VERSO L'ANALISI STATICA DEGLI ANSWER SET PROGRAMS

TOWARDS STATIC ANALYSIS OF ANSWER SET PROGRAMS[1]

Stefania Costantini

Abstract

In questo lavoro viene proposto un metodo di analisi statica per l'"Answer Set Programming", che é un nuovo paradigma basato sulla semantica degli "answer sets" introdotta da Gelfond e Lifschitz (chiamata anche semantica dei "modelli stabili"). Il metodo si basa sui seguenti punti: identificare i cicli che compaiono in un programma, mostrando che i modelli stabili del programma stesso si possono ottenere dai modelli stabili di particolari sottoprogrammi corrispondenti ai cicli; definire il Grafo dei Cicli, dove ogni nodo corrisponde ad un ciclo ed ogni arco ad un letterale detto "handle", che puó essere visto come una connessione fra due cicli in quanto l'atomo che in esso occorre appartiene ad entrambi i cicli. Sul Grafo dei Cicli é possibile verificare diverse proprietá di un programma, inclusa la consistenza.

In this paper we propose a static analysis methodology for Answer Set Programming, which ia a new logic programming paradigm based on Gelfond-Lifschitz's answer set semantics (originally defined as 'stable model semantics'). The method is based on: identifying the cycles contained in the program, showing that stable models of the overall program are composed of stable models of suitable subprograms, corresponding to the cycles; defining the Cycle Graph, *where each vertex corresponds to one cycle, and each edge corresponds to one* handle, *which is a literal containing an atom that, occurring in both cycles, actually determines a connection between them. Several properties of a program, including consistency, can be checked on its Cycle Graph.*

*We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

1 Introduction

Answer Set Programming is a branch of Logic Programming based on the Stable Models and Answer Sets declarative semantics defined by Gelfond and Lifschitz [8] [9]. While in standard logic programming program statements designate properties of a first-class object which is to be computed, Answer Set Programming (ASP) is based on the understanding of program statements as constraints on a set of atoms that encode a solution to the problem at hand.

Several implemented systems for answer sets computations are now available [13] and their performance is rapidly improving, approaching that of state-of-art SAT model checkers. The programmer shall write the logic program π with negation-as-failure but without function symbols. By feeding π to an ASP solver, one obtains the *answer sets*, and solutions to given problem can be *read off* the answer sets. Several answer sets corresponds to several different solutions. However, as it is well-known, under the answer set semantics a program may have no answer sets, i.e. it is not necessarily *consistent*.

In this paper we propose a static analysis methodology for Answer Set Programming. The method is based on the following points. (1) Identifying the cycles contained in the program, showing that stable models of the overall program are composed of stable models of suitable subprograms, corresponding to the cycles. (2) Defining the *Cycle Graph,* where each vertex corresponds to one cycle, and each edge corresponds to one *handle,* which is a literal containing an atom that, occurring in both cycles, actually determines a connection between them. In particular, existence of stable models is guaranteed if and only if for every odd cycle we can find a subgraph with certain properties. (3) Providing a procedure for checking consistency on the Cycle Graph without actually computing the answer sets. In fact, the necessary and sufficient condition that we introduce is syntactic in the sense that it does not refer either to models or derivations.It just requires to look at the program (represented by the Cycle Graph) and at the rules

composing the cycles.

Checking for the existence of stable models is a hard a computational problem (in fact, NP-complete) [**?**]. Being able to check for the existence of stable models syntactically for every answer set program can be useful in practice in many ways. In [4] it is argued that the approach can be useful for improving answer set computation, for defining classes of programs that are consistent by construction, and as a first step toward a component-based methodology for the construction and analysis of answer set programs. Here we show how it can be in principle used for checking many properties, e.g. for detecting whether an atom can belong to some answer set.

The method is presented for programs which are in a *canonical form* to which any logic program can be reduced by means of a tractable transformation [6]. Canonical programs focus the attention on cyclic dependencies. Rules are kept short, so as to make the syntactic analysis of the program easier. The stable models of any general logic program coincide (up to the language) to those of the corresponding canonical program. We will shortly discuss how the method might be rephrased for general programs, even in non-grounded form.

2 Cycles and Handles

We assume the standard definitions of a general logic program Π (or simply "logic program") [10], well-founded [12] stable model [8] and answer set semantics [9]. Whenever we mention consistency (or stability) conditions, we refer to the conditions introduced in [8]. For the sake of simplicity, we consider here the definition of *stable model* instead of that of answer set, which is an extension given for programs that contain the explicit negation operator $\neg$. In fact, this is not going to make a difference in the context of this work, and we will often interchange the terms "stable models" and "answer sets". The methodology proposed in this paper is based on identifying the negative cycles contained in the program.

Definition 1 *A set of rules C is called a* negative cycle, *or for short a* cycle, *if it has either the form*

$\lambda_1 \leftarrow not\ \lambda_2, \Delta_1$
$\lambda_2 \leftarrow not\ \lambda_3, \Delta_2$
$\ldots$
$\lambda_n \leftarrow not\ \lambda_1, \Delta_n$

where $n > 1$ and $\lambda_1, \ldots, \lambda_n$ are distinct atoms or the form

$\lambda_1 \leftarrow not\ \lambda_1, \Delta_1$

Each Δ_i, $i \leq n$, is a (possibly empty) conjunction $\delta_{i_1}, \ldots, \delta_{i_h}$ of literals (either positive or negative), where for each δ_{i_j}, $i_j \leq i_h$, $\delta_{i_j} \neq \lambda_i$ and $\delta_{i_j} \neq not\ \lambda_i$. The Δ_i's are called the AND handles *of the cycle. We say that Δ_i is an AND handle for atom λ_i, or, equivalently, an AND handle referring to λ_i.*

We say that *C* has size *n* and it is even (respectively odd) if $n = 2k$, $k \geq 1$ (respectively, $n = 2k + 1$, $k \geq 0$). For $n = 1$ we have the (odd) self-loop.

A *positive cycle* is similar to a negative cycle, except that we have positive literals λ_i's in the body of rules instead of negative ones. In the rest of the paper we will consider negative cycles unless differently specified explicitly.

For any cycle *C*, we will denote by *Composing_atoms*(C) the set $\{\lambda_1, \ldots, \lambda_n\}$, i.e., the set of atoms that *occur* in cycle *C*. We say that the rules listed above are *involved* in the cycle, or *form* the cycle. In the rest of the paper, sometimes it will be useful to see *Composing_atoms*(C) as divided into two subsets, that we indicate as two *kinds* of atoms: the set of the *Even_atoms(C)* composed of the λ_i's with *i* even, and the set *Odd_atoms(C)*, composed of the λ_i's with *i* odd. Notice that the sets of atoms composing different cycles are not required to be disjoint. In fact, the same atom may be involved in more than one cycle.

