Stable Model implementation of Layer Supported Models by program transformation

Luís Moniz Pereira and Alexandre Miguel Pinto
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

Abstract. For practical applications, the use of top-down query-driven proof-procedures is convenient for an efficient use and computation of answers using Logic Programs as knowledge bases. A 2-valued semantics for Normal Logic Programs (NLPs) allowing for top-down query-solving is thus highly desirable, but the Stable Models semantics (SM) does not allow it, for lack of the relevance property. To overcome this limitation we introduced in [11], and summarize here, a new 2-valued semantics for NLPs — the Layer Supported Models semantics — which conservatively extends the SM semantics, enjoys relevance and cumulativity, guarantees model existence, and respects the Well-Founded Model. In this paper we exhibit a space and time linearly complex transformation, TR, from one propositional NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can then be computed by extant Stable Model implementations, providing a tool for the immediate generalized use of the new semantics and its applications. TR can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs.Moreover, TR can be employed in combination with the top-down query procedure of XSB-Prolog, and be applied just to the residual program corresponding to a query (in compliance with the relevance property of Layer Supported Models). The XSB-XASP interface then allows the program transform to be sent to Smodels for 2-valued evaluation.
Keywords: Stable Models, Layer Supported Models, Relevance, Layering, Program Transformation.

1 Introduction and Motivation

The semantics of Stable Models (SM) [7] is a cornerstone for the definition of some of the most important results in logic programming of the past two decades, providing an increase in logic programming declarativity and a new paradigm for program evaluation. When we need to know the 2-valued truth value of all the literals in a logic program for the problem we are modeling and solving, the only solution is to produce complete models. Depending on the intended semantics, in such cases, tools like *SModels* [9] or *DLV* [2] may be adequate because they can indeed compute whole models according to the SM semantics. However, the lack of some important properties of language semantics, like relevance, cumulativity and guarantee of model existence (enjoyed by, say, Well-Founded Semantics [6] (WFS)), somewhat reduces its applicability in practice, namely regarding abduction, creating difficulties in required pre- and post-processing. But WFS in turn does not produce 2-valued models, though these are often desired, nor guarantees 2-valued model existence.

SM semantics does not allow for top-down query-solving precisely because it does not enjoy the relevance property — and moreover, does not guarantee the existence of a model. Furthermore, frequently there is no need to compute whole models, like its implementations do, but just the partial models that sustain the answer to a query. Relevance would ensure these could be extended to whole models.

To overcome these limitations we developed in [11] (reviewed here) a new 2-valued semantics for NLPs — the Layer Supported Models (LSM) — which conservatively extends the SMs, enjoys relevance and cumulativity, guarantees model existence, and respects the Well-Founded Model (WFM) [6]. Intuitively, a program is conceptually partitioned into "layers" which are subsets of its rules, possibly involved in mutual loops. An atom is considered *true* in some layer supported model iff there is some rule for it at some layer, where all the literals in its body which are supported by rules of lower layers are also *true*. That is, a rule in a layer must, to be usable, have the support of rules in the layers below.

1.1 Integrity Constraints and Inconsistencies

There is a generalized consensus that Integrity Constraints (ICs) are denials, i.e., rules with $\perp$ as head, e.g,

$$\perp \leftarrow IC\_Body$$

where IC_Body is the body of the IC we wish to prevent from being true. When an NLP is enriched with such an IC, we say an interpretation containing IC_Body violates the IC and is, thus, inconsistent.

The classical notion of support — an atom a is supported in an interpretation I iff there is some rule in the NLP where *all* the literals of the body of the rule are true in I — the one upon which the SM semantics is defined, focuses on one rule at a time. This approach misses the global structure of the whole program. A consequence of such strict notion of support is that, even when an NLP is not enriched with any ICs, some particular patterns of rules might act as ICs. One of such patterns of rules is the Odd Loop Over Negation (OLON) [1]. This is, in fact, one of the reasons why the SM semantics fails to guarantee model existence for every NLP: the stability condition it imposes on models, stemming from the classical notion of support, is impossible to be complied with by OLONs. Actually, under SM semantics, sometimes OLONs are purposefully written as a means to impose ICs. Unfortunately, this confuses the two distinct aspects of rule-writing (wether with OLONs or not) and IC-writing (as denials).

Example 1. **OLON as IC.** Using SMs, one can write an IC, in order to prevent IC_Body from being in any model, with the single rule for some atom 'a':

$$a \leftarrow not\ a, IC\_Body$$

[1] OLON is a loop with an odd number of default negations in its circular call dependency path.

Since the SM semantics cannot provide a semantics to this rule whenever IC_Body holds, this type of OLON is used as IC. When writing such ICs under SMs one must be careful and make sure there are no other rules for a. But the really unintuitive thing about this kind of IC used under SM semantics is the meaning of the atom a. What does a represent?

The LSM semantics instead provides a semantics to all NLPs (OLONs included) thereby separating OLON semantics from IC compliance. In a practical implementation allowing top-down querying, the latter can be enforced by top-down checking that $not\ \bot$ is also part of the answer to the query.

