Hypothetical Reasoning with Well Founded Semantics

Luís Moniz Pereira, Joaquim Nunes Aparício, José Júlio Alferes

AI Centre, Uninova

2825 Monte da Caparica, Portugal

December 16th, 1990

Abstract: Well Founded Semantics is shown adequate to capture hypothetical reasoning if we interpret the Well Founded model of a program P as a (possibly incomplete) view of the world. Thus the Well Founded model may be accepted as a partially definite view of the world, and the other extended stable models as alternative extended consistent views of the world.
The original contributions of this paper are: showing that Well Founded Semantics is useful for representing hypothetical reasoning problems; to present a framework for representing always within the language itself: definite, default and generally applicable rules; preference among defaults; exceptions to rules; exceptions to exceptions; abduction; and integrity constraints.

Introduction

Well Founded Semantics (WFS) [van Gelder et al. 1990] is adequate to capture hypothetical reasoning if we interpret the well-founded model (WFM) of a program P as a (possibly incomplete) view of the world. Thus the WFM may be accepted to be a view of the world and the extended stable models (XSMs) being extended alternative consistent views of the world, all of each containing the WFM. A world view may be completed (or refined) by hypothesizing (e.g. abducing, using default rules, etc.) about unknown information. Przymusinski [Przymusinski 1988; Przymusinski 1989a; Przymusinski 1989b] shows that well-founded semantics is also equivalent to suitable forms of four major formalizations of non-monotonic reasoning (Circumscription, Closed World Assumption, Autoepistemic Logic and Default Logic). An attractive aspect is that well-founded semantics is defined for the class of all logic programs, the WFM being a minimal one.

In case the WFM is a 2-value model then no degree of freedom is left to the agent to conjecture or hypothesize about the world. If the WFM is 3-valued, hypothetical reasoning is then delimited by the unknown facts of the WFM. We have devised procedures for XSM semantics and WFM semantics [Pereira et al. 1990c; Pereira et al. 1990e]. For a practical introduction to WFS c.f. [Pereira et al. 1990a]. For additional examples c.f. [Pereira et al. 1990d].

Technically it seems a nice approach to interpret hypothesizing as searching XSMs restricted to some conditions. We will show that asking if A is true when we may abduce B, maps into the process of searching for XSMs where A is true, whenever B is true. On the other hand, other interesting problems appear as finding the minimal change so that A becomes true. With the well-founded model being the intersection of XSMs, WF semantics seems a rather natural 3-valued extension to capture richer notions of entailment.

Our main original contribution is to present a framework for representing always within the language itself most of the mechanisms used in hypothetical reasoning

(namely: definite, default and generally applicable rules; preference among defaults; exceptions to rules; exceptions to exceptions; abduction; and integrity constraints). Another important original contribution is to show that the Well Founded Semantics is useful for representing hypothetical reasoning problems.

We will begin by giving a definition of WF Model Semantics for logic programs with classical negation. Next we propose a representation of hierarchical taxonomies in these semantics. In the section "Possible worlds" we represent hypothetical reasoning problems and interpret the results. Based on these results we define abduction within the WF semantics and establish a relationship between it and the XS Models. Then we systematize the representation methodology used in the above sections. After this we show how to deal with integrity constraints, and finally we draw additional comparisons with other work.

1. The Well Founded Semantics

There are two main motivations for this section. First, it makes the paper self-contained; second, it makes possible for the reader to confirm the results presented.

A reader familiarized with these semantics, or simply trusting that the models presented in the examples are correct and not caring much about their actual construction, can skip this section and proceed directly to section 2.

1.1. The Extended Stable Model Semantics

Here we will characterize the Extended Stable Model Semantics, as in [Przymusinska et al. 1990], which are an extension to the Stable Model Semantics (SMS) of [Gelfond et al. 1988]. Since the semantics are 3-valued, we begin by defining 3-valued interpretations.

Definition *(3-valued interpretation)*

By a 3-valued Herbrand interpretation *I* of a language L we mean any pair <T;F>, where T and F are disjoint subsets of the Herbrand base H. The set T contains all ground atoms true in *I*, the set F contains all ground atoms false in *I* and the truth value of the remaining atoms, in U = H - (T U F), is unknown (or undefined). o

An alternative way to represent an interpretation I = <T;F> is I = T U {~LIL F}. This is the representation used throughout the paper.

