Abstract Answer Set Solvers for Cautious Reasoning

REMI BROCHENIN and MARCO MARATEA
University of Genova, Italy
(email: {remi.brochenin,marco.maratea}@unige.it)

submitted 29 April 2015; accepted 5 June 2015

Abstract

Abstract solvers are a recently employed method to formally analyze algorithms that earns some advantages w.r.t. traditional ways such as pseudo-code-based description. Abstract solvers proved to be a useful tool for describing, comparing and composing solving techniques in various fields such as SAT, SMT, ASP, CASP. In ASP, abstract solvers have been so far employed for describing solvers for brave reasoning tasks.

In this paper we apply, for the first time, this methodology to the analysis of ASP solvers for cautious reasoning tasks. We describe and compare the available approaches in the literature, which employ techniques for computing over- and under-approximations of the solution, the last including "coherence tests" for deciding the inclusion of a single atom in the solution, a technique borrowed from backbone computation of CNF formulas. Then, we show how to improve the current abstract solvers with new techniques, in order to design new solving algorithms.

KEYWORDS: Answer Set Programming, Abstract solvers, Cautious reasoning

1 Introduction

Abstract solvers are a relatively recently employed method to formally analyse algorithms that earns some advantages w.r.t. traditional ways such as pseudo-code-based description. In this methodology, the states of 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. Abstract solvers proved to be a useful tool for describing, comparing and composing solver design techniques in various fields such as CNF satisfiability (SAT) and Satisfiability Modulo Theories (SMT) (Nieuwenhuis et al. 2006), Answer Set Programming (Lierler 2011; Lierler and Truszczynski 2011; Brochenin et al. 2014), and Constraint ASP (Lierler 2014). In ASP, such methodology led to the development of a new ASP solver, SUP (Lierler 2011); however, abstract solvers have been so far 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).

In cautious reasoning, instead, answers must be witnessed by all stable models. Two well-known ASP solvers, i.e. DLV (Leone et al. 2006) and CLASP (Gebser et al. 2012),

allow the computation of cautious consequences of ASP programs. They use devoted techniques, i.e. over- and under-approximations of the solution, which are built on top of their procedures for computing stable models. Very recently, Alviano et al. (2014) presented a unified, high-level view of such solving procedures, and designed several algorithms for cautious reasoning in ASP. The under-approximation technique relies on a "coherence test" for deciding the inclusion of a single atom in the solution, a technique borrowed from the backbone computation of CNF formulas (Janota et al. 2015): all these techniques are implemented (and tested) on top of the ASP solver WASP (Alviano et al. 2013).

In this paper, starting from the algorithms description of Alviano et al. (2014), we apply, for the first time, abstract solvers to the analysis of ASP solvers for cautious reasoning tasks. We first describe and compare the mentioned available techniques, and show how they can be modeled by adding edges to the graphs describing ASP solvers in a modular way. Finally, we outline how to include in our picture further (optimization) techniques, like backjumping and learning, as well as other algorithms of Janota et al. (2015).
To sum up, the main contributions of this paper are:

- We employ abstract solvers for analyzing ASP solvers for cautious reasoning tasks.
- We define a framework that can ease the import of other techniques originally developed in solvers for computing backbones of CNF formulas in the abstract procedures for cautious reasoning.
- We outline how to include further (optimization) techniques in our graphs.

The paper is structured as follows. Section 2 introduces needed preliminaries, while Section 3 reviews abstract solvers for SAT and ASP, in a uniform way. Section 4 presents abstract solving algorithms for cautious reasoning in ASP. The paper ends by presenting how to add further techniques in our graphs, in Section 5, and with related work and conclusions in Section 6.

2 Preliminaries

Syntax. An *atom* is a Boolean variable over $\{\top, \bot\}$. A *positive literal* is an atom. A *negative literal* is the negation of an atom $\neg a$. A *literal* is a positive literal or a negative literal. A *rule* is a pair (A, B), written $A \leftarrow B$, where B is a set of literals and A is a set of positive literals of size at most 1. A *program* is a finite set of rules. The complement $\overline{l}$ of a literal l is the literal a for $l = \neg a$ and the literal $\neg a$ for $l = a$. For a set of literals M, by $\overline{M}$ we denote the set of the complements of the literals of M, by M^+ is the set of positive literals of M and M^- is the set of negative literals of M. A set of literals is *consistent* if it does not contain both l and $\overline{l}$. In a rule $A \leftarrow B$,

1. A is called the *head* of the rule;
2. B is the *body* of the rule, of which elements are traditionally written separated by a comma as in $a \leftarrow b, \neg c$.

