ASPeRiX, a First Order Forward Chaining Approach for Answer Set Computing ∗

Claire Lefèvre, Christopher Béatrix, Igor Stéphan, Laurent Garcia

LERIA, University of Angers,
2 Boulevard Lavoisier,
49045 Angers Cedex 01, France
Email: {claire,beatrix,stephan,garcia}@info.univ-angers.fr

submitted 24 March 2014; revised 24 November 2014; accepted 25 February 2015

Abstract

The natural way to use Answer Set Programming (ASP) to represent knowledge in Artificial Intelligence or to solve a combinatorial problem is to elaborate a first order logic program with default negation. In a preliminary step this program with variables is translated in an equivalent propositional one by a first tool: the grounder. Then, the propositional program is given to a second tool: the solver. This last one computes (if they exist) one or many answer sets (stable models) of the program, each answer set encoding one solution of the initial problem. Until today, almost all ASP systems apply this two steps computation.

In this article, the project `ASPeRiX` is presented as a first order forward chaining approach for Answer Set Computing. This project was amongst the first to introduce an approach of answer set computing that escapes the preliminary phase of rule instantiation by integrating it in the search process. The methodology applies a forward chaining of first order rules that are grounded on the fly by means of previously produced atoms. Theoretical foundations of the approach are presented, the main algorithms of the ASP solver `ASPeRiX` are detailed and some experiments and comparisons with existing systems are provided.

KEYWORDS: Answer Set Programming, solver implementation, grounding on the fly, first order, forward chaining.

1 Introduction

Answer Set Programming (ASP) is a very convenient paradigm to represent knowledge in Artificial Intelligence (AI) and to encode combinatorial problems (Baral 2003; Niemelä 1999). It has its roots in nonmonotonic reasoning and logic programming and has led to a lot of works since the seminal paper (Gelfond and Lifschitz 1988). Beyond its ability to formalize various problems from AI or to encode combinatorial problems, ASP provides also an interesting way to practically solve such problems since some efficient solvers are available. In few words, if someone wants

∗ This work was supported by ANR (National Research Agency), project ASPIQ under the reference ANR-12-BS02-0003.

to use ASP to solve a problem, he has to write a logic program in term of rules in a purely declarative manner in such a way that the answer sets (initially called stable models in (Gelfond and Lifschitz 1988)) of the program represent the solutions of his original problem.

Illustration of ASP formalism Let us take two typical examples for which ASP is suitable: the first example is devoted to knowledge representation in Artificial Intelligence and the second one is a combinatorial problem.

KR problem This first example deals with default reasoning on incomplete information. It consists in describing knowledge about birds.

$bird(titi)$.
$ostrich(lola)$.
$bird(X) \leftarrow ostrich(X)$.
$fly(X) \leftarrow bird(X), not\ ostrich(X)$.
$non\_fly(X) \leftarrow ostrich(X)$.

The meaning of the two first rules is that we have two objects: *titi* which is a bird and *lola* which is an ostrich. The meaning of the other rules is that an ostrich is a bird, a bird which is not an ostrich flies and an ostrich does not fly. Here, we are interested in deducing some properties about titi and lola. Intuitively, we want that titi flies, lola is a bird and lola does not fly. Concerning the information that lola does not fly, let us notice that it is obtained by applying the last rule since lola is an ostrich and, then, the next to last rule cannot be applied in presence of ostrich lola due to the part *not* of this rule, called *default negation*. Here, there is only one answer set which contains all the deduced pieces of information: $\{bird(titi), fly(titi), ostrich(lola), bird(lola), non\_fly(lola)\}$.

CSP problem The second example deals with the representation of a combinatorial problem: possibles worlds are represented by nonmonotonic "guess" rules and choice between these worlds is expressed by constraints. The problem is then to find (at least) one solution corresponding to a world verifying the constraints. This example is about graph 2-coloring.

$vertex(1)$.
$vertex(2)$.
$edge(1, 2)$.
$red(X) \leftarrow vertex(X), not\ blue(X)$.
$blue(X) \leftarrow vertex(X), not\ red(X)$.
$\leftarrow red(X), red(Y), edge(X, Y)$.
$\leftarrow blue(X), blue(Y), edge(X, Y)$.

This represents a graph with two vertices and an edge between them (three first rules). The two following rules are guess rules. The fourth (resp. fifth) rule means that a vertex which is not colored in blue (resp. red) has to be colored in red (resp. blue). The two last rules are constraints. They mean that two adjacent vertices can not have the same color. Here, we want to find how the two vertices should be colored (knowing that two colors are available). Intuitively, we have two solutions:

one with vertex 1 colored in blue and vertex 2 colored in red and the other one with vertex 1 colored in red and vertex 2 colored in blue. This corresponds to the two answer sets of the program: $\{vertex(1), vertex(2), edge(1,2), blue(1), red(2)\}$ and $\{vertex(1), vertex(2), edge(1,2), red(1), blue(2)\}$. However, let us note that, in this kind of problem, we are often interested in finding one solution rather than finding all the possible solutions (and the determination of only one answer set is enough).

As regards the form of the rules, we can notice that a program usually contains different kind of rules. The simplest ones are facts as $(bird(titi).)$ or $(vertex(1).)$ representing data of the particular problem. Some ones are about background knowledge as $(bird(X) \leftarrow ostrich(X).)$. Some others can be nonmonotonic as $(fly(X) \leftarrow bird(X), not\ ostrich(X).)$ for reasoning with incomplete knowledge. In other cases, especially for combinatorial problems, nonmonotonic rules can be used to encode alternative potential solutions of a problem as $(red(X) \leftarrow vertex(X), not\ blue(X).)$ and $(blue(X) \leftarrow vertex(X), not\ red(X).)$ expressing the two exclusive possibilities to color a vertex in a graph. Last, special headless rules are used to represent constraints of the problem to solve as $(\leftarrow red(X), red(Y), edge(X,Y).)$, here, in order to not color with red two vertices linked by an edge.

With the examples above we can point out that knowledge representation in ASP is done by means of *first order* rules. But, from a theoretical point of view, answer set definition is given for propositional programs and the answer sets of a first order program are those of its ground instantiation with respect to its Herbrand universe (i.e. without variables). The first order program has to be seen as an intensional version of the grounded propositional corresponding program.

ASP systems Concerning the ASP sytems, their main goal is how to compute answer sets in an efficient way. Let us recall some of their main features. Until today, almost all systems available to compute the answer sets of a program follow the architecture described in Fig. 1.

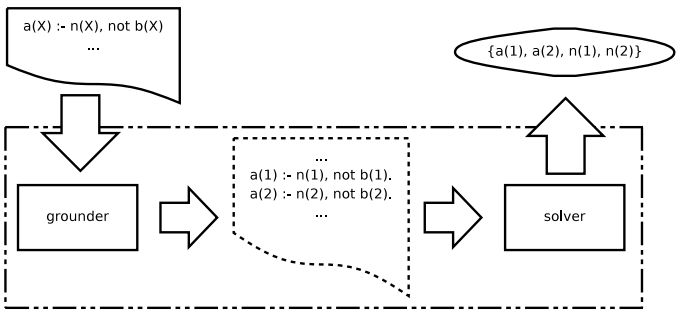

Fig. 1. Architecture of answer set computation

An ASP system begins its work by an instantiation phase in order to obtain a propositional program (and, as said above, the answer sets of the first order program will be those of its ground instantiation). After this first *grounding* phase realized by a *grounder* the solver starts the real phase of answer set computation by dealing

with a finite, but sometimes huge, propositional program. The main goal of each grounding system is to generate all propositional rules that can be relevant for a solver and only these ones, while preserving answer sets of the original program. Current intelligent grounders simplify rules as much as possible. Simplifications can lead to compute the unique answer set of some programs (for instance, programs that does not contain default negation) but it is no longer possible once the problem is combinatorial. Anyway, the grounding phase is firstly and fully processed before calling the solver.

For the grounder box we can cite `Lparse` (Syrjänen 1998) and `Gringo` (Gebser et al. 2011), and for the solver box `Smodels` (Simons et al. 2002) and `Clasp` (Gebser et al. 2012). A particular family of solvers are `Assat` (Lin and Zhao 2004), `Cmodels` (Giunchiglia et al. 2006) and `Pbmodels` (Liu and Truszczynski 2005), since they transform the answer set computation problem into a (pseudo) boolean model computation problem and use a (pseudo) SAT solver as an internal black box. In the system `DLV` (Leone et al. 2006), symbolized in Fig. 1 by the dash-line rectangle, the grounder ((Calimeri et al. 2008) describes a parallel version) is incorporated as an internal function. In the same way, `WASP` (Alviano et al. 2013) uses the `DLV` grounder (Faber et al. 2012).

Grounding The main drawback of the preliminary grounding phase is that it may lead to a lot of useless work as illustrated in the following examples.

The first examples illustrate the fact that the separation between the instantiation phase and the computation phase can prevent the (efficient) use of information relevant to the computation.

Example 1. Let P_{1a} be the following ASP program:

$$P_{1a} = \left\{ \begin{array}{l} a \leftarrow not\ b., \\ b \leftarrow not\ a., \\ \leftarrow a., \\ p(0)., \\ p(X+1) \leftarrow a,\ p(X). \end{array} \right\}$$

Grounding of P_{1a} is infinite (if an upper bound for integers is not fixed) while it has a unique (and finite) answer set $\{b,\ p(0)\}$.

Let P_{1b} be the following ASP program:

$$P_{1b} = \left\{ \begin{array}{ll} p(1)., p(2)., \ldots,\ p(N)., & \\ a \leftarrow not\ b., & aa(X,Y) \leftarrow pa(X),\ pa(Y),\ not\ bb(X,Y)., \\ b \leftarrow not\ a., & bb(X,Y) \leftarrow pb(X),\ pb(Y),\ not\ cc(X,Y)., \\ pa(X) \leftarrow a,\ p(X)., & cc(X,Y) \leftarrow aa(X,Y),\ X < Y., \\ pb(X) \leftarrow b,\ p(X)., & \leftarrow a. \end{array} \right\}$$

From the program P_{1b}, current grounders generate roughly $2.5 \times N^2$ rules.

In both programs, because of the constraint ($\leftarrow a.$) that eliminates from the possible solutions every atom set containing a, it is easy to see that rules ($p(X+1) \leftarrow$

$a,\ p(X).$) for P_{1a} and ($pa(X) \leftarrow a,\ p(X).$) for P_{1b} are useless since they can never contribute to generate an answer set of the corresponding program. In P_{1a} these useless rules are infinite ($p(X+1) \leftarrow a, p(X).$) while they are "only" large in P_{1b}: N rules with positive body containing a, like ($pa(1) \leftarrow a, p(1).$), and then, the N^2 rules with $pa(X)$ in their positive body are useless too. In defense of the actual grounders, their inability to eliminate these particular rules is not surprising since the reason justifying this elimination is the consequence of a reasoning taking into account the semantics of ASP. Thus, if we want to limit as much as possible the number of rules and atoms to deal with, we have not to separate grounding and answer set computing.

Example P_{1a} is a typical situation for planning problems where step $i+1$ must be generated only if the goal is not reached at step i. Such situations are not tractable by grounders. That is the reason why the number of steps needed to reach the goal (or at least the maximum number of allowed steps) is given as input of planning problems (in ASP competition for example). Yet it is rather counterintuitive having to know the step number to solve the problem before solving.

The next example illustrates that the grounding phase generates too much information regarding the computation of one answer set.

Example 2. Let P_2 be the program, as given in (Niemelä 1999), encoding a 3-coloring problem on a N vertices graph organized as a bicycle wheel (see below). v stands for *vertex*, e for *edge*, c for *color*, col for *colored by*, $ncol$ for *not colored by*.

$$P_2 = \left\{ \begin{array}{l} v(1)., \ldots, v(N).,\ \ c(red).,\ c(blue).,\ c(green)., \\ e(1,2)., \ldots, e(1,N)., \\ e(2,3).e(3,4)., \ldots, e(N,2)., \\ col(V,C) \leftarrow v(V),\ c(C),\ not\ ncol(V,C)., \\ ncol(V,C) \leftarrow col(V,D),\ c(C),\ C \neq D., \\ \leftarrow e(V,U),\ col(V,C),\ col(U,C). \end{array} \right\}$$

From P_2, current grounders generate about $18N$ rules. If N is even then P_2 has no answer set and if N is odd then it has 6 answer sets.

Suppose that P_2 has an answer set in which there is $col(1, red)$. Obviously, all the $N-1$ constraints like ($\leftarrow e(1,U),\ col(1,red),\ col(U,red).$) for all $U \in \{2, \ldots, N\}$ are necessary because they have to be checked. But, all the other constraints like ($\leftarrow e(1,U),\ col(1,blue),\ col(U,blue).$), and ($\leftarrow e(1,U),\ col(1,green),\ col(U,green).$) for all $U \in \{2, \ldots, N\}$ can be considered as useless since vertex 1 is not colored by *blue* or *green*. However, all these $2N-2$ constraints have been generated. So, the time consumed by this task is clearly a lost time and the memory space used by these data could have been saved. Thus, if we are searching for a single answer set, a lot of work would be done for nothing since the grounded program contains the enumeration of all solutions when only one is searched.

The last exemple shows that when the number of solutions is very important ASP solvers have more difficulty to find one solution due to the grounding phase generating a lot of information concerning all the solutions.

Example 3. Let P_3 be the program, inspired from one given in (Niemelä 1999), encoding the Hamiltonian cycle problem in a N vertices complete oriented graph. v

stands for *vertex*, a for *arc*, hc for *in Hamiltonian cycle*, nhc for *not in Hamiltonian cycle*, s for *start* and r for *reached*.

$$P_3 = \left\{ \begin{array}{l} s(1)., \quad v(1)., \ldots, v(N)., \qquad a(X,Y) \leftarrow v(X),\ v(Y)., \\ hc(X,Y) \leftarrow s(X),\ a(X,Y),\ not\ nhc(X,Y)., \\ hc(X,Y) \leftarrow r(X),\ a(X,Y),\ not\ nhc(X,Y)., \\ nhc(X,Y) \leftarrow hc(X,Z),\ a(X,Y),\ Y \neq Z., \\ nhc(X,Y) \leftarrow hc(Z,Y),\ a(X,Y),\ X \neq Z., \\ r(Y) \leftarrow hc(X,Y)., \\ \leftarrow v(X),\ not\ r(X). \end{array} \right\}$$

This program has $(N-1)!$ answer sets. Whatever the number of desired solutions, current grounders generate about $2N^2$ rules with hc predicate as head and about $2N^3$ rules with nhc predicate as head. Thus, even if we restrict our attention to the computation of one answer set, all the ASP solvers preceded by a grounding phase consume a huge amount of time when the graph has a few hundred vertices.

This previous example illustrates another strange phenomenon. Sometimes, solving a trivial problem, as finding one Hamiltonian cycle in a complete graph, is impossible for ASP systems. This is very counterintuitive since, in whole generality, in combinatorial problem solving the more solutions the problem has, the easier it is to find one of them. Again, the bottleneck for ASP systems seems to come from the huge number of rules and atoms that are generated in first, delaying and making the resolution more difficult than it should be.

Beyond these particular examples, the point to stress is that grounders generate in extension all the search space (for all potential solutions) that they give then to the solver. But, this is clearly not the approach of usual search algorithms. A classical coloring algorithm does not firstly enumerate, in extension, all possible colorations for every vertex in the graph. A finite domain solver makes choices by instantiating some variables, propagates the consequences of these choices, checks the constraints and by backtracking explores its search space. Following this strategy it instantiates and desinstantiates variables describing the problem to solve all along its search process. But, it does not build, a priori and explicitly, all the possible tuples of variables and constraints representing the problem to solve. That is why we think that if we want to use ASP to solve very large problems we have to realize the grounding process during the search process and not before it.