Conventionally, in the rest of the paper *C* and C_i denote cycles in general, *OC* and OC_i denote odd cycles, and *EC* or EC_i denote even cycles.

Definition 2 *A rule is called an* auxiliary rule of cycle *C if it is of this form:*
$\lambda_i \leftarrow \Delta$
where $\lambda_i \in$ Composing_Atoms(C), and Δ is a non-empty conjunction $\delta_{i_1}, \ldots, \delta_{i_h}$ of literals where for each $i_j \leq i_h$ either $\delta_{i_j} = \alpha_{i_j}$, or $\delta_{i_j} = not\ \alpha_{i_j}$ for some atom α_{i_j} and $\alpha_{i_j} \notin$ Composing_atoms(C). Δ is called an OR handle *of cycle C (more specifically, an OR handle for λ_i or, equivalently, and OR handle referring to λ_i).*

A cycle may possibly have several auxiliary rules, corresponding to different OR handles. In the following, we will call *Aux(C)* the set of the auxiliary rules of a cycle *C*. A cycle with no AND handles and no OR handles is called *unconstrained.*

Consider for instance the following program π.

— OC_1
$q \leftarrow not\ q$
—— $Aux(OC_1)$
$q \leftarrow f$
— OC_2
$p \leftarrow not\ p, not\ f$
— EC_1
$e \leftarrow not\ f$
$f \leftarrow not\ e$

It can be seen as divided into the following parts, each one corresponding to $C_i + Aux(C_i)$ for cycle C_i. The first part is composed of odd cycle OC_1, with an auxiliary rule (OR handle):

$q \leftarrow not\ q$
$q \leftarrow f$

The second part is composed of odd cycle OC_2, without auxiliary rules but with an AND handle:

$p \leftarrow not\ p, not\ f$

The third part is composed of the unconstrained even cycle EC_1:

$e \leftarrow not\ f$
$f \leftarrow not\ e$

Notice that the basic definition of a cycle corresponds to that of a *negative cycle* in the *Atom Dependency Graph* as defined in [7]. However, as discussed in [3], on the dependency graph it impossible to identify the handles, and there are different programs with different answer set, but the same dependency graph. Cycles and handles can be identified unambiguously on the *Extended Dependency Graph* as defined and discussed in [1] and [5].

In the following, without loss of generality [6] we refer to logic programs which are in the *canonical form* defined below. All definitions and results introduced in the rest of the paper might be rephrased for the general case, but the choice of referring to programs in canonical form is a significant conceptual simplification that leads without loss of generality to a more readable and intuitive formalization.

Definition 3 *A logic program* Π *is in canonical form (or, equivalently,* Π *is a canonical program) if it is WF-irreducible (i.e., all its atoms are deemed undefined under the well-founded semantics), and fulfills the following syntactic conditions.*

1. Π *does not contain positive cycles;*
2. *every atom in* Π *occurs both in the head of some rule and in the body of some (possibly the same) rule;*
3. *every atom in* Π *is involved in some cycle;*
4. *each rule of* Π *is either involved in a cycle, or is an auxiliary rule of some cycle;*
5. *each (AND/OR) handle of a cycle C consists of exactly one literal, either* α *or not* α*, where atom* $\alpha \notin$ *Composing_atoms*(C).

The above definition implies that in a canonical program Π: (i) the body of each rule which is involved in a cycle consists of either one or two literals; (ii) the body of each rule which is an auxiliary rule of some cycle consists of exactly one literal. For instance, program π introduced before is in canonical form.

3 Active handles and consistency

Truth or falsity of the atoms occurring in the handles of a cycle (w.r.t. a given set of atoms) affects truth/falsity of the atoms involved in the cycle. This creates the conditions for stable models to exist or not. The idea is formalized in the following definitions of *active handles.*

Definition 4 *Let* $\mathcal{I}$ *be a set of atoms. An AND handle* Δ *of cycle C is* active *w.r.t.* $\mathcal{I}$ *if it is false w.r.t.* $\mathcal{I}$*. We say that the rule where the handle occurs has an* active AND handle*. An OR handle* Δ *of cycle C is* active *w.r.t.* $\mathcal{I}$ *if it is true w.r.t.* $\mathcal{I}$*. We say that the rule where the handle occurs has an* active OR handle*.*

Assume that $\mathcal{I}$ is a model. We can make the following observations. (i) The head λ of a rule ρ with an active AND handle is not required to be true in $\mathcal{I}$. (ii) The head of a rule $\lambda \leftarrow \Delta$ where Δ is an active OR handle is necessarily true in $\mathcal{I}$: since the body is true, the head λ must also be true.

As we will see, the active handles of a cycle C gives relevant indications about whether a set of atoms $\mathcal{I}$ is a stable model.

Given cycle C, let H_C be the set of the handles of C, which are either of the form $(\Delta : AND : \beta)$ or of the form $(\Delta : OR : \beta)$, where $\beta \in$ *Composing_Atoms*(C): Whenever we need not care about β we shorten $(\Delta : K : \beta)$ as $(\Delta : K)$, K = AND/OR. We call "handles" the expressions in both forms, and whenever necessary we implicitly shift from one form to the other one. Informally, we will say for instance "the OR (resp. AND) handle Δ of β" meaning $(\Delta : OR : \beta)$ (resp. $(\Delta : AND : \beta)$). In general however the indication of β is necessary. In fact, different atoms of a cycle may have handles with the same Δ, but although active/not active at the same time, they may affect the existence of stable models differently.

Given any subset Z of H_C, it is useful to identify the set of atoms occurring in the handles belonging to Z, that we call *Atoms*(Z). Given any subset Z of H_C, let *ActivationAt*$_C(Z)$ the atoms that are required to be true, in order to make all the handles in Z active (implicitly, to this aim all the other atoms are required to be false). Vice versa, any subset V of *Atoms*(H_C) corresponds to a subset of the handles of C, called *Active*$_C(V)$, that become active, if atoms in V are true.

Finally, it is useful to introduce a short notation for the union of different sets of rules. By $I_1 + \ldots + I_q$, with $I_1, \ldots I_q$ sets of rules, we mean the program consisting of the union of all the rules belonging to $I_1, \ldots I_q$. As a special case, some of the I_j's can be sets of atoms, where each atom $\beta \in I_j$ is understood as a fact.

In [4] it is proved that for checking whether a logic program has stable models (and, possibly, for finding these models) one can do the following (where we let $X_i \subseteq$ *Atoms*(H_{C_i}), and we assume to add atoms in X_i as facts to $C_i +$ *Aux*(C_i)).