We may also 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 $A \leftarrow \{l\} \cup B$. Note that we focus on non-disjunctive programs in this article, hence A can only be empty or a singleton.

Finally, for a program Π, by $atoms(\Pi)$ we denote the set of atoms occurring in Π; we also use this notation for sets of literals and conjunctions. For a set of atoms X, a literal relative to X is a literal of which atom belongs to X, and $lit(X)$ is the set of all literals relative to X.

Semantics. An *assignment* to a set X of atoms is a total function from X to $\{\bot, \top\}$. We identify a consistent set M of literals with an assignment to $atoms(M)$: $a \in M$ iff a is mapped to $\top$, and $\neg a \in M$ iff a is mapped to $\bot$. A *classical model* of a program Π is an assignment M to $atoms(M)$ such that for each rule R of Π, $R \cap M \neq \emptyset$. 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 $A \leftarrow B^+$. A *stable model* of a program Π is an assignment M to $atoms(M)$ such that M^+ is minimal among the M_0^+ such that M_0 is a classical model of Π^{M^+}. If a program Π has at least one stable model then $cautious(\Pi)$ is the intersection of all the stable models of Π.

Following Leone et al. (1997), for a program Π and an assignment M over $atoms(\Pi)$, a subset X of $atoms(\Pi)$ is said to be *unfounded* on M w.r.t. Π when for each atom a in X and each rule $a \leftarrow B$ in Π, one of the following condition holds: (i) $M \cap \overline{B} \neq \emptyset$, or (ii) $X \cap B^+ \neq \emptyset$.

3 Abstract solvers

The Davis-Putnam-Logemann-Loveland procedure (DPLL) (Davis et al. 1962) exhaustively explores assignments which are good candidates for classical models of CNF formulas. An abstract transition system for DPLL has been proposed by Nieuwenhuis et al. (2006), and then extended to procedures that exhaustively explore assignments which are good candidates for stable models of answer set programs (Lierler 2008). We present here these abstract transition systems, with slight modifications that will allow us to introduce abstract solvers for cautious reasoning with more clarity.

States and Basic Rules. For a set of atoms X, an *action relative to* X is an element of the set $\{init, over\} \cup \{under_{\{l\}} | l \in lit(X)\}$. A *decision literal relative to* X is a literal relative to X annotated with d, as in a^d. An *annotated literal relative to* X is a literal or a decision literal relative to X. For a set X of atoms, a *record* relative to X is a string L composed of annotated literals relative to X without repetitions. A record L is *consistent* if it does not contains both a literal and its negation, whether annotated or not. We may view a record as the set containing all its elements stripped from their annotations. For example, we may view $b^d\ \neg a$ as $\{\neg a, b\}$, and hence as the assignment that maps a to $\bot$ and b to $\top$. A record L is *decision-free* if it does not contains any annotated literal.

The set of *states relative to* X, written V_X, is the union of:

- The set of *core states relative to* X, that are all $L_{O,U,A}$, s.t. L is a record relative to X, O and U are sets of literals relative to X, and A is an action relative to X.
- The set of *control states relative to* X, that are all the $Cont(O, U)$ where O and U are sets of literals relative to X.

Oracle rules

$Backtrack$	$Ll^d L'_{O,U,A}$	$\Longrightarrow L\overline{l}_{O,U,A}$	if $\begin{cases} Ll^d L' \text{ is inconsistent and} \\ L' \text{ contains no decision literal} \end{cases}$
$Decide$	$L_{O,U,A}$	$\Longrightarrow Ll^d{}_{O,U,A}$	if $\begin{cases} L \text{ is consistent and} \\ \text{neither } l \text{ nor } \overline{l} \text{ occur in } L \end{cases}$

Return rule

$Fail_{init}$	$L_{O,U,init}$	$\Longrightarrow Failstate$	if $\{$ L is inconsistent and decision-free

Fig. 1. The basic transition rules of $base$.

- The *fail state* $Failstate$;
- The set of *final states relative to* X, that are all the $Ok(W)$ where W is a set of literals relative to X.

