TR-0804

Magic Sets and Bottom-Up Evaluation of Stable Model

by

M. Fujita, N. Iwayama & R. Hasegawa

September, 1992

Magic Sets and Bottom-Up Evaluation of Stable Model

Masayuki FUJITA, Noboru IWAYAMA, and Ryuzo HASEGAWA

Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
email: mfujita@icot.or.jp

June 22, 1992

Abstract

We provide a query-answering procedure with respect to stable models for general logic programs. The procedure consists of a model construction part that computes proof tree for a query and a model check part that confirms that literals in the proof tree are included in some stable model. The model construction part is useful for reducing the number of irrelevant model generations to the query.

The most important characteristic of the procedure is that the procedure utilizes magic set technique in the model construction part. Although the model construction part is the top-down computation by nature, the computation is embedded into the bottom-up computation by magic set technique. Since the embedded bottom-up computation and the model check part are based on a bottom-up model generation theorem prover for first order formulas, the whole procedure computes in a bottom-up manner.

1 Introduction

For general logic programs, the stable model semantics was provided by Gelfond and Lifschitz [Gelfond88]. Stable model semantics is used in various fields such as deductive database and AI to formulate common-sense reasoning. Therefore, efficient procedures for computing stable models are needed.

There are two types of procedures to compute stable models; bottom-up and top-down. Bottom-up procedures construct stable models. Top-down procedures answer whether a goal is valid in some stable model. Sacca and Zaniolo [Sacca90] developed a simple bottom-up procedure. Satoh [Satoh91] and Iwayama [Iwayama91] developed bottom-up procedures, which were versions of Sacca's procedure but with some speed-up heuristics. Inoue [Inoue92] showed that general logic programs are translated into disjunctive logic programs and that minimal models for the disjunctive logic programs correspond to stable models for the original general logic programs. The minimal models for the disjunctive logic programs are computed by a model generation theorem prover. Since the model generation theorem prover computes in a bottom-up manner, Inoue's method is a bottom-up method.

Eshghi [Eshghi89] and Kakas [Kakas90] developed top-down procedures. Their procedures were not correct for all general logic programs, because the procedures return correct answers only for call-consistent programs. Satoh [Satoh92] developed a procedure which returns answers correctly for the program with at least one stable model; the procedure computes using both top-down and bottom-up techniques.

In this paper, we firstly give a naive query-answering procedure for range-restricted general logic programs. The procedure consists of a model construction part and a model check part.

The model construction part computes in a top-down manner from a query, and constructs a candidate subset of a stable model relevant to the query. The model check part checks that there is a stable model which includes the candidate subset obtained in the model construction part. Since the check part is based on Inoue's framework, the part is computed in a bottom-up manner by a model generation theorem prover.

The main point of this paper is that the model construction part in the naive procedure is embedded into a bottom-up procedure by magic set technique. Magic set technique in deductive database allows us to incorporate top-down information with bottom-up reasoning. In our case, magic set technique gives us this similar effect; the model construction part, which is computed in a top-down manner by nature, is simulated in bottom-up manner computation by means of magic rules.

Although the above naive procedure is not complete, the procedure with magic rules is complete if magic rules are still range-restricted. Since the top-down information by magic set technique cuts down the generation of irrelevant models to the query, the procedure with magic rules answers more efficiently than do bottom-up procedures without top-down information. Moreover, the procedure is very simple and small; it is written in only one page of Prolog code.

The rest of the paper is organized as follows: In section 2 we present our definitions and notation, and introduce the problem to be resolved. In section 3, we present a naive query-answering procedure which combines a bottom-up model check procedure with the Prolog interpreter. The naive procedure is rewritten to a bottom-up procedure with magic rules in section 4. In section 5, we conclude the paper and show many future directions for our work. In the appendixes, we show the Prolog codes of all the procedures as well as two examples.

2 General Logic Programs and Their Disjunctive Representations

We define general logic programs and their semantics. Then we show what we want to compute.