(i) Divide program Π into pieces, of the form $C_i +$ *Aux*(C_i), and check whether every odd cycle has handles; if not, then the program is inconsistent.

(ii) For every cycle C_i with handles, find the sets X_i that make the subprogram $C_i +$ *Aux*(C_i) consistent, and find the stable models S_{C_i} of each $C_i +$ *Aux*$(C_i) +$

X_i. Each S_{C_i} is called a partial stable model[1] of Π relative to C, and X_i is called its *base*. Notice that in the case of unconstrained even cycles, H_{C_i} is empty, and we have two stable models, namely $M^1_{C_i}$ = Even_atoms(C_i) and $M^2_{C_i}$ = Odd_atoms(C_i).

(iii) Check whether there exists a collection of X_i's, one for each cycle, such that the corresponding S_{C_i}'s form a collection of partial stable models which is *compatible*, i.e., which fulfills conditions (1)-(3) that follow. (1) If some atom A occurs in two cycles C_i and C_j, then their bases X_i and X_j must agree A, i.e., $A \in X_i$ if and only if $A \in X_j$. (2) If an atom A is supposed to be true in a base X_j of some cycle C_j, then it must be actually concluded true in some other cycle C_h. Notice that "concluded" does not mean "assumed", and thus A must occur in the partial stable model S_h of C_h without being in its base X_h. (3) If an atom A is supposed to be false in the base of some cycle C_j, it cannot be concluded true in any of the other cycles, i.e., A does not occur in the partial stable model S_k of any C_k. If the above conditions are fulfilled, then the program is consistent, and its stable model(s) can be obtained as the union of the S_{C_i}'s.

To show how the method works, consider program π introduced before. Cycle OC_1 in itself is inconsistent, but if we take $X_{OC_1} = \{f\}$ we get the partial stable model $\{f, q\}$: the active OR handle forces q to be true. Similarly, if we take for OC_2 $X_{OC_2} = \{f\}$, we get the partial stable model $\{f\}$: the active AND handle forces p to be false. Cycle EC_1 is consistent, with partial stable models $\{e\}$ and $\{f\}$. If we now select the partial stable model $\{f\}$, we get a compatible set of partial stable models thus obtaining the stable model $\{f, q\}$ for the overall program.

Instead, the partial stable model $\{e\}$ for EC_1 does not serve to the purpose of obtaining a stable model for the overall program, since atom f, which is in the positive base of both the odd cycles, is not concluded true in this partial stable model. Therefore, with this choice the handles of the odd cycles are not active and no overall consistency can be achieved.

4 Handle assignments and admissibility

In previous sections we have discussed how to split a stable model of a program into a compatible set of partial stable models of the cycles. In this section we define syntactic conditions that specify how active handles affect consistency of extended cycles.

Notice that the cycles where it is possible to derive an atom α are the cycles α is involved in, which are the cycles the handle Δ *comes from,* or equivalently the *source cycles* of the handle. Handles in H_C are called the *incoming handles* of C. The same handle of a cycle C may come from different cycles, and may refer to different atoms of C. Vice versa, the set *Out_handles*(C) denotes the atoms involved in C that occur in the handles of some other cycle.

Moreover, we say that:
$(\alpha : AND)$ and $(\alpha : OR)$ are opposite handles;
$(not\ \alpha : AND)$ and $(not\ \alpha : OR)$ are opposite handles;
$(\alpha : AND)$ and $(not\ \alpha : AND)$ are contrary handles;
$(\alpha : OR)$ and $(not\ \alpha : OR)$ are contrary handles;
$(\alpha : OR)$ and $(not\ \alpha : AND)$ are sibling handles;
$(\alpha : AND)$ and $(not\ \alpha : OR)$ are sibling handles;

Below we introduce the definition of *handle assignment*, which is a consistent hypothesis on (some of) the handles of a cycle C. Namely, it is a quadruple composed of the following sets. IN^A_C contains the incoming handles which are assumed to be active. From IN^A_C one can immediately derive a corresponding assumption on X_C. In particular, $X_C = ActivationAt_C(IN^A_C)$, i.e. it is exactly the set of the atoms that make the handles in IN^A_C active. Vice versa, IN^N_C contains the incoming handles which are assumed to be not active. Handles of C which are not in $IN^A_C \cup IN^N_C$ can be either active or not active, but their status is either unknown or irrelevant in the context where the handle assignment is used.

OUT^+_C is the set of out-handles which are required to be concluded true. This in order to make some handle of some other cycle active, as we have seen in the example above. Similarly, OUT^-_C is the set of the out-handles which are required to be concluded false, for the same reason. Of course, the OUT_C's must be disjoint, since no atom can be required to be simultaneously true and false.

Definition 5 *A* basic handle assignment *to (or for) cycle C is a quadruple of sets*

$$\langle IN^A_C, IN^N_C, OUT^+_C, OUT^-_C \rangle$$

where the (possibly empty) composing sets are such that:
$IN^A_C \cup IN^N_C \subseteq H_C$;
$IN^A_C \cap IN^N_C = \emptyset$;
neither IN^A_C and IN^N_C contain pairs of either opposite or contrary handles;
$OUT^+_C \cup OUT^-_C \subseteq$ *Out_handles*(C);
$OUT^+_C \cap OUT^-_C = \emptyset$.

For short, when talking of both IN^A_C and IN^N_C we will say "the IN_C's".

Definition 6 *A handle assignment will be called* trivial *if $OUT^+_C = OUT^-_C = \emptyset$, otherwise it will be called* non-trivial.

In a trivial handle assignment, no requirement is specified on the out-handles of C.

Definition 7 *A handle assignment will be called* effective *whenever $IN^A_C \neq \emptyset$, otherwise it will be called* non-effective.

[1] this term has been used in [10] with a very different meaning.

If IN_C^A is empty, there are two possible situations. (i) $H_C = \emptyset$, i.e., the cycle is unconstrained. (ii) $H_C \neq \emptyset$ but no active incoming handle is assumed: in this case, we say that the cycle is *actually unconstrained* w.r.t. this handle assignment.

We have to cope with the relationship between opposite, contrary, and sibling handles, whenever they should occur in the same cycle C. Notice that two opposite or contrary handles are never simultaneously active, i.e., whenever on is active the other one is not active, and vice versa. Instead, two sibling handles are either both active or both not active.