After notation and background definitions, we summarize the formal definition of LSM semantics and its properties. Thereafter, we present a program transformation, TR, from one propositional (or ground) NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can be computed by extant Stable Model implementations, which also require grounding of programs. TR's linear space and time complexities are then examined. The transformation can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs, say. In the Implementation section we show how TR can be employed, in combination with the top-down query procedure of XSB-Prolog, it being sufficient to apply it solely to the residual program corresponding to a query (in compliance with the relevance property of Layer Supported Model). The XSB-XASP interface allows the program transform to be sent for Smodels for 2-valued evaluation. Conclusions and future work close the paper.

2 Background Notation and Definitions

Definition 1. ***Logic Rule.*** *A Logic Rule r has the general form*

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

where H, the B_i and the C_j are atoms.

We call H the head of the rule — also denoted by $head(r)$. And $body(r)$ denotes the set $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ of all the literals in the body of r. Throughout this paper we will use 'not ' to denote default negation. When the body of the rule is empty, we say the head of rule is a fact and we write the rule just as H.

Definition 2. ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of ground Logic Rules of the form in Definition 1.*

In this paper we focus only on NLPs, those whose heads of rules are positive literals, i.e., simple atoms; and there is default negation just in the bodies of the rules. Hence, when we write simply "program" or "logic program" we mean an NLP.

We write $heads(P)$ to denote the set of heads of rules of P, i.e., $heads(P) = \{head(r) : r \in P\}$. Abusing the default negation notation we write $not\ S$, where S is a set of literals, to denote $\{not\ s : s \in S\}$, and confound $not\ not\ s \equiv s$.

The shifting rule [5, 8] may be used to reduce disjunctive programs into NLPs, as may other known transformations, say from Extended LPs into NLPs ([3]).

3 Layering of Logic Programs

The well-known notion of stratification of LPs has been studied and used for decades now. But the common notion of stratification does not cover all LPs, i.e., there are some LPs which are non-stratified. The usual syntactic notions of dependency are mainly focused on atoms. They are based on a dependency graph induced by the rules of the program. Useful as these notions might be, for our purposes they are insufficient since they leave out important structural information about the call-graph of P. To encompass that information we define below the notion of a rule's dependency. Indeed, layering puts rules, not atoms, in layers.

Rules that depend on no other rule (except, possibly, itself) are placed in layer 1. Two rules are in the same layer if they depend on each other. The dependency relation between two rules is the transitive closure of the direct dependency relation, where rule r directly depends on rule r' (denoted as $r \leftarrow r'$) iff $head(r') \in body(r)$ or $not\ head(r') \in body(r)$. Moreover, a rule r is in a layer strictly above that of r' if r depends on r' but not vice-versa. We write $r \twoheadleftarrow r'$ to denote r depends (possibly indirectly) on r'.

The *relevant* part of P for some atom A, represented by $Rel_P(A)$, is the subset of rules of P with head A plus the set of rules of P whose heads the atom A depends on, cf. [4]. The relevant part of P for rule r, represented by $Rel_P(r)$, is the set containing the rule r itself plus the set of rules relevant for each atom r depends on.

Definition 3. ***Rule Layering of a Normal Logic Program*** P***.*** *Let P be an NLP. A rule layering function $Lf/1$ is any function defined over the rules of P, assigning each rule $r \in P$ an ordinal, such that the following holds:*

$$\forall_{r_1,r_2 \in P} \begin{cases} Lf(r_1) = 1 & \Leftarrow \neg\,(r_1 \twoheadleftarrow r_2) \wedge \quad r_2 \neq r_1 \\ Lf(r_1) = Lf(r_2) & \Leftarrow \quad (r_1 \twoheadleftarrow r_2) \wedge \quad (r_2 \twoheadleftarrow r_1) \\ Lf(r_1) > Lf(r_2) & \Leftarrow \quad (r_1 \twoheadleftarrow r_2) \wedge \neg\,(r_2 \twoheadleftarrow r_1) \end{cases}$$

The three cases above, which are patently mutually exclusive, leave out independent rules, i.e., rules that have no dependencies amongst themselves. According to this definition there is no restriction on which ordinal to assign to each independent rule in what the other rules' assignations are concerned.

A rule layering of program P is a partition $\ldots, P^i, P^j, \ldots$ of P such that P^i contains all rules r having $Lf(r) = i$. We write $P^{<\alpha}$ as an abbreviation of $\bigcup_{\beta<\alpha} P^\beta$, $P^{\leq\alpha}$ as an abbreviation of $P^{<\alpha} \cup P^\alpha$, and define $P^0 = P^{\leq 0} = \emptyset$. It follows immediately that $P = \bigcup_\alpha P^\alpha = \bigcup_\alpha P^{\leq\alpha}$, and also that the $\leq$ relation between layers is a total-order.

Amongst the several possible layerings of a program P we can always find the least one, i.e., the layering with least number of layers and where the ordinals of the layers are smallest, whilst respecting the rule layering function assignments. This least layering is easily seen to be unique. In the following, when referring to the program's "layering", we mean just such least layering.

