Stable versus Layered Logic Program Semantics

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. Additionally, abductive reasoning on demand is intrinsically a top-down search method. 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 [24], and review 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 also exhibit a 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.
In the context of abduction in Logic Programs, when finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, that is without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. To accomplish this mechanism, we present the concept of Inspection Point in Abductive Logic Programs, and show, by means of examples, how one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions.
Keywords: Stable Models, Layer Supported Models, Relevance, Layering, Abduction, Side-effects, Program Transformation.

1 Introduction and Motivation

The semantics of Stable Models (SM) [14] 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* [19] or *DLV* [7] 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 [13] (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.

Furthermore, with SM semantics, in an abductive reasoning situation, computing the whole model entails pronouncement about every abducible whether or not it is relevant to the problem at hand, and subsequently filtering the irrelevant ones. When we just want to find an existential answer to a query, we either compute a whole model and check if it entails the query (the way SM semantics does), or, if the underlying semantics we use enjoys the *relevance* property — which SM semantics do not — we can simply use a top-down proof-procedure (*à la* Prolog), and abduce by need. In this second case, the user does not pay the price of computing a whole model, nor the price of abducing all possible abducibles or their negations, and then filtering irrelevant ones, because the only abducibles considered will be those needed to answer the query.

To overcome these limitations we developed in [24] (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) [13]. We say a semantics is a conservative extension of another one if it gives the same semantics to programs as the latter, whenever the latter is defined, and also provides semantics to programs for which the latter does not.

The LSM semantics builds upon the foundational step of the Layered Models semantics presented in [27] by imposing a layered support requirement resulting in WFM consonance. In [27], neither layered support nor WFM respect are required. Intuitively, a program is conceptually partitioned into "layers" which are subsets of its rules, possibly involved in mutual loops. An atom is considered *true* if 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*. Otherwise that conclusion is false. That is, a rule in a layer must, to be usable, have the support of rules in the layers below.

Although Stable Models is a great semantics with a simple definition, it nevertheless fails to provide semantics for all normal programs. LSM coincides with Stable Models whenever the latter is defined (no odd loops ou infinite chains), — [12] — and thus can afford the same simple definition in that class of programs. The definition for the class of all programs necessarily requires a more complex definition, essentially resting on a required generalization of stratification — the layering — and then simply using minimal models on successive layers.

The core reason SM semantics fails to guarantee model existence for every NLP is that the stability condition it imposes on models is impossible to be complied with by Odd Loops Over Negation (OLONs). An OLON is a loop with an odd number of default negations in its circular call dependency path. In fact, the SM semantics community uses

that inability as a means to impose Integrity Constraints (ICs), such as $a \leftarrow not\ a, X$, where the OLON over a prevents X from being *true* in any model.

The LSM semantics provides a semantics to all NLPs. In the $a \leftarrow not\ a, X$ example above, whenever X is *true*, the only LSM is $\{a, X\}$. For LSM semantics OLONs are not ICs. ICs are implemented with rules for reserved atom $falsum$, of the form $falsum \leftarrow X$, where X is the body of the IC we wish to prevent being true. This does not prevent $falsum$ from being in some models. From a theoretical standpoint this means that the LSM semantics does not include an IC compliance enforcement mechanism. ICs must be dealt with in two possible ways: either by 1) a syntactic post-processing step, as a "test phase" after the model generation "generate phase"; or by 2) embedding the IC compliance in a query-driven (partial) model computation, where such method can be a top-down query-solving one *a la* Prolog, since the LSM semantics enjoys relevance. In this second case, the user must conjoin query goals with $not\ falsum$. If inconsistency examination is desired, like in the 1) case above, models including $falsum$ can be discarded *a posteriori*. This is how LSM semantics separates OLON semantics from IC compliance.