Definition 8 *Let: h and h^- be a pair of opposite handles; h and h^n be a pair of contrary handles; and h and h^s be a pair of sibling handles. A* complete handle assignment*, or simply a* handle assignment*, to cycle C is a basic handle assignment to C where, for each pair of opposite, contrary or sibling handles the occur in C, the following conditions hold:*
$h \in IN_C^A$ if and only if $h^- \in IN_C^N$;
$h \in IN_C^A$ if and only if $h^n \in IN_C^N$;
either $h, h^s \in IN_C^A$ and $h, h^s \notin IN_C^N$ or $h, h^s \in IN_C^N$ and $h, h^s \notin IN_C^A$.

A basic handle assignment can be *completed*, i.e., turned into a complete handle assignment, by an obvious update of the IN_C's.

What the definition does not state yet is that IN_C's and the OUT_C's should be compatible, in the sense that the handles in IN_C^A and IN_C^N being active should not prevent the out-handles in OUT_C's from being true/false as required. This is formalized in the following:

Definition 9 *A handle assignment $HA = \langle IN_C^A, IN_C^N, OUT_C^+, OUT_C^- \rangle$ to a cycle C is* admissible *if and only if the program $C + Aux(C) + ActivationAt_C(IN_C^A)$ is consistent, and for some stable model $S^{IN_C^A}$ of this program, $OUT_C^+ \subseteq S^{IN_C^A}$ and $OUT_C^- \cap S^{IN_C^A} = \emptyset$. We say that $S^{IN_C^A}$* corresponds *to HA.*

It is easy to see [4] that a non-effective handle assignment cannot be admissible for an odd cycle, and is admissible for an even cycle C if and only if either $OUT_C^+ \subseteq Even\_atoms(C)$ or $OUT_C^+ \subseteq Odd\_atoms(C)$.

Observe that whenever a handle assignment is effective the corresponding program fragment is locally stratified, and thus, according to [10], has a unique stable model that coincides with its well-founded model. It may also be observed that a trivial handle assignment, which does not state requirements on the out-handles, is always admissible for an even cycle, and it is admissible for an odd cycle only if it is effective (otherwise as seen before the cycle is inconsistent).

The admissibility of a non-trivial effective handle assignment for cycle C can be checked syntactically, by means of the criterion that we state below. The advantage of this check is that it does not require to compute the well-founded model of $C + Aux(C) + ActivationAt_C(IN_C^A)$, but it just looks at the rules of C. Although the syntactic formulation may seem somewhat complex, it simply states in which cases an atom in OUT_C^+, which is required to be concluded true w.r.t. the given handle assignment (or, conversely, an atom in OUT_C^- which is required to be concluded false), is actually allowed to take the specified truth value without raising inconsistencies. Notice that OUT_C^+ and OUT_C^- must be mutually coherent, in the sense that truth of an atom in OUT_C^+ cannot rely on truth of an atom in OUT_C^- (that instead is required to be concluded false), and vice versa.

Proposition 1 *A non-trivial effective handle assignment $\langle IN_C^A, IN_C^N, OUT_C^+, OUT_C^- \rangle$ to cycle C is admissible if and only if for every $\lambda_i \in OUT_C^+$ the following condition 1 holds, and for every $\lambda_k \in OUT_C^-$ the following condition 2 holds.*

- *Condition 1.*
  - *Either there exists an OR handle h_o for λ_i, $h_o \in IN_C^A$ or*
  - *for every AND handle h_a for λ_i, $h_a \in IN_C^N$ and $\lambda_{i+1} \notin OUT_C^+$, and condition 2 holds for λ_{i+1}.*
- *Condition 2.*
  - *For every OR handle h_o for λ_k, $h_o \in IN_C^N$, and*
  - *either there exists an AND handle h_a for λ_k such that $h_a \in IN_C^A$, or $\lambda_{k+1} \notin OUT_C^-$, and condition 1 holds for λ_{k+1}.*

The fact that conditions 1 and 2 refer to each other is not surprising, since they are to be applied to cycles. Notice however that since the given handle assignment is effective, some of the handles of the cycle are assumed to be active in this assignment. This means that some of the λ's is forced to be either true (by an OR handle) or false (via an AND handle). Then, the program $C + Aux(C) + ActivationAt_C(IN_C^A)$ is consistent. What conditions 1 and 2 check is simply that the given active/non-active handles are make the resulting partial stable model contain the OUT_C^+'s but not the OUT_C^-'s. Admissibility of a handle assignment in fact intuitively means that what is given "in input" to a cycle actually entails what is expected "in output".

Definition 10 *An admissible handle assignment $\langle IN_C^A, IN_C^N, OUT_C^+, OUT_C^- \rangle$ is* minimal *if there is no other sets $IN_C^{A'} \subset IN_C^A$ and $IN_C^{N'} \subset IN_C^N$ such that $\langle IN_C^{A'}, IN_C^{N'}, OUT_C^+, OUT_C^- \rangle$ is still admissible.*

There can be alternative minimal sets of incoming active handles for the same out-handles. However, there may also be the case there is none.

5 Cycle Graph and support sets

In this section we introduce the Cycle Graph of a program, that represent cycles and handles, and we show that the concepts and principles that we have previously introduced allow us to define syntactic conditions for consistency on the Cycle Graph.

Definition 11 *Given program* Π*, the* Cycle Graph CG_Π*, is a directed graph defined as follows:*

- **Vertices.** *One vertex for each one of the cycles* $C_1, \ldots, C_w$ *that occur in* Π*. Vertices corresponding to even cycles are labeled as* EC_i*'s while those corresponding to odd cycles are labeled as* OC_j*'s.*
- **Edges.** *An edge* (C_j, C_i) *marked with* $(\Delta : K : \lambda)$ *for each handle* $(\Delta : K : \lambda) \in H_{C_i}$ *of cycle* C_i*, that comes from* C_j*.*

Each marked edge will be denoted by $(C_j, C_i|\Delta : K : \lambda)$, where either ($C_j$ or C_i or λ) will be omitted whenever they are clear from the context, and we may write for short $(C_j, C_i|h)$, h standing for a handle that is either clear from the context or does not matter in that point. An edge on the *CG* connects the cycle a handle comes from to the cycle to which the handle belongs.

The Cycle Graph of a program directly represents cycles, that correspond to its vertices. It also indirectly represents extended cycles, since its edges are marked by the handles. Paths on the Cycle Graph graph represent direct or indirect connections between cycles through the handles. In order to relate admissible handle assignments for the cycles of Π to subgraphs of its cycle graph CG_Π we introduce the following definitions.

