Stratified logic programs

Marek Sergot
Department of Computing
Imperial College, London

January 2005 v1.0c

Several groups of people — Apt, Blair, and Walker, Przymusiński, van Gelder, Naqui, Topor (and others) — came up with the same idea at about the same time. Some of them were working in logic programming, some of them were working in databases. Either way, the same basic idea:

- restrict attention to a special class of stratified logic programs/databases;
- then show that every stratified logic program/database has a unique model which is minimal and supported, and moreover which can be obtained by modifying the usual fixpoint construction \( T_P \) in a very natural way.

I follow Apt, Blair and Walker’s (ABW) presentation here. I omit most proofs and concentrate on motivating the basic ideas and stating the key results.


The notion of stratified program/database is very important. Very similar ideas have appeared elsewhere (‘separability’ in the literature on circumscription, for example).

Minimal supported (Herbrand) models

As already observed, a normal logic program/database such as \( P \):

\[
\begin{align*}
p(1) & \leftarrow \\
q(2) & \leftarrow \\
r(X) & \leftarrow \neg q(X)
\end{align*}
\]

has two different minimal Herbrand models:

\[
\{p(1), q(2), r(1)\} \quad \text{and} \quad \{p(1), q(2), q(1)\}
\]

The first of these is supported and the second one is not. (The atom \( q(1) \) in the second model has no support.)

A natural candidate for the semantics of a normal logic program/database is the supported minimal (Herbrand) model.

We just need to figure out when such a model exists, and when it is unique.

We know that, even for a normal logic program \( P \):

- \( T_P(I) \subseteq I \) means that \( I \) is a model of \( P \) (\( I \) is closed under \( T_P \));
- \( T_P(I) \supseteq I \) means that \( I \) is supported.

Unfortunately, we cannot find a fixpoint \( T_P(I) = I \) just by constructing \( T_P \) (or \( T_P(\emptyset) \)) because \( T_P \) is not sufficiently well behaved when \( P \) contains clauses with occurrences of negation-by-failure not in the body. \( T_P \) is not even monotonic in that case. But imposing a reasonable restriction on the form of \( P \) allows us to devise a natural means of constructing fixpoints of \( T_P \).

Stratified programs/databases: outline

Suppose we do the following:

- \( P_0 \) — any set of definite clauses (no negation), whose unique minimal model is \( M(P_0) = T_{P_0} \) (as usual — no problem). Call this model \( M_0 \).
- \( P_1 \) — any set of clauses (including negative literals in bodies) that define new predicates, except that negative literals are only allowed to refer to predicates that are already defined in \( P_0 \). Let \( M_1 \) be the closure of facts \( M_0 \) under the rules \( P_1 \); since all negative conditions in clauses of \( P_1 \) are fully determined by \( M_0 \), we can expect that this closure will be \( M_1 = T_{P_1}(M_0) \).
- \( P_2 \) — any set of clauses that define new predicates, except that negative literals are only allowed to refer to predicates that are already defined in \( P_0 \cup P_1 \). Let \( M_2 \) be the closure of facts \( M_1 \) under the rules \( P_2 \); we can expect that this closure will be \( M_2 = T_{P_2}(M_1) \).

\[ \vdots \]

etc

We would end up with a normal logic program/database \( P \) that is made up of several distinct strata, like this:

\[
P = P_0 \cup P_1 \cup \cdots \cup P_n \quad (P_i \text{ and } P_j \text{ disjoint for all } i \neq j)
\]

Such a program/database \( P \) is stratified. If our intuition is correct, then the last \( M_n = T_{P_n}(M_{n-1}) \) will be a model of \( P \). This remains to be checked. (\( M_n \) will turn out to be a minimal supported model of \( P \).)

The above description suggests we construct a stratified program/database \( P \) by successively constructing a sequence of strata. But this is not necessary. A stratified program/database \( P \) is one that could have been constructed in this way. It is only necessary that it can be partitioned into strata as described above, not that it was actually constructed in this way.
**Stratified programs/databases**

There are several alternative (and equivalent) definitions of *stratified*. Here below is one. Note the key feature: a stratified program forbids recursion ‘inside negation’. For example, any program containing a clause of the form

\[ p ← q, \neg p \]

is not stratified. Nor is any program containing clauses of the form

\[ p ← q, \neg r \]
\[ r ← s, p \]

And so on.