The graphs we will define with these states represent algorithms that search for assignments with certain properties. In a core state, L represents what is known of the assignment currently studied. The fail state means that the computation failed, in other words there is no such mathematical object as the one we were searching for. The use of control and final states, as well as of the subscripts O, U, A in core states, does not influence our current graphs and will come in next section. Also, in this section the actions A will always be $init$.

The *basic rules* are the rules from the set of transition rules $base = \{Backtrack, Decide, Fail_{init}\}$, presented in Figure 1. The basic rules along with the states relative to X will be part of many transition graphs we will present in the article. The rule $Decide$ means that we make an arbitrary decision to assign a value to an atom. The rules specific to a problem that we will add later will provide deterministic consequences to the choices already made, as literals that will not be decision literals. Since decisions with $Decide$ are arbitrary, if we find a contradiction we should be able to backtrack to an earlier point, and the rule $Backtrack$ means that the present state of computation is inconsistent and we reverse our latest arbitrary decision. The rule $Fail$ means that we have found a contradiction and no backtracking is possible.

Graphs. Intuitively, every reachable state of the graphs represents a possible state of a computation, and a path in the graph from the initial state is the description of possible search for a model.

We now define the program $\Pi_{O,U,init}$ to be Π. From the transition rules presented in Figure 2, we define $dp = \{UnitPropagate\}$, $cl = \{UnitPropagate, Unfounded\}$ and $sm = \{UnitPropagate, AllRulesCancelled, BackchainTrue, Unfounded\}$ (i.e. rules employed in, e.g. SMODELS (Simons et al. 2002)). The rules in sm mean that we can add a literal that is a straightforward consequence of our previous decisions and the studied formula or program. Such deterministic consequences can be different, depending on whether we are searching for a classical model or a stable model. In case of a search for a classical model one may use dp, but since we search for a stable model we will use cl or sm.

Oracle rules

$$\begin{array}{lll}
UnitPropagate & L_{O,U,A} \Longrightarrow Ll_{O,U,A} & \text{if } \left\{ \begin{array}{l} \text{in } \Pi_{O,U,A}, \text{ there is :} \\ l \leftarrow B \text{ such that } B \subseteq L \text{ or} \\ A \leftarrow l, B \text{ such that } \overline{A} \cup B \subseteq L \end{array} \right. \\
AllRulesCancelled & L_{O,U,A} \Longrightarrow L\neg a_{O,U,A} & \text{if } \left\{ \begin{array}{l} \text{for all } a \leftarrow B \text{ in } \Pi_{O,U,A}, B \cap L \neq \emptyset \end{array} \right. \\
BackchainTrue & L_{O,U,A} \Longrightarrow Ll_{O,U,A} & \text{if } \left\{ \begin{array}{l} \text{in } \Pi_{O,U,A}, \text{ there is } a \leftarrow l, B \text{ s. t. } a \in L \text{ and} \\ \text{for all other } a \leftarrow B' \text{ holds } B' \cap L \neq \emptyset \end{array} \right. \\
Unfounded & L_{O,U,A} \Longrightarrow L\neg a_{O,U,A} & \text{if } \left\{ \begin{array}{l} \text{there is } X \text{ such that } a \in X \text{ and} \\ X \text{ is unfounded on } L \text{ w.r.t } \Pi_{O,U,A} \end{array} \right.
\end{array}$$

Fig. 2. The transition rules of sm.

The family of graphs $(V_X, base \cup sm)$ defines a general graph that include the transitions that characterise the behavior of backtracking-based SAT and (some) ASP solvers: To find a stable model of Π it is enough to find a path leading from a proper initial vertex to a terminal vertex in the graph employing the cl or sm set of transition rules. If the terminal vertex is $Failstate$, then no such model exists. Otherwise, L is a stable model of Π.

The use of the sets of transition rules $cl \cup base$ and $sm \cup base$ respectively characterise the behavior of CLASP or WASP, and DLV, without backjumping and learning[1], i.e. the solvers on top of which approaches for cautious reasoning are implemented. Thus, in our graphs they will describe the search of an ASP oracle call.

4 Abstract Solvers for Computing Cautious Consequences

In this section we abstract the algorithms from Alviano et al. (2014). After explaining some general properties about the graphs we will define, we first abstract the over-approximation strategy, and then we abstract the under-approximation strategy and show how these strategies can be mixed.