Is is important to notice that few works advocate the grounding of the program during the search of an answer set and not by a preprocessing. Some aim at solving the grounding bottleneck by combining ASP to constraint programming: (Baselice et al. 2005) proposes to reduce the memory requirements for a very specific class of programs, i.e. multi-sorted logic programs with cardinality constraints, (Balduccini 2009) proposes an algorithm to make cooperate an ASP solver and a Constraint Logic Programming solver in such a way that ASP is viewed as a specification language for constraint satisfaction problems and (Ostrowski and Schaub 2012) describes the `Clingcon` system which is a tight cooperation between the ASP solver `Clasp` and the Constraint Programming solver `GeCode`. The theory solvers (mainly arithmetic solvers) forbid instances that are in conflict with the constraints reduc-

ing by this way the size of the grounding image. Some others works use a forward chaining of rules that are instantiated as and when required: `GASP` (Dal Palù et al. 2009) and `ASPeRiX` (Lefèvre and Nicolas 2009a; Lefèvre and Nicolas 2009b)) developed at the same time, and more recently `OMiGA` (Dao-Tran et al. 2012). They are all based on the notion of computation given in (Liu et al. 2010). `GASP` is implemented in Prolog and Constraint Logic Programming over finite domains. Each rule instantiation and propagation is realized by building and solving a CSP. `OMiGA` is implemented in Java and uses an underlying Rete network for instantiation and propagation. `ASPeRiX`, which is the one presented in this article, is implemented in C++. Instantiation and propagation are inspired by previous work realized on the `DLV` grounder which is based on the semi-naive evaluation technique of (Ullman 1989).

Last, concerning a direct handling of first order programs, let us note that there exists some works (Gottlob et al. 1996; Eiter et al. 1997; Ferraris et al. 2007; Lin and Zhou 2007; Truszczynski 2012) dealing with first order nonmonotonic logic programs. These works establish some relations between stable model semantics and constraints systems or second order logic or circumscription but they are not really concerned by the explicit computation of answer sets.

The present paper is an extended version of (Lefèvre and Nicolas 2009a; Lefèvre and Nicolas 2009b). It details our approach of answer set computation that escapes the preliminary grounding phase by integrating it in the search process and includes:

- theoretical foundations of the approach, "mbt `ASPeRiX` computation", with complete proofs; these computations are based on those of (Liu et al. 2010) and include use of constraints and must-be-true propagation in order to guide the search;
- a detailed description of the main algorithms;
- experimentations of the resulting system, `ASPeRiX`, and comparisons with other similar systems and other "classical" ASP systems. Our methodology is particularly well suited for:
  - — solving easy problems with a large grounding,
  - — finding only one answer set for a program whose search space is large and proportional to the desired number of solutions,
  - — solving problems for which pre-grounding is impossible because domains are infinite or open, or because some pieces of knowledge come from outside (distributed systems for example).

The paper is organized as follows. In Section 2 we recall the theoretical backgrounds about ASP necessary to the understanding of our work. In Section 3 we present our first order rule oriented approach of answer set computation and its implementation in the solver `ASPeRiX`. In Section 4 experimental results are presented. We conclude in Section 5 by citing some new perspectives for ASP as a result of our innovative approach. Proofs of theorems are reported in the online appendix of the paper, Appendix B.

2 Theoretical Background

In this section, we give the main backgrounds of ASP framework useful to the understanding of this article.

Set $\mathcal{V}$ denotes the infinite countable set of *variables*, set $\mathcal{FS}$ denotes the set of *function symbols*, set $\mathcal{CS}$ denotes the set of *constant symbols* and set $\mathcal{PS}$ denotes the set of *predicate symbols*. It is assumed that the sets $\mathcal{V}$, $\mathcal{CS}$, $\mathcal{FS}$ and $\mathcal{PS}$ are disjoint and that the set $\mathcal{CS}$ is not empty. Function ar denotes the arity function from $\mathcal{FS}$ to $\mathbb{N}^*$ and from $\mathcal{PS}$ to $\mathbb{N}$ which associates to each function or predicate symbol its arity. Set $\mathbf{T}$ denotes the set of *terms* defined by induction as follows:

- if $v \in \mathcal{V}$ then $v \in \mathbf{T}$,
- if $c \in \mathcal{CS}$ then $c \in \mathbf{T}$,
- if $f \in \mathcal{FS}$ with $ar(f) = n > 0$ and $t_1, \ldots, t_n \in \mathbf{T}$ then $f(t_1, \ldots, t_n) \in \mathbf{T}$.

A *ground term* is a term built over only the two last items of the previous definition. The *Herbrand universe* is the set of all ground terms. Set $\mathbf{A}$ denotes the set of *atoms* defined as follows:

- if $a \in \mathcal{PS}$ with $ar(a) = 0$ then $a \in \mathbf{A}$,
- if $p \in \mathcal{PS}$ with $ar(p) = n > 0$ and $t_1, \ldots, t_n \in \mathbf{T}$ then $p(t_1, \ldots, t_n) \in \mathbf{A}$.

A *ground atom* is an atom built over only ground terms. The *Herbrand base* denoted $\mathcal{A}$ is the set of all ground atoms.

A *normal logic program* (or simply *program*) is a set of *rules* like

$$c \leftarrow a_1,\ \ldots,\ a_n,\ not\ b_1,\ \ldots,\ not\ b_m. \quad n \geq 0, m \geq 0 \tag{1}$$

where $c, a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms.

The intuitive meaning of such a rule is: "if all the a_i's are true and it may be assumed that all the b_j's are false then one can conclude that c is true". Symbol *not* denotes the *default negation*. A rule with no default negation is a *definite rule* otherwise its is a *nonmonotonic rule*. A program with only definite rules is a *definite logic program*. A program is a *propositional program* if all the predicate symbols are of arity 0.

For each program P, we consider that the set $\mathcal{CS}$ (resp. $\mathcal{FS}$ and $\mathcal{PS}$) consists of all constant (resp. function and predicate) symbols appearing in P. These sets determine the set of ground terms and the set of ground atoms of the program. A *substitution* for a rule $r \in P$ is a mapping from the set of variables from r to the set of ground terms of P. A ground rule r' is a *ground instance* of a rule r if there is a substitution θ for r such that $r' = \theta(r)$, the rule obtained by substituting every variable in r by the corresponding ground term in θ. The program P (with variables) has to be seen as an intensional version of the program $ground(P)$ defined as follows: given a rule r, $ground(r)$ is the set of all ground instances of r and then, $ground(P) = \bigcup_{r \in P} ground(r)$. Program $ground(P)$ may be considered as a propositional program. Let us note that the use of function symbols leads to an infinite Herbrand universe, this point will be discussed in Section 3.5.

Example 4. The program

$$P_4 = \left\{ \begin{array}{l} n(1).,\ n(2)., \\ a(X) \leftarrow n(X),\ not\ b(X)., \\ b(X) \leftarrow n(X),\ not\ a(X). \end{array} \right\}$$

is a shorthand for the program

$$ground(P_4) = \left\{ \begin{array}{l} n(1).,\ n(2)., \\ a(1) \leftarrow n(1),\ not\ b(1)., \\ b(1) \leftarrow n(1),\ not\ a(1)., \\ a(2) \leftarrow n(2),\ not\ b(2)., \\ b(2) \leftarrow n(2),\ not\ a(2). \end{array} \right\}$$

For a rule r (or by extension for a rule set), we define:

- $head(r) = c$ its *head*,
- $body^+(r) = \{a_1, \ldots, a_n\}$ its *positive body* and
- $body^-(r) = \{b_1, \ldots, b_m\}$ its *negative body*.

The immediate consequence operator for a definite logic program P is $T_P : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}}$ such that $T_P(X) = \{head(r) \mid r \in P,\ body^+(r) \subseteq X\}$. The *least Herbrand model* of P, denoted $Cn(P)$, is the smallest set of atoms closed under P, i.e., the smallest set X such that $T_P(X) \subseteq X$. It can be computed as the least fix-point of the consequence operator T_P.

The *reduct* P^X of a normal logic program P w.r.t. an atom set $X \subseteq \mathcal{A}$ is the definite logic program defined by:

$$P^X = \{head(r) \leftarrow body^+(r). \mid r \in P,\ body^-(r) \cap X = \emptyset\}$$

and it is the core of the definition of an *answer set*.

Definition 1. (Gelfond and Lifschitz 1988) Let P be a normal logic program and X an atom set. X is an answer set of P if and only if $X = Cn(P^X)$.

For instance, the propositional program $\{a \leftarrow not\ b.,\ b \leftarrow not\ a.\}$ has two answer sets $\{a\}$ and $\{b\}$.

Example 5. Taking again the program P_4, $ground(P_4)$ has four answer sets:
$\{a(1), a(2), n(1), n(2)\}$, $\{a(1), b(2), n(1), n(2)\}$,
$\{a(2), b(1), n(1), n(2)\}$, $\{b(1), b(2), n(1), n(2)\}$
that are thus the answer sets of P_4.

There is another definition of an anwer set for a normal logic program based on the notion of *generating rules* which are the rules participating to the construction of the answer set. These rules are important in our approach because they are exactly the rules fired in the `ASPeRiX` computation presented in the next section.

Definition 2. (Konczak et al. 2006) Let P be a normal logic program and X be an atom set. $GR_P(X)$, the set of *generating rules* of P, is defined as $GR_P(X) = \{r \in P \mid body^+(r) \subseteq X \text{ and } body^-(r) \cap X = \emptyset\}$.

Theorem 1. (Konczak et al. 2006) Let P be a normal logic program and X be an atom set. Then, X is an answer set of P if and only if $X = head(GR_P(X))$.

Special headless rules, called *constraints*, are admitted and considered equivalent to rules like ($bug \leftarrow \ldots,\ not\ bug.$) where bug is a new symbol appearing nowhere else. For instance, the program $\{a \leftarrow not\ b.,\ b \leftarrow not\ a., \leftarrow a.\}$ has one, and only one, answer set $\{b\}$ because constraint ($\leftarrow a.$) prevents a to be in an answer set.

When dealing with default negation, we call a *literal* an atom, a, or the negation of an atom, $not\ a$. A literal a is said to be *positive*, and $not\ a$ is said to be *negative*. The corresponding atom a of a literal l is denoted by $at(l)$. For a literal l where $at(l) = a$, let us denote $pred(l)$ the function such that $pred(not\ a) = pred(a) = p$ with p the predicate symbol of the atom a.

For purposes of knowledge representation, one may have to use conjointly strong negation (like $\neg a$) and default negation (like $not\ a$) inside a same program. This is possible in ASP by means of an *extended logic program* (Gelfond and Lifschitz 1991) in which rules are built with *classical* literals (i.e. an atom a or its strong negation $\neg a$) instead of atoms only. Semantics of extended logic programs distinguishes inconsistent answer sets from absence of answer set. But, if we are not interested in inconsistent answer sets, the semantics associated to an extended logic program is reducible to answer set semantics for a normal logic program using constraints by taking into account the following conventions:

- every classical literal $\neg x$ is encoded by the atom nx,
- for every atom x, the constraint ($\leftarrow x,\ nx.$) is added.

By this way, only consistent answer sets are kept. In this article, we do not focus on strong negation and literal will never stand for classical literal.

Let us note that one can also use some particular atoms for (in)equalities and simple arithmetic calculus on (positive and negative) integers. Arithmetic operations are treated as a functional arithmetic and comparison relations are treated as built-in predicates.

Finally, a program P is said to be *stratified* iff there is a mapping $strat$ from $\mathcal{PS}$ to $\mathbb{N}$ such that, for each ground rule like (1), the two following conditions hold:

- $strat(pred(c)) \geq str(pred(a_i))$ for all $i \in [1..n]$
- $strat(pred(c)) > str(pred(b_j))$ for all $j \in [1..m]$

3 A First Order Forward Chaining Approach for Answer Set Computing

3.1 `ASPeRiX` Computation

In this section, a characterization of answer sets for first-order normal logic programs, based on a concept of *`ASPeRiX` computation*, is presented. This concept is itself based on an abstract notion of *computation* for ground programs proposed in (Liu et al. 2010). This computation fundamentally uses a forward chaining of rules. It builds incrementally the answer set of the program and does not require the whole set of ground atoms from the beginning of the process. So, it is well suited to deal directly with first order rules by instantiating them during the computation.

The only syntactic restriction required by this methodology is that every rule of a

program must be *safe*. That is, all variables occurring in the head and all variables occurring in the negative body of a rule occur also in its positive body. Note that this condition is already required by all standard evaluation procedures. Moreover, every constraint (i.e. headless rule) is considered given with the particular head $\bot$ and is also safe. For the moment we do not consider function symbols but their use will be discussed in Section 3.5.

An *ASPeRiX computation* is defined as a process on a computation state based on a *partial interpretation* which is defined as follows.

Definition 3. A *partial interpretation* for a program P is a pair $\langle IN, OUT\rangle$ of disjoint atom sets included in the Herbrand base of P.

Intuitively, all atoms in IN belong to a search answer set and all atoms in OUT do not.

The notion of partial interpretation defines different status for rules.

Definition 4. Let r be a ground rule and $I = \langle IN, OUT\rangle$ be a partial interpretation.

- r is *supported* w.r.t. I when $body^+(r) \subseteq IN$,
- r is *blocked* w.r.t. I when $body^-(r) \cap IN \neq \emptyset$,
- r is *unblocked* w.r.t. I when $body^-(r) \subseteq OUT$,
- r is *applicable* w.r.t. I when r is supported and not blocked.[1]

An ASPeRiX computation is a forward chaining process that instantiates and fires one unique rule at each iteration according to two kinds of inference: a monotonic step of *propagation* and a nonmonotonic step of *choice*. To fire a rule means to add the head of the rule in the set IN.

Definition 5. Let P be a set of first order rules, I be a partial interpretation and R be a set of ground rules.

- $\Delta_{pro}(P, I, R)$ is the set of all supported definite rules and supported unblocked nonmonotonic rules from $ground(P) \setminus R$.
- $\Delta_{cho}(P, I, R)$ is the set of all applicable nonmonotonic rules from $ground(P) \setminus R$.

It is important to notice that the two sets defined above, like the set $ground(P)$, do not need to be explicitly computed. It is in accordance with the principal aim of this work that is to avoid their extensive construction. When necessary, a first-order rule of P can be selected and grounded with propositional atoms occurring in IN and OUT in order to define a new (not already occurring in R) fully ground rule member of Δ_{pro} or Δ_{cho}. Because of the safety constraint on rules this full grounding is always possible. These mechanisms are specified in more details in Subsection 3.3. The sets Δ_{pro} and Δ_{cho} are used in the following definition of an ASPeRiX computation. Specific case of constraints (rules with $\bot$ as head) is treated by adding $\bot$ into OUT set. By this way, if a constraint is fired (violated), $\bot$ should be added into IN and thus, $\langle IN, OUT\rangle$ would not be a partial interpretation.

[1] The negation of blocked, *not blocked*, is different from *unblocked*.