Definition 4. ***Part of the body of a rule in loop.*** *Let P be an NLP, and r a rule of P^i. r^l is the subset of literals in the body of r in loop with r, i.e., their corresponding atoms have rules in the same layer as r.*

Formally, $r^l = body(r) \cap (heads(P^i) \cup not\ heads(P^i))$.

Theorem 1. ***The literals in*** r^l ***are in loop with*** r***.*** *All the atoms of the literals in* r^l *have at least one rule in loop with* r*.*

Proof. Clearly, r depends on rules for all literals in r^l because this is, by definition, a subset of $body(r)$. If it were true for any atom a of some literal $A \in r^l$ that it had no rule in loop with r, then, by definition 3, that all rules for a would be in layers strictly below that of r. In that case a would not be in $heads(P^i)$ (nor $not\ a$ in $not\ heads(P^i)$) since P^i is the layer where r is placed. Thus, a would not be in r^l contradicting the initial assumption "for any atom a of some literal $A \in r^l$". □

Example 2. **Layering example.** Consider the following program P, depicted along with the layer numbers for its least layering:

$c \leftarrow not\ d, not\ y, not\ a$		Layer 3
$d \leftarrow not\ c$		
$y \leftarrow not\ x$	$b \leftarrow not\ x$	Layer 2
$x \leftarrow not\ x$	b	Layer 1

Atom b has a fact rule: its body is empty, and therefore it is placed in Layer 1. The unique rule for x is also placed in Layer 1 in the least layering of P, because this rule only depends on itself. Both $b \leftarrow not\ x$ and $y \leftarrow not\ x$ are placed in Layer 2 because they depend on $not\ x$, and the unique rule for x is in Layer 1. The unique rule for c is placed in Layer 3 because it depends on $not\ y$. The rule for d is also placed in the same Layer 3 because it is in a loop with the rule for c. This program has two LSMs: $\{b, c, x\}$, and $\{b, d, x\}$.

4 Layer Supported Models Semantics

The Layer Supported Models semantics we now present is the result of the two new notions we introduced: the layering, formally introduced in section 3, which is a generalization of stratification; and the layered support, as a generalization of classical support. These two notions are the means to provide the 2-valued Layer Supported Models semantics.

Definition 5. ***Classically Supported interpretation.*** *An interpretation* M *of* P *is classically supported iff every atom* a *of* M *is classically supported in* M*, and this holds iff* a *has a rule* r *where all the literals in* $body(r)$ *are true in* M*.*

Definition 6. ***Layer Supported interpretation.*** *An interpretation* M *of* P *is layer supported iff every atom* a *of* M *is layer supported in* M*, and this holds iff* a *has a rule* r *where all the literals in* $(body(r) \setminus r^l)$ *are true in* M*.*

Theorem 2. ***Classical Support implies Layered Support.*** *Given an NLP* P*, an interpretation* M*, and an atom* a *such that* $a \in M$*, if* a *is classically supported in* M *then* a *is also layer supported in* M*.*

Proof. Trivial from definitions 5 and 6. □

In programs without odd loops layered supported models are classically supported too.

Intuitively, the minimal layer supported models up to and including a given layer, respect the minimal layer supported models up to the layers preceding it. It follows trivially that layer supported models are minimal models, by definition. This ensures the truth assignment to atoms in loops in higher layers is consistent with the truth assignments in loops in lower layers and that these take precedence in their truth labeling. As a consequence of the layered support requirement, layer supported models of each layer comply with the WFM of the layers equal to or below it. Combination of the (merely syntactic) notion of layering and the (semantic) notion of layered support makes the LSM semantics.

Definition 7. ***Layer Supported Model of*** P. *Let P be an NLP. A layer supported interpretation M is a Layer Supported Model of P iff*

$$\forall_\alpha M_{\leq\alpha} \textit{ is a minimal layer supported model of } P^{\leq\alpha}$$

where $M_{\leq\alpha}$ denotes the restriction of M to heads of rules in layers less or equal to α:

$$M_{\leq\alpha} \subseteq M \cap \{head(r) : Lf(r) \leq \alpha\}$$

The Layer Supported semantics of a program is just the intersection of all of its Layer Supported Models.

Example 3. **Layer Supported Models semantics.** Consider again the program from example 2. Its LS models are $\{b, c, x\}$, and $\{b, d, x\}$. According to LSM semantics b and x are *true* because they are in the intersection of all models. c and d are *undefined*, and a and y are *false*.

Layered support is a more general notion than that of perfect models [12], with similar structure. Perfect model semantics talks about "least models" rather than "minimal models" because in strata there can be no loops and so there is always a unique least model which is also the minimal one. Layers, as opposed to strata, may contain loops and thus there is not always a least model, so layers resort to minimal models, and these are guaranteed to exist (it is well known every NLP has minimal models).

The arguments in favor of the LSM semantics are presented in [10, 11], and are not detailed here. This paper assumes the LSM semantics and focuses on a program transformation.