2.1 Basic Definitions

We give a definition of general logic programs. Later, general logic programs are transformed into disjunctive forms. Both original and transformed forms of general logic programs are used by the naive query-answering procedure in the next section.

Definition 2.1 *Let A_i be atoms. A* general logic program *is a set which consists of the following rules:*

$$A_0 \leftarrow A_1, ..., A_m, not\, A_{m+1}, ..., not\, A_n.$$

or integrity constraints *of the form:*

$$\leftarrow A_1, ..., A_m, not\, A_{m+1}, ..., not\, A_n.$$

where $n \geq m \geq 0$. In the case of $n = m = 0$, we call the rule a fact.

In the definition, a rule (or an integrity constraint) is *range-restricted*, if every variable in the rule (or integrity constraint) has at least one occurrence in the positive part of its body. If all rules and integrity constraints in a general logic program are range-restricted, we also say that the program is range-restricted.

Next, we gives the semantics of general logic programs, which is based on [Gelfond88].

Definition 2.2 *A stable model M for K is a Herbrand model satisfying the following conditions:*

1. *M is equal to the minimal Herbrand model for the ground Horn program K^M where $K^M = \{A_0 \leftarrow A_1, ..., A_m \mid A_0 \leftarrow A_1, ..., A_m, not\, A_{m+1}, ..., not\, A_n$ is a rule in ground(K) and $A_i \notin M$ for each $i = m+1, ..., n\}$.*

2. *For every ground integrity constraint $C \in ground(K)$, $M \models C$.*

Our objective is the *query-answering problem*, as follows. For a general logic program and a goal [1] (an atom), we want to find a stable model for the program such that the stable model satisfies the goal; if there exists such a stable model, then the stable model provides us with an answer substitution for the query.

2.2 Disjunctive Representations of General Logic Programs

The naive procedure for query-answering proposed in the next section uses two representation forms of rules. The forms of rules defined above are used for the (top-down) model construction. Transformed rules are used for the (bottom-up) model check. Original forms of integrity constraints are used only in the model check part.

Transformed rules for rules in a general logic program K are the following positive disjunctive clauses:

$$A_1, ..., A_m \rightarrow A_0, \sim A_{m+1}, ..., \sim A_n \mid \sim\sim A_{m+1} \mid ... \mid \sim\sim A_n.$$

In the above forms of rules, all negation-as-failure formula are removed. $\sim\sim A$ expresses that there is no proof for $\sim A$. In fact, $\sim\sim A$ expresses that A should be proven, because $\sim A$ means negation as failure. We call the transformed rules of K and integrity constraints in K as a *disjunctive program* $TR(K)$.

As original forms of general logic programs, the range-restrictedness of disjunctive programs should be considered. A disjunctive rule is range-restricted if every variable in the rule has at least one occurrence in its antecedents. If all rules and integrity constraints in a disjunctive program are range-restricted, we say that the program is range-restricted. If a general logic program is range-restricted, then the disjunctive program is also range-restricted.

To compute stable models for the original form of a general logic program K, we compute minimal models for $TR(K)$ which satisfy the next two conditions related to the symbol $\sim$:

- For a minimal model M for $TR(K)$, $A \in M$ implies $\sim A \notin M$. (1)
- For a minimal model M, $\sim\sim A \in M$ implies $A \in M$. (2)

The above transformation and the correspondence between stable models of general logic programs and minimal models of disjunctive programs are first considered in Inoue [Inoue92]. $\sim A$ and $\sim\sim A$ in our transformation correspond to $\neg\mathrm{K}A$ and $\mathrm{K}A$ respectively in Inoue's transformation.

3 Naive Procedure

In this section, we give a naive query-answering procedure which we will rewrite to a procedure using magic rules. We implement the procedure in Prolog and show the entire code.