After notation and background definitions, we summarize the formal definition of LSM semantics and its properties. Thereafter, in a section that may be skipped without loss of continuity, 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 space and time complexities are then examined. TR can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs, say. 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 for Smodels for 2-valued evaluation. Thus, for existential query answering there is no need to compute total models, but just the partial models that sustain the answer to the query, or one might simply know a model exists without producing it; relevance ensures these can be extended to total models.

In its specific implementation section we indicate 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. Implementation details and attending algorithms can be found in [25].

Next, issues of reasoning with logic programs are addressed in section 7 where, in particular, we look at abductive reasoning and the nature of backward and forward chaining in their relationship to the issues of query answering and of side-effects examination within an abductive framework. In section 8 we then introduce inspection points to address such issues, illustrate their need and use with examples, and provide for them a declarative semantics. The touchstone of enabling inspection points can be construed as meta-abduction, by (meta-)abducing an "abduction" to check (i.e. to passively verify) that a certain concrete abduction is indeed adopted in a purported abductive solution. In

section 8.2 we surmise their implementation, the latter's workings being fully given and exemplified in [23]. We have implemented inspection points on top of already existing 3- and 2-valued abduction solving systems — ABDUAL and XSB-XASP — in a way that can be adopted by other systems too.

Our semantics can easily be extended to deal with Disjunctive Logic Programs (DisjLPs) and Extended Logic Programs (ELPs, including explicit negation), by means of the shifting rule and other well-known program transformations [1, 10, 15], thus providing a practical, comprehensive and advantageous alternative to SMs-based Answer-Set Programming. 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. 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 mainly 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.

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 these notions might be, for our purposes they are insufficient as they leave out important structural information about the call-graph of P. To cover that information we also define below the notion of a rule's dependency. Indeed, layering puts rules in layers, not atoms. An atom B directly depends on atom A in P iff there is at least one rule with head B and with A or $not\ A$ in the body. An atom's dependency is just the transitive closure of the atom's direct dependency. A rule directly depends on atom B iff any of B or $not\ B$ is in its body. A rule's dependency is just the transitive closure of the rule's direct dependency. 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. [9]. Likewise for the relevant part for an atom A notion [9], we define and present the notion of relevant part for a rule r. 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 2. ***Parts of the body of a rule.*** *Let $r = H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ be a rule of P. Then, $r^l = \{B_i, not\ C_j : B_i$ depends on $H \wedge C_j$ depends on $H\}$. Also, $r^B = \{B_i : B_i \in (body(r) \setminus r^l)\}$, and $r^C = \{C_j : not\ C_j \in (body(r) \setminus r^l)\}$.*

Definition 3. ***HighLayer function.*** *The $HighLayer$ function is defined over a set of literals: its result is the highest layer number of all the rules for all the literals in the set, or zero if the set is empty.*

Definition 4. ***Layering of a Logic Program*** P**.** *Given a logic program P a layering function $L/1$ is just any function defined over the rules of P', where P' is obtained from P by adding a rule of the form $H \leftarrow falsum$ for every atom H with no rules in P, assigning each rule $r \in P'$ a positive integer, such that:*

- $L(r) = 0$ *if* $falsum \in body(r)$*, otherwise*
- $L(r) \geq max(HighLayer(r^l), HighLayer(r^B), (HighLayer(r^C) + 1))$

A layering of program P is a partition $P^1, \ldots, P^n$ of P such that P^i contains all rules r having $L(r) = i$, i.e., those which depend only on the rules in the same layer or layers below it.

This notion of layering does not correspond to any level-mapping [16], since the later is defined over atoms, and the former is defined over rules. Also, due to the definition of dependency, layering does not coincide with stratification [3], nor does it coincide with the layer definition of [28]. However, when the program at hand is stratified (according to [3]) it can easily be seen that its respective layering coincides with its stratification. In this sense, layering, which is always defined, is a generalization of the stratification.

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 integers of the layers are smallest. In the remainder of the paper when referring to the program's layering we mean such least layering (easily seen to be unique).