4.1 Respect for the Well-Founded Model

Definition 8. ***Interpretation*** M ***of*** P ***respects the WFM of*** P. *An interpretation M respects the WFM of P iff M contains the set of all the* true *atoms of WFM, and it is contained by the set of* true *or* undefined *atoms of the WFM. Formally, $WFM^+(P) \subseteq M \subseteq WFM^{+u}(P)$.*

Theorem 3. ***Layer Supported Models respect the WFM.*** *Let P be a NLP. Each sub-LSM $M_{\leq\alpha}$ of LSM M respects the WFM of $P^{\leq\alpha}$.*

Proof. (The reader can skip this proof without loss for the following). By definition, each $M_{\leq\alpha}$ is a full LSM of $P^{\leq\alpha}$.

Consider $P^{\leq 1}$. Every $M_{\leq 1}$ contains the facts of P, and their direct positive consequences, since the rules for all of these are necessarily placed in the first layer in the least layering of P. Necessarily, $M_{\leq 1}$ contains all the *true* atoms of the WFM of $P^{\leq 1}$. Layer 1 also contains whichever loops that do not depend on any other atoms besides those which are the heads of the rules forming the loop. These loops that have no negative literals in the bodies are deterministic and, therefore, the heads of the rules forming the loop will be all *true* or all *false* in the WFM, depending if the bodies are fully supported by facts in the same layer, or not. In any case, a minimal model of this layer will necessarily contains all the *true* atoms of the WFM of $P^{\leq 1}$, i.e., $WFM^{+}(P^{\leq 1})$. For loops involving default negation, the atoms head of rules forming such loops are *undefined* in the WFM; some of them might be in $M_{\leq 1}$ too. Assume now there is some atom a *false* in the WFM of $P^{\leq 1}$ such that $a \in M_{\leq 1}$. a can only be *false* in the WFM of $P^{\leq 1}$ if either it has no rules or if every rule for a has a false body. In the first case, by definition, a cannot be $M_{\leq 1}$ because only heads of rules can be part of LSMs. In the second case, since in a LSM every atom must be layer supported, if all the bodies of all rules for a are false, a will not be layer supported and so it will not be in any LSM, in particular, not in $M_{\leq 1}$. Since $M_{\leq 1}$ contains all *true* atoms of WFM of$P^{\leq 1}$ and it contains no *false* atoms, it must be contained by the *true* or *undefined* atoms of WFM of $P^{\leq 1}$.

Consider now $P^{\leq i+1}$, and $M_{\leq i}$ a LSM of $P^{\leq i}$. Assuming in P^{i+1} all the atoms of $M_{\leq i}$ as *true*, there might be some bodies of rules of P^{i+1} which are true. In such case, a minimal model of P^{i+1} will also consider the heads of such rules to be *true* — these will necessarily comply with the layered support requirement, and will be *true* in the WFM of $P^{i+1} \cup M_{\leq i}$. For the same reasons indicated for layer 1, no *false* atom in the WFM of $P^{i+1} \cup M_{\leq i}$ could ever be considered *true* in $M_{\leq i+1}$. □

5 Program Transformation

The program transformation we now define provides a syntactical means of generating a program P' from an original program P, such that the SMs of P' coincide with the LSMs of P. It engenders an expedite means of computing LSMs using currently available tools like Smodels [9] and DLV [2]. The transformation can be query driven and performed on the fly, or previously preprocessed.

5.1 Top-down transformation

Performing the program transformation in top-down fashion assumes applying the transformation to each atom in the program in the call-graph of a query. The transformation involves traversing the call-graph for the atom, induced by its dependency rules, to detect and "solve" the OLONs, via the specific LSM-enforcing method described below. When traversing the call-graph for an atom, one given traverse branch may end by finding (1) a fact literal, or (2) a literal with no rules, or (3) a loop to a literal (or its default negation conjugate) already found earlier along that branch.

To produce P' from P we need a means to detect OLONs. The OLON detection mechanism we employ is a variant of Tarjan's Strongly Connected Component (SCC) detection algorithm [15], because OLONs are just SCCs which happen to have an odd number of default negations along its edges. Moreover, when an OLON is detected, we need another mechanism to change its rules, that is to produce and add new rules to the program, which make sure the atoms a in the OLON now have "stable" rules which do not depend on any OLON. We say such mechanism is an "OLON-solving" one. Trivial OLONs, i.e. with length 1 like that in Example 1 ($a \leftarrow not\ a, IC\_Body$), are "solved" simply by removing the $not\ a$ from the body of the rule. General OLONs, i.e. with length ≥ 3, have more complex (non-deterministic) solutions, described below.

Minimal Models of OLONs In general, an OLON has the form

$R_1 = \lambda_1 \leftarrow not\ \lambda_2, \Delta_1$

$R_2 = \lambda_2 \leftarrow not\ \lambda_3, \Delta_2$

$\vdots$

$R_n = \lambda_n \leftarrow not\ \lambda_1, \Delta_n$

where all the λ_i are atoms, and the Δ_j are arbitrary conjunction of literals which we refer to as "contexts". Assuming any λ_i *true* alone in some model suffices to satisfy any two rules of the OLON: one by rendering the head *true* and the other by rendering the body *false*.