Definition 6. Let P be a first order normal logic program. An *ASPeRiX computation* for P is a sequence $\langle R_i, I_i\rangle_{i=0}^{\infty}$ of ground rule sets R_i and partial interpretations $I_i = \langle IN_i, OUT_i\rangle$ that satisfies the following conditions:

- $R_0 = \emptyset$ and $I_0 = \langle \emptyset, \{\bot\}\rangle$,

  (Propagation) $R_i = R_{i-1} \cup \{r_i\}$ with $r_i \in \Delta_{pro}(P, I_{i-1}, R_{i-1})$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1}\rangle$
  or (Rule choice) $\Delta_{pro}(P, I_{i-1}, R_{i-1}) = \emptyset$,
  $R_i = R_{i-1} \cup \{r_i\}$ with $r_i \in \Delta_{cho}(P, I_{i-1}, R_{i-1})$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1} \cup body^-(r_i)\rangle$
  or (Stability) $R_i = R_{i-1}$ and $I_i = I_{i-1}$,

- (Convergence) $\exists i \geq 0,\ \Delta_{cho}(P, I_i, R_i) = \emptyset$.

The computation is said to converge to the set $IN_\infty = \bigcup_{i=0}^{\infty} IN_i$.

Example 6. Let P_6 be the following program:

$$\left\{ \begin{array}{l} n(1). \\ n(X+1) \leftarrow n(X), (X+1) <= 2. \\ a(X) \leftarrow n(X), not\ b(X), not\ b(X+1). \\ b(X) \leftarrow n(X), not\ a(X). \\ c(X) \leftarrow n(X), not\ b(X+1). \end{array} \right\}$$

The following sequence is an `ASPeRiX` computation for P_6:

$$\begin{array}{ll} I_0 & = \langle \emptyset, \{\bot\}\rangle \\ & \\ r_1 & = n(1). \in \Delta_{pro}(P_6, I_0, \emptyset) \\ I_1 & = \langle \{n(1)\}, \{\bot\}\rangle \\ & \\ r_2 & = n(2) \leftarrow n(1). \in \Delta_{pro}(P_6, I_1, \{r_1\}) \\ I_2 & = \langle \{n(1), n(2)\}, \{\bot\}\rangle \\ & \\ & \Delta_{pro}(P_6, I_2, \{r_1, r_2\}) = \emptyset \\ r_3 & = a(1) \leftarrow n(1), not\ b(1), not\ b(2). \in \Delta_{cho}(P_6, I_2, \{r_1, r_2\}) \\ I_3 & = \langle \{n(1), n(2), a(1)\}, \{\bot, b(1), b(2)\}\rangle \\ & \\ r_4 & = c(1) \leftarrow n(1), not\ b(2). \in \Delta_{pro}(P_6, I_3, \{r_1, r_2, r_3\}) \\ I_4 & = \langle \{n(1), n(2), a(1), c(1)\}, \{\bot, b(1), b(2)\}\rangle \\ & \\ & \Delta_{pro}(P_6, I_4, \{r_1, r_2, r_3, r_4\}) = \emptyset \\ r_5 & = a(2) \leftarrow n(2), not\ b(2), not\ b(3). \in \Delta_{cho}(P_6, I_4, \{r_1, r_2, r_3, r_4\}) \\ I_5 & = \langle \{n(1), n(2), a(1), c(1), a(2)\}, \{\bot, b(1), b(2), b(3)\}\rangle \end{array}$$

$$
\begin{aligned}
r_6 &= c(2) \leftarrow n(2), not\ b(3). \in \Delta_{pro}(P_6, I_5, \{r_1, r_2, r_3, r_4, r_5\}) \\
I_6 &= \langle \{n(1), n(2), a(1), c(1), a(2), c(2)\}, \{\bot, b(1), b(2), b(3)\} \rangle \\
& \\
& \Delta_{pro}(P_6, I_6, \{r_1, r_2, r_3, r_4, r_5, r_6\}) = \emptyset \\
& \Delta_{cho}(P_6, I_6, \{r_1, r_2, r_3, r_4, r_5, r_6\}) = \emptyset \\
I_7 &= I_6
\end{aligned}
$$

The previous ASPeRiX computation converges to the set $\{n(1), n(2), a(1), c(1), a(2), c(2)\}$ which is an answer set for P_6.

The following theorem establishes a connection between the results of any ASPeRiX computation and the answer sets of a normal logic program.

Theorem 2. Let P be a normal logic program and X be an atom set. Then, X is an answer set of P if and only if there is an ASPeRiX computation $\langle R_i, I_i \rangle_{i=0}^{\infty}$, $I_i = \langle IN_i, OUT_i \rangle$, for P such that $IN_\infty = X$.

Let us note that in order to respect the revision principle of an ASPeRiX computation each sequence of partial interpretations must be generated by using the propagation inference based on rules from Δ_{pro} as long as possible before using the choice based on Δ_{cho} in order to fire a nonmonotonic rule. Then, because of the non determinism of the selection of rules from Δ_{cho}, the natural implementation of this approach leads to a usual search tree where, at each node, one has to decide whether or not to fire a rule chosen in Δ_{cho}. Persistence of applicability of the nonmonotonic rule chosen to be fired is ensured by adding to OUT all ground atoms from its negative body. On the other branch, where the rule is not fired, the translation of its negative body into a new constraint ensures that it becomes impossible to find later an answer set in which this rule is not blocked.

Propagation can be improved by using "must-be-true"[2] atoms: atoms which have to be in the answer set to avoid a contradiction or, in other words, atoms already determined to be in IN but which are not yet be proved to be in.

Example 7. Let $(\bot \leftarrow not\ b.)$ be a constraint whose body contains only one literal $not\ b$ with $b \notin IN \cup OUT$. In order to have an answer set, b must be in IN so that the constraint is not applicable but b is not yet proved (it is not the head of a fired rule). Thus, one can only conclude that b must be true.

Must-be-true atoms can be used during the propagation step in order to reduce the search space.

Example 8. Let $(c \leftarrow a, b.)$ be a rule with $a \in IN$ and $b \notin IN$ but b has been determined to be a must-be-true atom. The rule may be fired during the propagation step but one can only conclude that the rule head c must be true (because b is not yet proved).

Must-be-true atoms can also be used to reduce the size of Δ_{cho}, the set of nonmonotonic rules that can be chosen to be fired.

[2] The term "must be true" is first used in (Faber et al. 1999).

Example 9. Let $(c \leftarrow a, not\ d.)$ be a rule with $a \in IN$ and $d \notin IN$ but d has been determined to be a must-be-true atom. The rule may already be considered to be blocked, even if d is not yet proved, and thus may be excluded from Δ_{cho}.

Note that must-be-true atoms are first used to improve propagation and choice but have to be proved later, otherwise the computation can not lead to an answer set.

Notions of partial interpretation, rule status and `ASPeRiX` computation can be modified in order to consider these new elements.

Definition 7. Let P be a logic program. A *mbt partial interpretation* for P is a triplet $\langle IN, MBT, OUT\rangle$ of disjoint atom sets included in the Herbrand base of P.

Definition 8. Let r be a ground rule and $I = \langle IN, MBT, OUT\rangle$ be a mbt partial interpretation.

- r is *supported* w.r.t. I when $body^+(r) \subseteq IN$,
- r is *weakly supported* w.r.t. I when $body^+(r) \subseteq (IN \cup MBT)$
- r is *blocked* w.r.t. I when $body^-(r) \cap (IN \cup MBT) \neq \emptyset$,
- r is *unblocked* w.r.t. I when $body^-(r) \subseteq OUT$,
- r is *applicable* w.r.t. I when r is supported and not blocked.

Propagation is extended by Mbt-propagation: if some rule is weakly supported and unblocked w.r.t. mbt partial interpretation $\langle IN, MBT, OUT\rangle$ (but is not supported, i.e., does not belong to Δ_{pro}), then the head of the rule can be added in MBT set. And Δ_{cho}, the set of rules that can be chosen, is restricted to the rules that are not blocked w.r.t. mbt partial interpretation.

Definition 9. Let P be a set of first order rules, $I = \langle IN, MBT, OUT\rangle$ be a mbt partial interpretation and R be a set of ground rules.

- $\Delta_{pro}(P, I, R) = \{r \in ground(P) \setminus R \mid body^+(r) \subseteq IN \text{ and } body^-(r) \subseteq OUT\}$
- $\Delta_{pro\_mbt}(P, I, R) = \{r \in ground(P) \setminus R \mid body^+(r) \subseteq IN \cup MBT, body^+(r) \not\subseteq IN \text{ and } body^-(r) \subseteq OUT\}$
- $\Delta_{cho\_mbt}(P, I, R) = \{r \in ground(P) \setminus R \mid body^+(r) \subseteq IN, \text{ and } body^-(r) \cap (IN \cup MBT) = \emptyset\}$

A mbt `ASPeRiX` computation is an `ASPeRiX` computation with this additional kind of propagation and with the possibility to block a rule from Δ_{cho_mbt} instead of firing it ("Rule exclusion"). To block a rule is to add a constraint with the negative literals of the rule body. If there is only one literal in the negative body, this constraint can be expressed by adding an atom in MBT set (see Example 7). These possibilities restrict rule choice in Δ_{cho_mbt} and thus forbid some computations: if a rule r is blocked, computation can only converge to an answer set whose generating rules do not contain r. Note that Convergence principle impose that, at the end of a computation, no constraint is applicable and each atom from MBT set has been proved (i.e., was moved from MBT to IN set).

Definition 10. Let P be a first order normal logic program. A *mbt* ***ASPeRiX*** *computation* for P is a sequence $\langle K_i, R_i, I_i\rangle_{i=0}^{\infty}$ of ground rule sets K_i and R_i and mbt partial interpretations $I_i = \langle IN_i, MBT_i, OUT_i\rangle$ that satisfies the following conditions:

- $K_0 = \emptyset$, $R_0 = \emptyset$ and $I_0 = \langle \emptyset, \emptyset, \{\bot\}\rangle$,
- (Revision) $\forall i \geq 1$,

  (Propagation) $K_i = K_{i-1}$,
  $R_i = R_{i-1} \cup \{r_i\}$ with $r_i \in \Delta_{pro}(P, I_{i-1}, R_{i-1})$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, MBT_{i-1} \setminus \{head(r_i)\}, OUT_{i-1}\rangle$

  or (Mbt-propagation) $K_i = K_{i-1}$, $R_i = R_{i-1}$,
  and $I_i = \langle IN_{i-1}, MBT_{i-1} \cup \{head(r_i)\}, OUT_{i-1}\rangle$
  with $r_i \in \Delta_{pro\_mbt}(P, I_{i-1}, R_{i-1})$

  or (Rule choice) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}) = \emptyset$,
  $\Delta_{pro\_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}) = \emptyset$,
  $K_i = K_{i-1}$,
  $R_i = R_{i-1} \cup \{r_i\}$ with $r_i \in \Delta_{cho\_mbt}(P, I_{i-1}, R_{i-1})$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, MBT_{i-1} \setminus \{head(r_i)\}, OUT_{i-1} \cup body^-(r_i)\rangle$

  or (Rule exclusion) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}) = \emptyset$,
  $\Delta_{pro\_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}) = \emptyset$,

    $K_i = K_{i-1}$, $R_i = R_{i-1}$
    and $I_i = \langle IN_{i-1}, MBT_{i-1} \cup body^-(r_i), OUT_{i-1}\rangle$
    with $r_i \in \Delta_{cho\_mbt}(P, I_{i-1}, R_{i-1})$ and $|body^-(r_i)| = 1$

    or $K_i = K_{i-1} \cup \{\bot \leftarrow \cup_{b \in body^-(r_i)} not\ b.\}$, $R_i = R_{i-1}$ and $I_i = I_{i-1}$
    with $r_i \in \Delta_{cho\_mbt}(P, I_{i-1}, R_{i-1})$ and $|body^-(r_i)| > 1$

  or (Stability) $K_i = K_{i-1}$, $R_i = R_{i-1}$ and $I_i = I_{i-1}$,
- (Convergence) $\exists i \geq 0$, $\Delta_{cho\_mbt}(P \cup K_i, I_i, R_i) = \emptyset$ and $MBT_i = \emptyset$.

Mbt `ASPeRiX` computations characterize answer sets of a normal logic program. Completeness and correctness are established by the following theorem.

Theorem 3. Let P be a normal logic program and X be an atom set. Then, X is an answer set of P if and only if there is a mbt `ASPeRiX` computation $\langle K_i, R_i, I_i\rangle_{i=0}^{\infty}$, $I_i = \langle IN_i, MBT_i, OUT_i\rangle$, for P such that $IN_\infty = X$.

Note that computations model only successfull branches of a search tree. On the other hand, must-be-true atoms and rules blocking enable to prune failed branches of the tree and to reduce non determinism of the search by restricting the possible choices for the oracle (because some rules are explicitly excluded, and others are blocked by must-be-true atoms). So, these new elements do not improve the number of steps of a computation but they improve the number of steps needed to find a computation when there is no oracle to guide the search and, then, they make easier the search of answer sets.

3.2 ASPeRiX Main Algorithm

Now, we are interested in the practical computation of an answer set. The ASPeRiX algorithm, following the principle of mbt ASPeRiX computation seen in section 3.1, is based on the construction of three disjoint atom sets IN, MBT and OUT during the search for an answer set. It alternates two steps. On the one hand, a propagation step which instantiates all supported and unblocked rules which may be built from IN, MBT and OUT and fires them, i.e. adds their head in IN (or MBT). On the other hand, a choice step which forces or prohibits a nonmonotonic instantiated applicable rule to be fired during the next propagation step.

In order to treat the information more efficiently, the rules of a program P are ordered following the *strongly connected components* (SCC) of the dependency graph of P: the nodes of the dependency graph of a program P are its predicate symbols and the arcs are defined by $\{(p,q)|\exists r \in P,\ p = pred(head(r)),\ q \in pred(body^+(r) \cup body^-(r))\}$. The strongly connected components $\{C_1, ..., C_n\}$ are ordered in such a way that if $i < j$ then no node (i.e. predicate symbol) of C_i depends of a node of C_j. A rule is said to belong to a SCC C if the predicate symbol of its head is in the component C. Note that constraints are not really concerned by ordering of rules but, for standardizing notations, constraints are considered to belong to a unique component whose number is greater than that of the last SCC, i.e., if C_n is the last SCC then constraints are considered to belong to C_{n+1}.

Example 10. (Example 6 continued)

The strongly connected components (SCC) of the graph of the program P_6 are $C_1 = \{n\}$, $C_2 = \{a, b\}$ and $C_3 = \{c\}$ (Figure 2).

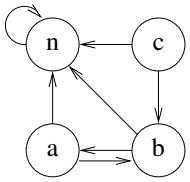

Fig. 2. Dependency graph of P_6

The ASPeRiX algorithm solves one by one the SCC $\{C_1, ..., C_n\}$ of a program P by starting by C_1. When no propagation nor choice can no longer be done on the current SCC, the predicate symbols of the SCC are said to be *solved* and the SCC too. It means that nothing can be deduced anymore for those predicate symbols. The atoms which are instances of the predicate symbols of the current SCC and which are not in IN are implicitly added to OUT. Note that they are not explicitly added to OUT because ground instances of a predicate are not known (not computed): they could be infinite and, even if finite, to compute and store them is useless.