Definition 12 *Given program* Π*, let a* CG support set *be a pair* $S = \langle ACT^+, ACT^- \rangle$ *of subsets of the handles marking the edges of* CG_Π *(represented in the form* $(\Delta : K)$ *with* $K = AND/OR$*) such that the following conditions hold:*
(i) $ACT^+ \cap ACT^- = \emptyset$*.*
(ii) if two opposite handles h and h^- *both occur on the CG, then* ACT^+ *contains handle h if and only if* ACT^- *contains its opposite handle* h^-*.*
(iii) if two contrary handles h and h^n *both occur on the CG, then* ACT^+ *contains handle h if and only if* ACT^- *contains its contrary handle* h^n*.*
(iv) if two sibling handles h and h^s *both occur on the CG, then either* $h, h^s \in ACT^+$ *and* $h, h^s \notin ACT^-$*, or vice versa* $h, h^s \in ACT^-$ *and* $h, h^s \notin ACT^+$

For given S, we will indicate its two components with $ACT^+(S)$ and $ACT^-(S)$. As stated in Section 3, we have to restrict the attention on CG support sets including at least one active handle for each odd cycle. Then, we have to check that the assumptions on the handles are mutually coherent, and are sufficient for ensuring consistency.

Definition 13 *A CG support set S is* potentially adequate *if for every odd cycle C in* Π *there exists a handle* $h \in H_C$ *such that* $h \in ACT^+(S)$*.*

A CG support set S induces a set of handle assignments, one for each of the cycles $\{C_1, \ldots, C_w\}$ occurring in Π.

The induced assignments are obtained on the basis of the following observations.

Each handle in $h \in ACT^+(S)$ is supposed to be active, and therefore it must be active for each cycle C_i such that $h \in H_{C_i}$.

Each handle in $h \in ACT^-(S)$ is supposed to be not active, and therefore it must be not active for each of cycle C_j such that $h \in H_{C_j}$.

If a handle h in S requires, in order to be active/not active, an atom β to be false, then it must be concluded false *in all the extended cycles of the program h* comes from.

If a handle h in S requires, in order to be active/not active, an atom β to be true, then it must be concluded true *in all the extended cycles of the program h* comes from. This point deserves some comment, since one usually assumes that it suffices to conclude β true *somewhere* in the program. Consider however that any rule $\beta \leftarrow$ *Body* that allows β to be concluded true in some cycle is an auxiliary rule to all the other cycles β is involved into. This is why β is concluded true *everywhere it occurs.* This is the mechanism for selecting partial stable models of the cycles that agree on shared atoms, in order to assemble stable models of the overall program.

Definition 14 *Let* $S = \langle ACT^+, ACT^- \rangle$ *be a CG support set which is potentially adequate. For each cycle* C_k *occurring in* Π*,* $k \leq w$*, the (possibly empty) handle assignment induced by this set is determined as follows.*

1. *Let* $IN^A_{C_k}$ *be* $H_{C_k} \cap ACT^+(S)$*.*
2. *Let* $IN^N_{C_k}$ *be* $H_{C_k} \cap ACT^-(S)$*.*
3. *Let* $OUT^+_{C_k}$ *be the (possibly empty) set of all atoms* $\beta \in Out\_handles(C_k)$ *such that there is a handle* $h \in ACT^+(S)$ *either of the form* $(\beta : OR)$ *or* $(not\ \beta : AND)$*.*
4. *Let* $OUT^-_{C_k}$ *be the (possibly empty) set of all atoms* $\alpha \in Out\_handles(C_k)$ *such that there is a handle* $h \in ACT^-(S)$ *either of the form* $(\alpha : AND)$ *or* $(not\ \alpha : OR)$*.*
5. *Verify that* $OUT^-_{C_k} \cap OUT^+_{C_k} = \emptyset$*.*

If this is the case for each C_k*, then S actually induces a set of handle assignments, and is called* coherent*. Otherwise, S does not induce a set of handle assignments, and is called* incoherent*.*

The above definition does not guarantee that the assignments induced by a coherent support set are admissible, that the same atom is not required to be both true and false

in the assignments of different cycles, and that the incoming handles of a cycle being supposed to be active/not active corresponds to a suitable setting of the out-handles of the cycles they come from. Consider for instance cycle C_i which has an incoming handle, e.g. $h = (\beta : OR : \lambda)$, in $IN^A_{C_i}$: h is supposed to be active, which in turn means that β must be concluded true elsewhere in the program; then, for all cycles C_j where β is involved into, we must have $\beta \in OUT^+_{C_j}$, in order to fulfill the requirement. Of course, we have to consider both IN^A_C and IN^C_N, and both the AND and the OR handles.

The following definition formalizes this more strict requirements.

Definition 15 *A coherent CG support set S is* adequate *(w.r.t. not adequate) if for the induced handle assignments the following conditions hold:*

1. *they are admissible;*
2. *for each two cycles C_i, C_j in Π, $OUT^+_{C_i} \cap OUT^-_{C_j} = \emptyset$.*
3. *For every C_i in Π, for every handle $h \in IN^A_{C_k}$ of the form either $(\beta : OR : \lambda)$ or $(not\ \beta : AND : \lambda)$, and for every handle $h \in IN^N_{C_k}$ of the form either $(\beta : AND : \lambda)$ or $(not\ \beta : OR : \lambda)$, for every other cycle C_j in Π, $i \neq j$, such that $\beta \in Out\_handles(C_j)$, we have $\beta \in OUT^+_{C_j}$.*
4. *For every C_i in Π, for every handle $h \in IN^A_{C_k}$ of the form either $(not\ \beta : OR : \lambda)$ or $(\beta : AND : \lambda)$, and for every handle $h \in IN^N_{C_k}$ of the form either $(not\ \beta : AND : \lambda)$ or $(\beta : OR : \lambda)$, for every other cycle C_j in Π, $i \neq j$, such that $\beta \in Out\_handles(C_j)$, we have $\beta \in OUT^-_{C_j}$.*

In [4] we have introduced a necessary and sufficient syntactic condition for consistency based on the Cycle Graph of the program.

Theorem 1 *A program Π has stable models if and only if there exists and adequate CG support set S for Π.*

Therefore, it is useful to define a procedure for identifying adequate support sets of a program on its Cycle Graph.

6 Identifying adequate support sets on the Cycle Graph

The above definitions allow us to define a procedure for trying to *find* adequate support sets S starting from the odd cycles, and following the dependencies on the *CG*.

For the sake of readability we introduce some simplifying assumptions.