General Structure. The basic architecture of all solutions, including the one implemented in WASP (Alviano et al. 2013), for determining the cautious consequences of a program is presented in Algorithm 1 of Alviano et al. (2014). Given a program, over-approximation is set to all atoms in the program, while the under-approximation is empty. A first stable model is searched (ASP oracle call at line 2), which is characterised by the action $init$ in the $base$ transition rules and, if found, serves as a first over-approximation of the cautious consequences of the program. Then, iteratively either under-approximation or over-approximation (as shown in Algorithms A1, ..., A4 of Alviano et al. (2014)) are applied. When under- and over-approximations coincide, the set of cautious consequences O has been found and the computation terminates by going to the state $Ok(O)$ in the graph.

Hence, the full extent of states relative to X becomes useful. As previously, the fail state

[1] CLASP and WASP are conflict-driven solvers relying heavily on learning, so it is not easy to describe such solvers as backtracking-based procedures without learning. For CLASP we refer to the considerations in Sec. 8.2 of Lierler (2011); WASP has a similar behavior to that of CLASP, which is confirmed by personal communications with the authors. For DLV we refer to Giunchiglia et al. (2008). More precise characterizations, including backjumping and learning techniques, will be presented in Section 5.

Return rules

$Fail_{over}$	$L_{O,U,over}$	$\Longrightarrow Ok(O)$	if $\{$ L is inconsistent and decision-free
$Find$	$L_{O,U,A}$	$\Longrightarrow Cont(O \cap L, U)$	if $\{$ no other return or oracle rule applies

Control rules

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

Fig. 3. The transition rules of ov.

$Failstate$ is reached when there is no model. The final states $Ok(W)$ are the only other type of terminal states, and W is what we want to compute in the graph – in this section the cautious consequences of the program. The initial state will always be $\emptyset_{atoms(\Pi),\emptyset,init}$ in this section. The core states $L_{O,U,A}$ and the control states $Cont(O,U)$ represent all the intermediate steps of the computation; they are such that:

- L is the current state of the computation of a model;
- O is the current over-approximation of the solution stored as a set;
- U is the current under-approximation of the solution stored as a set;
- A is the action currently carried out: $init$ if we search for a first model, $over$ (resp. $under_{\{l\}}$) action if over-approximation (resp. under-approximation on a literal l) 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. We say that *no cycle is reachable* if there is no reachable state which is reachable from itself.

Definition 1
We say that a graph G *finds* a statement M that defines a set of literals, or "finds M", if the following conditions hold:

1 G is finite, and no cycle is reachable in G,
2 the only reachable terminal state is $Failstate$ if M does not define a set of literals, and
3 the only reachable terminal state is $Ok(M)$ otherwise.

In the following we will define graphs that find the cautious consequences of a program.

Over-approximation. We first abstract Algorithm A2 of Alviano et al. (2014). It corresponds to the strategy of CLASP and DLV to compute cautious consequences.[2]

We define $\Pi_{O,U,over}$ as $\Pi \cup \{\leftarrow O\}$. We call ov the set containing all the rules presented in Figure 3, that is $ov = \{Fail_{over}, Find, Success, OverApprox\}$. These rules are organised into Return rules, that handle the result of an oracle call, and Control rules, that

[2] The strategy of DLV slightly differs from that of CLASP as the studied program in DLV is Π to which is added one constraint per found stable model so as to avoid that precise stable model. CLASP, instead, add a constraint built on the over-approximation. Moreover, DLV does not forget the added constraints later, thus it may run in exponential space.

Return rule
$Fail_{under}$ $\quad L_{O,U,under_{\{l\}}} \Longrightarrow Cont(O \setminus \{\overline{l}\}, U \cup \{l\})$ if $\{$ L is inconsistent and decision-free

Control rule
$UnderApprox$ $\quad Cont(O,U) \Longrightarrow \emptyset_{O,U,under_{\{l\}}}$ if $\{$ $l \in O \setminus U$

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

direct the search given the actual values of O and U. Intuitively, $Fail_{over}$ means that a call to an oracle did not find a stable model, so O is the solution. If $Find$ is triggered, instead, we go to a control state where the over-approximation O is updated according to the stable model found: then, if $O = U$ a solution is found through $Success$, otherwise the search is restarted ($L = \emptyset$) in an oracle state with $OverApprox$. For any Π, the graph OS_Π is $(V_{atoms(\Pi)}, base \cup ov \cup sm)$.