**Preliminary definitions**

- If \( P \) contains a clause of the form
  \[ A ← \ldots, p(\ldots), \ldots \]
  then predicate \( p \) occurs *positively* in the clause.
- If \( P \) contains a clause of the form
  \[ A ← \ldots, \neg p(\ldots), \ldots \]
  then predicate \( p \) occurs *negatively* in the clause.

Note that a predicate could occur both positively and negatively, even in the same clause.

**Definition** A normal logic program/database \( P \) is *stratified* when there is a partition

\[ P = P_0 ∪ P_1 ∪ \cdots ∪ P_n \quad (P_i \text{ and } P_j \text{ disjoint for all } i \neq j) \]

such that, for every predicate \( p \)

- the definition of \( p \) (all clauses with \( p \) in the head) is contained in one of the partitions/strata \( P_i \)

and, for each \( 1 \leq i \leq n \):

- if a predicate occurs *positively* in a clause of \( P_i \) then its definition is contained within \( \bigcup_{j ≤ i} P_j \)

- if a predicate occurs *negatively* in a clause of \( P_i \) then its definition is contained within \( \bigcup_{j > i} P_j \)

Note that a program/database is stratified if there is *any* such partition.

Notice also that the above definition does allow normal logic programs in the bottom stratum, as long as the negated predicates are undefined.

---

**Example** Logic program/database \( P \):

\[
\begin{align*}
p(X) & ← q(X), \not r(X) \\
p(X) & ← q(X), \not t(X) \\
r(X) & ← s(X), \not t(X) \\
t(a) & ← \\
s(a) & ← \\
s(b) & ← \\
qu(a) & ← \\
\end{align*}
\]

Here is one stratification of \( P \):

\[
\begin{align*}
P & = \{p(X) ← q(X), \not r(X), p(X) ← q(X), \not t(X)\} ∪ \\
& \{r(X) ← s(X), \not t(X)\} ∪ \\
& \{t(a) ←, s(a) ←, s(b) ←, q(a) ←\} \\
\end{align*}
\]

Here is another stratification of \( P \):

\[
\begin{align*}
P & = \{p(X) ← q(X), \not r(X), p(X) ← q(X), \not t(X)\} ∪ \\
& \{r(X) ← s(X), \not t(X)\} ∪ \\
& \{t(a) ←, s(a) ←, s(b) ←, q(a) ←\} \\
\end{align*}
\]

Here is another:

\[
\begin{align*}
P & = \{p(X) ← q(X), \not r(X), p(X) ← q(X), \not t(X)\} ∪ \\
& \{r(X) ← s(X), \not t(X), s(a) ←, s(b) ←, q(a) ←\} ∪ \\
& \{t(a) ←\} \\
\end{align*}
\]

etc, etc.

**Example** Logic program/database \( P \):

\[
\begin{align*}
p & ← \not q, \not r \\
q & ← s \\
r & ← t \\
s & ← \\
t & ← \\
\end{align*}
\]

Here is one stratification of \( P \):

\[
\begin{align*}
P & = \{p ← \not q, \not r\} ∪ \\
& \{q ← s\} ∪ \\
& \{r ← t\} ∪ \\
& \{s ←, t ←\} \\
\end{align*}
\]
Here is another stratification of $P$:

$$P = \{ p \leftarrow \text{not } q, \text{not } r \} \cup \\ \{ q \leftarrow \text{not } s, r \leftarrow \text{not } t \} \cup \\ \{ s \leftarrow, t \leftarrow \}$$

etc, etc.

**Example** Logic program/database $P$:

- $p \leftarrow q, \text{not } r$
- $r \leftarrow s, \text{not } p$
- $q \leftarrow$
- $s \leftarrow$

This program cannot be stratified. Notice that it contains a ‘recursion through negation’.

**Example** Logic program/database $P$:

- $p \leftarrow q$
- $q \leftarrow \text{not } p, s$
- $q \leftarrow q, \text{not } t$

$P$ is stratified. One stratum is enough:

$$P = \{ p \leftarrow q, q \leftarrow \text{not } r \}$$

It is easy to check that the stratification criteria are satisfied, because $r$ is undefined in $P$.