Rules of the program are instantiated on the fly during the propagation phase and the choice step. Hence the propositional program $ground(P)$ which contains all the instantiated rules of the program is never really computed. The propagation step

and the choice step are realized in the ASPeRiX algorithm thanks to the functions γ_{pro} and γ_{cho} (which are selection functions in $\Delta_{pro} \cup \Delta_{pro\_mbt}$ and Δ_{cho_mbt} sets used in the mbt ASPeRiX computation of Subsection 3.1). The γ_{pro} function searches for a weakly supported unblocked rule amongst the current and next non-solved SCC. So propagation operates on several components: each rule is fired as soon as possible to quickly detect a possible conflict. Rules instantiated by γ_{pro} are stored in a set $Subst = \{subst(r_i), ..., subst(r_n)\}$ during the answer set search to mark the substitution of rules that have already been used. For each first-order rule r_i, $subst(r_i)$ denotes the set of all substitutions θ such that $\theta(r_i)$ has already been fired. And $subst\_rule(r_i) = \bigcup_{\theta \in subst(r_i)} \{\theta(r_i)\}$ is the set of instantiated rules obtained thanks to substitutions $subst(r_i)$. The notation is extended to a set R of first-order rules: $subst\_rule(R) = \bigcup_{r_i \in R} subst\_rule(r_i)$. The γ_{cho} function chooses an applicable rule in the current SCC when nothing can no longer be propagated. So choice, unlike propagation, operates only on the current component. This strategy, consisting of solving the SCC one after another, makes it possible to solve efficiently stratified programs (or some stratified parts of programs).

Functions γ_{pro} and γ_{cho} are specified in more details in Subsection 3.3 and are defined informally as follows:

- $\gamma_{pro}(P, S, S', T, SCC, Subst)$: nondeterministic function which selects a rule (or a constraint) r belonging to a SCC greater or equal to the current SCC in the dependency graph of a program P such that $body^+(r) \subseteq S \cup S'$, $body^-(r) \subseteq T$ and $r \in ground(P) \backslash subst\_rule(P)$ or returns **NULL** if no such a rule exists.
- $\gamma_{cho}(P, S, S', T, SCC, Subst)$: nondeterministic function which selects a rule r belonging to the current SCC in the dependency graph of a program P such that $body^+(r) \subseteq S$, $body^-(r) \cap (S \cup S') = \emptyset$ and $r \in ground(P) \backslash subst\_rule(P)$ or returns **NULL** if no such a rule exists.

The function *solve* of Algorithm 1 specifies the algorithm of the search of one answer set for a program P. The set P_K is the set of constraints (rules with the symbol $\bot$ at their heads) of P and P_R the other rules. By default, $\bot$ is into the set OUT. Then, if a constraint is fired, a contradiction is immediately detected since $\bot$ is added into the set IN and the sets IN and OUT are no longer disjoint. The algorithm of the function *solve* computes one answer set (or none if the program is incoherent) thanks to the variable *stop* which stops the search once an answer set has been found. This algorithm may be easily extended to compute an arbitrary number of answer sets. Let us note that, for sake of simplicity, the function *solve* will return either a set (when there is an answer set) or the constant *no_answer_set* if there is no answer set.

The main parts of the function *solve* are now described. Initially, $IN = \emptyset$, $MBT = \emptyset$, $OUT = \{\bot\}$, SCC is the index of the first SCC and $Subst = \emptyset$.

The propagation phase successively fires each weakly supported and unblocked instantiated rule r_0. At each step, the call $\gamma_{pro}(P_R \cup P_K, IN, MBT, OUT, SCC, Subst)$ selects and instantiates a unique unblocked rule r_0 such that $body^+(r_0) \subseteq IN \cup MBT$ (line 4). If such a rule exists, its head atom $head(r_0)$ must belong to the answer set. This head atom is added into the set IN (line 9) if the positive body

Algorithm 1: *solve*

```
1  Function solve(P_R, P_K, IN, MBT, OUT, SCC, Subst);
2  // search of one answer set for a program P = P_R ∪ P_K
3  repeat//Propagation phase
4      r_0 ← γ_pro(P_R ∪ P_K, IN, MBT, OUT, SCC, Subst);
5      if r_0 ≠ NULL then
6          if (body^+(r_0) ∩ MBT) ≠ ∅ then
7              MBT ← MBT ∪ {head(r_0)};
8          else
9              IN ← IN ∪ {head(r_0)};
10             if (head(r_0) ∈ MBT) then
11                 MBT ← MBT\{head(r_0)};
12 until r_0 = NULL;
13 if ((IN ∪ MBT) ∩ OUT ≠ ∅) then //Contradiction detected
14     return no_answer_set;
15 else
16     r_0 ← γ_cho(P_R, IN, MBT, OUT, SCC, Subst);
17     if r_0 ≠ NULL then //Choice point
18         stop ← solve(P_R, P_K, IN, MBT, OUT ∪ body^-(r_0), SCC, Subst);
19         if stop = no_answer_set then
20             atoms ← {a|a ∈ body^-(r_0), pred(a) ∈ pred(SCC)};
21             if (|atoms| = 1) then
22                 MBT ← MBT ∪ atoms;
23             else
24                 P_K ← P_K ∪ {⊥ ← ∪_{a_i∈atoms} not a_i};
25             stop ← solve(P_R, P_K, IN, MBT, OUT, SCC, Subst);
26         return stop ;
27     else // The SCC is solved
28         if pred(MBT) ∩ pred(SCC) = ∅ then
29             if ¬last(SCC) then
30                 return solve(P_R, P_K, IN, MBT, OUT, SCC + 1, Subst);
31             else
32                 if γ_check(P_K, IN, MBT, OUT, SCC) then // a constraint is
                   violated
33                     return no_answer_set;
34                 else // An answer set has been found
35                     return IN;
36         else // a MBT atom can not be proved
37             return no_answer_set;
```

of the rule is included in the set IN or added into the set MBT (line 7) otherwise since at least one atom a of the positive body of the rule has not yet proved its membership to the set IN ($a \in MBT$ but $a \notin IN$). Moreover, a head atom which is added into IN must be deleted from MBT since a proof of its membership to the answer set has been found (line 10). When there is no more unblocked rule r_0 such that $body^+(r_0) \subseteq IN \cup MBT$, $(IN \cup MBT) \cap OUT = \emptyset$ is checked in order to detect a contradiction (line 13). If no contradiction is detected, the algorithm begins the choice step.

The choice point forces or forbids a nonmonotonic applicable rule to be fired. The call $\gamma_{cho}(P_R,\ IN,\ MBT,\ OUT,\ SCC,\ Subst)$ selects and instantiates a unique applicable rule of P_R whose head belongs to the current SCC (line 16). If such a rule exists, r_0 is forced to be unblocked and then will be fired during the next propagation phase: its negative body is added to the OUT set and function *solve* is recursively called with its new parameters (line 18). If a recursive call to the function *solve* detects a contradiction, the algorithm backtracks on the last choice point on the rule r_0 which has been forced to be fired and blocks it (lines 19-25): if a is the only atom of the negative body of r_0 then a is added to the set MBT (line 22) else a constraint including all the atoms of the negative body of r_0 is added to the program (line 24). More precisely, the only atoms of the negative body that are considered are those with a predicate symbol belonging to the current SCC because atoms from a lower SCC are already solved, i.e. they are in IN or OUT. When there is no more choice point, the current SCC is solved (line 27) but it must be checked that no atom of the MBT set has a predicate symbol in the current SCC (line 28). If such an atom exists, MBT and OUT sets are not disjoint. Indeed, if a SCC is solved, atoms which are instances of predicate symbols of the SCC and which are not in IN are implicitly added to OUT. Then if a MBT atom is an instance of a predicate symbol of the current SCC, a failure is observed and the backtrack process continues (line 36). If the last SCC is solved, the set IN represents an answer set of P if no constraint is applicable. This test is realized thanks to the nondeterministic function γ_{check} (line 32) which is specified in more details in Subsection 3.3 and is defined informally as follows:

$\gamma_{check}(P, S, S', T, SCC)$: function which checks if there is any constraint c such that $body^+(c) \subseteq S$, $body^-(c) \cap S = \emptyset$ and $c \in ground(P)$.

Example 11. The execution of the `ASPeRiX` algorithm for program P_6 of Example 6 is represented by a tree in Figure 3. At the beginning $IN = \emptyset$, $MBT = \emptyset$, $OUT = \{\bot\}$ and the current SCC is the component $C_1 = \{n\}$. After the first propagation, $n(1)$ and $n(2)$ are in IN thanks to the two rules $(n(1).)$ and $(n(2) \leftarrow n(1), (1+1) <= 2.)$. No choice point exists and the first SCC is solved since the MBT set is empty. The component $C_2 = \{a, b\}$ becomes the current SCC.

The first choice is realized on the current SCC (choice point $CP1$): the rule $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ becomes unblocked by adding $b(1)$ and $b(2)$ into the set OUT (left branch after choice point $CP1$). A new propagation phase shows that $a(1)$ and $c(1)$ are in IN since $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ and $(c(1) \leftarrow n(1), not\ b(2).)$ can be fired. Then, a new choice is realized (choice point $CP2$) and

the rule $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ is forced to be unblocked (left branch after choice point $CP2$). The atom $b(3)$ is added into the set OUT. A new propagation phase shows that $a(2)$ and $c(2)$ are in IN since $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ and $(c(2) \leftarrow n(2), not\ b(3).)$ can be fired. The second SCC is solved since no other rule is applicable and the MBT set is still empty. In the same way, no propagation nor choice point is possible in the SCC $C_3 = \{c\}$. Since no constraint is applicable, a first answer set is obtained: $\{a(1), a(2), c(1), c(2), n(1), n(2)\}$.

If another answer set is wished, the algorithm backtracks to the last choice point

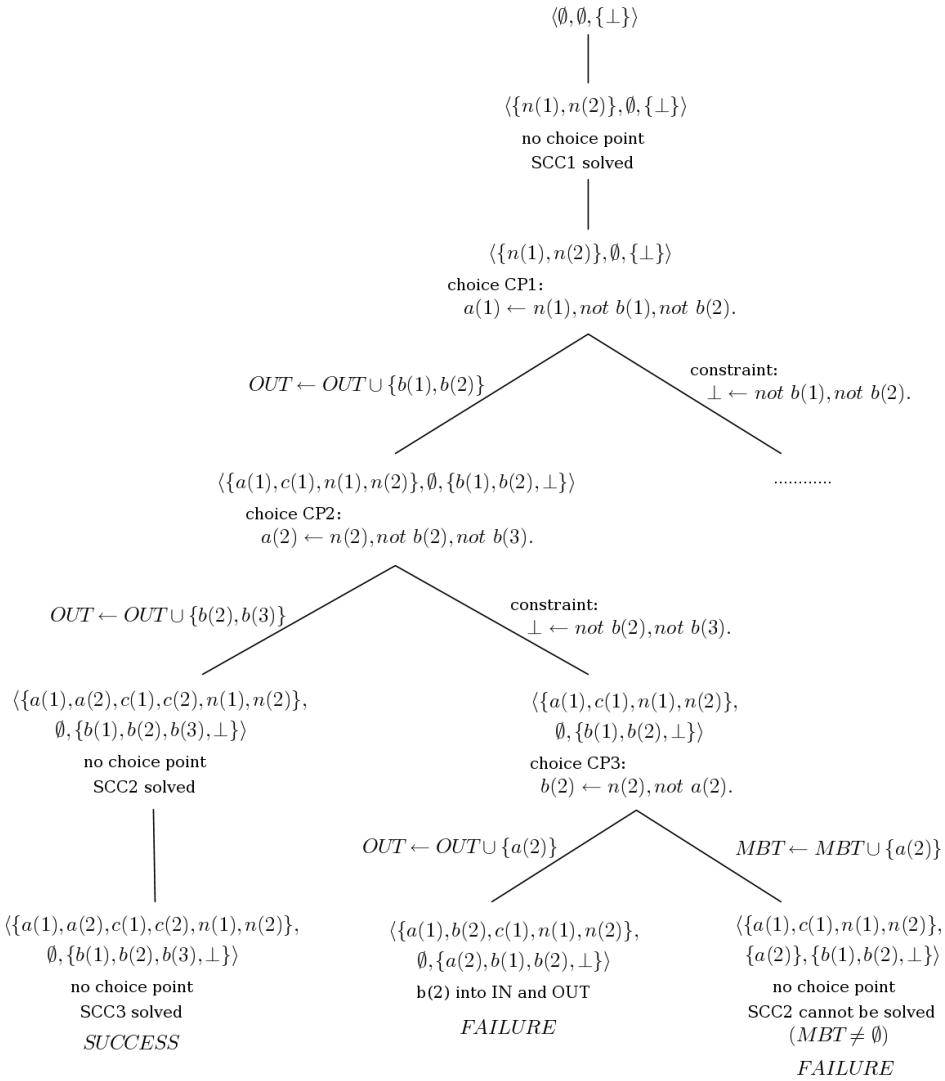

Fig. 3. The tree-shaped execution of the answer sets of program P_6.

on $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ of the component C_2 and blocks the rule (right branch after choice point $CP2$) by adding a constraint $(\bot \leftarrow not\ b(2), not\ b(3).)$ into P_K. A new choice is realized (choice point $CP3$) and the rule $(b(2) \leftarrow n(2), not\ a(2).)$ is forced to be unblocked (left branch after choice point $CP3$) by adding $a(2)$ into the OUT set. During the propagation step, $b(2)$ is added into the IN set since $(b(2) \leftarrow n(2), not\ a(2).)$ is fired. The atom $b(2)$ is then simultaneously in the sets IN and OUT which leads to a contradiction.

The algorithm backtracks to the choice point $(b(2) \leftarrow n(2), not\ a(2).)$ of the component C_2 (choice point $CP3$) and the rule is blocked by adding the atom $a(2)$ into the MBT set (right branch after choice point $CP3$). Since there is no more possible choice and the MBT set contains an atom whose predicate symbol is in the current SCC, this atom cannot be proved and this leads to a failure. The algorithm backtracks to the first choice point on $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ of the component C_2 (choice point $CP1$) and blocks the rule and searches for a new possible answer set (right branch after choice point $CP1$). The process keeps going until the whole tree is computed when all the answer sets are required. Let us note that when dealing with the computation of one answer set like explained in the algorithm, only the first branch is considered.

3.3 Functions γ

Functions γ have a crucial role in two important steps of the search of an answer set. The function γ_{pro} is called during the propagation step in order to choose the rules to fire and then to add their heads into IN (or MBT). The function γ_{cho} is called during the choice step in order to force or to forbid a rule to be fired during the next propagation step. The function γ_{check} is called during the verification step in order to verify that no constraint is applicable. Since the principle of the solver `ASPeRiX` is to instantiate the rules on the fly during the search of an answer set, functions γ need to call a function *instantiateRule* which searches for a substitution for the atoms of a rule. This function is detailed in its own Subsection 3.4.

Function γ_{pro}. The function γ_{pro} searches for a rule to fire w.r.t. IN, MBT and OUT sets. This function computes a complete instantiation of a rule such that the positive body is in $IN \cup MBT$ and the negative body is in OUT. The rule to instantiate is chosen amongst a set of rules R consisting of rules that could lead to new, unprocessed instances. These rules are those whose body contains some predicate symbol of what we call an *atom to propagate*. Atoms to propagate are atoms recently added into IN, MBT and OUT sets, and not yet used for propagation phase. Thereby, when an atom a is added into IN or MBT (resp. OUT) set, the rules containing $pred(a)$ in their positive body (resp. negative body) will be in the R set for the next call to γ_{pro} in order to propagate this atom, i.e. to use its presence in IN or MBT (resp. OUT) for building new instances of rules to be fired. During the first call of the function *solve*, atoms to propagate are the facts of the program, and the set R contains all the rules which have some predicate symbols of the facts in their positive body. During a call after a choice

point, atoms to propagate are those added into OUT during this choice point, and the set R contains all the rules which have some predicate symbols of these atoms in their negative body. During a call after the access to the next SCC, the predicate symbols of the current SCC are solved and then all instances of these predicate symbols that are not in IN are implicitly added into OUT. Atoms to propagate are all these instances determined to be false, and then the set R contains all the rules which have in their negative body some of these solved predicate symbols.