Proposition *(Interpretations as functions)*

Any interpretation I = <T;F> can be equivalently viewed as a function I: HV where V = {0, 1/2, 1}, defined by:

- I(A) = 0 if A F
- I(A) = 1/2 if A U
- I(A) = 1 if A T o

Definition *(truth valuation)*

The function I: HV can be recursively extended to the truth valuation function Î: LitV defined on the set Lit of all literals of the language.

Given an interpretation I: HV then the truth valuation Î corresponding to I is a function Î: LitV , as follows, where A is a ground atom:

- Î(A) = I(A).
- Î(~A) = 1 - I(A). o

Definition *(Non-negative program)*

By a non-negative program we mean a logic program whose premises are either positive atoms or the special proposition **u**. We assume that every interpretation I satisfies I(**u**) = 1/2 and thus Î(~**u**) = 1/2. This special proposition denotes **unknown** or **undefined**. o

Theorem *(Generalization of* [van Emden et al. 1976]*)*

Every non-negative logic program has a unique least 3-valued model. o

Next we define the program transformation P/M (P modulo M), which is an extension to the GL-transformation defined in [Gelfond et al. 1988].

Definition *(Modulo transformation) [Przymusinska et al. 1990]*

Let P be a logic program and let I be a 3-valued interpretation. By the extended GL-transformation of P modulo I we mean a new (non-negative) program P/I obtained from P by performing the following three operations:

- Removing from P all rules which contain a negative premise L = ~A such that Î(L) = 0.
- Replacing in all remaining rules those negative premises L = ~A which satisfy Î(L) = 1/2 by **u**.
- Removing from all the remaining rules those negative L = ~A which satisfy Î(L) = 1. o

Since the resulting program P/I is non-negative, by theorem 4.2 it has a unique least 3-valued model. We define $\Gamma^*(I)$ (a generalization of the Γ operator [Gelfond et al. 1988]) to be the 3-valued least model of P/I.

Definition *(Extended Stable Model)*

A 3-valued interpretation I of a logic program P is called an Extended Stable Model of P iff $\Gamma^*(I) = I$. o

These semantics were proved equivalent to the WF Semantics, in [Przymusinska et al. 1990] with the following theorem:

Theorem *[Przymusinska et al. 1990]*

The Well Founded Model of a program P is the F-least Extended Stable Model of

P. Consequently the WFS coincides with the XSMS. o

To obtain a constructive definition of the WFM of a program P, we use the following sequence of $\{I_\alpha\}$ of interpretations of P:

- $I_0 = <\{\},\{\}>$.
- $I_{\alpha+1} = \Gamma^*(I_\alpha)$.

The WF Model of P is the fixed point of this sequence, i.e. I_λ [Przymusinska et al. 1990].

1.2. Introducing classical negation *[Przymusinski 1990]*

The 3-valued stable model semantics can be extended to programs with classical negation, by using a simple program transformation:

• First, we denote all classically negated (ground) literals ¬A by a new atomic symbol, say, A' and make a suitable substitution everywhere in the program P. As result we obtain a standard program P* without classical negation.

• Then for any 3-valued stable model M* of the transformed program P* (in particular, for the well-founded model) we define the corresponding 3-valued stable model M of the extended program P as follows:

- If A (resp. A') is true in M*, then we take A (resp. ¬A) to be true in M.
- If A (resp. A') is false in M*, then we take ~A (resp. ~(¬A)) to be true in M, i.e., we view A (resp. ¬A) as false by default.
- Otherwise, if A (resp. A') is unknown in M*, then we consider the status of A (resp. ¬A) in M as also unknown.

• We throw out all *contradictory* models M, i.e., those containing both A and ¬A, for some atom A.

2. Hierarchy Representation

In this section we will illustrate how to represent a hierarchy in extended logic programs with the above semantics. In this representation we wish to express general rules, and their exceptions, as well as exceptions to exceptions, these exceptions being possible general rules in the sense that they might be defeated. So, for instance, we want to represent general rules such as "birds normally fly" and "penguins normally don't fly" where the latter can act as an exception to the former, and vice-versa, given that penguins are (always) birds. Furthermore, we wish to be able to express preference for one rule over another in case they conflict and are both applicable. Thus in case of multiple extensions, we can prefer the most specific information (e.g. for a penguin, which is a bird we want to conclude that it doesn't fly, unless there is even more specific information).

Our presentation is made step by step using the well known "Birds Fly" hierarchical taxonomy.

2.1. One level hierarchy

Let us begin with the simplest version of this problem:

Normally birds fly.