- Given handle $h = (\Delta : K : \lambda)$, by $ACT^+(S) \cup \{h\}$ (resp. $ACT^-(S) \cup \{h\}$) we mean $ACT^+(S) \cup \{(\Delta : K)\}$ (resp. $ACT^+(S) \cup \{(\Delta : K)\}$).
- Given handle $h \in ACT^+(S)$ (resp. $h \in ACT^-(S)$) of the form $(\Delta : K)$, by $IN^A_C \cup \{h\}$ (resp. $IN^N_C \cup \{h\}$) we mean: to identify the set $H = \{(\Delta : K : \lambda) \in H_C\}$ and perform $IN^A_C \cup H$ (resp. $IN^N_C \cup H$).
- By $H_C \cap ACT^+(S)$ (resp. $H_C \cap ACT^+(S)$) we mean $\{(\Delta : K : \lambda) \in H_C | (\Delta : K) \in ACT^+(S)\}$ (resp. $(\Delta : K) \in ACT^+(S)$).

Assumption 1 Whenever for handle h either h^s or h^- or h^n or some of them occur on the *CG*, by $A \cup \{h\}$ (where A can be either $ACT^+(S)$ or $ACT^-(S)$ or IN^A_C or IN^N_C for some cycle C) we implicitly mean that the sibling handle is added to A as well, while the opposite and contrary handles are added to the "opposite" set.

Definition 16 Procedure PACG for finding adequate CG support sets for program Π

1. *Let initially $S = \langle \emptyset; \emptyset \rangle$.*
2. *For each cycle C_k occurring in Π, $k \leq w$, let initially $HA_{C_k} = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$*
3. *For each odd cycle OC in Π do:*
   (a) *Choose $h \in H_{OC}$. If $H_{OC} = \emptyset$, than FAIL.*
   (b) *For chosen h:*
      i. *$ACT^+(S) := ACT^+(S) \cup \{h\}$;*
      ii. *for each cycle C_k in Π such that $h \in H_{C_k}$: $IN^A_{C_k} := IN^A_{C_k} \cup h$;*
      iii. *If h is either of the form $(\beta : OR)$ or $(not\ \beta : AND)$, for each cycle C_k in Π where $\beta \in Out\_handles(C_k)$, do: $OUT^+_{C_k} := OUT^+_{C_k} \cup \{\beta\}$;*
      iv. *If h is either of the form $(not\ \beta : OR)$ or $(\beta : AND)$, for each cycle C_k in Π where $\beta \in Out\_handles(C_k)$, do: $OUT^-_{C_k} := OUT^-_{C_k} \cup \{\beta\}$*
4. *REPEAT*
   (a) *Verify that $ACT^+(S) \cap ACT^-(S) = \emptyset$. If not, FAIL.*
   (b) *For each cycle C_k in Π such that $OUT^+_{C_k} \neq \emptyset$ or $OUT^-_{C_k} \neq \emptyset$:*
      i. *Verify that $OUT^+_{C_k} \cap OUT^-_{C_k} = \emptyset$. If not, FAIL.*
      ii. *Update (if needed) $IN^A_{C_k}$ and $IN^N_{C_k}$ w.r.t. $OUT^+_{C_k}$ and $OUT^-_{C_h}$, and check that the resulting handle assignment is admissible. If not, FAIL.*
      iii. *For each other cycle C_h in Π do: verify that $OUT^+_{C_k} \cap OUT^-_{C_h} = \emptyset$, and that $OUT^-_{C_k} \cap OUT^+_{C_h} = \emptyset$. If not, FAIL.*
   (c) *For each cycle C_k in Π, for each $h \in IN^A_{C_k}$:*

i. *do* $ACT^+(S) := ACT^+(S) \cup \{h\}$;

ii. *For each cycle* C_h *in* Π *such that* $h \in H_{C_h}$: $IN^A_{C_h} := IN^A_{C_h} \cup h$;

(d) If h is either of the form $(\beta : OR)$ *or* $(not\ \beta : AND)$, *for each cycle* C_k *in* Π *where* $\beta \in Out\_handles(C_k)$, *do:* $OUT^+_{C_k} := OUT^+_{C_k} \cup \{\beta\}$;

(e) If h is either of the form $(not\ \beta : OR)$ *or* $(\beta : AND)$, *for each cycle* C_k *in* Π *where* $\beta \in Out\_handles(C_k)$, *do:* $OUT^-_{C_k} := OUT^-_{C_k} \cup \{\beta\}$

5. *For each cycle* C_k *in* Π, *for each* $h \in IN^N_{C_k}$:

(a) do $ACT^-(S) := ACT^-(S) \cup \{h\}$;

(b) For each cycle C_h *in* Π *such that* $h \in H_{C_h}$: *do* $IN^N_{C_h} := IN^N_{C_h} \cup h$;

(c) If h is either of the form $(\beta : OR)$ *or* $(not\ \beta : AND)$, *for each cycle* C_k *in* Π *where* $\beta \in Out\_handles(C_k)$, *do:* $OUT^-_{C_k} := OUT^-_{C_k} \cup \{\beta\}$;

(d) If h is either of the form $(not\ \beta : OR)$ *or* $(\beta : AND)$, *for each cycle* C_k *in* Π *where* $\beta \in Out\_handles(C_k)$, *do:* $OUT^+_{C_k} := OUT^+_{C_k} \cup \{\beta\}$

UNTIL no set is updated by the previous steps.

Proposition 2 *Procedure PACG either fails, or returns an adequate CG support set.*

7 Example

Consider the following collection of cycles.

— OC_0	— OC_1	— OC_3
$p \leftarrow not\ s, not\ c$	$p \leftarrow not\ s, not\ c$	$r \leftarrow not\ r, not\ e$
$s \leftarrow not\ t$	$s \leftarrow not\ t$	
$t \leftarrow not\ p$	$t \leftarrow not\ p$	
	$s \leftarrow a$	

— OC_2	— OC'_2
$q \leftarrow not\ q$	$q \leftarrow not\ q$
$q \leftarrow not\ e$	$q \leftarrow not\ e$
	$q \leftarrow a$

— EC_1	— EC_2	— EC_3
$a \leftarrow not\ c$	$a \leftarrow not\ b$	$e \leftarrow not\ f$
$c \leftarrow not\ a$	$b \leftarrow not\ a$	$f \leftarrow not\ e$

Let $\pi_1 = OC_0 \cup EC_1$, $\pi_2 = OC_1 \cup EC_1 \cup EC_2$, $\pi_3 = OC_1 \cup EC_1 \cup EC_2 \cup EC_3 \cup OC_2 \cup OC_3$, and $\pi_4 = OC_1 \cup EC_1 \cup EC_2 \cup EC_3 \cup OC'_2 \cup OC_3$. The Cycle Graphs of these programs are shown in Figures 1, 2, 3 and 4 respectively.

Let's now apply the definition of handle assignment.

For $\pi_1 = OC_0 \cup EC_1$, the odd cycle OC_0 admits the unique potentially active handle $(not\ c : AND : p)$ Then, we let S_{π_1} be such that $ACT^+(S_{\pi_1}) = \{(not\ c : AND)\}$ and $ACT^-(S_{\pi_1}) = \emptyset$. The induced sets of handle assignments are as follows.