The Algorithm 2 of the function γ_{pro} chooses a first-order rule r amongst the set R (the first one, line 5) and tries to find a weakly supported unblocked instantiation of the rule. It calls the function *instantiateRule* which returns this next instantiation if any (line 6). If there is no more weakly supported unblocked instantiated rule which may be extracted from r (line 7), γ_{pro} deletes from R the rule r and treats the next rule. This process is repeated until a weakly supported unblocked rule is found or there is no more rule in R. When a rule allows a substitution (line 10), the latter is stored in *Subst* in order to find some others at the next call to γ_{pro}.

Algorithm 2: γ_{pro}

```
1  Function γ_pro(P, IN, MBT, OUT, SCC, Subst);
2  R ← Set of rules (including constraints) containing predicate symbols to
     propagate;
3  if R ≠ ∅ then
4      repeat
5          r ← first(R);
           /* Searching for an instantiation of the rule r with
              body⁺(r) ⊆ IN ∪ MBT and body⁻(r) ⊆ OUT                    */
6          θ ← instantiateRule(r, γ_pro, IN, MBT, OUT, subst(r));
7          if θ = NULL then
8              R ← R\{r};
9      until θ ≠ NULL or R = ∅;
10     if θ ≠ NULL then
           /* An unblocked weakly supported instantiated rule is found
              */
11         subst(r) ← subst(r) ∪ {θ};
12         return θ(r);
13     else
14         return NULL;
15 else
16     return NULL;
```

Example 12. Example 11 is taken again. An answer set is searched after the choice point on the rule $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ (choice point $CP1$): the atoms $b(1)$ and $b(2)$ are added into the set OUT in order to force the rule to be fired

(left branch after choice point $CP1$). During the propagation step, many calls to the function γ_{pro} are executed. During the first call the set R consists of all the rules containing in their negative body the predicate symbol b of the atoms $b(1)$ and $b(2)$ that must be propagated. This set R then contains the rules $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$ and $(c(X) \leftarrow n(X), not\ b(X+1).)$ Arbitrarily, the rule $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$ of the set R is chosen and a supported unblocked instantiation $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ is found. The function γ_{pro} returns the instantiation of the rule and the *solve* function adds $a(1)$ into IN. During the next call to γ_{pro}, the set R must contain, in addition to the previous rules, any rule containing in its positive body the predicate symbol a of the atom to be propagated $a(1)$ (since $a(1)$ has been added into IN). Since no rule respects this condition, the set R still contains only the two previously added rules. The function γ_{pro} searches for a new weakly supported unblocked instantiation of the rule $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$. No such instantiation is found and the rule is deleted from the set R. The function γ_{pro} searches for a new weakly supported unblocked instantiation of the rule $(c(X) \leftarrow n(X), not\ b(X+1).)$. The instantiation $(c(1) \leftarrow n(1), not\ b(2).)$ is then returned to the *solve* function which adds $c(1)$ into IN. Then during the next call to the function γ_{pro}, the set R must be updated with the rules containing in their positive body the predicate symbol c of the atom to propagate $c(1)$. As previously, no rule respects this condition and the set R still contains the only rule $(c(X) \leftarrow n(X), not\ b(X+1).)$. A new weakly supported unblocked instantiation is sought but this rule leads to a failure. The rule $(c(X) \leftarrow n(X), not\ b(X+1).)$ is then deleted from the set R which becomes empty. Then the function γ_{pro} returns the value **NULL** and the propagation step of the function *solve* stops.

Function γ_{cho}. The function γ_{cho} is executed when no rule can be fired anymore and there is some SCC to be solved. This function searches an applicable instantiated rule belonging to the current SCC. The Algorithm 3 of function γ_{cho} is similar to the algorithm of the function γ_{pro}. The function γ_{cho} searches for an applicable instantiated rule amongst a set R of rules which have in their negative body at least one predicate symbol from the current SCC (otherwise, if all predicate symbols from negative body belong to previous SCC, they are already solved and then the rule can be considered as a monotonic one and is only used for propagation). The function γ_{cho} chooses a rule in this set R before calling the function *instantiateRule* searching for the next applicable instantiation for the considered rule. In a similar way as the function γ_{pro}, the process is repeated until an applicable instantiated rule is found for a rule of R or there is no more rule in R.

Example 13. Example 12 is taken again. After the first SCC has been solved, a first choice is realized on the current SCC, $C_2 = \{a, b\}$, by the function γ_{cho}. The rules of this component which contains in their negative body at least one predicate symbol a or b of C_2 are added into the set R of the rules that may be chosen. Then, the rules $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$ and $(b(X) \leftarrow n(X), not\ a(X).)$ are in R. Arbitrarily, the function γ_{cho} searches for an applicable instantiation of the first rule

Algorithm 3: γ_{cho}

```
1  Function γ_cho(P, IN, MBT, OUT, SCC, Subst);
2  R ← Set of rule belonging to the current SCC such that the negative body
     contains at least a predicate symbol not solved;
3  if R ≠ ∅ then
4      repeat
5          r ← first(R);
           /* Searching for an instantiation of the rule r with
              body⁺(r) ⊆ IN and body⁻(r) ∩ (IN ∪ MBT) = ∅                  */
6          θ ← instantiateRule(r, γ_cho, IN, MBT, OUT, subst(r));
7          if θ = NULL then
8              R ← R\{r};
9      until θ ≠ NULL or R = ∅;
10     if θ ≠ NULL then
           /* An applicable instantiated rule is found                    */
11         subst(r) ← subst(r) ∪ {θ};
12         return θ(r);
13     else
14         return NULL;
15 else
16     return NULL;
```

of this set and a choice point on $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ is returned to the calling function *solve* (choice point $CP1$). After the propagation step, γ_{cho} searches for a new applicable instantiation of the rule $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$ and a choice point on $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ is returned to the calling function *solve* (choice point $CP2$). After a new propagation step, γ_{cho} searches in vain a new applicable instantiation of the rule $(a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).)$ This last rule is then deleted from the set R and γ_{cho} searches for an applicable instantiation of the rule $(b(X) \leftarrow n(X), not\ a(X).)$ which leads to a failure. The set R is now empty and the function γ_{cho} returns **NULL** to the calling function *solve* to mean that no other choice may be realized on the current SCC.

Function γ_{check}**.** The function γ_{check} is executed when no more choice point is possible for the last SCC. This function verifies that no constraint containing at least one predicate symbol of the last SCC is applicable in order to determine if the set IN is an answer set. The Algorithm 4 of the function γ_{check} is similar to the algorithm of the function γ_{cho}. The function γ_{check} searches for an applicable instantiated constraint amongst a set C of constraints whose negative body contains at least a not-solved predicate symbol of the last SCC. The function γ_{check} chooses a constraint in the set C and calls the function *instantiateRule* which searches for an applicable instantiated constraint. If no instantiated constraint is applicable,

the algorithm returns **false** and the set IN is an answer set of the program. If a constraint is applicable, the algorithm returns **true** which means there is a failure on the branch (the search of answer sets keeps going on other branches if any).

Algorithm 4: Function γ_{check}

```
1  Function γ_check(P, IN, MBT, OUT, SCC);
2  C ← Set of constraints such that the negative body contains at least a
     predicate symbol not solved;
3  if C ≠ ∅ then
4      repeat
5          c ← first(C);
           /* Searching for an instantiation of the constraint c such
              that body⁺(c) ⊆ IN and body⁻(c) ∩ IN = ∅                  */
6          θ ← instantiateRule(c, γ_check, IN, MBT, OUT, ∅);
7          if θ = NULL then
8              C ← C\{c};
9      until θ ≠ NULL or C = ∅;
10     if θ ≠ NULL then
           /* An applicable instantiated constraint is found           */
11         return true;
12     else
13         return false;
14 else
15     return false;
```

3.4 Rule Instantiation

In this section is described the process of instantiation of a rule. This process is a lazy one only called when needed. Since we only consider safe rules, the instantiation of a rule is in fact the instantiation of its positive body. In a forward chaining approach, the only rule instantiations of interest are those that lead to a not blocked supported rule or an unblocked weakly supported rule. Hence, the rule instantiation is mainly directed by the instantiated atoms already present in the sets IN and MBT.

The algorithm used in the ASPeRiX solver and described below is inspired by the previous work realized on the DLV grounder (Faber et al. 2012; Perri et al. 2007) which is based on the semi-naive evaluation technique of (Ullman 1989). The goal is to find a substitution for all the literals of the body of a rule r thanks to the atoms already in IN, MBT or OUT. To do this, a partial substitution θ is built as possible values are found for the variables of the literals of the body of the rule r. It is assumed that the literals $l_1, l_2, \ldots, l_n$ of the body of the rule r are ordered following a list $[l_1, l_2, \ldots, l_n]$: $firstLiteral(r)$ (resp. $lastLiteral(r)$) corresponds to l_1 (resp.

l_n) and $previousLiteral(r)$ (resp. $nextLiteral(r)$) corresponds to the literal which precedes (resp. follows) the literal under consideration in the list. The substitution calculus for a literal l of a rule r is realized thanks to the functions $firstMatch$ and $nextMatch$. These functions look for a substitution which has not already been computed, i.e. not leading to a substitution for r present in the set $subst(r)$ of all substitutions θ such that $\theta(r)$ has already been fired. If the literal l is positive, a substitution such that the substituted atom is in the set IN (or $IN \cup MBT$) is searched. If the literal is negative, (a) a substitution such that the substituted corresponding atom is in the set OUT is searched if the goal is an unblocked rule or (b) the non membership of the substituted atom to the set $IN \cup MBT$ is checked if the goal is a not blocked rule[3].

In the functions $firstMatch$ and $nextMatch$ which follow, the parameter γ shows if an unblocked weakly supported or not blocked supported rule is looked for.

- $firstMatch(l, \theta, \gamma, IN, MBT, OUT, subst)$ is a function which searches for the first possible substitution for a literal l w.r.t. the sets IN, MBT and OUT, selection criterion γ (unblocked weakly supported or applicable rule) and the current partial substitution θ. $firstMatch$ returns true and updates the partial substitution θ in case of success. Otherwise, the function returns false.
- $nextMatch(l, \theta, \gamma, IN, MBT, OUT, subst)$ is a function which searches for the next possible substitution for literal l given the already realized substitutions.

For a rule r, a *free variable* of a literal l is an occurrence of a variable X such that it is its first occurrence in the body of r when starting traversing the literal l. In other words, no other literal which precedes l in the body of r contains an occurrence of the variable X. During the instantiation of a rule, a possible substitution is sought for all the free variables of every traversed literal and the substitutions of the previously calculated variables are kept. If a literal has no free variable, the validity of the substitution w.r.t. the selection criterion γ is checked (i.e. the substituted corresponding atom $\theta(at(l))$ is in IN or $IN \cup MBT$ if $l \in body^+(r)$ and $\theta(at(l))$ is in OUT or $\theta(at(l))$ is not in IN if $l \in body^-(r)$).

Example 14. Let $(a(X, Y, Z) \leftarrow b(X, Y), c(X, Y), d(X, Z).)$ be a rule. The ordered list of the body of the rule is $[l_1 = b(X, Y), l_2 = c(X, Y), l_3 = d(X, Z)]$ with:

- $freeVariables(l_1) = \{X, Y\}$
- $freeVariables(l_2) = \emptyset$
- $freeVariables(l_3) = \{Z\}$.

Function *instantiateRule* of Algorithm 5 specifies the instantiation principles of a rule for constant sets IN, MBT and OUT. This function is initialized with the partial substitution θ which is the last found substitution (thanks to the function *lastSubstitution*) for the rule r if any (line 2). If it is the first attempt for the instantiation of this rule, θ is empty (line 3) and the function searches a first substitution

[3] In this case, the body of the rule is ordered in such a way that negative literals appear after the positive literals containing their variables.

Algorithm 5: *instantiateRule*

```
1  Function instantiateRule(r, γ, IN, MBT, OUT, subst);
2  θ ← lastSubstitution(r);
3  if θ = ∅ then
       /* Searching for the first possible substitution of first
          literal                                                  */
4      l ← firstLiteral(r);
5      matchFound ← firstMatch(l, θ, γ, IN, MBT, OUT, subst);
6  else
       /* Searching for the next possible substitution of last
          literal                                                  */
7      l ← lastLiteral(r);
8      θ ← θ\freeVariableSubstitutions(l);
9      matchFound ← nextMatch(l, θ, γ, IN, MBT, OUT, subst);
10 while true do
11     if matchFound then
12         if l ≠ lastLiteral(r) then
13             l ← nextLiteral(r);
14             matchfound ← firstMatch(l, θ, γ, IN, MBT, OUT, subst)
15         else
               /* A complete substitution is found                */
16             return θ;
17     else
           /* No substitution for literal l. Bactrack to previous
              literal (if any) to find its next possible substitution
              */
18         if l ≠ firstLiteral(r) then
19             l ← previousLiteral(r);
20             θ ← θ\freeVariableSubstitutions(l);
21             matchFound ← nextMatch(l, θ, γ, IN, MBT, OUT, subst);
22         else
23             return NULL;
```

for the first literal of the body of the rule r using the function $firstMatch$. Otherwise, a substitution for r has already been computed (line 6), the function searches a new possible substitution for the rule. For this, the function searches the next possible instance of the last literal of the rule r by deleting from θ the substitutions of the free variables of this literal (thanks to the function $freeVariableSubstitutions$) and by calling the function $nextMatch$. During the execution of the main loop, the function first checks if a substitution has been found for the current literal a (line 11). If it is the case, it searches a first substitution for the next literal of the rule body respecting the partial substitution θ. When all the atoms have been considered,

a complete substitution is found (line 15). The function returns this substitution. When the instantiation of a literal fails (i.e. there is no possible substitution for it), the function backtracks on the previous literal (line 18) and updates θ by deleting the substitutions of the free variables of this literal. Hence the function calls the function $nextMatch$ which searches the next possible instantiation for this literal. The instantiation of a rule r fails when no more substitution is possible for the first literal (line 23).

Actually, the instantiation algorithm of a rule is slightly more complicated than the Algorithm 5 since the atoms dynamically added into IN, MBT and OUT sets during the answer set computation, called *atoms to propagate*, have to be taken into account: if possible, each substitution has to be computed once and only once. Hence `ASPeRiX` uses a queue called $propagate\_IN$ (resp. $propagate\_MBT$ and $propagate\_OUT$) which contains the atoms to be added into the set IN (resp. MBT and OUT). When all the instances of a rule r, for given sets IN_0, MBT_0 and OUT_0, have been generated, an atom to propagate a_p whose predicate symbol p appears in the body of the rule r is extracted. Now $I_1 = \langle IN_1, MBT_1 OUT_1 \rangle$ denotes the mbt partial interpretation obtained by adding a_p into $I_0 = \langle IN_0, MBT_0, OUT_0 \rangle$. The body of the rule is ordered in such a way that the first literals are those whose predicate symbol is the one of the atom to propagate a_p (they are the literals that might unify with a_p). Then these literals whose predicate symbol is p are successively marked and placed at the beginning of the rule. The marked literal might only take the value of the atom to propagate a_p whereas the following (non marked) literals might take any values in I_1. Then, if the instantiation of the first literal fails, it is unmarked, the next literal of predicate p becomes the first literal of the rule body and is marked in turn, and the instantiation of the rule is started again. The unmarked literals might then take any values in I_0 (which excludes the values of a_p already used) while the marked literal can only take the value of the atom to propagate, and the non marked literals always take their values in I_1. If the instantiation of the first literal fails and there is no other literal to be marked, the instantiation of the rule fails.