What we want is a rule *flies(X) bird(X)* that applies whenever possible, but can be defeated by exceptions. These exceptions can be made, both by explicit exception to the conclusion predicate (flies) and by a weaker exception made only to the rule. In other words we want to be able to say that a given bird doesn't fly and also that for a given bird this rule doesn't apply.

In order to allow exceptions on the conclusions, we must say that a bird flies only if there is no evidence that it doesn't. So:

flies(X) bird(X), ~ ¬flies(X)

To allow exceptions to the rule, we have to name that rule:

(flies(X) bird(X), ~ ¬flies(X)) bird_flies(X).

which is equivalent to:

flies(X) bird(X), ~ ¬flies(X), bird_flies(X). (1)

To capture the notion that the rule applies whenever possible we introduce the rule:

bird_flies(X) ~ ¬bird_flies(X). (2)

which with (1) captures the notion that if there is no way to prove that an object is an exception to the rule, then assume that the rule applies.

¬bird_flies(X) represents an exception to bird_flies(X). By renaming ¬bird_flies (X) to McCarthy's abnormal_bird(X) [McCarthy 1987], then from (1) and (2) there follow the clauses:

flies(X) bird(X), ~ abnormal_bird(X). and
abnormal_bird(X) ¬ flies(X).

We prefer having rule (2), it being a modular statement of an assumption about a predicate.

Note also that (2) can be seen as a closed world assumption about ¬bird_flies (X).

Abridging, we represent the "birds fly" taxonomy with the program:

$\prod_1$: f(X) b(X), ~ ¬f(X), bf(X).
bf(X) ~¬bf(X).

where *f* stands for flies, *b* for bird and *bf* for bird_flies.

Let us see what are the Extended Stable Models (XSM) of $\prod_1$ if we add to it the facts:

$Facts_1$: b(a). b(b). ¬f(b).

$\prod_1 \approx Facts_1$ has a unique XS Model coinciding with the Well Founded Model (WF), which is :

WF = { f(a), ~ ¬f(a), ¬f(b), ~f(b),
b(a), b(b),
bf(a), ~ ¬bf(a), bf(b),~ ¬bf(b)}

The fact that this model is 2-valued reflects that no choices are available to obtain other XSMs. This seems quite an acceptable result. Regarding *b* we know for sure that it is a bird and it doesn't fly. We can say that the rule might apply, because there are no exceptions defined for it. For *a* we know that it is a bird, and as there are no possible exception for flies, when X is bound to *a*, we can say for sure that it flies. Thus the birds fly rule is applied maximally, as default rules are.

This problem domain can be extended by saying that all penguins are birds, penguins don't fly, and b is a penguin.

2.2. Exceptions to exceptions

In a hierarchy, instead of saying that "penguins don't fly", we would like to say that normally penguins don't fly. This will allow us to express an exception to an exception rule of birds fly, and hence the possibility that a given exceptional penguin may fly.

So the problem statement now is:

Normally birds fly.
Penguins are birds.
Normally penguins don't fly.

According to what was said before this problem is represented as:

$\prod_2$: f(X) b(X), ~ ¬f(X), bf(X). % Normally birds fly
bf(X) ~¬bf(X).

¬f(X) p(X), ~ f(X), pnf(X). % Normally penguins don't fly
pnf(X) ~ ¬pnf(X).

b(X) p(X). % Penguins are birds

where *p* stands for penguin and *pnf* stands for penguin_not_flies.

With this program, we will consider two birds, one of them being a penguin:

$Facts_2$: b(a). p(b).

The WF Model of $\prod_2 \approx Facts_2$ is:

WF = { f(a), ~¬f(a), bf(a), ~¬bf(a), pnf(a), ~¬pnf(a), b(a), ~p(a),
bf(b), ~¬bf(b), pnf(b), ~¬pnf(b), b(b), p(b) }

About *a* everything is defined, and we are exactly at the same case as before.

About *b* this model only tells us that it is a bird and a penguin, leaving predicates flies and ¬flies undefined. This is due to the fact that there are two exception rule that might apply, each of which inhibits the conclusion of the other. So nothing can be said for sure about b flying or not.

But the program has two more XS Models:

$XSM_1 = WF \approx \{ f(b), \sim \neg f(b)\}$
$XSM_2 = WF \approx \{ \neg f(b), \sim f(b) \}$

These models can be seen as different models (i.e. consistent beliefs) an agent may have of the world. This is a quite intuitive result, since we can believe that b flies (because it is a bird) or that it doesn't (because it is a penguin). The WF Model is the intersection of these two, and intuitively gives us our *real* world, the things we believe unconditionally, and where nothing can be said about b flying or not. This topic will be further detailed in the sequel.