In OS_Π, first an oracle is called under the $init$ action. If it fails there is no stable model at all and the computation ends on $Failstate$. If it succeeds, the obtained set is used as a first over-approximation of the cautious consequences of the studied program. Then, iteratively, the oracle is called to find other stable models that reduce the over-approximation O. The $over$ action is used for this purpose. If a stable model M is found, since $\Pi_{O,U,over}$ is $\Pi \cup \{\leftarrow O\}$, for sure $M \cap O \neq \emptyset$. Hence some progress has been made, the over-approximation can be updated to $M \cap O$. Otherwise, if no stable model is found, there is no stable model that does not contain all the elements of O. Also, since at least one stable model has been found during the $init$ action, all the elements of O belong to a stable model. Hence O is the set of cautious consequences of the studied program. Formally:

Theorem 1
For any program Π, the graph OS_Π finds $cautious(\Pi)$.

Under-approximation. We now abstract the strategy A3 of Alviano et al. (2014). We define $\Pi_{O,U,under_l}$ as $\Pi \cup \{\leftarrow l\}$. We call un the set $\{Fail_{under}, Find, Success, UnderApprox\}$ containing the rules presented in Figure 4 plus $Find$ and $Success$ from Figure 3. Intuitively, $Fail_{under}$ updates over- and under-approximations in case a test on l 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 $(V_{atoms(\Pi)}, base \cup un \cup sm)$.

In US_Π, again, a first oracle call takes place with the action $init$. This provides a first over-approximation. Then, iteratively, for one element (i.e. the tested literal l) of the current over-approximation O the oracle is called to find a stable model of the program that does not contain this element. If such a stable model M is found then the element can be removed from the over-approximation, along with any other element of the over-approximation that would not belong to M, with $O \cap M$ in the rule $Find$ kept identical from over-approximations. On the other hand, if no such stable model is found then this element can be added to the under-approximation with the rule $Fail_{under}$. Finally, at some point the over-approximation and the under-approximation are equal. The computation can end with $Success$.

Theorem 2

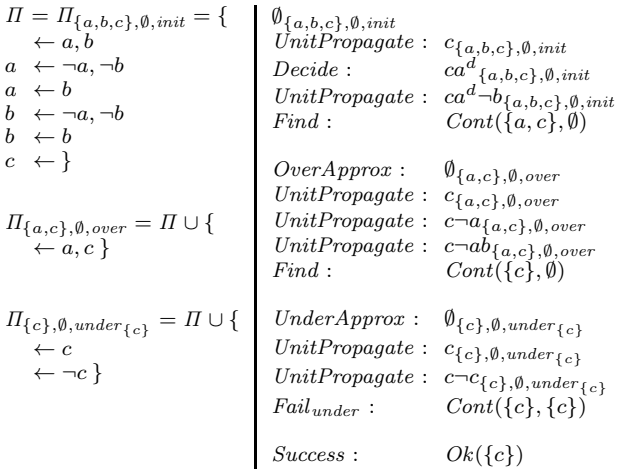

Fig. 5. A path in $MixS_{\Pi}$.

For any program Π, the graph US_{Π} finds $cautious(\Pi)$.

The actual strategy of WASP mixes both over-approximations and under-approximations. We now abstract such a mixed strategy. We define $MixS_{\Pi}$ as $(V_{atoms(\Pi)}, base \cup un \cup ov \cup sm)$, which summarises Algorithm 1 in Alviano et al. (2014). In Figure 5, we introduce an example of path through this mixed graph, where for each rule the destination state follows, while the source state is in the line above.

Theorem 3
For any program Π, the graph $MixS_{\Pi}$ finds $cautious(\Pi)$.

5 Extensions and Discussion

In this section we discuss further techniques that can be abstracted and added to the graphs described in the previous sections. After introducing backjumping and learning, it will be meaningful to add restarts. Finally, we discuss about the opportunity to apply other methods from backbone computation to the computation of cautious consequences.

Backjumping and Learning. The techniques described above are actually often implemented on top of solvers employing two additional techniques: backjumping and learning. We now show how to add these techniques to the graphs defined above. For two programs Π and Π', we say that Π is a logical consequence of Π' if any classical model of Π' is also classical model of Π.