$\lambda_{i-1} \leftarrow \sim \lambda_i, \Delta_{i-1}$, and

$\lambda_i \leftarrow \sim \lambda_{i+1}, \Delta_i$

A minimal set of such λ_i is what is needed to have a minimal model for the OLON. Since the number of rules n in OLON is odd we know that $\frac{n-1}{2}$ atoms satisfy $n-1$ rules of OLON. So, $\frac{n-1}{2} + 1 = \frac{n+1}{2}$ atoms satisfy all n rules of OLON, and that is the minimal number of λ_i atoms which are necessary to satisfy all the OLON's rules. This means that the remaining $n - \frac{n+1}{2} = \frac{n-1}{2}$ atoms λ_i must be *false* in the model in order for it to be minimal.

Taking a closer look at the OLON rules we see that λ_2 satisfies both the first and second rules; also λ_4 satisfies the third and fourth rules, and so on. So the set $\{\lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$ satisfies all rules in OLON except the last one. Adding λ_1 to this set, since λ_1 satisfies the last rule, we get one possible minimal model for OLON: $M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$. Every atom in M_{OLON} satisfies 2 rules of OLON alone, except λ_1, the last atom added. λ_1 satisfies alone only the last rule of OLON. The first rule of OLON — $\lambda_1 \leftarrow not\ \lambda_2, \Delta_1$ — despite being satisfied by λ_1, was already satisfied by λ_2. In this case, we call λ_1 the *top literal* of the OLON under M. The other Minimal Models of the OLON can be found in this manner simply by starting with λ_3, or λ_4, or any other λ_i as we did here with λ_2 as an example. Consider the $M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}\}$. Since $\sim \lambda_{i+1} \in body(R_i)$ for every $i < n$, and $\sim \lambda_1 \in body(R_n)$; under M_{OLON} all the $R_1, R_3, R_5, \ldots, R_n$ will have their bodies false. Likewise, all the $R_2, R_4, R_6, \ldots, R_{n-1}$ will have their bodies true under M_{OLON}.

This means that all $\lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}$ will have classically supported bodies (all body literals true), namely via rules $R_2, R_4, R_6, \ldots, R_{n-1}$, but not λ_1 — which has only layered support (all body literals of strictly lower layers true). "Solving an OLON" corresponds to adding a new rule which provides classical support for λ_1. Since this new rule must preserve the semantics of the rest of P, its body will contain only the conjunction of all the "contexts" Δ_j, plus the negation of the remaining $\lambda_3, \lambda_5, \lambda_7, \ldots, \lambda_n$ which were already considered *false* in the minimal model at hand.

These mechanisms can be seen at work in lines 2.10, 2.15, and 2.16 of the Transform Literal algorithm below.

Definition 9. ***Top-down program transformation.***

```
    input  : A program P
    output: A transformed program P'
1.1 context = $\emptyset$
1.2 stack = empty stack
1.3 P' =P
1.4 foreach atom $a$ of P do
1.5     Push ($a$, stack)
1.6     P' =P' $\cup$Transform Literal ($a$)
1.7     Pop ($a$, stack)
1.8 end
```

Algorithm 1: TR Program Transformation

The TR transformation consists in performing this literal transformation, for each individual atom of P. The Transform Literal algorithm implements a top-down, rule-directed, call-graph traversal variation of Tarjan's SCC detection mechanism. Moreover, when it encounters an OLON (line 2.9 of the algorithm), it creates (lines 2.13–2.17) and adds (line 2.18) a new rule for each literal involved in the OLON (line 2.11). The newly created and added rule renders its head true *only when the original OLON's* context *is* true*, but also only when that head is not classically supported, though being layered supported under the minimal model of the OLON it is part of.*

Example 4. **Solving OLONs.** Consider this program, coinciding with its residual:

$$a \leftarrow not\ a, b \qquad b \leftarrow c \qquad c \leftarrow not\ b, not\ a$$

Solving a query for a, we use its rule and immediately detect the OLON on a. The leaf $not\ a$ is removed; the rest of the body $\{b\}$ is kept as the Context under which the OLON on a is "active" — if b were to be false there would be no need to solve the OLON on a's rule. After all OLONs have been solved, we use the Contexts to create new rules that preserve the meaning of the original ones, except the new ones do not now depend on OLONs. The current Context for a is now just $\{b\}$ instead of the original $\{not\ a, b\}$.