In a hierarchy however, since we always want to apply the most specific information, there should be an explicit preference between the rules bird_flies and penguin_not_flies, which establishes the properties that in a hierarchy a property is either true or false for an individual in a taxonomy.

Having rules named, this preference is quite easy to represent. What we want to say is that if we can apply the rule penguin_not_flies to a penguin we are in presence of an exception to the bird_flies rule. So the preference rule is simply:

¬bf(X) p(X), pnf(X).

With this rule the only XS Model is the WF Model:

WF = { f(a) , ~¬f(a), bf(a), ~¬bf(a), pnf(a), ~¬pnf(a), b(a), ~p(a),
¬f(b), ~ f(b) , ¬bf(b), ~ bf(b), pnf(b), ~¬pnf(b), b(b), p(b) }

which expresses the intuitive notions about the hierarchy. **b** doesn't fly and is an exception to bird_flies rule.

Now we can represent exceptions to exceptions:

$Facts_3$: f(X) fp(X). % Flying penguins fly

p(X) fp(X). % Flying penguins are penguins

fp(c). % c is a flying penguin

where *fp* is short for flying_penguin.

The WF Model of $\prod_2 \approx Facts_3$ is:

WF = { f(c), ~ ¬f(c), ¬bf(c), ~ bf(c), pnf(c), ~ ¬pnf(c), b(c), p(c), fp(c) }

Note that c is also an exception to the bird_flies rule. Nevertheless it flies, but because of the more specific and absolute rule that flying penguins fly.

2.3. A More Complex Hierarchy

For better exemplifying our representation of hierarchies, and the results obtained in them, we will illustrate it on a more complex example.

The problem statement is:

Mammals are animals.
Bats are mammals.
Birds are animals.
Penguins are birds.
Dead animals are animals.

Normally animals don't fly.
Normally bats fly.
Normally birds fly.
Normally penguins don't fly.

Dead animals don't fly.

Pluto is a mammal
Tweety is a bird.
Joe is a penguin.
Jack is a bat.
Jack is a dead animal.

Our representation of this is the program:

Π_3:
```
animal(X)      mammal(X).          % Mammals are animals.
mammal(X)      bat(X).                   % Bats are mammals
animal(X)      bird(X).                  % Birds are animals.
bird(X)   penguin(X).          % Penguins are birds.
animal(X)      dead_animal(X).   % Dead animals are animals.

¬flies(X)   animal(X),~flies(X), anf(X).   % Normally animals don't fly.
anf(X)      ~ ¬anf(X).
flies(X)    bat(X), ~ ¬flies(X), btf(X).          % Normally bats fly.
btf(X)      ~ ¬btf(X).
flies(X)    bird(X), ~ ¬flies(X), bf(X).          % Normally birds fly.
bf(X)       ~ ¬bf(X).
¬flies(X)   penguin(X),~flies(X),pnf(X). %Normally penguins don't fly.
pnf(X)      ~ ¬pnf(X).

¬flies(X)   dead_animal(X). % Dead animals don't fly.

mammal(pluto).                  % Pluto is a mammal
bird(tweety).                   % Tweety is a bird.
penguin(joe ).             % Joe is a penguin.
bat(jack).                      % Jack is a bat.
dead_animal(jack).         % Jack is a dead animal.

% Preference rules
```

¬anf(X) bat(X), btf(X).
¬anf(X) bird(X), bf(X).
¬bf(X) penguin(X), pnf(X).

As expected, this program has exactly one XS Model (coinciding with the WF Model), no choice being possible and everything being defined in the hierarchy. The full model is:

WF =
{¬flies(pluto),animal(pluto),mammal(pluto),anf(pluto),btf(pluto),bf(pluto), pnf(pluto),
flies(tweety),animal(tweety),bird(tweety),¬anf(tweety),btf(tweety),bf(tweety),pnf(tweety),
¬flies(joe),animal(joe),bird(joe),penguin(joe),anf(joe),btf(joe),¬bf(joe),pnf(,joe),
¬flies(jack),animal(jack),mammal(jack),bat(jack),dead_animal(jack),
anf(jack),¬btf(jack),bf(jack),pnf(jack) }

Thus pluto doesn't fly, and isn't exception to any of the rules; tweety flies because it's a bird and an exception to the "animals don't fly" rule; joe doesn't fly because it's a penguin and an exception to the "birds fly" rule and finally jack doesn't fly because it is dead. Note that jack is an exception to the "animals don't fly" rule, being expected to fly. But as it's dead, and as the "dead animals don't fly" rule can't be exceptioned, it overrides the "bats fly" rule. Note also that the "dead animals don't fly" rule doesn't interfere unless we have a dead_animal.