Example 1. **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
$a \leftarrow falsum$		Layer 0

Atom a has no rules so its now created unique rule $a \leftarrow falsum$ is placed in Layer 0. Atom b has a fact rule r_{b_1}: its body is empty, and therefore all $HighLayer(r^l_{b_1})$, $HighLayer(r^B_{b_1})$, and $HighLayer(r^C_{b_1})$ are 0 (zero). Hence, $L(r_{b_1}) = max(0, 0, (0 + 1)) = max(0, 0, 1) = 1$, where r_{b_1} is the fact rule for b, placed in Layer 1.

The unique rule for x, r_x is also placed in Layer 1 in the least layering of P because $HighLayer(r^l_x) = L(r_x)$, $HighLayer(r^B_x) = 0$, and $HighLayer(r^C_x) = 0$. So, $L(r_x) = max(L(r_x), 0, (0 + 1)) = max(L(r_x), 0, 1) = 1$, in the least layering.

The unique rule for c, r_c is placed in Layer 3 because $HighLayer(r^C_c) = 2$, $HighLayer(r^B_c) = 0$, and $HighLayer(r^l_c) = HighLayer(r_d) = 3$. By the same token, r_d is placed in the same Layer 3. Both r_{b_2} and r_y are placed in Layer 2.

This program has two LSMs: $\{b, c, x\}$, and $\{b, d, x\}$.

The complexity of computing the Layering of a given ground NLP is conjectured to be in the same order as that of the Tarjan's Strongly Connected Components detection Algorithm, which is known to be linear in the number of nodes and edges on the dependency graph: $O(|N| + |E|)$.

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 desired 2-valued semantics which respects the WFM, as we will see below.

An interpretation M of P is classically supported iff every atom a of M is classically supported in M, i.e., *all* the literals in the body of some rule for a are *true* under M in order for a to be supported under M.

Definition 5. ***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 literals in $(body(r) \setminus r^l)$ are true in M. Otherwise, it follows that a is false.*

Theorem 1. ***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 the definitions of classical support and layered support. □

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

Example 2. **Layered Unsupported Loop.** Consider program P:

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

The only rule for b is in the first layer of P. Since it is a fact it is always *true* in the WFM. Knowing this, the body of the rule for a is *false* because unsupported (both classically and layered). Since it is the only rule for a, its truth value is *false* in the WFM, and, consequently, c is *true* in the WFM. This is the intuitively desirable semantics for P, which corresponds to its LSM semantics. LM and the LSM semantics differences reside both in their layering notion and the layered support requisite of Def. 5. In this example, if we used LM semantics, which does not exact layered support, there would be two models: $LM_1 = \{b, a\}$ and $LM_2 = \{b, c\}$. $\{b\}$ is the only minimal model for the first layer and there are two minimal model extensions for the second layer, as a is not necessarily false in the LM semantics because Def. 5 is not adopted. Lack of layered support lets LM semantics fail to comply with the WFM. Note that adding a rule like $b \leftarrow a$ would not affect the semantics of the program, according to LSM. This is so because, such rule would be placed in the same layer with the rules for a and c, but leaving the fact rule b in the strictly lower layer.

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 6. ***Layer Supported Model of*** P**.** *Let* $P^1, \ldots, P^n$ *be the least layering of* P*. A layer supported interpretation* M *is a Layer Supported Model of* P *iff*

$$\forall_{1 \leq i \leq n} M|_{\leq i} \textit{ is a minimal layer supported model of } \cup_{1 \leq j \leq i} P^j$$

where $M|_{\leq i}$ *denotes the restriction of* M *to heads of rules in layers less or equal to* i*:*

$$M|_{\leq i} \subseteq M \cap \{head(r) : L(r) \leq i\}$$

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 1. 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 [30], 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 simple transformation step adding a rule of the form $a \leftarrow falsum$ for atoms a with no rules in P enforces compliance with the Closed World Assumption (CWA).

The principle used by LSMs to provide semantics to any NLP — whether with OLONs or not — is to accept all, and only, the minimal models that are layer supported, i.e., that respect the layers of the program. The principle used by SMs to provide semantics to only some NLPs is a "stability" (fixed-point) condition imposed on the SMs by the Gelfond-Lifschitz operator.