[1] In general, a goal can be a conjunction of atoms and/or negated atoms. We can deal with such a conjunctive goal $L_0, ..., L_n$ by adding a new rule $A \leftarrow L_0, ..., L_n$ to the program and regarding a new atom A as a goal (atom A contains all the variables that occur in the conjunction).

```
vanilla(G,M):- vi(G,[],M).
vi(true,Mi,Mi):- !.
vi((G,GL),Mi,Mo):- !, vi(G,Mi,M1),vi(GL,M1,Mo).
vi((G1;G2),Mi,Mo):- !, vi(G1,Mi,Mo);vi(G2,Mi,Mo).
vi(~G,Mi,Mo):- !,(vi(G,[],_) -> fail;Mo=[~G|Mi]).
vi(G,Mi,[G|Mo]):- clause(G,SG),vi(SG,Mi,Mo).
```

Figure 1: Vanilla interpreter returning model candidate during proof tree construction

To solve a query-answering problem, we can use bottom-up procedures to compute stable models; the satisfiability of the goal is checked after the generation of stable models. However, this approach has the possibility of computing models irrelevant to the goal. To prevent the generation of irrelevant models, we change the order of the model construction and the satisfiability check. In the following, we reconsider the query-answering problem.

The main job of the query-answering procedure for general logic programs with respect to stable model semantics is divided into the following two parts:

(a) Finding candidate sets of literals which derive the query.

(b) Checking whether the candidate sets are included in some stable models.

In this way, computing answers for a query is quite efficient for applications that involve query-answering, because the generation of irrelevant candidate sets to the query are cut down. In Prolog, the top level of the procedure is for a query G.

```
glp(G):- vanilla(G,M), assert_models(M), satisfiable, exist_evidence.
```

The code `vanilla(G,M), assert_models(M)` corresponds to phase (a) (finding candidate sets), and the remaining code `satisfiable, exist_evidence` corresponds to phase (b) (checking candidate sets). In the following, we show the codes of each part and explain them.

Phase (a) – Finding Candidate Sets

Finding a candidate set deriving query is the same as constructing a proof tree deriving query. During the proof tree construction, all positive and negative literals are gathered, though the validity of literals are not checked; the validity is checked in phase (b). This phase is realized by a Prolog meta-interpreter with memo function. The Prolog codes are in Figure 1; for query G, `vanilla(G,M)` returns model candidate M which derives G. In Figure 1, `clause(Head,Body)` is a Prolog code for a rule of a general logic program: a rule

$$A_0 \leftarrow A_1, ..., A_m, not\, A_{m+1}, ..., not\, A_n.$$

is coded in

clause(A_0,(A_1, ..., A_m, ~A_{m+1}, ..., ~A_n)).

As a disjunctive rule in section 2.2, this coding for a rule shows that negative literals are represented as positive atoms.

The Prolog predicate `assert_models(M)` asserts each literal in the candidate set. You can find the codes for the predicate in appendix A.

```
satisfiable :-
        is_violated(C),!,
        satisfy(C),
        satisfiable.
satisfiable.

is_violated(C) :-
        (A--->C),
        A, not C.

satisfy(C) :-
        component(X,C),
        casserta(X),
        on_backtracking(cretract(X)),
        not false.

component(X,(Y;Z)) :-
        !, (X=Y ; component(X,Z)).
component(X,X).

on_backtracking(_).
on_backtracking(X) :-
        X,!, fail.

casserta((A,B)):-
        !, asserta(A),casserta(B).
casserta(A):- asserta(A).

cretract((A,B)):-
        (retract(A),fail;cretract(B)).
cretract(A):- retract(A).
```

Figure 2: SATCHMO interpreter

Phase (b) – Checking Candidate Sets
Validity checking of candidate sets, i.e., checking whether the candidate sets are included in some stable models, is done by trying to construct a stable model from the candidate sets as initial models. Since minimal models for disjunctive program $TR(K)$ correspond to stable models for general logic program K under two conditions described in the previous section, we construct minimal models for $TR(K)$ by the model generation theorem prover. Although there might be many alternative ways of doing this, here, we use a small Prolog prover (Figure 2), SATCHMO [Bry88]. SATCHMO generates minimal models for range-restricted first-order formulas. Therefore, we restrict the programs to range-restricted ones.