3. Possible worlds

In hierarchies, as seen, everything is defined, leaving no choices available. This is not the case for general hypothetical reasoning problems.

In the next section we represent hypothetical reasoning problems in the Extended Stable Models Semantics and we interpret the results.

3.1. The "Nixon diamond" problem

Let us look now at another well known problem:

Normally quakers are pacifists. (1)
Normally republicans are hawks. (2)
Pacifists are non hawks.
Hawks are non pacifists.
Nixon is quaker and republican.

If we represent this example as before the names of rules (1) and (2) are in the well founded model, leaving no possibility of choice between them. This was correct when we had a hierarchy and there was always an explicit preference between rules. In this problem that is not the case.

Here, as there is no explicit preference, we don't want a rule to be maximally applicable (in the sense that it is true whenever possible). Instead we wish to choose between having or not the rule applied.

In hierarchies *quaker_pacifist(X)* ~ ¬*quaker_pacifist(X)* (C1) expresses that rule (1) is true whenever possible. We must have a clause for saying that rule (1) is not true whenever possible, leaving this way freedom to apply or not the rule.

This clause is clearly:

¬quaker_pacifist(X) ~ quaker_pacifist(X). (C2)

These two clauses (C1 and C2) forces the name of the rule (quaker_pacifist), and consequently its conclusion, to be unknown in the Well Founded Model.

According to this, our representation of the "Nixon diamond" problem is as follows:

```
% Normally quakers are pacifists
pacifist(X)              quaker(X), quaker_pacifist(X).
quaker_pacifist(X)   ~ ¬quaker_pacifist(X).
¬quaker_pacifist(X)      ~ quaker_pacifist(X).

% Normally republicans are hawks
hawk(X)                  republican(X), republican_hawk(X).
republican_hawk(X)       ~ ¬republican_hawk(X).
¬republican_hawk(X)      ~ republican_hawk(X).

¬hawk(X)      pacifist(X).   % Pacifists are non hawks
¬ pacifist(X)  hawk(X).      % Hawks are non pacifists

quaker(nixon).               % Nixon is a quaker.
republican(nixon).   % Nixon is a republican.
```

The WF Model of this program is:

WF = { quaker(nixon), republican(nixon) }

and the other consistent XS Models are:

XSM_1 = { pacifist(nixon), ¬hawk(nixon), quaker(nixon), republican(nixon),
quaker_pacifist(nixon), ¬republican_hawk(nixon) }

XSM_2 = { ¬pacifist(nixon), hawk(nixon), quaker(nixon), republican(nixon),
¬quaker_pacifist(nixon), republican_hawk(nixon) }

XSM_3 = { quaker(nixon), republican(nixon),
¬quaker_pacifist(nixon), ¬republican_hawk(nixon) }

We interpret the WF Model as being what is taken for certain in our view of the real world, i.e. all things we believe unconditionally. In this we are not able to conclude anything about nixon being pacifist or hawk.

The remaining XS Models can be seen as possible extended world views in which some choices of belief have been made. So XSM_1 represents a world where nixon is pacifist and non hawk, by choosing to apply the rule quaker_pacifist and not to apply the republican_hawk one; XSM_2 represents a world where nixon is hawk and non pacifist, by choosing to apply the rule republican_hawk and not to apply the quaker_pacifist one; XSM_3 is a world where we choose not to apply any of the rules, so that we can prove nothing about nixon being a pacifist or a hawk.

The interpretation of the above results, in particular in what concerns XSM's where literals not in the WF model are true, suggested to us a relationship between XS Models of a program with rules like (1) and (2) and a notion of abduction within WF Semantics. This relationship will be explored latter in this paper in section 5.

3.2. Salvador's problem

For a better understanding of the proposed representation let us look now at the following example:

Someone is saved by Salvador unless that someone saves himself. (1)
Someone is not saved by Salvador unless that someone does not save himself. (2)
John saves himself and Mary doesn't.
John saves Peter.