In [27] the authors present an example (7) where three alternative joint vacation solutions are found by the LM semantics, for a vacation problem modeled by an OLON. The solutions found actually coincide with those found by the LSM semantics. We recall now a syntactically similar example (from [29]) but with different intended semantics, and show how it can be attained in LSM by means of ICs.

Example 4. **Working example [29].**

$$tired \leftarrow not\ sleep \qquad sleep \leftarrow not\ work \qquad work \leftarrow not\ tired$$

As in the example 7 of [27], the LSM semantics would provide three solutions for the above program: $\{work, tired\}, \{work, sleep\}, \{sleep, tired\}$. Although some (or even all!) of these solutions might be actually plausible in a real world case, they are not, in general, the intended semantics for this example. With the LSM semantics, the way to prune away some (or all) of these solutions is by means of ICs. For example, to eliminate

the $\{work, sleep\}$ solution we would just need to add the IC $falsum \leftarrow work, sleep$. Again, an IC compliance mechanism, such as conjoining $not\ falsum$ to the query, must be employed by the user in order to eliminate the undesired models.

Example 5. **Jurisprudential reasoning.** A murder suspect not preventively detained is likely to destroy evidence, and in that case the suspect shall be preventively detained:

$$\begin{aligned} likely\_destroy\_evidence(murder\_suspect) \leftarrow & \\ & not\ preventively\_detain(murder\_suspect) \\ preventively\_detain(murder\_suspect) \quad \leftarrow & \\ & likely\_destroy\_evidence(murder\_suspect) \end{aligned}$$

There is no SM, and a single $LSM = \{preventively\_detain(murder\_suspect)\}$. This jurisprudential reasoning is carried out without need for a $murder\_suspect$ to exist now. Should we wish, LSM's cumulativity (cf. below) allows adding the model literal as a fact.

Example 6. **Program with depth-unbound layering.** A typical case of an infinitely ground program (actually the only one with real theoretical interest, to the best of our knowledge) was presented by François Fages in [12]. We repeat it here for illustration and explanation.

$$p(X) \leftarrow p(s(X)) \qquad p(X) \leftarrow not\ p(s(X))$$

Ground (layered) version of this program, assuming there only one constant 0 (zero):

$$\begin{array}{cc} p(0) \leftarrow p(s(0)) & p(0) \leftarrow not\ p(s(0)) \\ p(s(0)) \leftarrow p(s(s(0))) & p(s(0)) \leftarrow not\ p(s(s(0))) \\ p(s(s(0))) \leftarrow p(s(s(s(0)))) & p(s(s(0))) \leftarrow not\ p(s(s(s(0)))) \\ \vdots \leftarrow \vdots & \vdots \leftarrow \vdots \end{array}$$

The only layer supported model of this program is $\{p(0), p(s(0)), p(s(s(0)) \ldots\}$ or, in a non-ground form, $\{p(X)\}$. The theoretical interest of this program lies in that, although it has no OLONs it still has no SMs either because no rule is supported (under the usual notion of support), thereby showing there is a whole other class of NLPs to which the SMs semantics provides no model, due to the notion of support used.

5 Properties of the Layer Supported Models semantics

Although the definition of the LSMs semantics is different from the LMs of [27], this new refinement enjoys the desirable properties of the LMs semantics, namely: guarantee of model existence, relevance, cumulativity, ([9]) and being a conservative extension of the SMs semantics. Guarantee of existence ensures all normal programs have a semantics; relevance allows for simple top-down querying concerning truth in a model (like in Prolog) and is essential for abduction by need; and cumulativity means that atoms true in the semantics can be added as facts without changing it — none of these properties are enjoyed by Stable Models. A semantics is a conservative extension of

another one if it gives the same semantics to programs as the latter, whenever the latter is defined, and also provides semantics to programs for which the latter does not, that is the models of the semantics being extended are kept.

