science, vol. 2833. Springer, 842–846.

JANOTA, M., LYNCE, I., AND MARQUES-SILVA, J. 2015. Algorithms for computing backbones of propositional formulae. *AI Communications 28,* 2, 161–177.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

LIERLER, Y. 2008. Abstract answer set solvers. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 377–391.

LIERLER, Y. 2011. Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming 11*, 135–169.

LIERLER, Y. 2014. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence 207*, 1–22.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2011. Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming 11,* 4-5, 629–646.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2013. Modular answer set solving. In *Late-Breaking Developments in the Field of Artificial Intelligence*. AAAI Workshops, vol. WS-13-17. AAAI.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2014. Abstract modular inference systems and solvers. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages (PADL 2014)*, M. Flatt and H. Guo, Eds. Lecture Notes in Computer Science, vol. 8324. Springer, 49–64.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2015. An abstract view on modularity in knowledge representation. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, B. Bonet and S. Koenig, Eds. AAAI Press, 1532–1538.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2016. On abstract modular inference systems and solvers. *Artificial Intelligence 236*, 65–89.

LYNCE, I. AND SILVA, J. P. M. 2004. On computing minimum unsatisfiable cores. In *Proc. of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*.

MANNA, M., RICCA, F., AND TERRACINA, G. 2013. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory and Practica of Logic Programming 13,* 2, 227–252.

MAREK, V. W. AND TRUSZCZYŃSKI, M. 1998. Stable models and an alternative logic programming paradigm. *CoRR cs.LO/9809032*.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 241–273.

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.

SEBASTIANI, R. 2007. Lazy satisability modulo theories. *Journal fo Satisfiability, Boolean Modeling and Computation 3,* 3-4, 141–224.