For OC_0: $IN^A_{OC_0} = \{(not\ c : AND)\}$, $OUT^+_{OC_0} = OUT^-_{OC_0} = \emptyset$. This assignment is trivially admissible,

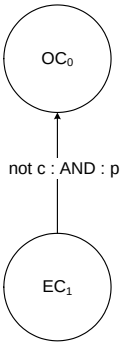

Figure 1: The Cycle Graph of π_1.

since there is no requirement on the out-handles. For EC_1: $OUT^+_{EC_1} = \{c\}$, $OUT^-_{EC_1} = \{\emptyset\}$. $IN_{EC_1} = \emptyset$, since EC_1 is unconstrained. It is easy to verify that this handle assignment is admissible, by letting $\lambda_1 = a$ and $\lambda_2 = c$, where of course for c to be true a must be false. This handle assignment corresponds to selecting the partial stable model $\{c\}$ for EC_1, while discarding the other partial stable model $\{a\}$. Then, S_{π_1} is an adequate CG support set.

Consider program $\pi_2 = OC_1 \cup EC_1 \cup EC_2$. Cycles EC_1 and EC_2 are not independent. In fact, rule $a \leftarrow not\ b$ of EC_2 is an auxiliary rule for EC_1, and, vice versa, $a \leftarrow not\ c$ of EC_1 is an auxiliary rule for EC_2. Then, here we have a cyclic connection between the even cycles. This is evident on the Cycle Graph of π_2, reported in Figure 2.

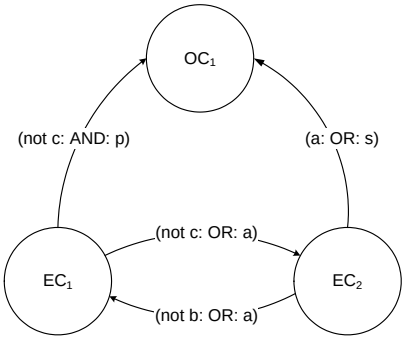

Figure 2: The Cycle Graph of π_2.

The odd cycle OC_1 has two handles, of which at least one must be active. Let us first assume that $(not\ c : AND : p)$ is active.

According to the PACG procedure, we try to assemble a CG support set S, by letting at first $ACT^+(S_{\pi_2}) = \{(not\ c : AND)\}$ and $ACT^-(S_{\pi_2}) = \{(not\ c : OR)\}$. In fact, since $not\ c$ is an incoming OR handle for a in EC_2, when assuming $(not\ c : AND)$ to be active, we also have to assume its opposite handle and its contrary handle to be not active.

Accordingly, we let $IN^A_{OC_1} = \{(not\ c : AND)\}$ and $IN^N_{EC_2} = \{(not\ c : OR)\}$ Now, we have to put $OUT^-_{OC_1} =$

$\{p\}$ and $OUT^{+}_{EC_1} = \{c\}$. To form an admissible handle assignment for EC_1, this implies to let $IN^{N}_{EC_1} = \{(not\ b : OR)\}$. Consequently, we have to update $ACT^{-}(S_{\pi_2})$ which becomes: $ACT^{-}(S_{\pi_2}) = \{(not\ c : AND), (not\ b : OR)\}$. This leads to put $OUT^{+}_{EC_1} = \{b\}$.

Further iteration of the procedure changes nothing, and thus the pair of sets $ACT^{+}(S_{\pi_2}) = \{(not\ c : AND)\}$ and $ACT^{-}(S_{\pi_2}) = \{(not\ c : OR), (not\ b : OR)\}$ form, as it is easy to verify, an adequate CG support set.

Notice that this kind of reasoning requires neither to find the stable models of the cycles, nor to consider every edge of the *CG*. In fact, we do not need to consider the second incoming handle of OC_1.

Let us now make the alternative assumption, i.e. assume that $(a : OR : s)$ is active for OC_1. This means at first $ACT^{+}(S_{\pi_2}) = \{(a : OR)\}$ and $ACT^{-}(S_{\pi_2}) = \emptyset$, since *not a* does not occur in handles of the *CG*. This implies $OUT^{+}_{EC_1} = \{a\}$. Thus, there is no requirement on IN_{EC_2} for forming an admissible handle assignment, and then the procedure stops here.

For program $\pi_3 = OC_1 \cup EC_1 \cup EC_2 \cup EC_3 \cup OC_2 \cup OC_3$, the only incoming handles to OC_2 and OC_3 are opposite handles, that cannot be both active. For the other cycles, the situation is exactly the same as for π_2.

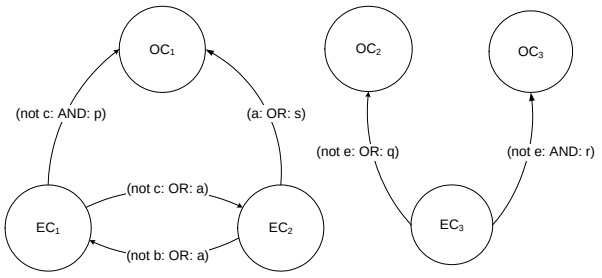

Figure 3: The Cycle Graph of π_3.

On its Cycle Graph (Figure 3) it is apparent that consistency problems of this program arise from subprogram $EC_3 \cup OC_2 \cup OC_3$. We can fix these problems for instance by replacing OC_2 with OC'_2, which means that we add an auxiliary rule (and then an OR handle) to OC_2. We thus obtain program π_4 (CG in Figure 4) where we can exploit handle $(a : OR)$ for both OC_1 and OC'_2. It is easy to verify that the CG support set S composed of $ACT^{+}(S_{\pi_4}) = \{(a : OR), (not\ e : AND)\}$ and $ACT^{-}(S_{\pi_4}) = \{(not\ e : OR)\}$ is adequate. The need to support OC'_2 rules out the possibility of supporting OC_1 by means of the handle $(not\ c : AND : p)$.

8 How to exploit the results

We have identified and discussed in depth two aspects of consistency checking: (1) the odd cycles must be (either directly or indirectly) supported by the even cycles; (2) this

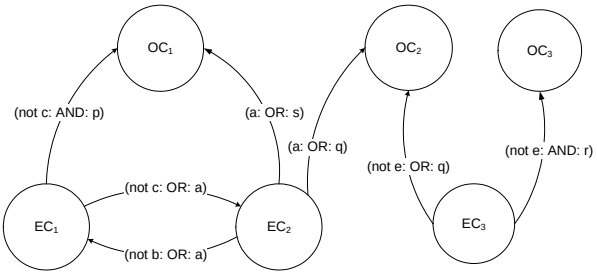

Figure 4: The Cycle Graph of π_4.

support must be consistent, in the sense that no contrasting assumptions on the handles can be made.