Moreover, and this is the main contribution of the LSMs semantics, it respects the Well-Founded Model. It is not hard to recognize that the LM semantics [27] is the conservative generalization LSM semantics one obtains by removing the layered support condition on interpretations prescribed in Definition 6.

The complexity analysis of this semantics is left out of this paper. Nonetheless, a brief note is due. Model existence is guaranteed for every NLP, hence the complexity of finding if one LSM exists is trivial, when compared to SMs semantics. Brave reasoning — finding if there is any model of the program where some atom a is true — is an intrinsically NP-hard task from the computational complexity point of view. But since LSM semantics enjoys relevance, the computational scope of this task can be reduced to consider only $Rel_P(a)$, instead of the whole P. From a practical standpoint, this can have a significant impact in the performance of concrete applications. Cautious reasoning (finding out if some atom a is in all models) in the LSM semantics should have the same computational complexity as in the SMs, i.e., coNP-complete.

5.1 Respect for the Well-Founded Model

Definition 7. ***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 the WFM of P, and it contains no* false *atoms of the WFM of P.*

Theorem 2. ***Layer Supported Models respect the WFM.*** *Let P be an NLP, and $P^{\leq i}$ denote $\bigcup_{1 \leq j \leq i} P^j$, where P^j is P's j layer. Each LSM $M|_{\leq i}$ of $P^{\leq i}$, where $M \supseteq M|_{\leq i}$, respects the WFM of $P^i \cup M|_{<i}$.*

Proof. By hypothesis, each $M|_{\leq i}$ is a full LSM of $P^{\leq i}$. Consider P^1. Any $M|_{\leq 1}$ 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. Hence, $M|_{\leq 1}$ contains all the *true* atoms of the WFM of P^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. Any such loops having 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 of P^1, depending on whether the bodies are fully supported by facts in the same layer, or not, and, in the latter case, if the rules are not involved in other types of loop making their heads undefined. In any case, an LSM of this layer will by necessity contain all the *true* atoms of the WFM of P^1. On the other hand, assume there is some atom b which is *false* in the WFM of P^1. b being *false* in the WFM means that either b has no rules or that every rule for b has an unsatisfiable body in P^1. In the first case, by definition 6 we know that b cannot be in any LSM. In the second case, every unsatisfiable body is necessarily unsupported, both classically and layered. Hence, b cannot be in any LSM of P^1. This means that any LSM contains no atoms *false* in the WFM of P^1, and, therefore, that they must respect the WFM of P^1.

By hypothesis $M|_{\leq i+1}$ is an LSM of $P^{\leq i+1}$ iff $M|_{\leq i+1} \supseteq M|_{\leq i}$, for some LSM $M|_{\leq i}$ of $P^{\leq i}$, which means the LSMs $M|_{\leq i+1}$ of $P^{\leq i+1}$ are exactly the LSMs $M|_{\leq i+1}$

of $P^{i+1} \cup M|_{\leq i}$. Adding the $M|_{\leq i}$ atoms as facts imposes them as *true* in the WFM of $P^{i+1} \cup M|_{\leq i}$. The then deterministically *true* consequences of layer $i+1$ — the *true* atoms of the WFM of $P^{i+1} \cup M|_{\leq i}$ — become necessarily present in every minimal model of $P^{i+1} \cup M|_{\leq i}$, and therefore in its every LSM $M|_{\leq i+1}$. On the other hand, every atom b *false* in the WFM of $P^{i+1} \cup M|_{\leq i}$ has now unsatisfiable bodies in all its rules (up to this layer $i+1$). Hence, b cannot be in any LSM of $P^{i+1} \cup M|_{\leq i}$. Therefore, every $M|_{\leq i+1}$ respects the WFM of $P^{i+1} \cup M|_{\leq i}$. Hence, more generally, every $M|_{\leq i}$ respects the WFM of $P^i \cup M|_{<i}$