For a set of states V and a program Π, we define the set $V^{BL(\Pi)}$ as the set of all the x^{Λ}, where $x \in V$ and Λ is a program of which rules have no nonempty head and of which atoms belong to $atoms(\Pi)$. Note that there is only a finite number of possible rules using a given finite set of atoms, hence there is only a finite number of possible such programs; so there is only a finite number of elements in $V^{BL(\Pi)}$. These elements are states that can

Oracle rules

$$\textit{Backjump} \quad Ll^dL'^{\Lambda}_{O,U,A} \Longrightarrow L\bar{l}^{\Lambda}_{O,U,A} \quad \text{if} \begin{cases} Ll^dL' \text{ is inconsistent and} \\ \leftarrow \bar{l}, L' \text{ is a logical consequence of } \Pi_{O,U,A} \end{cases}$$

$$\textit{UnitLearn} \quad L^{\Lambda}_{O,U,A} \Longrightarrow Ll^{\Lambda}_{O,U,A} \quad \text{if} \{ \text{ in } \Lambda, \text{ there is } \leftarrow l, B \text{ such that } B \subseteq L$$

Other rule

$$\textit{Learn} \quad L^{\Lambda}_{O,U,A} \Longrightarrow \emptyset^{\Lambda \cup \{\leftarrow B\}}_{O,U,A} \quad \text{if} \begin{cases} \leftarrow B \text{ is a logical consequence of } \Pi_{O,U,A} \text{ and} \\ \leftarrow B \notin \Lambda \text{ and} \\ \leftarrow B \text{ is a clause} \end{cases}$$

Fig. 6. The transition rules of bl.

hold the learnt rules, and are hence suitable for backjumping and learning. We now define a trivial transformation of edges; for a set E of edges between elements of V we define $E^{BL(\Pi)}$ as the set of edges between elements of $V^{BL(\Pi)}$ such that $(x_1^{\Lambda_1}, x_2^{\Lambda_2}) \in E^{BL(\Pi)}$ iff $(x_1, x_2) \in E$ and $\Lambda_1 = \Lambda_2$. This transformation allows us to extend existing rules to the framework of backjumping and learning. We can now define the transition rules specific to backjumping and learning in Figure 6: $\textit{Backjump}$ restores an inconsistent state by flipping the last decision literal responsible for the conflict, $\textit{UnitLearn}$ applies unit propagation on the added rules, and $\textit{Learn}$ stores an implied rule. We define bl as the set containing these three transitions rules, where $\textit{Backjump}$ and $\textit{UnitLearn}$ are oracle rules, while $\textit{Learn}$ is neither an oracle rule nor a return rule. Abstractly, $\textit{Learn}$ is an oracle rule, but has been classified in a different way for a technical issue, i.e. in order to ensure correct interaction with (the conditions enabling) rule $\textit{Find}$. Finally, for a graph $G = (V, E)$, we define $G^{BL(\Pi)}$ as $(V^{BL(\Pi)}, E^{BL(\Pi)} \cup bl)$.

If G is finite and no cycle is reachable then it is possible to show that $G^{BL(\Pi)}$ is also finite and no cycle is reachable. Note that the procedure obtained from such graphs may not run in polynomial space: hence, in general such algorithms also include a mechanism for forgetting learnt rules The following result extends the one in the previous section. For any program Π, and for any set $S \subseteq \{un, ov, ch\}$ such that $A \neq \emptyset$,

- the graph $(V_{atoms(\Pi)}, base \cup \bigcup_{x \in S} x \cup dp)^{BL(\Pi)}$ finds $backbone(\Pi)$, and
- for any set $cl \subseteq S' \subseteq sm$, the graph $(V_{atoms(\Pi)}, base \cup \bigcup_{x \in S} x \cup S')^{BL(\Pi)}$ finds $cautious(\Pi)$.

Adding restarts. In the previous graphs, it is meaningful to consider restart by adding the rule $\textit{Restart}$. Note that $\textit{Restart}$, like $\textit{Learn}$, is neither an oracle rule nor a return rule.