Solving a query for b, we go on to solve c — $\{c\}$ being b's current Context. Solving c we find leaf $not\ b$. We remove c from b's Context, and add c's body $\{not\ b, not\ a\}$ to it. The OLON on b is detected and the $not\ b$ is removed from b's Context, which finally is just $\{not\ a\}$. As can be seen so far, updating Contexts is similar to performing an unfolding plus OLON detection and resolution by removing the dependency on the

```
input  : A literal $l$
output: A partial transformed program Pa'
2.1  previous context =context
2.2  Pa' =P
2.3  atom =atom $a$ of literal $l$; //removing the eventual $not$
2.4  if $a$ has been visited then
2.5      if $a$ or $not$ $a$ is in the stack then
2.6          scc root indx =lowest stack index where $a$ or $not$ $a$ can be found
2.7          nots seq = sequence of neg. lits from (scc root indx +1) to top indx
2.8          loop length = length of nots seq
2.9          if loop length is odd then
2.10             # nots in body = $\frac{(\text{loop length}-1)}{2}$
2.11             foreach 'not $x$' in nots seq do
2.12                 idx = index of $not$ $x$ in nots seq
2.13                 newbody = context
2.14                 for i=1 to # nots in body do
2.15                     newbody = newbody $\cup$
2.16                             {nots seq $((idx + 2 * i)$ mod loop length )}
2.17                 end
2.18                 newrule = $x$ $\leftarrow$newbody
2.19                 Pa' =Pa' $\cup$ {newrule }
2.20             end
2.21         end
2.22     end
2.23 else    // $a$ has not been visited yet
2.24     mark $a$ as visited
2.25     foreach rule $r = a \leftarrow b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m$ of P do
2.26         foreach $(not\ )b_i$ do
2.27             Push ($(not\ )b_i$, stack)
2.28             context =context $\cup\{b_1, \ldots, (not\ )b_{i-1}, (not\ )b_{i+1}, \ldots, not\ b_m\}$
2.29             Transform Literal ($(not\ )b_i$)
2.30             Pop ($(not\ )b_i$, stack)
2.31             context =previous context
2.32         end
2.33     end
2.34 end
```

Algorithm 2: Transform Literal

OLON. The new rule for b has final Context $\{not\ a\}$ for body. I.e., the new rule for b is $b \leftarrow not\ a$. Next, continuing a's final Context calculation, we remove b from a's Context and add $\{not\ a\}$ to it. This additional OLON is detected and $not\ a$ is removed from a's Context, now empty. Since we already exhausted a's dependency call-graph, the final body for the new rule for a is empty: a will be added as a fact. Moreover, a new rule for b will be added: $b \leftarrow not\ a$. The final transformed program is:

$a \leftarrow not\ a, b \qquad a \qquad b \leftarrow c \qquad b \leftarrow not\ a \qquad c \leftarrow not\ b, not\ a$

it has only one $SM = \{a\}$ the only LSM of the program. Mark layering is respected when solving OLONs: a's final rule depends on the answer to b's final rule.

Example 5. **Solving OLONs (2).** Consider this program, coinciding with its residual:

$$\begin{array}{l} a \leftarrow not\ b, x \\ b \leftarrow not\ c, y \\ c \leftarrow not\ a, z \\ x \\ y \\ z \end{array}$$

Solving a query for a we push it onto the stack, and take its rule $a \leftarrow not\ b, x$. We go on for literal $not\ b$ and consider the rest of the body $\{x\}$ as the current Context under which the OLON on a is "active". Push $not\ b$ onto the stack and take the rule for b. We go on to solve $not\ c$, and add the y to the current Context which now becomes $\{x, y\}$. Once more, push $not\ c$ onto the stack, take c's rule $c \leftarrow not\ a, z$, go on to solve $not\ a$ and add z to the current Context which is now $\{x, y, z\}$. When we now push $not\ a$ onto the stack, the OLON is detected and it "solving" begins. Three rules are created and added to the program $a \leftarrow not\ c, x, y, z$, $b \leftarrow not\ a, x, y, z$, and $c \leftarrow not\ b, x, y, z$. Together with the original program's rules they render "stable" the originally "non-stable" LSM $\{a, b, x, y, z\}$, $\{b, c, x, y, z\}$, and $\{a, c, x, y, z\}$. The final transformed program is:

$$\begin{array}{l} a \leftarrow not\ b, x \\ a \leftarrow not\ c, x, y, z \\ b \leftarrow not\ c, y \\ b \leftarrow not\ a, x, y, z \\ c \leftarrow not\ a, z \\ c \leftarrow not\ b, x, y, z \\ x \\ y \\ z \end{array}$$

TR transformation correctness The TR transformation steps occur only when OLONs are detected, and in those cases the transformation consists in adding extra rules. So, when there are no OLONs, the TR transformation's effect is $P' = P$, thus preserving the SMs of the original P. The additional Layer Supported Models of P are obtained by "solving" OLONs (by adding new rules in P'), so that the order of OLON solving complies with the layers of P. This is ensured because the top-down search, by its nature, solves OLONs conditional on their Context, and the latter will include same or lower layer literals, but not above layer ones. Finally, note Stable Models evaluation of P' itself respects the Well-Founded Semantics and hence Contexts evaluation respects layering, by Theorem 3.

5.2 Number of Models

Loop detection and the variety of their possible solutions concerns the number of Strongly Connect Components (SCCs) of the residual program.