As proposed above, this problem should be represented as:

```
% Someone is saved by Salvador.
saves(salvador,X)   salv_saves(X), ~ ¬saves(salvador,X).
salv_saves(X) ~ ¬salv_saves(X).
¬salv_saves(X)      ~ salv_saves(X).

% Someone is not saved by Salvador.
¬saves(salvador,X)  ¬salv_saves(X), ~ saves(salvador,X).

% Those who save themselves are an exception to the second rule
¬salv_saves(X)      saves(X,X).

% Those who don't save themselves are an exception to the first rule
salv_saves(X)       ¬saves(X,X).

saves(john,john).           % John does save himself.
¬saves(mary,mary).% Mary doesn't save herself.
saves(john,peter).          % John saves Peter.
```

The WF Model of this program is:

{saves(salvador,mary),salv_saves(mary),~¬saves(salvador,mary),~¬salv_saves(mary),
¬saves(salvador,john),¬salv_saves(john), ~saves(salvador,john), ~salv_saves(john),
¬saves(mary,mary), saves(john,john), saves(john,peter) }

Note that there are some atoms undefined in the WF Model (namely, saves (salvador,salvador), salv_saves(salvador), saves(salvador,peter), salv_saves (peter), etc). Only two additional XS Models exist: one containing saves (salvador,peter), and another containing ¬saves(salvador,peter). In both, saves (salvador,salvador) remains undefined.

4. Abduction within Well Founded Semantics

We will begin by giving a definition of abduction within WF Semantics very similar to the ones given for classical logic in [Eshghi et al. 1989], [Kakas et al. 1988], [Pereira et al. 1990b]. Finally we present a theorem that relates the XS Models of

a modified program with the abduction within the WF Semantics of the original program.

Definition : We define Lit(P) as being the set of all literals appearing in the program P, which are not of the form ~L.

Definition : An abductive theory is pair < P, Ab > where P is a program and Ab is a set of literals in Lit(P).

Ab is the set of all literals we are prepared to assume if that contributes to prove some goal. We call these literals the abducible literals.

Definition : For a given abductive theory < P, Ab >, a subset S of Ab whose elements are not in the WF Model of P is an abductive solution for literal L iff L is in the WF Model of P ≈ S.

Since WF Models are always consistent by definition, we have no need to impose consistency.

Example 1:

< { p q, ¬p r, p r }, { q, r } > is an abductive theory with one abductive solutions for literal p namely {q}.o

Now that we have defined abduction within WF Semantics, we define a modification of a given program based on the abducible literals, and then establish results between the XS Models of the modified program (*abducing program*) and abductive solutions in the original one.

Definition : An abducing program for an abductive theory < P, Ab > is a program obtained by adding to P, for all literals L in Ab, two clauses of the form:

L ~ ¬L. and ¬L ~ L.

Example 2:

The program:

p q.	¬p r.	p r.
q ~ ¬q.	¬q ~ q.	
r ~ ¬r.	¬r ~ r.	

is the abducing program for the theory in example 1.o

Theorem : S is an abductive solution for L in an abductive theory < P, Ab > iff all XS Models of the abducing program for the theory that contain S also contain L.

This provides a technique for performing abductive inference, namely by considering the XS models where some desired conclusion holds.

Example 3:

The abducing program of example 2 has three XS Models:

$XSM_1 = \{ \}$

$XSM_2 = \{ \sim p, \sim \neg p, \neg q, \sim q, \neg r, \sim r \}$
$XSM_3 = \{ p, \sim \neg p, q, \sim \neg q, \neg r, \sim \neg r \}$

The abductive solution for p in the corresponding original program of example 1 can be readily found by applying the above theorem to these models. For instance, in all models where q is present (XSM_3), p is also present.o

Example 4:

In the Salvador's problem (section 3.2) salv_saves(peter) is an abductive solution for saves(salvador,peter). o

5. Integrity constraints

Definition : An integrity constraint is a logical formula of the form IC, where IC is a conjunction of literals.

Definition : A set of integrity constraints is satisfied by a model of a program P iff none is false in the model (i.e. each is true or undefined).

Since we allow for facts with classical negation, we may transform each integrity constraint IC into $\bot \leftarrow IC$ and add to the program the fact $\neg \bot$, where $\bot$ is a new predicate name.

Theorem : M is a model of the new program iff M - $\{ \neg \bot \}$ is a model of the original program satisfying the integrity constraints.

6. Summary of our representation method

In this section we summarize and systematize the representation method adopted in all the above examples. The type of rules for which we propose a representation is, in our view, general enough to capture a wide domain of hypothetical reasoning problems.

Each type of rule is described in a subsection by means of a schema in natural language and its corresponding representation schema .

6.1. Definite Rules

If A then B. The representation is: $B \leftarrow A$.