If you prefer, you could think of this as if there were a bottom stratum $\emptyset$, and $r$ is 'defined' in $\emptyset$.

### Dependency graphs

Here is a way of visualising the stratification conditions which some people find helpful.


- If program $P$ contains a clause of the form
  
  $$p(\ldots) \leftarrow \ldots, q(\ldots)\ldots$$

  then predicate $p$ refers (+) to $q$ (or 'depends (+) on $q$' if you prefer).

- If $P$ contains a clause of the form
  
  $$p(\ldots) \leftarrow \ldots, \text{not } q(\ldots)\ldots$$

  then predicate $p$ refers (-) to $q$ (or 'depends (-) on $q$').

Now the dependency graph for $P$:

- predicates at nodes;
- refers to edges labelled (+/−) between nodes.

**Example**

- $p \leftarrow q, r$
- $p \leftarrow \text{not } q, s$
- $q \leftarrow q, \text{not } t$

(Thanks to Silvia Dobrota (MEng4, 2013-14) for correcting a mistake in the original graph.)

**Lemma** The logic program $P$ is stratified iff the dependency graph for $P$ contains no cycles containing a negative edge.
Fixpoint semantics for stratified programs

$T_P$ is the immediate consequence operator for the normal logic program/database $P$. We use the ‘progressive’ version:

$$T_P(I) \overset{def}= T_P(I) \cup I$$

Here is the recipe:

- Stratify the program
  $$P = P_0 \cup P_1 \cup \cdots \cup P_n \quad (P_i \text{ and } P_j \text{ disjoint for all } i \neq j)$$

- Now construct
  $$M_0 = T_{P_0}^\uparrow(\emptyset) \quad \text{only the clauses in } P_0$$
  $$M_i = T_{P_i}^\uparrow(M_{i-1}) \quad \text{only the clauses in } P_i$$

- Take $M_P = M_n$ as the intended semantics of $P$.

I often refer to this as the ‘ABW iterated fixpoint construction’.

It still remains to establish that $M_P$ is a model of $P$. This is a coincidence.

Example Logic program/database $P$:

\begin{align*}
p(X) & \leftarrow q(X), \text{not } r(X) \\
r(X) & \leftarrow s(X), \text{not } t(X) \\
t(a) & \leftarrow \\
s(a) & \leftarrow \\
s(b) & \leftarrow \\
q(a) & \leftarrow
\end{align*}

One possible stratification of $P$:

\begin{align*}
P & = \{p(X) \leftarrow q(X), \text{not } r(X)\} \cup \cdots \cdots \cup \{t(a) \leftarrow, s(a) \leftarrow, s(b) \leftarrow, q(a) \leftarrow\}
\end{align*}

Applying the recipe:

$$M_0 = T_{P_0}^\uparrow(\emptyset) = \{t(a), s(a), s(b), q(a)\}$$
$$M_1 = T_{P_1}^\uparrow(M_0) = \{r(b), t(a), s(a), s(b), q(a)\}$$
$$M_2 = T_{P_2}^\uparrow(M_1) = \{p(a), r(b), t(a), s(a), s(b), q(a)\}$$

$M_2$ is a fixpoint of $T_P$. (Check it!) So $M_2$ is a supported model of $P$.

Suppose we take some other stratification of $P$? (Some were given earlier for this example.) If you apply the recipe, you will see that you get the same model, whatever stratification of $P$ you take. This is not a coincidence.

Example Here is another of the earlier examples. Logic program/database $P$:

\begin{align*}
p & \leftarrow q \\
q & \leftarrow \text{not } r
\end{align*}

To stratify $P$, one stratum is enough (because $r$ is not defined in $P$).

Applying the recipe:

\begin{align*}
T_{P_0}^\uparrow(\emptyset) & = \emptyset \\
T_{P_0}^\uparrow(\emptyset) & = \{q\} \\
T_{P_0}^\uparrow(\emptyset) & = \{p, q\} \\
\vdots \\
T_{P_0}^\uparrow(\emptyset) & = \{p, q\}
\end{align*}

$\{p, q\}$ is a fixpoint of $T_P$ (check) and so a supported model of $P$. 
**Stratified programs: Key results**

**Theorem** For a stratified normal logic program/database $P$, the ‘ABW iterated fixpoint’ construction described above constructs a set of ground atoms $M_P$ such that:

- $M_P$ is a fixpoint of $T_P$
- $M_P$ is a model of $P$
- $M_P$ is supported
- $M_P$ is a minimal model of $P$

**Theorem** $M_P$ is independent of the way the program $P$ is stratified.

That’s good! And finally there is a link to the Clark completion.