Example 15. Let r_0 be a rule and IN_0, $propagate\_IN$ and IN_1 be sets of atoms defined as follow:

$$r_0 = a(X+Y) \leftarrow a(X), b(X,Y), a(Y).$$
$$IN_0 = \{b(1,1), b(1,2)\}$$
$$propagate\_IN = \{a(1)\}$$
$$IN_1 = \{b(1,1), b(1,2), a(1)\}$$

The atom $a(1)$ has to be propagated by instantiating the rule r_0. Table 1 shows the different steps of the instantiation. The literals to be marked (whose predicate symbol is a) of the body of the rule are $a(X)$ and $a(Y)$. These literals are placed at the beginning of the body of r_0 like this: $[l_1 = a(X), l_2 = a(Y), l_3 = b(X,Y)]$. In Table 1, for clarity, the sequence of literals of the rule body is not changed when the marked literal changes. But the marked literal (shown in bold) is processed first, which is the same. The first attempt for an instantiation begins and for the first time with atom to propagate $a(1)$. The literal $l_1 = a(X)$ is then marked and takes

$a(X+Y)$	$\leftarrow$	$a(X)$,	$a(Y)$,	$b(X,Y)$	
		*** first call to *instantiateRule* ***			
(1.1)		$\mathbf{a(1)}$	-	-	(l_1 marked)
(1.2)		$\mathbf{a(1)}$	$a(1)$	-	
(1.3)		$\mathbf{a(1)}$	$a(1)$	$b(1,1)$	$\Rightarrow$ complete instantiation
		*** second call to *instantiateRule* ***			
(2.1)		$\mathbf{a(1)}$	$a(1)$	NO	
(2.2)		$\mathbf{a(1)}$	NO	-	
(2.3)		**NO**	-	-	$\Rightarrow$ failure
(2.4)		-	$\mathbf{a(1)}$	-	(l_2 marked)
(2.5)		NO	$\mathbf{a(1)}$	-	$\Rightarrow$ failure

Table 1. *Decomposition of the instantiations of the rule r_0 for the atom to propagate $a(1)$ (Example 15)*

as unique value that of the atom to propagate $a(1)$ ((1.1) Table 1). Hence, value 1 is substituted to the variable X in θ. Then, the following literal in the body of the rule, $l_2 = a(Y)$, becomes the current literal and takes as value the first amongst those into IN_1 which is also $a(1)$. Hence, value 1 is substituted to the variable Y in θ ((1.2) Table 1). Then the last literal, $l_3 = b(X,Y)$, is reached. This literal has no free variable and the membership into IN_1 is simply checked for $b(1,1)$ which is obtained from $b(X,Y)$ by substituting X and Y by the values in θ ((1.3) Table 1). There is no more literal to consider then a complete substitution has been found. The atom of the head $a(X+Y)$ takes the values of the substitution θ. Hence, the forward chaining algorithm can add $a(2)$ into the *propagate_IN* queue.

Now, during a new instantiation attempt of the rule for the atom to propagate $a(1)$, the function restarts with the last substitution of the rule $\theta = \{X/1, Y/1\}$ in order to find a new substitution for the literal $l_3 = b(X,Y)$. The second attempt for an instantiation begins with atom to propagate $a(1)$ for the second time. Since $b(X,Y)$ has no free variable, there can be no other substitution than the current one ((2.1) Table 1). The process then backtracks to the literal $l_2 = a(Y)$ which has no other substitution in IN_1 ($a(2)$ has been inferred after $a(1)$ and is not into the current set IN_1) ((2.2) Table 1). Since literal $l_1 = a(X)$ can only take the value $a(1)$, it also fails ((2.3) Table 1). Since the last literal has failed, the literal $l_2 = a(Y)$ is now marked instead of $a(X)$, and is instantiated with the atom to propagate $a(1)$. Hence, value 1 is substituted to the variable Y in θ ((2.4) Table 1). Literal $l_1 = a(X)$ is unmarked and can only take the values of the atoms of IN_0, thus no substitution is possible. Hence the algorithm fails on the first literal ((2.5) Table 1). Since there is no more literal to be marked, the rule instantiation ends by a failure for the atom to propagate $a(1)$. The sets becomes as follow:

$$IN_0 = \{b(1,1), b(1,2), a(1)\}$$
$$propagate\_IN = \{a(2)\}$$
$$IN_1 = \{b(1,1), b(1,2), a(1), a(2)\}$$

The next atom $a(2)$ is extracted from the queue to propagate. The third attempt for an instantiation of r_0 begins with atom to propagate $a(2)$ for the first time. Table 2 shows the different steps of the instantiation. The literals $a(X)$ and $a(Y)$

$a(X+Y)$	$\leftarrow$	$a(X)$,	$a(Y)$,	$b(X,Y)$	
		*** third call to *instantiateRule* ***			
(1.1)		**a(2)**	-	-	(l_1 marked)
(1.2)		**a(2)**	$a(1)$	-	
(1.3)		**a(2)**	$a(1)$	NO	
(1.4)		**a(2)**	$a(2)$	-	
(1.5)		**a(2)**	$a(2)$	NO	
(1.6)		**a(2)**	NO	-	
(1.7)		**NO**	-	-	$\Rightarrow$ failure
(1.8)		-	**a(2)**	-	(l_2 marked)
(1.9)		$a(1)$	**a(2)**	-	
(1.10)		$a(1)$	**a(2)**	$b(1,2)$	$\Rightarrow$ complete instantiation
		*** fourth call to *instantiateRule* ***			
(2.1)		$a(1)$	**a(2)**	NO	
(2.2)		NO	**a(2)**	-	
(2.3)		-	**NO**	-	$\Rightarrow$ failure

Table 2. *Decomposition of the instantiations of the rule r_0 for the atom to propagate $a(2)$ (Example 15 continued)*

are again to be marked. The rule instantiation is restarted with the literal $l_1 = a(X)$ which is the marked literal. The variable X is substituted by the value 2 since the only allowed value is that of the atom to propagate $a(2)$ ((1.1) Table 2). The current literal is now $l_2 = a(Y)$ where Y is substituted by the value 1 since $a(1)$ is into IN_1 ((1.2) Table 2). The literal $b(X,Y)$ has no free variable and since the atom $b(2,1)$ which respects the substitution $\theta = \{X/2, Y/1\}$ is not in IN_1, the literal $b(X,Y)$ has no possible substitution ((1.3) Table 2). Then a new instantiation for $l_2 = a(Y)$ is sought: its next possible value is 2 (since $a(2)$ is in IN_1) ((1.4) Table 2). Again, since the atom $b(2,2)$ which respects the substitution $\theta = \{X/2, Y/2\}$ is not in IN_1, the literal $b(X,Y)$ has no possible substitution ((1.5) Table 2). The process backtracks to the literal $l_2 = a(Y)$ which has no possible value ((1.6) Table 2). Hence, the process backtracks to the literal $l_1 = a(X)$ which has no possible value since the only possible value was that of the atom to propagate $a(2)$ ((1.7) Table 2). Since the first literal has failed, the process restarts by marking the second literal $a(Y)$ (and unmarking the first $a(X)$). The marked literal $l_2 = a(Y)$ is processed first, it substitutes Y by the value 2 of the atom to propagate $a(2)$ ((1.8) Table 2). The unmarked literal $l_1 = a(X)$ may only take its values into IN_0. The variable X is then substituted by the value 1 ((1.9) Table 2). The literal $l_3 = b(X,Y)$ has no free variable and since $b(1,2)$ which respects the substitution $\theta = \{X/1, Y/2\}$ is in IN_1 a complete substitution is found ((1.10) Table 2). The atom $a(X+Y)$ of the head takes then the value of the substitution θ. Hence, the forward chaining algorithm can add $a(3)$ into IN_1 and into *propagate_IN*.

Then, during a new instantiation attempt of the rule r_0, the atom to propagate is still $a(2)$. The process restarts from the last substitution $\theta = \{X/1, Y/2\}$ and search for a new substitution for the literal $l_3 = b(X,Y)$. A fourth attempt for an instantiation begins with atom to propagate $a(2)$ for the second time. Since $b(X,Y)$ has no free variable, there can be no other substitution than the current one ((2.1) Table 2). The process then backtracks to the literal $l_1 = a(X)$ that has no other

substitution since the only possible values are those from IN_0 (then neither the atom to propagate $a(2)$ nor $a(3)$ appeared after $a(2)$ are possible) ((2.2) Table 2). The literal $l_2 = a(Y)$ also fails since the marked literal only accepts the value of the atom to propagate $a(2)$ ((2.3) Table 2). Since there is no more literal to be marked, the instantiation of the failing rule ends for this atom to propagate. The process continues with the atom $a(3)$ which also leads to a failure.

3.5 ASPeRiX language

The core language of `ASPeRiX` is that of normal logic programs (Gelfond and Lifschitz 1988) with function symbols and true (or strong) negation without inconsistent answer set. `ASPeRiX` also provides dedicated treatment of lists with built-in predicates, as in `DLV-complex` (Calimeri et al. 2008), an extension of `DLV` with lists and sets. On the other side, `ASPeRiX` does not provide aggregate atoms and optimization statements (Buccafurri et al. 2000) which are accepted by the main current systems.

One of the important issues in ASP is the treatment of function symbols. Uninterpreted function symbols are important because they enable representation of recursive structures such as lists and trees. But reasoning becomes undecidable if no restriction is enforced. A lot of work has been made for identifying program classes for which reasoning is decidable (Alviano et al. 2011; Alviano et al. 2010; Calimeri et al. 2011; Lierler and Lifschitz 2009; Baselice and Bonatti 2010; Greco et al. 2013).

The inherent difficulty with functions in general (and arithmetic in particular) in the framework of ASP is that it makes the Herbrand universe infinite in whole generality. ASP grounders `Lparse` (Syrjänen 1998) and versions up to 3.0 of `Gringo` (Gebser et al. 2007) accept programs respecting some syntactic domain restrictions and are able to deal with some restricted versions of functions.

`DLV` grounder (Faber et al. 2012) and `Gringo` (since version 3.0) (Gebser et al. 2011) only require programs to be safe and can deal with all programs having a finite instantiation. `DLV` guarantees finite instantiation for finitely ground programs but membership in this class is not decidable. It integrates a Finite Checker module which can check if a program belongs to a sub-class of finitely ground programs (argument-restricted programs). For programs that are not member of this subclass, answer sets can be computed without preliminary check but ending is not guaranteed.

`ASPeRiX` can deal with these programs and with some other programs whose instantiation is infinite but whose answer sets are finite. For example, the program P_{1a} from Section 1 is not finitely ground: intelligent instantiation of the program must be finite to be finitely ground. The key points of intelligent instantiation are that rules are instantiated with atoms appearing in head of rules of the program, and simplifications are performed relatively to facts and rule heads of preceding components of the dependency graph. In example P_{1a}, choice between a and b makes both possible for the grounder, and constraint has no effect on intelligent instantiation of the program. Thus, the grounding of rules from P_{1a} will be the same with or with-

out the constraint ($\leftarrow a.$): infinite in both cases. `ASPeRiX` halts on P_{1a} and is thus able to halt on non finitely ground programs but it is not able to verify in advance if answer sets are finite or not, and thus if computation will end or not. Nevertheless, ending can be guaranteed by means of command-line options specifying the maximum allowed nesting level for functional terms and the biggest admissible integer (`DLV` grounder provides similar possibilities). These restrictions ensure that our computations always converge to an answer set if it exists. Formalizing the class of programs for which `ASPeRiX` halts will be the subject of a forthcoming work.

4 Experimental results

Following Algorithm 1 of Section 3.2, the solver called `ASPeRiX` has been implemented in C++ and is available at `http://www.info.univ-angers.fr/pub/claire/asperix`.

There are two other ASP systems, `GASP` (Dal Palù et al. 2009) and `OMiGA` (Dao-Tran et al. 2012), that realize the grounding of the program during the search of an answer set.

`GASP` is an implementation in Prolog and Constraint Logic Programming over finite domains of the notion of computation (see Section 3.1). The main ideas are the same as those of `ASPeRiX`. Notable differences are the following. Well founded consequences of the program are computed first. Then propagation is close to ours. `GASP` does not deal with must-be-true atoms but two special cases of propagation, not treated by `ASPeRiX`, are implemented: (a) if the head of a rule is known to be in OUT set and the body of the rule is satisfied except for one positive literal, then this literal must be false (added to OUT) and (b) if, for some undefined atom a, there is no applicable rule whose head is a, then a can be added to OUT. For each rule, instantiation and propagation are realized by building and solving a CSP that determines atoms derivable from the rule. Representation of interpretations uses Finite Domain Sets, such a data structure is efficient to represent compactly intervals but it need to code tuples (instances of predicates) by integers (very big integers if domain is large and arity of predicate too). This representation impose the set of ground terms of the program to be finite and thus function symbols are excluded. On the other hand, `GASP` supports some cardinality constraints. To our knowledge, `GASP` remained at the prototype stage and is no longer developed.

`OMiGA` is implemented in Java. Functional symbols (of non-zero arity) are not supported. Principles of propagation and choice are the same as those of `ASPeRiX` but implementation uses Rete algorithm for improving the speed of propagation. First order rules are represented by a Rete network. Each node represents a literal (or a set of literals) from the body of a rule or the atom of the head of a rule. It stores all instances of the node that are true w.r.t. current partial interpretation. Thereby all partial instantiations of rules are stored in the network. This lead to an efficient propagation regarding computation time, but memory space is sacrified. Dependency graph and solved predicates seems to be treated in a similar manner to that of `ASPeRiX`. Current version (Weinzierl 2013) uses must-be-true propagation and tries to introduce methods for conflict-driven learning of non-ground rules:

when a constraint is violated, a new constraint is built by unfolding of rules whose firing contributes to the conflict. This learned constraint is then transformed into special rules so as to be used for propagation.

In the following we give some results of evaluation of `ASPeRiX 0.2.5` highlighting its adequacy to some particular problems. It is compared with `Clingo` (composed by `Gringo 3.0.5` and `Clasp 1.3.10`) (Gebser et al. 2011; Gebser et al. 2012), `DLV Dec 16 2012` (Leone et al. 2006), `GASP (june 2009)` (Dal Palù et al. 2009) and `OMiGA Dec 3 2012` (Dao-Tran et al. 2012). Version without learning is used for `OMiGA` because learning lowers its performances. All the systems have been run on an Intel Core i7-3520M PC with 4 cores at 2.90GHz and about 4GB RAM, running Linux Ubuntu 12.04 64 bits. For each instance of a problem, the memory usage is limited to 3.000MB and computation time to 600 seconds. RunLim1.7 is used for these limitations tasks. Tables of results use OoM (resp. OoT) to indicate Out of Memory (resp. Out of Time). Results for `GASP` are only given for the first two examples, because it does not accept other tested programs.

Schur problem The Schur number problem is to partition N numbers into M sets such that all of the sets satisfy: if x and y are assigned to the same set, then $x + y$ is not in the set. The following program (Dal Palù et al. 2009) is for $M = 3$ sets and $N = 4$ numbers.

$$P_{Schur-4} = \left\{ \begin{array}{l} number(1)., \quad number(2)., \quad number(3)., \quad number(4)., \\ part(1)., \qquad part(2)., \qquad part(3)., \\ inpart(X,1) \leftarrow not\ inpart(X,2),\ not\ inpart(X,3),\ number(X)., \\ inpart(X,2) \leftarrow not\ inpart(X,1),\ not\ inpart(X,3),\ number(X)., \\ inpart(X,3) \leftarrow not\ inpart(X,1),\ not\ inpart(X,2),\ number(X)., \\ \leftarrow\ number(X),\ number(Y),\ part(P), \\ \quad inpart(X,P),\ inpart(Y,P),\ inpart(Z,P), \\ \quad T = Y + 1,\ X < T,\ Z = X + Y. \end{array} \right\}$$

The results are shown in Table 3 for $M = 3$. AS reports the number of answer sets which are all computed. For all $N \geq 14$, Schur-N has no answer set.