6.2. Definite Facts

A is true. The representation is: A.

6.3. Default (or maximally applicable) Rules

Normally if A then B. The representation is:

$$B \leftarrow A, name\_A\_B, \sim \neg B.$$
$$name\_A\_B \leftarrow \sim \neg name\_A\_B .$$

where *name_A_B* is a predicate symbol that "names" this rule only. Its has arguments are those arguments in *A* or *B*.

We will consider **Default Facts** as a special case of **Default Rules** where A is absent.

As an example consider the rule "Normally birds fly". Its representation is:

fly(X) bird(X), bird_flies(X), ~ ¬fly(X).

bird_flies(X) ~ ¬bird_flies(X).

6.4. Abduction Rules

Rule "If A then B" may or may not be abduced. Its representation is:

B *A*, *name_A_B*, ~ ¬*B*.

name_A_B ~ ¬*name_A_B* .

¬ *name_A_B* ~ *name_A_B* .

where *name_A_B* is a predicate symbol that "names" the first rule only. Its arguments are all those arguments in *A* or *B*.

As an example consider the rule "Quakers might be pacifists". Its representation is:

pacifist(X) quaker(X), quaker_pacifist(X), ~ ¬pacifist(X).

quaker_pacifist(X) ~ ¬quaker_pacifist(X).

¬quaker_pacifist(X) ~ quaker_pacifist(X).

6.5. Exceptions to Default Rule

Under certain conditions COND there are exceptions to the default rule named NAME.

¬*NAME* *COND*.

As an example, the representation of the exception "Penguins are exceptions to "normally birds fly" rule" is:

¬bird_flies(X) penguin(X).

6.6. Preference Rules

Under conditions COND, prefer to apply the default rule NAME+ instead of the default rule named NAME-.

¬*NAME-* *COND*, *NAME+*.

As an example consider "For penguins if the rule that says that "normally penguins don't fly" is applicable then inhibit "normally birds fly" rule". This is represented as:

¬ bird_flies(X) penguin(X), penguins_no_flies(X).

6.7. Integrity constraints

It is not possible to have $COND_1$, …, $COND_n$ *simultaneously.*

⊥ $COND_1$, …, $COND_n$.

As an example consider "No one can fly and not fly at the same time".

$\bot$ fly(X), ¬fly(X).

7. Comparisons with other work using non-classical semantics of logic programs

7.1. Logic Programs With Exceptions *[Kowalski et al. 1989]*

Above is shown how we represent and solve the “rules with exceptions” examples presented in [Kowalski et al. 1989].

The following distinctions seem to apply:

• We deal with exceptions to exceptions in a uniform and modular way. Because of its inherent asymmetry, the “rules with exceptions” approach requires changing previous clauses in the program each time an exception to an exception is made, because head literals need to change. For instance, a three level hierarchy of birds, penguins and flying penguins requires rules like

fly(X) bird(X)

nofly(X) penguin(X)

fly(X) flying_penguin(X)

and the exceptions

¬ fly(X) nofly(X)

¬ nofly(X) flying_penguin(X)

• We allow both positive and negative conclusions in rules, inclusively for the same predicate.

• The extension of well-founded semantics to classical negation provides a (non contradictory) well-founded model of definite conclusions in cases where e-answer set semantics provides only alternative models. For instance, in the pacifist/hawk example we obtain a well-founded model containing the facts {quaker, republican}, besides the two alternative e-answer sets. This additional model is, by design, the intersection of the other two.

• Of course, the use of well-founded semantics also provides a richer variety of answers, given its three-valued nature.

• By introducing “name predicates“ in rules, we have shown how to express exceptions to rules, rather than exceptions to whole predicates, and how to express preference amongst rules. This technique can be imported into the “rules with exceptions” approach, but then why distinguish between rules and exceptions? Thus our approach appears more uniform.

7.2. Generalized Stable Models *[Kakas et al. 1989]*

Above, we have shown how a literal P can be made abducible by directly introducing into a program the two rules P ~ ¬P and ¬P ~P. It follows that each

extended stable model of a program shows which abducible literals may be consistently added to the program to make that particular model the new well-founded one. This provides a technique for performing abductive inference, namely by considering the extended stable models where some desired conclusion holds and making it the new well-founded one.

We have also shown how we can deal with integrity constraints.

Thus, compared to Generalized Stable Models, we can observe that:

• Our approach can deal with the class of problems of GSM.

• The three-valued semantics subsumes the two-valued semantics of GSM and, additionally, allows the use of classical negation.