In our query-answering procedure, SATCHMO is called by `satisfiable` after phase (a). In SATCHMO, an input formula:

$$A_{l+1}, \ldots, A_m \rightarrow A_{1,1}, \ldots, A_{1,k_1} | \ldots | A_{l,1}, \ldots, A_{l,k_l},$$

where $m \geq l \geq 0, k_i \geq 1 (1 < i \leq l)$, is coded in Prolog as a clause:

$$\mathtt{A}_{1+1}, \ldots, \mathtt{A}_{m} \; \texttt{--->} \; \mathtt{A}_{1,1}, \ldots, \mathtt{A}_{1,k_1} ; \ldots ; \mathtt{A}_{1,1}, \ldots, \mathtt{A}_{1,k_1}.$$

For an integrity constraint:

$$\leftarrow A_1, \ldots, A_m,$$

the Prolog code for SATCHMO is a clause:

$$\texttt{false :- } \mathtt{A}_1, \ldots, \mathtt{A}_m.$$

To obtain stable models of general logic programs through computing minimal models of the disjunctive programs, the two conditions stated in section 2.2 should be satisfied. Since SATCHMO asserts facts in a model candidate during model generation, the two conditions are checked by using asserted facts. Condition (1) is expressed in the following Prolog code:

```
false :- ~X, X.
```

This clause is a scheme which expresses integrity constraints for the disjunctive program, and is checked in the clause `satisfy(C)`. Condition (2) is checked after all other procedures have finished. This is done by a Prolog clause `exist_evidence`:

```
exist_evidence :- not (~~X, not X).
```

4 Bottom-Up Procedure with Magic Rules

In the naive procedure in the previous section, the Prolog interpreter computes phase (a) in a top-down manner. The top-down search of Prolog interpreter is not complete[2].

Magic set technique [Bancilhon86, Fuchi88, Bry90] provides us with one of the keys to overcoming the limitation. With magic set technique, we incorporate top-down information, binding information in the query, with bottom-up reasoning so that we can prevent redundant fact generation and keep the reasoning complete in Horn or stratified cases. In our case, magic set technique allows us to embed phase (a) in the naive procedure into the bottom-up procedure. Therefore, we can prevent redundant model generation and keep the procedure complete in non-Horn cases[3].

4.1 Magic-Sets Transformation for Disjunctive Representations of Rules

Here, we provide magic-sets transformation method from disjunctive representations of general logic programs: we do not use forms of rules in our magic interpreter. For Horn or stratified programs, we use only magic rules to answer queries. For general logic programs, however, both magic rules and original disjunctive forms of rules are used in the magic interpreter. This point is different from previous works in that magic set technique is applied to query-answering for Horn or stratified programs.

For a positive disjunctive clause,

$$A_1, \ldots, A_m \quad \rightarrow \quad A_0, \sim A_{m+1}, \ldots, \sim A_n \mid \sim\sim A_{m+1} \mid \ldots \mid \sim\sim A_n.$$

where $m, n > 1$ (the rule is not a fact), magic rules are the following positive disjunctive clauses:

$$\begin{array}{rcl} \textit{magic-}A_0 & \rightarrow & \textit{magic-}A_1. \\ \textit{magic-}A_0, A_1 & \rightarrow & \textit{magic-}A_2. \\ & \ldots & \\ \textit{magic-}A_0, A_1, \ldots, A_{m-1} & \rightarrow & \textit{magic-}A_m. \\ \textit{magic-}A_0, A_1, \ldots, A_{m-1}, A_m & \rightarrow & A_0, \sim A_{m+1}, \ldots, \sim A_n \mid \sim\sim A_{m+1} \mid \ldots \mid \sim\sim A_n. \end{array}$$

where each *magic-A* is a new predicate symbol which expresses the 'negation' of atom A.

For a query `G`, the following magic rule is needed:

$$\textit{true} \quad \rightarrow \quad \textit{magic-}G.$$

This magic rule ignites other magic rules.