**Theorem** $M_P$ is a model of the Clark completion $\text{comp}(P)$.

**Implications for SLDNF**

**Implication 1**

$M_P$ is only one model of $\text{comp}(P)$. There may be other models. So, in general, more answers to a query will be correct with respect to the ABW semantics $M_P$ (true in the model $M_P$) than are correct with respect to $\text{comp}(P)$ (where we require true in all models of $\text{comp}(P)$ and not just $M_P$).

We know that SLDNF is incomplete with respect to the Clark completion $\text{comp}(P)$ – there may be consequences of $\text{comp}(P)$ that cannot be computed using the SLDNF procedure. So SLDNF is ‘even more incomplete’ with respect to the ABW semantics $M_P$ of $P$.

**Implication 2**

Since $M_P$ is a model of $\text{comp}(P)$, for every stratified program $P$

- $\text{comp}(P)$ is consistent.

**Locally stratified programs**

Consider this program

$p(a) ← \neg p(c)$
$p(b) ← \neg p(c)$

It is clearly not stratified. But suppose it had been written with three different 0-ary predicates instead of one unary predicate, like this:

$p_a ← \neg p_c$
$p_b ← \neg p_c$

This program is stratified.

This suggests that we could have a slightly more general notion of stratification – locally stratified (Przymusinski). Locally stratified does to ground atoms and ground instances of clauses what stratified does to predicates and clauses with variables.

**Locally stratified program/database $P$**:

- Replace $P$ by all ground instances of its clauses.
- Instead of assigning every predicate to a partition/stratum of $P$, assign every ground atom to a partition/stratum of $P$: the definition of that atom (the ground clauses with that atom as head) are put in this partition/stratum.
- Re-express the stratification criteria in terms of ground atoms rather than predicates: if a ground atom occurs positively in a clause in partition/stratum $P_i$, the definition of that ground atom must be contained in $\bigcup_{j ≤ i} P_j$; and if a ground atom occurs negatively in a clause in partition/stratum $P_i$, the definition of that ground atom must be contained in $\bigcup_{j < i} P_j$.

**Theorem** (Obvious)

Every stratified program is locally stratified.

**Remark** The generalisation to locally stratified is not particularly useful in practice, but there are references to it in the literature, and it can be a very useful device when trying to prove, e.g., the existence of a supported minimal model of a logic program/database.
Here is an idea that comes up frequently in the study of non-monotonic and default reasoning.

Let \( A \) be any set of formulas (not necessarily clauses) of some language and \( \alpha \) any formula of that language.

Standard logical consequence:
\[ A \models \alpha \implies \alpha \text{ is true in all models of } A \]

‘Preferential entailment’:
\[ A \models_{\text{pref}} \alpha \implies \alpha \text{ is true in all special preferred models of } A \]

Since the special ‘preferred’ models of \( A \) are a subset (usually a proper subset) of the set of all models of \( A \), more formulas will be ‘preferentially entailed’ by \( A \) than are logical consequences of \( A \).

The exact specification of special/preferred differs depending on context and application.

Here are some possibilities straightaway. Let \( P \) be a normal logic program/database. The special, preferred models of \( P \) could be, e.g.
- the models of \( \text{comp}(P) \)
- the minimal models of \( P \)
- the minimal supported models of \( P \)
- the ABW iterated fixpoint model, if it exists

(There are other possibilities)

Example

Compare some alternative semantics for the following databases/programs, in each case considering what answers are correct for queries \( ?-p \) and \( ?-q \).