The program is a typical "guess and check" program. The seach space is expressed by the three rules with *inpart* as head predicate, and constraint eliminates "bad choices". The grounding of the program is rather small but the search space is large. The problem is very easy for `Clingo` and `DLV` but very hard for `ASPeRiX` and `GASP`. Systems using grounding on the fly have to repeat instantiation of the same rules in each branch of the search tree. Moreover, constraints are not efficiently managed by systems like `ASPeRiX`: it does not use constraints for propagation but only checks if a constraint is violated. Compared to `ASPeRiX`, `OMiGA` performs well for computation time, certainly because Rete network improve speed of instantiation (partial instantiations are stored in the network) and the network remains relatively small in such an example. This example illustrates a large class of programs that `ASPeRiX` mismanage: programs with many choices and little propagation by forward chaining.

Conversely, the following examples illustrate problems for which grounding on the fly is well adapted.

			ASPeRiX	Clingo	DLV	OMiGA	GASP
$N = 1$	$AS = 3$	time in sec	<0.1	<0.1	<0.1	<0.1	<0.1
		memory in MB	<2.0	<2.0	<2.0	20.0	4.4
$N = 2$	$AS = 6$	time in sec	<0.1	<0.1	<0.1	<0.1	<0.1
		memory in MB	<2.0	<2.0	<2.0	20.0	5.2
$N = 3$	$AS = 18$	time in sec	<0.1	<0.1	<0.1	0.1	0.3
		memory in MB	<2.0	<2.0	<2.0	20.0	6.3
$N = 4$	$AS = 30$	time in sec	<0.1	<0.1	<0.1	0.2	1.0
		memory in MB	<2.0	<2.0	<2.0	24.0	8.3
$N = 5$	$AS = 66$	time in sec	<0.1	<0.1	<0.1	0.3	3.6
		memory in MB	<2.0	<2.0	<2.0	50.0	8.3
$N = 6$	$AS = 120$	time in sec	<0.1	<0.1	<0.1	0.4	11.0
		memory in MB	<2.0	<2.0	<2.0	63.0	8.3
$N = 7$	$AS = 258$	time in sec	0.3	<0.1	<0.1	0.6	41.0
		memory in MB	2.0	<2.0	<2.0	99.0	7.5
$N = 8$	$AS = 288$	time in sec	1.0	<0.1	<0.1	0.8	113.0
		memory in MB	2.0	<2.0	<2.0	100.0	5.7
$N = 9$	$AS = 546$	time in sec	3.6	<0.1	<0.1	1.3	370.0
		memory in MB	2.0	<2.0	<2.0	165.0	7.5
$N = 10$	$AS = 300$	time in sec	11.1	<0.1	<0.1	1.9	OoT
		memory in MB	2.0	<2.0	<2.0	220.0	-
$N = 11$	$AS = 186$	time in sec	39.7	<0.1	<0.1	2.8	OoT
		memory in MB	2.2	<2.0	<2.0	290.0	-
$N = 12$	$AS = 114$	time in sec	131.0	<0.1	<0.1	4.1	OoT
		memory in MB	2.2	<2.0	<2.0	290.0	-
$N = 13$	$AS = 18$	time in sec	448.0	<0.1	<0.1	6.5	OoT
		memory in MB	2.2	<2.0	<2.0	285.0	-
$N = 14$	$AS = 0$	time in sec	OoT	<0.1	<0.1	11.0	OoT
		memory in MB	-	<2.0	<2.0	287.0	-

Table 3. *Experimental results for Schur*

Birds problem Problem *birds* is a stratified program encoding a taxonomy about flying and non flying birds. b stands for $bird$, f for $flying$, nf for $nonflying$, p for $penguin$, sp for $superpenguin$, and o for $ostrich$.

$$P_{birds} = \left\{ \begin{array}{lll} p(X) \leftarrow sp(X)., & b(X) \leftarrow p(X)., & b(X) \leftarrow o(X)., \\ f(X) \leftarrow b(X),\ not\ p(X),\ not\ o(X)., & & f(X) \leftarrow sp(X)., \\ nf(X) \leftarrow p(X),\ not\ sp(X)., & & nf(X) \leftarrow o(X). \end{array} \right\}$$

We add to this program the atoms encoding N birds with 10% of ostriches, 20% of penguins whose half of them are super penguins.

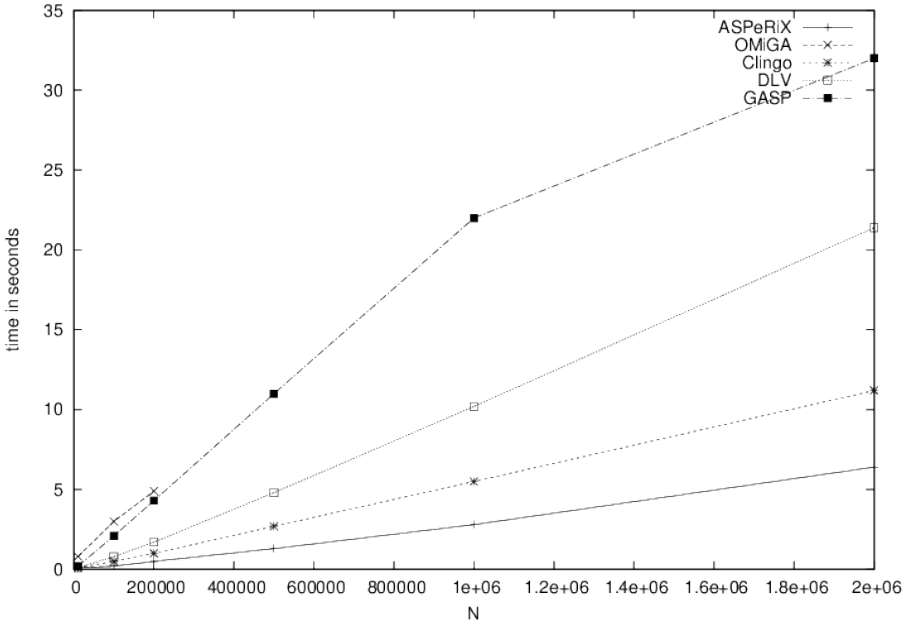

Fig. 4. Time for birds

The unique answer set of such a program can be computed polynomially. `ASPeRiX` uses only propagation step, without choice point, and grounders completely evaluate the program so that the solver has nothing to do. Experimental results for *birds* are comparable for `ASPeRiX`, `Clingo`, `GASP` and `DLV`. `ASPeRiX` has the best results for CPU time, and `DLV` for memory usage (Figure 4 and 5). For such a problem, the number of instantiated rules must be nearly the same for all systems. On the other side, `OMiGA` system uses a very large amount of memory space, certainly due to the Rete network which is designed to sacrifice memory for increased speed. Unfortunately, memory gains expected by the first order approach are lost.

Cutedge problem *cutedge* program is proposed in (Dao-Tran et al. 2012): given a random graph with 100 vertices and N edges, each answer set is obtained by deleting an edge and compute some transitive closure on the remaining edges.

$$P_{cutedge} = \left\{ \begin{array}{l} delete(X,Y) \leftarrow edge(X,Y),\ not\ keep(X,Y). \\ keep(X,Y) \leftarrow edge(X,Y),\ delete(X1,Y1),\ X1! = X., \\ keep(X,Y) \leftarrow edge(X,Y),\ delete(X1,Y1),\ Y1! = Y., \\ reachable(X,Y) \leftarrow keep(X,Y)., \\ reachable(X,98) \leftarrow reachable(X,Z),\ reachable(Z,98). \end{array} \right\}$$

Computing each answer set is only based on propagation, and the number of answer sets equals the number of edges. The number of rules needed to compute all answer sets is proportional to N^2 while the rule number needed to compute

			ASPeRiX	Clingo	DLV	OMiGA
$N = 2.8K$	$AS = 1$	time in sec	<0.1	21	115	0.6
		memory in MB	14.8	345	103	85
	$AS = 10$	time in sec	0.2	21	226	1.8
		memory in MB	14.8	345	103	200
	$AS = 100$	time in sec	2.7	32	OoT	11.4
		memory in MB	15.1	345	-	1042
	$AS = 500$	time in sec	16	78	OoT	48
		memory in MB	16.4	345	-	1050
	$AS = 1000$	time in sec	36	123	OoT	84
		memory in MB	18.0	345	-	1050
	$AS = all$	time in sec	167	189	OoT	165
		memory in MB	24.1	345	-	1050
$N = 4.9K$	$AS = 1$	time in sec	0.1	60	325	1
		memory in MB	23.7	881	144	177
	$AS = 10$	time in sec	0.5	63	OoT	3.7
		memory in MB	23.8	881	-	425
	$AS = 100$	time in sec	5.8	94	OoT	29
		memory in MB	24.0	881	-	1100
	$AS = 500$	time in sec	30.7	228	OoT	125
		memory in MB	25.3	881	-	1150
	$AS = 1000$	time in sec	67	373	OoT	245
		memory in MB	27	881	-	1190
$N = 5.9K$	$AS = 1$	time in sec	0.1	94	465	1
		memory in MB	28.5	1167	202	132
	$AS = 10$	time in sec	0.8	94	OoT	5
		memory in MB	28.6	1168	-	680
	$AS = 100$	time in sec	7.6	114	OoT	41
		memory in MB	29.1	1168	-	1135
	$AS = 500$	time in sec	42	210	OoT	192
		memory in MB	31.3	1168	-	1125
	$AS = 1000$	time in sec	92	316	OoT	352
		memory in MB	34.1	1168	-	1132

Table 4. *Experimental results for cutedge*

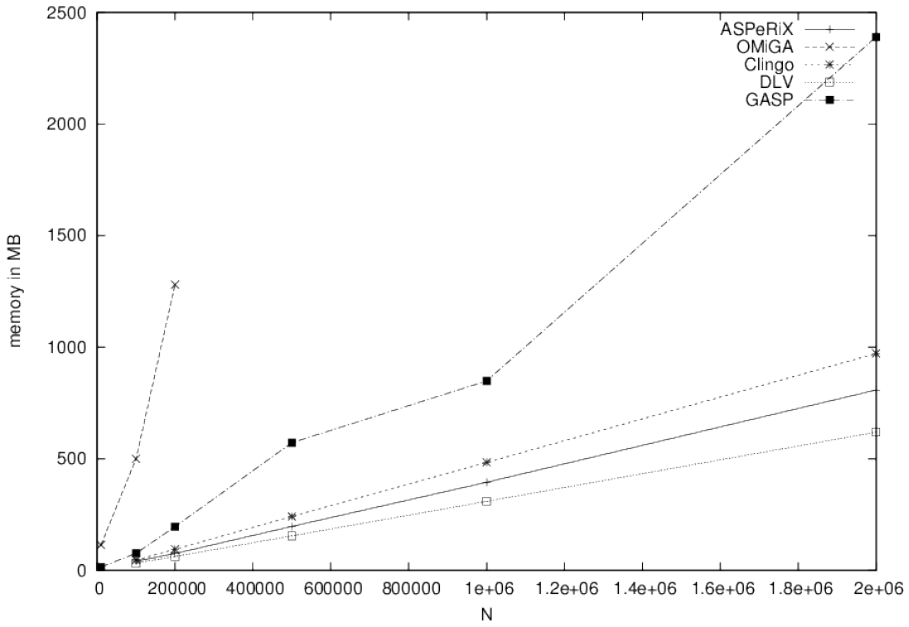

Fig. 5. Space for birds

one is proportional to N. But systems with pregrounding phase must generate all ground instances of rules even if only one answer set is required. The results are shown in Table 4. `ASPeRiX` has the best results for this program both for CPU time and memory usage. `OMiGA` and `Clingo` use much more memory and are much slower than `ASPeRiX`. As expected, memory usage of `Clingo` is independent of the number of answer sets required and is close to the square of that used by `ASPeRiX`. For its part, `DLV` quickly exceeds the time limit imposed.

Hamiltonian cycle problem The program P_3 from Example 3, Hamiltonian cycle in a complete graph, is another easy problem with a lot of answer sets. Each answer set is easy to compute but the whole instantiation is huge. Experiments for the computation of one answer set in a graph with N vertices are represented in Figures 6 and 7. `ASPeRiX` performs well on this example whereas `OMiGA` has time and memory problems similar to that of `Clingo` and `DLV`. One more time, a simple problem becomes intractable by systems with pregrounding phase because they drown it in a lot of useless information so that memory used quickly becomes prohibitive.

Hanoi problem *Hanoi* example illustrates a planning problem where the maximum number of allowed steps is given as input. NbD is the number of disks in the problem and NbM is the maximum number of moves that are allowed to move all disks from the first rod to the third. The least value of NbM is the minimum required to achieve the goal, then its value is gradually increased to evaluate its impact. The complete program is given in the online appendix of the paper, Appendix

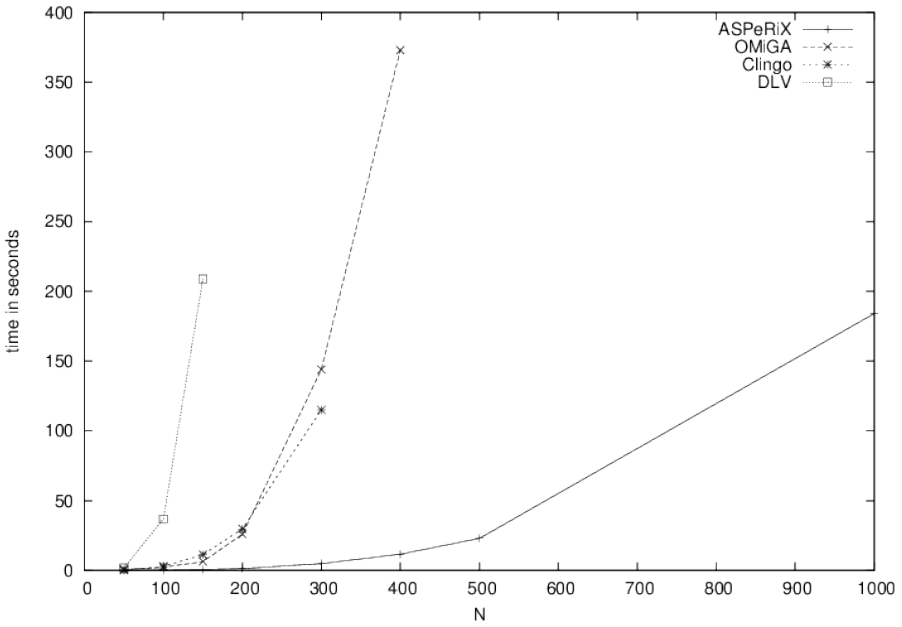

Fig. 6. Time for Hamiltonian cycle

A. Experimental results are shown in Table 5. `ASPeRiX` performances are (almost) independent of the given number of moves: search, and therefore grounding, are stopped when a solution is found. Conversely, grounders are quickly overwhelmed as they are obliged to fully instantiate the program with all hypothetical (and unnecessary in this case) calculation steps[4]. This example cannot be computed by `OMiGA` due to restrictions on its input language (function symbols are not supported).