In this magic-set transformation, it is possible that the range-restrictedness in magic rules is broken. Since SATCHMO, which is the basis of our magic interpreter, requests formulas to be

[2]The completeness means that, if there is a finite proof tree, then computation terminates. Ordinary Prolog interpreters, even those which use the breadth-first search strategy, are not complete because of negation as failure.

[3]Actually, the completeness of our magic interpreter has not been proved yet.

```
naf(G):-
        on_backtracking(abolish(found,1)),
        query(G),
        satisfiable,% Satchmo's clause
        (not not_exist_evidence),
        asserta(found(G)).

query(G):- goal(G),not found(G).
query(G):-
        magic_rule_is_violated(C),!,
        satisfy(C), % SATCHMO's clause
        query(G).
```

```
magic_rule_is_violated(C) :-
        (A---->C),
        A, not C.

false:- (~(X)), X.

not_exist_evidence:-
        (~~X),not (call(X)),!.
```

Figure 3: Magic interpreter

range-restricted, magic-set transformation cause some troubles. In the last section, we discuss this point again.

In addition to the above magic rules, the following rule is needed as the special magic rule:

$$G \rightarrow goal(G).$$

The atom $goal(G)$ terminates the magic part of the procedure.

4.2 Magic Interpreter

The naive procedure in section 3 is rewritten with magic set technique in the following *magic interpreter*:

```
nag(G):- query(G), satisfiable, exist_evidence.
```

A predicate query(G) constructs the model candidate in a bottom-up manner, and asserts literals until goal(G) is asserted; in other words, the predicate does phase (a). query(G) tries to construct a model in which magic rules are satisfied. To construct this model, a model generation theorem prover (SATCHMO, in this case) is used[4]. Facts and magic rules are referenced in the predicate query(G), while ordinary rules (and also disjunctive representations) and integrity constraints are not. The entire code is shown in Figure 3. To discriminate a original form and the magic rule, a magic rule:

$$A_{l+1},...,A_m \rightarrow A_{1,1},...,A_{1,k_1}|...|A_{l,1},...,A_{l,k_l},$$

where $m \geq l \geq 0, k_i \geq 1 (1 \leq i \leq l)$, is coded in Prolog as a clause:

$$\mathtt{A_{l+1},...,A_m \; \text{---->} \; A_{1,1},...,A_{1,k_1};...;A_{1,1},...,A_{1,k_1}}.$$

Predicates satisfiable, exist_evidence corresponding to phase (b) are the same as in the naive procedure. In these predicates, facts and original disjunctive forms of rules are referenced as well as integrity constraints.

[4]A complete theorem prover is need to keep magic interpreter complete, though SATCHMO shown in this paper is not complete.

5 Concluding Remarks

A query-answering procedure with respect to stable models is provided in this paper. The procedure is based on a bottom-up model generation theorem prover and utilizes magic set technique. We have two future directions for this work; one is related to magic set and the other is related to the theorem prover.

Future works related to magic set are as follows. Firstly, magic-set transformation may possibly break the range-restrictedness of magic rules. If an underlying theorem prover (SATCHMO, in our case) cannot deal with non-range-restricted rules, the procedure will fail. Magic-set transformation is originally done to "adorned" rules. The adornment allows us to specify input/output modes to rules. Although we do not consider adorned rules in this paper, the adornment can save the range-restrictedness of magic rules if the rules are adorned well.

Secondly, magic-set transformation for integrity constraints provides us with the possibility to make the magic interpreter more efficient. In the magic interpreter, integrity constraints are checked in the latter part of the interpreter. If the integrity check were done by magic rules for integrity constraints during model construction, that is phase (a), the integrity violation might be detected in the earlier phase of the magic interpreter. This might allow us to find the violation of constraints as early as possible.

Furthermore, in logic programming, magic set technique has been used not only for Horn programs but also for stratified programs. It is needed to investigate the difference between those related works and our approach. Techniques in first order theorem provers also have some relationship to our usage of magic set technique. The comparison will be needed.