Theorem 4. ***Maximum number of SCCs and of LSMs of a strongly connected residual component with N nodes.*** *They are, respectively, $\frac{N}{3}$ and $3^{\frac{N}{3}}$.*

Proof. Consider a component containing N nodes. A single odd loop with N nodes, by itself, contains N LSMs: each one obtained by minimally solving the implicit disjunction of the heads of the rules forming the OLON. Given only two OLONs in the component, with $N_1 + N_2 = N$ nodes, they could conceivably always be made independent of each other. Independent in the sense that each and every solution of one OLON combines separately with each and every solution of the other. To achieve this, iteratively duplicate as needed the rules of each OLON such that the combination of values of literals from the other loop are irrelevant. For example, in a program like

$$a \leftarrow not\ b \qquad b \leftarrow not\ c \qquad c \leftarrow not\ a, e$$
$$e \leftarrow not\ d \qquad d \leftarrow not\ f \qquad f \leftarrow not\ e, c$$

add the new rules $c \leftarrow not\ a, not\ e$ and $f \leftarrow not\ e, not\ c$ so that now the loop on a, b, c becomes independent of the truth of e, and the loop on e, d, f becomes independent of the truth of c. So, in the worst (more complex) case of two OLONs the number of LSMs is $N_1 * N_2$, in this case $3 * 3 = 9$.

Each loop over an even number of default negations (ELON), all by itself contains 2 LSMs, independently of the number N of its nodes. An OLON can always be made independent of an ELON by suitably and iteratively duplicating its rules, as per above. So an OLON with N_1 nodes dependent on a single ELON will, in the worst case, provide $2 * N_1$ LSMs.

It is apparent the highest number of LSMs in a component with N nodes can be gotten by combining only OLONs. Moreover, we have seen, the worst case is when these are independent.

Consider a component with N nodes and two OLONs with nodes $N_1 + N_2 = N$. The largest value of $N_1 * N_2$ is obtained when $N_1 = N_2 = \frac{N}{2}$. Indeed, since $N_2 = N - N_1$, take the derivative $\frac{d(N_1*(N-N_1))}{dN_1} = \frac{d(N*N_1)}{dN_1} - \frac{dN_1^2}{dN_1} = N - 2 * N_1$. To obtain the highest value make the derivative $N - 2 * N_1 = 0$, and hence $N_1 = \frac{N}{2}$.

Similarly, for $\sum_i N_i = N$, so $N_i = \frac{N}{i}$ gives the maximum value for $\prod_i N_i$. Thus, the maximum number of LSMs for a component of N nodes is obtained when all its (odd) loops have the same size.

And what is the size i that maximizes this value? Let us again use a derivative in i, in this case $\frac{di^{\frac{N}{i}}}{di}$ as the number of LSMs is $i^{\frac{N}{i}}$. Now $\frac{di^{\frac{N}{i}}}{di} = -N * i$. Equating it to zero we have $i = 0$. But i must be greater than zero and less than N. It is easy to see that the

i that affords the value of the derivative closest to zero is $i = 1$. But OLONs of length 1 afford no choices hence the least i that is meaningful is $i = 3$.

Hence the maximum number of LSMs of a component with N nodes is $3^{\frac{N}{3}}$. $\square$

Theorem 5. ***Maximum number of ELONs and of SMs of a SCC component with N nodes.*** *These are, respectively, $\frac{N}{2}$ and $2^{\frac{N}{2}}$.*

Proof. By the same reasoning as above, the maximum number of SMs of a component with N nodes is $2^{\frac{N}{2}}$, since there are no OLONs in SMs and so i can only be 2. $\square$

Corollary 1. ***Comparison between number of possible models of LSMs and SMs.*** *The highest number of models possible for LSMs, #LSMs, is larger than that for SMs, #SMs.*

Proof. By the two previous theorems, we know that for a component with N nodes, $\frac{\#LSMs}{\#SMs} = \frac{3^{\frac{N}{3}}}{2^{\frac{N}{2}}} = 3^{(N*[\frac{1}{3}-\frac{1}{2}*(log_3 2)])}$. $\square$

6 Implementation

The XSB Prolog system[2] is one of the most sophisticated, powerful, efficient and versatile implementations, with a focus on execution efficiency and interaction with external systems, implementing program evaluation following the WFS for NLPs. The XASP interface [1] (standing for XSB Answer Set Programming), is included in XSB Prolog as a practical programming interface to Smodels [9], one of the most successful and efficient implementations of the SMs over generalized LPs. The XASP system allows one not only to compute the models of a given NLP, but also to effectively combine 3-valued with 2-valued reasoning. The latter is achieved by using Smodels to compute the SMs of the so-called residual program, the one that results from a query evaluated in XSB using tabling [13]. A residual program is represented by delay lists, that is, the set of undefined literals for which the program could not find a complete proof, due to mutual dependencies or loops over default negation for that set of literals, detected by the XSB tabling mechanism. This coupling allows one to obtain a two-valued semantics of the residual, by completing the three-valued semantics the XSB system produces. The integration also allows to make use of and benefit from the relevance property of LSM semantics by queries.

In our implementation, detailed below, we use XASP to compute the query relevant residual program on demand. When the TR transformation is applied to it, the resulting program is sent to Smodels for computation of stable models of the relevant sub-program provided by the residue, which are then returned to the XSB-XASP side.

[2] XSB-Prolog and Smodels are freely available, at: http://xsb.sourceforge.net and http://www.tcs.hut.fi/Software/smodels.