Point (1) is related to the "coarse" structure of the program, and can be easily checked on the *CG*, so as to rule out a lot of inconsistent programs, thus leaving only the "potentially consistent" ones to be checked w.r.t. point (2).

The former definitions and result and the PACG procedure can be adapted to perform several static analysis tasks, beyond checking consistency. For lack of space we explain what we mean by means of examples and then we introduce a preliminary result.

Let us for instance reconsider program π_2 in the above example. Let us assume we want to know whether atom *p* may belong to an answer set of π_2. Since *p* occurs in cycle OC_1, there must exist a non-trivial effective handle assignment for OC_1 such that $p \in OUT^{+}_{OC_1}$. From Proposition 1 we know that (Condition 1) either there exists an OR handle for *p* (part (a)) which is not the case here, or there is no active AND handle (part (b)) and the atom in the cycle (namely, *s*) upon which *p* depends is in $OUT^{-}_{OC_1}$. Then, on the cycle graph we rule out handle $(not\ c : AND : p)$ as a support for OC_1. However, the other possibility is $(a : OR : s)$. Again according to Proposition 1, with an active OR handle *s* cannot be in $OUT^{-}_{OC_1}$. Therefore, *p* cannot be in any answer set of π_2.

Take program π_4 in the above example. Can *a* be in an answer set of this program? PACG can check this, if we let the procedure choose $(a : OR)$ as a supporting handle for both OC_1 and OC_2. As seen at the end of previous section, with this choice an adequate CG support set can be constructed, and then the answer is yes. Instead, atom *b* cannot be in any answer set of this program. In fact, the only cycle where *b* occurs is OC_2. Again by Proposition 1 for *b* to be true (i.e., for *b* to be in $OUT^{+}_{EC_2}$ *a* must be false (condition 1.(b)). Then, handle $(a : OR)$ cannot be active. It is easy to check on the *CG* that, in this case, OC_2 and OC_3 should be supported by opposite handles, which cannot be the case.

An easy way to initialize PACG is to start with a non-empty initial candidate support set.

Definition 17 *Let PACGE be the same as PACG, where in step 1 we let initially* $S = \langle ACT^{+}_{init}, ACT^{-}_{init} \rangle$*, where* ACT^{+}_{init}

and ACT^-_{init} can be non-empty.

Proposition 3 *An atom A can belong to some of the answer sets of program* Π *only if the following conditions hold.*

(i) There exists a cycle C where A is involved into and a non-trivial effective handle assignment for C such that $A \in OUT^+_C$.

(ii) PACGE returns an adequate CG support set given initially $ACT^+_{init} = OUT^+_C$ *and* $ACT^-_{init} = OUT^-_C$.

9 Discussion

An issue to consider is how much space is consumed by the Cycle Graph. In the case of canonical programs, a given program is just split into parts, and then the Cycle Graph does not take more space than the program itself. Non-canonical programs instead, because of long rules may contain an exponential number of cycles. A worst case for negative programs is represented by *extremal programs* ChoTru96, where a rule may contain all the atoms which occur in the program. However, although an extensive study has to be performed, many odd cycles presumably do not really need to be represented in the Cycle Graph. For instance, in extremal programs all rules of each odd cycle also belong to even cycles, all distinct: thus, whatever the choice of the partial stable models of the even cycles, the odd cycle turns out to be supported.

But, how can the method proposed in this paper be extended to non-canonical programs? (1) If an OR handle is composed of several literals, they must be *all* true for the handle to be active. (2) If an AND handle is composed of several literals, *at least one* must be false for the handle to be active. (3) If the connection between cycles are not direct but there are chains of dependencies, in order to state whether a literal in a handle is true/false these chains of dependencies must be followed.

How can the method be applicable to analyze non-grounded programs? In principle, a version of the Cycle Graph can be built for non-grounded programs as well. Point (1) of consistency checking can be at least partly performed. However, either for performing point (2) or for other analysis tasks, at least part of the program must be grounded. How to optimize "on the flight" partial grounding is a future topic of this research.

References

[1] BRIGNOLI, G., COSTANTINI, S., D'ANTONA, O. AND PROVETTI, A., 1999. Characterizing and computing stable models of logic programs: the non–stratified case. In *Proc. of CIT99 Conference on Information Technology*.

[2] CHOLEWIŃSKI, P. AND TRUSZCZYŃSKI, M., 1999. Extremal problems in logic programming and stable model computation. *J. of Logic Programming*, 38, 219–242.

[3] COSTANTINI, S., 2001. Comparing different graph representations of logic programs under the answer set semantics. In *Proc. AAAI Spring Symposium "Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning"*, 21–26.

[4] COSTANTINI, S., 2005. On the existence of stable models of unstratified logic programs. In *To appear in: "Theory and Practice of Logic Programming"*.

[5] COSTANTINI, S., D'ANTONA, O. AND PROVETTI, A., 2002. On the equivalence and range of applicability of graph-based representations of logic programs. *Information Processing Letters 84(2)*, 241–249.

[6] COSTANTINI, S. AND PROVETTI, A., 2004. Normal forms for answer set programming. To appear in: *Theory and Practice of Logic Programming.*

[7] FAGES, F., 1994. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science* 2, 51–60.

[8] GELFOND, M. AND LIFSCHITZ, V., 1988. The stable model semantics for logic programming. In *Proc. of the Fifth Joint International Conf. and Symp.*. The MIT Press, Cambridge, MA, 1070–1080.

[9] GELFOND, M. AND LIFSCHITZ, V., 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

[10] PRZYMUSINSKA, H. AND PRZYMUSINSKI, T. C., 1990. Semantic issues in deductive databases and logic programs. *Formal Tech. in Art. Intell., a Sourcebook*, Elsevier Sc. Publ. B.V., 321–367.

[11] SCHLIPF, J. S., 1995. The expressive power of logic programming semantics. J. of Comp. and Syst. Sciences 51 (1), 64-86.

[12] VAN GELDER A., ROSS K.A. AND SCHLIPF J., 1990. The well-founded semantics for general logic programs. *J. of the ACM 38(3)*, 620–650.

[13] WEB LOCATION OF THE MOST KNOWN ASP SOLVERS.
Cmodels: *http://www.cs.utexas.edu/users/yuliya/*
Aspps: *http://www.cs.uky.edu/ai/aspps/*
DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
NoMoRe: *http://www.cs.uni-potsdam.de/~linke/nomore/*
Smodels: *http://www.tcs.hut.fi/Software/smodels/*