• Abducibility, and preference among abducibles, are both expressible as rules in our language, but not so in the GSM approach.

7.3. Answer-set Semantics *[Gelfond et al. 1990]*

• The three-valued semantics subsumes the two-valued semantics of answer-set semantics.

• We have developed top-down procedures to compute our three-valued extended stable models comprising classical negations. These procedures are parametrizable to compute the two-valued special case of answer-sets.

• The techniques for hypothetical reasoning we've presented are in part adoptable by the answer-set semantics of extended logic programs.

Acknowledgements

We thank ESPRIT Basic Research Project COMPULOG (no. 3012), Instituto Nacional de Investigação Científica, Junta Nacional de Investigação Cientifica e Tecnologia, and Gabinete de Filosofia do Conhecimento for their support.

References

Eshghi, K. and R. Kowalski. (1989). Abduction Compared with Negation by Failure. In G. Levi and M. Martelli, Proceedings of ICLP89. 234–254, MIT Press.

Fitting, M. (1985). "A Kripke–Kleene Semantics for Logic Programs." Journal of Logic Programming. 2(4): 295–312.

Gelfond, M. and V. Lifschitz. (1988). The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, Proceedings of Fifth International Conference on Logic Programming. 1070–1080, MIT Press.

Gelfond, M. and V. Lifschitz. (1990). Logic Programs with Classical Negation. Proceedings of ICLP90. 579–597,

Kakas, A. C. and P. Mancarella. (1988). Anomalous Models and Abduction. Technical report, Imperial College

Kakas, A. C. and P. Mancarella. (1989). Generalized Stable Models:A Semantics for Abduction. Technical report, Imperial College

Kowalski, R. and F. Sadri. (1989). Logic Programs with Exceptions. Technical report, Imperial College

McCarthy, J. (1987). Applications of Circumscription to Formalizing Common-Sense Knowledge. Readings In Nonmonotonic Reasoning. Morgan Kaufman Publishers, Inc.

Pereira, L. M., J. J. Alferes and J. N. Aparício. (1990a). A Practical Introduction to Well Founded Semantics. Technical report, CRIA/UNINOVA

Pereira, L. M. and J. N. Aparício. (1990b). Default Reasoning as Abduction. Proceedings of

Pereira, L. M., J. N. Aparício and J. J. Alferes. (1990c). A Derivation Procedure for Extended Stable Model. Technical report, CRIA/UNINOVA

Pereira, L. M., J. N. Aparício and J. J. Alferes. (1990d). Nonmonotonic Reasoning with Well Founded Semantics. Technical report, CRIA/UNINOVA

Pereira, L. M., J. N. Aparício and J. J. Alferes. (1990e). Top-Down procedures for Well Founded Semantics. Technical report, CRIA/UNINOVA

Poole, D. L. (1987). A Logical Framework for Default Reasoning. Technical report, University of Waterloo

Przymusinska, H. and T. Przymusinski. (1990). Semantic Issues in Deductive Databases and Logic Programs. Formal Techniques in Artificial Intelligence. North Holland.

Przymusinski, T. (1990). Extended Stable Semantics for Normal and Disjunctive Programs. Proceedings of ICLP'90. 459-477,

Przymusinski, T. C. (1988). On The Relantionship Between Logic Programming and Non-monotonic Reasoning. Proceedings of AAAI88. 2: 444-448, Morgan Kaufmann.

Przymusinski, T. C. (1989a). Non-Monotonic Formalisms and Logic Programming. Proceedings of ICLP-89. 655-674, mit press.

Przymusinski, T. C. (1989b). Three-Valued Formalizations of Non-Monotonic Reasoning and Logic Programming. In R. Brachman, H. Levesque and R. Reiter, Proceedings of 1st Conference on Principles of Knowledge Representation and Reasoning. 341-348, Morgan Kaufmann Publishers, INC San Mateo, California.

Reiter, R. (1987). A Logic for default Reasoning. Readings In Nonmonotonic Reasoning. Morgan Kaufman Publishers, Inc.

van Emden, A. and R. Kowalski. (1976). "The Semantics of Predicate Logic as a Programming Language." JACM. 23(4): 733-742.

van Gelder, A., K. A. Ross and J. S. Schlipf. (1990). "The Well-Founded Semantics for General Logic Programs." JACM. (to appear): Available from first author as UCSC-CRL-88-16. Preliminary abstract appeared in Seventh ACM Symposium on Principles of Database Systems March 1988, pp. 221--230.