Residual Program After launching a query in a top-down fashion we must obtain the relevant residual part of the program for the query. This is achieved in XSB Prolog using the `get_residual/2` predicate. According to the XSB Prolog's manual " the predicate `get_residual/2` unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the subgoal is taken to be conditional on the truth of the elements in the delay list". The delay list is the list of literals whose truth value could not be determined to be *true* nor *false*, i.e., their truth value is *undefined* in the WFM of the program.

It is possible to obtain the *residual* clause of a solution for a query literal, and in turn the *residual* clauses for the literals in its body, and so on. This way we can reconstruct the complete relevant residual part of the KB for the literal — we call this a *residual program* or *reduct* for that solution to the query.

More than one such *residual program* can be obtained for the query, on backtracking. Each *reduct* consists only of partially evaluated rules, with respect to the WFM, whose heads are atoms relevant for the initial query literal, and whose bodies are just the *residual* part of the bodies of the original KB's rules. This way, not only do we get just the relevant part of the KB for the literal, we also get precisely the part of those rules bodies still *undefined*, i.e., those that are involved in Loops Over Negation.

Dealing with the Query and Integrity Constraints ICs are written as just $falsum \leftarrow IC\_Body$. An Smodels IC preventing $falsum$ from being *true* (`:- falsum`) is enforced whenever a transformed program is sent to Smodels. Another two rules are added to the Smodels clause store through XASP: one creates an auxiliary rule for the initially posed query; with the form: `lsmGoal :- Query`, where `Query` is the query conjunct posed by the user. The second rule just prevents Smodels from having any model where the `lsmGoal` does not hold, having the form: `:- not lsmGoal`.

The XSB Prolog source code for the meta-interpreter, based on this program transformation, is available at http://centria.di.fct.unl.pt/∼amp/software.html

7 Conclusions and Future Work

We have recapped the LSMs semantics for *all* NLPs, complying with desirable requirements: 2-valued semantics, conservatively extending SMs, guarantee of model existence, relevance and cumulativity, plus respecting the WFM.

We have exhibited a space and time linearly complex transformation, TR, from one propositional NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can then be computed by extant Stable Model implementations. TR can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs. Moreover, it can be employed in combination with the top-down query procedure of XSB-Prolog, and be applied solely to the residual program corresponding to a query. The XSB-XASP interface subsequently allows the program transform to be sent for Smodels for 2-valued evaluation.

The applications afforded by LSMs are *all* those of SMs, plus those where odd loops over default negation (OLONs) are actually employed for problem domain representation, as we have shown in examples 4 and 5. The guarantee of model existence is essential in applications where knowledge sources are diverse (like in the Semantic Web), and wherever the bringing together of such knowledge (automated or not) can give rise to OLONs that would otherwise prevent the resulting program from having a semantics, thereby brusquely terminating the application. A similar situation can be brought about by self- and mutually-updating programs, including in the learning setting, where unforeseen OLONs would stop short an ongoing process if the SM semantics were in use. Hence, apparently there is only to gain in exploring the adept move from SM semantics to the more general LSM one, given that the latter is readily implementable through the program transformation TR, introduced here for the first time.

Work under way [14] concerns an XSB engine level efficient implementation of the LSM semantics, and the exploration of its wider scope of applications with respect to ASP, and namely in combination with abduction and constructive negation.

Finally, the concepts and techniques introduced in this paper are readily adoptable by other logic programming systems and implementations.

References

1. L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*, 1999. http://xsb.sourceforge.net/packages/xasp.pdf.
2. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *Workshop in Logic Programming*, 1997.
3. C.V. Damásio and L. M. Pereira. Default negated conclusions: Why not? In R. Dyckhoff et al, editor, *Extensions of Logic Programming, ELP'96*, volume 1050 of *LNAI*, pages 103–118. Springer-Verlag, 1996.
4. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I, II. *Fundamenta Informaticae*, XXII(3):227–255, 257–288, 1995.
5. J. Dix, G. Gottlob, V.W. Marek, and C. Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996.
6. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
7. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
8. M. Gelfond, H. Przymusinska, V. Lifschitz, and M. Truszczynski. Disjunctive defaults. In *KR-91*, pages 230–237, 1991.
9. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. LPNMR'97*, LNAI 1265, pages 420–429, 1997.
10. L.M. Pereira and A.M. Pinto. Layered models top-down querying of normal logic programs. In *Procs. PADL'09*, volume 5418 of *LNCS*, pages 254–268. Springer, January 2009.
11. Luís Moniz Pereira and Alexandre Miguel Pinto. Layer supported models of logic programs. In E. Erdem, F. Lin, and T. Schaub, editors, *Procs. 10th LPNMR*, LNAI. Springer, September 2009. http://centria.di.fct.unl.pt/∼lmp/publications/online-papers/LSMs.pdf (long version).
12. T.C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
13. T. Swift. Tabling for non-monotonic programming. *AMAI*, 25(3-4):201–240, 1999.
14. Terrance Swift, Alexandre Miguel Pinto, and Luís Moniz Pereira. Incremental answer completion in xsb-prolog. In *Procs. 25th ICLP*, LNCS. Springer-Verlag, July 2009.
15. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.