Other rule

$$\textit{Restart} \quad L_{O,U,A} \Longrightarrow Cont(O, U)$$

This rule can be added to any of the graphs we have defined. We more particularly study the case of $MixS_{\Pi}$. Extending the graph $MixS_{\Pi}$ with this rule and the ones in the previous paragraph provides a good abstraction of a mixed strategy in Algorithms A2*, A3* and A4*=A4 of Alviano et al. (2014). Stating a formal result on this graph involves using properties about the frequency of the use of $\textit{Restart}$ rule, which are not specified in the article. Without such assumptions, we can state the following result: The graph is finite,

the only reachable terminal state is $Failstate$ if there is no stable model, and otherwise any reachable terminal state is of the form $Ok(cautious(\Pi))$.

We are now also in the position to better characterise the behaviors of CLASP, WASP and DLV, i.e. the ASP solvers on top of which procedures for computing cautious consequences are built. In particular, CLASP and WASP also employ the techniques in bl and $Restart$, while DLV employs $Backjump$ (Ricca et al. 2006).

Methods from backbone computation. We believe that the framework we have defined can be also used, with simple adaptations, to abstract the computation of the backbone of CNF formulas. This could allow importing some solving technique into the algorithms for cautious reasoning in ASP, and also outlining relation between the computation of the backbone of CNF formulas and cautious reasoning in ASP.

As an example, some algorithms from Janota et al. (2015) share numerous similarities with the ones described in this article; they contain over-approximation and under-approximation techniques as we have described. Other algorithms from Janota et al. (2015) include a more general version of under-approximation that allows to try to add several literals to the under-approximation at once. It is possible to include this "chunk" under-approximation in the framework we have defined in this article.

Also, core-based methods presented in (Janota et al. 2015) use the unsatisfiable cores returned by SAT solvers in case a formula is unsatisfiable (see also Asín et al. (2008)). There is no published article describing such method for ASP, although there is no theoretical obstacle to its existence[3]. In particular, in the case of ASP solvers based on SAT solvers, like CMODELS, one can easily imagine to use the core output by the underlying solver. Abstracting these core-based methods through graphs as we have defined in previous sections is possible, but properly defining and explaining such graphs would require substantially more space than allowed in this article.

6 Related Work and Conclusions

Transition systems for describing solving procedures have been introduced by Nieuwenhuis et al. (2006) for the DPLL procedure of SAT solving and for certain extensions implemented in SMT solvers. In ASP, Lierler (2008) introduced and compared the transition systems for the answer set solvers SMODELS and CMODELS on non-disjunctive programs, then extended by Lierler (2011) 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. Recently, Brochenin et al. (2014) presented transition systems for disjunctive answer set solvers CMODELS, GNT and DLV implementing plain backtracking. Lierler (2014) defined abstract frameworks for *constraint answer set programming solvers*, which solves problems that integrate ASP and constraint logic programming (Mellarkod et al. 2008).

All these papers describe ASP procedures for brave reasoning. Our contribution focuses,

[3] Our considerations about the feasibility of an ASP solver using core-based methods are corroborated by personal conversations with people involved in the development of WASP. Additionally, at least one unpublished implementation of WASP is able to return a core, employing one method in (Asín et al. 2008).

instead, on the description of ASP procedures for cautious reasoning. We show abstract procedures for the algorithms presented in Alviano et al. (2014), and outline how optimization techniques like backjumping, learning and restart, and techniques from backbone computation of CNF formulas, can be added to our graphs.

Future work will be devoted to the precise characterization of these additions, and the inclusion in our framework.

References

ALVIANO, M., DODARO, C., FABER, W., LEONE, N., AND RICCA, F. 2013. WASP: A native ASP solver based on constraint learning. In *Proceedings of the 12th International Conference of Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 54–66.

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

ASÍN, R., NIEUWENHUIS, R., OLIVERAS, A., AND RODRÍGUEZ-CARBONELL, E. 2008. Efficient generation of unsatisfiability proofs and cores in SAT. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008)*. 16–30.

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

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.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem proving. *Communications of the ACM 5(7)*, 394–397.

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

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

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., 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.

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.

LEONE, N., RULLO, P., AND SCARCELLO, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation 135(2)*, 69–112.

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.

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

MELLARKOD, V. S., GELFOND, M., AND ZHANG, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

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.

RICCA, F., FABER, W., AND LEONE, N. 2006. A backjumping technique for disjunctive logic programming. *AI Communications 19,* 2, 155–172.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.