As other future works, we will consider the underlying theorem prover of our magic interpreter. The MGTP [HFujita91], a parallel and refined version of SATCHMO, is being developed in ICOT. If we replace SATCHMO in our magic interpreter with MGTP, out interpreter works in parallel and more efficiently as a result.

Acknowledgment

We are grateful to Katsumi Inoue, Ken Satoh and Koichi Furukawa for variable discussions. We also thank Kazuhiro Fuchi for providing us with the chance to do this work and for his earlier work which showed us the way to this work.

References

[Bancilhon86] Bancilhon, F., Ramakrishnan, R., An Amateur's Introduction to Recursive Query Processing Strategies, *SIGMOD'86*, pp. 16 – 52, 1986.

[Bry88] Bry, F., SATCHMO: a theorem prover implemented in Prolog, *Proc. CADE-9*, pp. 415-434, 1988.

[Bry90] Bry, F., Upside-down Deduction, *Proc. 6th PODS*, 1990.

[Eshghi89] Eshghi, K., Kowalski, R. A., Abduction Compared with Negation by Failure, *Proc. of ICLP'89*, pp. 234 254 1989.

[Fuchi88] Fuchi, K., Logical Source of Magic Set, unpublished manuscript for ICOT meeting (in Japanese), 1988.

[Gelfond88] Gelnfond, M., Lifschitz, V., The Stable Model Semantics for Logic Programming, *Proc. of ICLP'88*, pp. 1070 – 1080, 1988.

[HFujita91] Fujita, H., and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm, *Proc. of ICLP'91*, pp. 535 – 562, 1991.

[Inoue92] Inoue, K., Koshimura, M., Hasegawa, R., Embedding negation as failure into a model generation theorem prover, to appear *Proc. of CADE-11*, 1992.

[Iwayama91] Iwayama N., Satoh. K., A Bottom-up Procedure with Top-down Expectation for General Logic Programs with Integrity Constraints, *ICOT-TR 625*, 1991.

[Kakas90] Kakas, A. C., Mancarella, P., On the Relation between Truth Maintenance and Abduction, *Proc. of PRICAI'90*, pp. 438 – 443 (1990).

[Sacca90] Sacca, D., Zaniolo, C., Stable Models and Non-Determinism in Logic Programs with Negation, *Proc. of PODS'90*, pp. 205 – 217, 1990.

[Satoh91] Satoh, K., Iwayama, N., Computing Abduction by Using the TMS, *Proc. of ICLP'91*, pp. 505 – 518, 1991.

[Satoh92] Satoh, K., Iwayama, N., A Correct Top-Down Proof Procedure for a General Logic Program with Integrity Constraints, *Proc. of 3rd Int. Workshop on Extensions of Logic Programming*, pp. 19 – 34, 1992.

Appendix

A Prolog Codes of The Naive Procedure

```
%%%%% Stable Model Prolog Interpreter %%%%%%%%%%
naf(G):- vanilla(G,M),exist_stable_model(M).
% prolog meta interpreter with memoization of lemmas
vanilla(G,M):- vi(G,[],M).
vi(true,Mi,Mi):- !.
vi((G,GL),Mi,Mo):- !, vi(G,Mi,M1),vi(GL,M1,Mo).
vi((G1;G2),Mi,Mo):- !, vi(G1,Mi,Mo);vi(G2,Mi,Mo).
vi(~G,Mi,Mo):- !,(vi(G,[],_) -> fail;Mo=[~G|Mi]).
vi(G,Mi,[G|Mo]):- clause(G,SG),vi(SG,Mi,Mo).
% axiom
false:- call(~X),call(X).
% integrity checker based on satchmo
exist_stable_model(M):-
          assert_naf(M),
          satisfiable, % Satchmo's clause
          \+not_exist_evidence, !.
% evidence check
not_exist_evidence:- call(~~X),\+(call(X)),!.
% assert proof tree
assert_naf([G|GL]):- !,
        (call(G) -> true;assert(G)),% with subsumption test
        on_backtracking(retract(G)),
assert_naf(GL).
assert_naf([]).
```