Three coloring problem The program P_2 (see Example 2), 3-coloring problem on a graph organized as a bicycle wheel, poses no problem for `Clingo` and `DLV` (cf. Table 6). But `ASPeRiX` and `OMiGA` have bad results on this example because they are mismanaging constraints. Once a vertex is colored, say *red*, constraint $(\leftarrow e(V,U), col(V,C), col(U,C).)$ prohibits coloring adjacent vertices of the same color. In propositional systems, unit propagation (or equivalent) works well and allows to infer that adjacent vertices are not colored red. But first-order approach does not allow, in general case, to use unit propagation and thus, constraints are mainly used for verification and not for propagation. A lot of work remains on these points. First-order constraints could instead allow more powerful propagation. Suppose for example a constraint $(\leftarrow p(X,Y), p(Y,Z).)$ and $p(1,2)$ is added in IN set then, for all Z, $p(2,Z)$ can be excluded at once from current solution, even if Z values are potentially infinite. But these opportunities are not exploited yet.

[4] `iClingo` (Gebser et al. 2008) was created to address this specific problem. Some directives are added to the program in order to incrementally instantiate some predicates of the program. But it does not escape the grounding/solving separation, it only introduces some tools to control the process.

			ASPeRiX	Clingo	DLV
$NbD = 4$	$NbM = 15$	time in sec	<0.1	<0.1	< 0.1
		memory in MB	-	-	-
	$NbM = 60$	time in sec	<0.1	0.7	0.8
		memory in MB	-	27	22
	$NbM = 100$	time in sec	<0.1	1.5	3.6
		memory in MB	-	54	51
	$NbM = 500$	time in sec	<0.1	11.6	-
		memory in MB	-	327	OoM
	$NbM = 1000$	time in sec	<0.1	27	-
		memory in MB	-	693	OoM
	$NbM = 2000$	time in sec	<0.1	66	-
		memory in MB	-	1523	OoM
	$NbM = 5000$	time in sec	<0.1	-	-
		memory in MB	-	OoM	OoM
	$NbM = 10000$	time in sec	<0.1	-	-
		memory in MB	-	OoM	OoM
	$NbM = 50000$	time in sec	0.1	-	-
		memory in MB	12.9	OoM	OoM
	$NbM = 100000$	time in sec	0.3	-	-
		memory in MB	23.9	OoM	OoM
$NbD = 5$	$NbM = 31$	time in sec	0.2	0.1	0.1
		memory in MB	3.7	6.4	4
	$NbM = 50$	time in sec	0.2	0.7	0.9
		memory in MB	3.7	28	31
	$NbM = 100$	time in sec	0.2	8.2	9.4
		memory in MB	3.8	245	270
	$NbM = 500$	time in sec	0.2	91	-
		memory in MB	3.8	2055	OoM
	$NbM = 1000$	time in sec	0.2	-	-
		memory in MB	3.8	OoM	OoM
	$NbM = 5000$	time in sec	0.2	-	-
		memory in MB	4.8	OoM	OoM
	$NbM = 10000$	time in sec	0.2	-	-
		memory in MB	5.8	OoM	OoM
	$NbM = 50000$	time in sec	0.4	-	-
		memory in MB	14.4	OoM	OoM
	$NbM = 100000$	time in sec	0.7	-	-
		memory in MB	25	OoM	OoM
$NbD = 6$	$NbM = 63$	time in sec	4.7	0.6	1.1
		memory in MB	10.3	24	29
	$NbM = 100$	time in sec	4.7	9	-
		memory in MB	10.3	272	OoM
	$NbM = 150$	time in sec	4.7	83	-
		memory in MB	10.3	1863	OoM
	$NbM = 200$	time in sec	4.7	-	-
		memory in MB	10.3	OoM	OoM
	$NbM = 500$	time in sec	4.7	-	-
		memory in MB	10.4	OoM	OoM
	$NbM = 1000$	time in sec	4.7	-	-
		memory in MB	10.5	OoM	OoM
	$NbM = 5000$	time in sec	4.7	-	-
		memory in MB	11.3	OoM	OoM
	$NbM = 10000$	time in sec	4.8	-	-
		memory in MB	12.4	OoM	OoM
	$NbM = 50000$	time in sec	5	-	-
		memory in MB	21	OoM	OoM
	$NbM = 100000$	time in sec	5.6	-	-
		memory in MB	31.7	OoM	OoM

Table 5. *Experimental results for Hanoi tower problem*

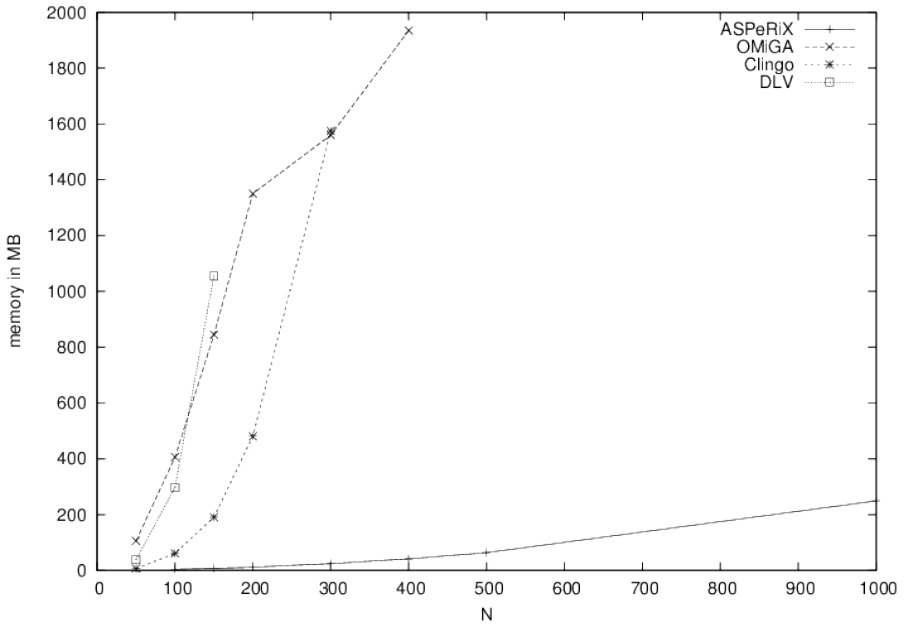

Fig. 7. Space for Hamiltonian cycle

			ASPeRiX	Clingo	DLV	OMiGA
$N = 11$	$AS = 1$	time in sec	<0.1	<0.1	<0.1	0.3
		memory in MB	<2	1	<1	45
$N = 11$	$AS = all(6)$	time in sec	3.4	<0.1	<0.1	7.7
		memory in MB	1.8	1	<1	132
$N = 101$	$AS = 1$	time in sec	<0.1	<0.1	<0.1	OoT
		memory in MB	3	1.5	1.1	-
$N = 101$	$AS = all(6)$	time in sec	OoT	<0.1	<0.1	OoT
		memory in MB	-	1.8	1.4	-
$N = 501$	$AS = 1$	time in sec	1.6	<0.1	<0.1	OoT
		memory in MB	8.8	3.3	3.3	-
$N = 501$	$AS = all(6)$	time in sec	OoT	<0.1	0.3	OoT
		memory in MB	-	3.3	3.3	-
$N = 1001$	$AS = 1$	time in sec	13.3	<0.1	0.1	OoT
		memory in MB	15.9	5.5	5.5	-
$N = 1001$	$AS = all(6)$	time in sec	OoT	<0.1	1.4	OoT
		memory in MB	-	5.5	5.5	-

Table 6. *Experimental time results for 3col*

To sum up, ASPeRiX is efficient to deal with stratified programs or simple problems whose instantiation is infinite or huge but much of which is useless to compute one specific answer set. On the other hand, the system is not competitive for more combinatorial problems, with a large search space and few solutions, because propositional methods for propagation, heuristics, learning lemmas did not apply to the first order case.

5 Conclusion

In this paper, we have presented the ASPeRiX approach to answer set computation. Our methodology deals with first order rules following a forward chaining with grounding process realized on the fly and has been implemented in the ASP solver ASPeRiX. This paper is the first comprehensive document in which a survey of the important techniques relevant to our approach is presented.

Starting from a short description of state-of-the-art ASP working principle, we have presented by many examples the main motivation of our approach: escaping the bottleneck of the preliminary phase of grounding in which many state-of-the-art systems fall. After a presentation of the theoretical foundations of ASP, we have described by an ASPeRiX computation our first order forward chaining approach for answer set computing and have established the soundness and completeness of this calculus w.r.t. the semantics of ASP (Proofs are reported in the online appendix of the paper, Appendix B). We have then described in details the main algorithms of ASPeRiX and particularly those which realize the selection of the first order rules to be instantiated and applied according to the current answer set in construction.

Our methodology allows very good performances for definite and stratified programs. It outperforms systems with a pregrounding phase for programs with large grounding but much of it is unnecessary to solve the problem. On the other side, performances quickly degrade for combinatorial problems with large search spaces, especially if forward chaining propagation can not be exploited.

We have shown that our approach escapes the bottleneck of the preliminary phase of grounding that is the only difficulty for some classes of programs. A direct consequence of our new approach is that the use of symbolic functions in general and arithmetic calculus in particular inside ASP is greatly facilitated.

The forward chaining with the grounding process realized on the fly as an operational semantics emphasizes the programming aspect of ASP in which the answer set is not only the result of a black box but the result of a process that may be followed. This is interesting in particular when dealing with knowledge coming from the web and expressed in description logic since the structure of information uses rules that are chained ones with the others (whereas this is not always the case for a program encoding a combinatorial problem). Moreover, when dealing with knowledge expressed in description logic, one important issue is the ability to query the knowledge base. The grounding process realized on the fly will then allow to focus only on the rules useful to find an answer to the query. For this category of programs, we think that our approach may be of great interest.

Furthermore, computing the answer sets of a program is a fundamental goal but

not an exclusive one. Debugging a program, controlling its behavior, introducing in it some features coming from other programming languages may be of great interest for ASP. We think that our methodology of answer set computing, guided by the rules of the program, is the good starting point towards these new goals.

The `ASPeRiX` project is still in progress. Improvements at the algorithmic level are underway by the development and implementation of backjumping and clause learning techniques. On the other hand, we plan to fully respect the core language ASP (Calimeri et al. 2014) by introducing, among others, minimization / maximization and aggregates and extend it by introducing existentially quantified variables in multi-head rules to encode fragments of Description Logics which are logical formalisms for ontologies and the Semantic Web.

Tribute

In memory of the late Pascal Nicolas who was at the origin of this work. He sadly passed away in 2010 but his enthusiasm, his passion for research and his great humanity are still with us.

References

Alviano, M., Calimeri, F., Faber, W., Ianni, G., and Leone, N. 2011. Function Symbols in ASP: Overview and Perspectives. In *Nonmonotonic Reasoning, Essays Celebrating its 30th Anniversary*, G. Brewka, V. Marek, and M. Truszczynski, Eds. Studies in Logic, vol. 31. College Publications, 1–24.

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 on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, P. Cabalar and T. C. Son, Eds. LNCS, vol. 8148. Springer, 55–67.

Alviano, M., Faber, W., and Leone, N. 2010. Disjunctive asp with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming 10,* 4-6, 497–512.

Balduccini, M. 2009. Representing constraint satisfaction problems in answer set programming. In *Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*. 16–30.

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

Baselice, S., Bonatti, P., and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proceedings of the 21st International Conference on Logic Programming (ICLP'05)*. LNCS, vol. 3668. Springer, 52–66.

Baselice, S. and Bonatti, P. A. 2010. A decidable subclass of finitary programs. *Theory and Practice of Logic Programming 10,* 4-6, 481–496.

Buccafurri, F., Leone, N., and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering 12,* 5, 845–860.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2008. Computable functions in ASP: Theory and implementation. In *Proceedings of the 24th International Conference on Logic Programming (ICLP'08)*, M. G. de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer, 407–424.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2011. Finitely recursive programs: Decidability and bottom-up computation. *AI Communications 24,* 4 (Dec.), 311–334.

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.

Calimeri, F., Perri, S., and Ricca, F. 2008. Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms 63,* 1-3, 34–54.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3 (Aug.), 297–322.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., and Weinzierl, A. 2012. OMiGA: An open minded grounding on-the-fly answer set solver. In *Proceedings of the 13th European Conference on Logics in Artificial Intelligence (JELIA'12)*. LNAI, vol. 7519. Springer, 480–483.

Eiter, T., Lu, J. J., and Subrahmanian, V. S. 1997. Computing non-ground representations of stable models. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach, and A. Nerode, Eds. LNCS, vol. 1265. Springer, 198–217.

Faber, W., Leone, N., and Perri, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. LNCS, vol. 7265. Springer, 247–264.

Faber, W., Leone, N., and Pfeifer, G. 1999. Pushing goal derivation in dlp computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, M. Gelfond, N. Leone, and G. Pfeifer, Eds. LNCS, vol. 1730. Springer, 177–191.

Ferraris, P., Lee, J., and Lifschitz, V. 2007. A new perspective on stable models. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*. 372–379.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. 2008. Engineering an incremental ASP solver. In *Proceedings of the 24th International Conference on Logic Programming (ICLP'08)*, M. Garcia de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer, 190–205.

Gebser, M., Kaminski, R., König, A., and Schaub, T. 2011. Advances in *gringo* Series 3. In *Proceedings of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. P. Delgrande and W. Faber, Eds. LNCS, vol. 6645. Springer, 345–351.

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

Gebser, M., Schaub, T., and Thiele, S. 2007. GrinGo : A New Grounder for Answer Set Programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. LNCS, vol. 4483. Springer, 266–271.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP'88)*, R. A. Kowalski and K. Bowen, Eds. The MIT Press, Cambridge, Massachusetts, 1070–1080.

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

Giunchiglia, E., Lierler, Y., and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

Gottlob, G., Marcus, S., Nerode, A., Salzer, G., and Subrahmanian, V. S. 1996. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science 166,* 1-2, 221–262.

Greco, S., Molinaro, C., and Trubitsyna, I. 2013. Logic programming with function symbols: Checking termination of bottom-up evaluation through program adornments. *Theory and Practice of Logic Programming 13,* 4-5, 737–752.

Konczak, K., Linke, T., and Schaub, T. 2006. Graphs and colorings for answer set programming. *Theory and Practice of Logic Programming 6*, 61–106.

Lefèvre, C. and Nicolas, P. 2009a. A first order forward chaining approach for answer set computing. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. LNCS, vol. 5753. Springer, 196–208.

Lefèvre, C. and Nicolas, P. 2009b. The first version of a new ASP solver : `ASPeRiX`. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. LNCS, vol. 5753. Springer, 522–527.

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. and Lifschitz, V. 2009. One more decidable class of finitely ground programs. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*. LNCS, vol. 5649. Springer, 489–493.

Lin, F. and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1-2, 115–137.

Lin, F. and Zhou, Y. 2007. From answer set logic programming to circumscription via logic of GK. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*. 441–446.

Liu, L., Pontelli, E., Son, T. C., and Truszczynski, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence 174,* 3-4, 295–315.

Liu, L. and Truszczynski, M. 2005. Pbmodels - software to compute stable models by pseudoboolean solvers. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. LNCS, vol. 3662. Springer, 410–415.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

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

Perri, S., Scarcello, F., Catalano, G., and Leone, N. 2007. Enhancing dlv instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence 51,* 2-4, 195–228.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

Syrjänen, T. 1998. Implementation of local grounding for logic programs for stable model semantics. Tech. rep., Helsinki University of Technology.

Truszczynski, M. 2012. Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. LNCS, vol. 7265. Springer, 543–559.

Ullman, J. D. 1989. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press.

Weinzierl, A. 2013. Learning non-ground rules for answer-set solving. In *2nd Workshop on Grounding and Transformations for Theories With Variables (GTTV'13).*