\( DB_1 = \{ q \leftarrow \neg p \} \)

i) logical consequences of the Clark completion \( \text{comp}(DB_1) \)
\[ \text{comp}(DB_1) = \{ q \leftrightarrow \neg p, \neg p \} \]
\[ \text{comp}(DB_1) \models \neg p, \text{comp}(DB_1) \models q. \]
So \( ?-q \) gets ‘yes’ and \( ?-p \) gets ‘no’.

ii) minimal Herbrand models
\{\( p \)\} and \{\( q \)\} are the minimal models. \( p \) is true in one and false in the other. So \( p \) is not ‘preferentially entailed’ by the minimal model semantics, and neither is \( \neg p \).
Likewise for \( q \) and \( \neg q \).
Queries \( ?-p \) and \( ?-q \) both get ‘don’t know’.

iii) minimal supported Herbrand models
Of the two minimal models, only \{\( q \)\} is supported. So according to this semantics:
\( ?-q \) gets ‘yes’ and \( ?-p \) gets ‘no’.

iv) ABW iterated fixpoint model
\[ T_{DB_1}^{\omega}(\emptyset) = \{ q \} \]
So \( ?-q \) gets ‘yes’ and \( ?-p \) gets ‘no’.

\( DB_2 = \{ q \leftarrow \neg p, p \leftarrow p \} \)

i) logical consequences of the Clark completion \( \text{comp}(DB_2) \)
\[ \text{comp}(DB_2) = \{ q \leftrightarrow \neg p, p \leftrightarrow p \} \]
Neither \( p \) nor \( \neg p \) are consequences of \( \text{comp}(DB_2) \). Likewise for \( q \) and \( \neg q \).
So queries \( ?-p \) and \( ?-q \) both get ‘don’t know’.

ii) minimal Herbrand models
\{\( p \)\} and \{\( q \)\} are the minimal models. So just as for \( DB_1 \),
\( ?-p \) and \( ?-q \) both get ‘don’t know’.

iii) minimal supported Herbrand models
Unlike \( DB_1 \), now both minimal models \{\( p \)\} and \{\( q \)\} are supported. So:
\( ?-p \) and \( ?-q \) both get ‘don’t know’. 
iv) ABW iterated fixpoint model

$DB_2$ can be stratified: $P_0 = \{ p \leftarrow p \}, P_1 = \{ q \leftarrow \neg p \}.$

$T_{P_1}^{\downarrow}(T_{P_0}^{\downarrow}(\emptyset)) = T_{P_1}^{\downarrow}(\emptyset) = \{ q \}.$ Same as $DB_1.$

$DB_3 = \{ q \leftarrow \neg p, p \leftarrow \neg q \}$

i) logical consequences of the Clark completion $comp(DB_3)$

$comp(DB_3) = \{ q \leftrightarrow \neg p, p \leftrightarrow \neg q \}.$

Neither $p$ nor $\neg p$ are consequences of $comp(DB_3)$ Likewise for $q$ and $\neg q$.
So queries $?-p$ and $?-q$ both get ‘don’t know’.

ii) minimal Herbrand models

$\{ p \}$ and $\{ q \}$ are the minimal models. Same as $DB_2$.
Queries $?-p$ and $?-q$ both get ‘don’t know’.

iii) minimal supported Herbrand models

$\{ p \}$ and $\{ q \}$ are the minimal models. Both are supported. Same as $DB_2$.
Queries $?-p$ and $?-q$ both get ‘don’t know’.

iii) ABW iterated fixpoint model

$DB_3$ is not stratified. The ABW semantics is not defined.

**Final remark**

**Theorem** For any finite normal logic program $P$, a set of atoms is a model of $comp(P)$ iff it is a supported model of $P$.

The proof is easy:

$Proof$ $comp(P)$ consists of the following, for every atom $\alpha$:

$\alpha \leftarrow Body_i$

$\vdots$

$\alpha \leftarrow Body_k$

$\alpha \rightarrow Body_1 \lor \cdots \lor Body_k$

where $\alpha \leftarrow Body_i (1 \leq i \leq k)$ are all the clauses in $P$ with atom $\alpha$ in the head, or $\neg \alpha$

if there are no clauses in $P$ with atom $\alpha$ in the head.

So, a set $X$ of atoms satisfies (is a model of) $comp(P)$ iff for every atom $\alpha$:

(1) for every clause $\alpha \leftarrow Body_i$ in $P$, if $X \models Body_i$ then $\alpha \in X$;

(2) if $\alpha \in X$ then $X \models Body_{\alpha}$ for some clause $\alpha \leftarrow Body_i$ in $P$.

(1) says that $X$ is a model of $P$ ($X$ is closed under the clauses of $P$); (2) says that $X$ is supported by $P$. 