B Prolog Codes of Magic Interpreter

```
%%%%% Stable Model Prolog Interpreter by Magic Set %%%%%
naf(G):-
        on_backtracking(abolish(found,1)),
        query(G),
        satisfiable,% Satchmo's clause
        (not not_exist_evidence),
        asserta(found(G)).
% revised SATCHMO
query(G):- goal(G),not found(G).
query(G):-
        magic_rule_is_violated(C),!,
        satisfy(C), % SATCHMO's clause
        query(G).
magic_rule_is_violated(C) :-
        (A---->C),
        A, not C.
% axiom
false:- call(~X),call(X).
% evidence check
not_exist_evidence:- call(~~X), not call(X),!.
```

C Examples

- Example 1

```
%%%%% Result %%%%%
?- naf(p(X)).
X=1;

X=3;

X=2;

X=4;

no

%%%%%% A Famous Problem %%%%%%
%p(1):- ~p(2).
%p(2):- ~p(3).
%p(3):- ~p(4).
%p(4):- ~p(1).
%%%%% Bottom Up Rule %%%%%
true ---> ((p(1), (~p(2)));(~ ~p(2))).
true ---> ((p(2), (~p(3)));(~ ~p(3))).
true ---> ((p(3), (~p(4)));(~ ~p(4))).
true ---> ((p(4), (~p(1)));(~ ~p(1))).
```

```
%%%%% Magic Set %%%%%
p(X) ----> goal(p(X)).
true ----> gp(X).
gp(1) ----> ((p(1), (~p(2)));(~ ~p(2))).
gp(2) ----> ((p(2), (~p(3)));(~ ~p(3))).
gp(3) ----> ((p(3), (~p(4)));(~ ~p(4))).
gp(4) ----> ((p(4), (~p(1)));(~ ~p(1))).
```

- Example 2 (reservation problem)[5]
  This example is computed by the naive procedure in section 3, in about 850 msec under Sistus Prolog on SUN3/260.

```
%%%%% Result %%%%%
% query
?- naf(holdmeeting(V1,V2,V3,V4)).
V1=mon,
V2=213,
V3=self,
V4=self ?;

no

%%%%% Sample of Room Reservation %%%%%%
% facts
person(mark).
person(donald).
day(mon).
day(tue).
room(212).
room(213).
reserved(212,mon).
reserved(213,tue).
busy(mark,agent,mon).
busy(donald,self,tue).

% rules
cand(X,self,Y) :-
        person(X),
        day(Y),
        ~busy(X,self,Y).
person(X), day(Y) ---> cand(X,self,Y),~(busy(X,self,Y));
                       ~(~(busy(X,self,Y))).
cand(X,agent,Y) :-
        busy(X,self,Y),
        ~busy(X,agent,Y).
busy(X,self,Y) ---> ~(busy(X,agent,Y)),cand(X,agent,Y);
```

[5]This problem is provided by Katumi Inoue.

```
                    ~(~(busy(X,agent,Y))).

openroom(Room,Day) :-
        day(Day),
        room(Room),
        ~reserved(Room,Day).
day(Day), room(Room) ---> ~(reserved(Room,Day)),openroom(Room,Day);
                          ~(~(reserved(Room,Day))).
selves_meeting :-
        holdmeeting(Day,Room,self,self).
holdmeeting(Day,Room,self,self) ---> selves_meeting).
holdmeeting(Day,Room,self,self) :-
        cand(mark,self,Day),
        cand(donald,self,Day),
        openroom(Room,Day).
cand(a,self,Day), cand(b,self,Day), openroom(Room,Day)
               ---> holdmeeting(Day,Room,self,self).
holdmeeting(Day,Room,Ida,Idb) :-
        cand(mark,Ida,Day),
        cand(donald,Idb,Day),
        openroom(Room,Day),
        ~selves_meeting.
cand(a,Ida,Day), cand(b,Idb,Day), openroom(Room,Day)
               ---> ~(selves_meeting),holdmeeting(Day,Room,Ida,Idb);
                    ~(~(selves_meeting)))).
```