□

5.2 Other Comparisons

We next argue informally that the LSMs and the Revised Stable Models (RSMs) defined in [21] are the same. The equivalence is rather immediate. Take Definitions 2 and 3 of [21] characterizing the RSMs:

Definition 8. ***Sustainable Set (Definition 2 in [21]).*** *Intuitively, we say a set S is sustainable in P iff any atom $a \in S$ does not go against the well-founded consequences of the remaining atoms in S, whenever, $S \setminus \{a\}$ itself is a sustainable set. The empty set by definition is sustainable. Not going against means that atom a cannot be false in the WFM of $P \cup S \setminus \{a\}$, i.e., a is either true or undefined. That is, it belongs to set $\Gamma_{P \cup S \setminus \{a\}}(WFM(P \cup S \setminus \{a\}))$. Formally, we say S is sustainable iff*

$\forall_{a \in S} S \setminus \{a\}$ *is sustainable* $\Rightarrow a \in \Gamma_{P \cup S \setminus \{a\}}(WFM(P \cup S \setminus \{a\}))$

where $\Gamma_P(I)$ is just the Gelfond-Lifschitz program division of P by interpretation I.

Definition 9. ***Revised Stable Models and Semantics (Definition 3 in [21]).***
Let $RAA_P(M) \equiv M \setminus \Gamma_P(M)$. M is a Revised Stable Model of a NLP P, iff:

- *M is a minimal classical model, with not interpreted as classical negation, and*
- *$\exists_{\alpha \geq 2}$ such that $\Gamma_P^{\alpha}(M) \supseteq RAA_P(M)$*
- *$RAA_P(M)$ is sustainable*

The first item goes without saying for both semantics. For the third item, it is apparent that sustainable models respect the WFS as required of LSM models; vice-versa, LSM models are sustainable models. Finally, for the 2nd item, consider its justification provided after Def. 3 of [21]. It should be clear, after due consideration, that the 2nd item holds for LSMs and, vice-versa, that RSMs, because they comply with it, will comply with the minimal models at each LSM layer, which always resolve the odd loops that may be present in it.

Only recently, we became aware of the work of Osorio et. al [20]. Their approach, couched in argumentation terms, addresses the same semantic concerns as our own, apparently with similar results for the case of finite propositional normal programs, though the paper does not produce a proof of respecting the WFS. In [22] we also address the same concerns in argumentation semantics terms having providing the Approved Models semantics. The latter can be made to respect the WFS only if desired, by means of an appropriate condition, and so appears to be more general than that of [20]. Moreover, Approved Models is conjectured to result, in that case, in a semantics equivalent to the RSMs [21] and the LSMs.

6 Program Transformation

The program transformation we now define (which can be skipped without loss of continuity) 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 [19] and DLV [7]. The transformation can be query driven and performed on the fly, or previously preprocessed.

6.1 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 [34], 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 the Introduction's example ($a \leftarrow not\ a, X$), 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 "context" 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 10. ***Top-down program transformation.***

```
    input  : A program P
    output: A transformed program P'
1.1 context = ∅
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' UTransform 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*

input : An 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 $((idx + 2 * i) \bmod$ 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

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.*

In the worst case, the whole P is a set of intricate OLONs. In such case there are exponentially many combinations of literals of P forming all the possible contexts, and for each of them TR adds a new rule. Hence, the complexity of TR is conjectured to be exponential in the number of atoms of P. Since the main purpose of this transformation is to be used for top-down query-solving, only the relevant atoms for the query are explored and their respective new rules produced. This way, the transformation can be applied on a "by need" basis.

Example 7. **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 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 8. **Solving OLONs (2).** Consider this program, coinciding with its residual:

$$a \leftarrow not\ b, x \qquad b \leftarrow not\ c, y \qquad c \leftarrow not\ a, z \qquad x \qquad y \qquad z$$

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:

$a \leftarrow not\ b, x \qquad b \leftarrow not\ c, y \qquad c \leftarrow not\ a, z \qquad x \qquad y \qquad z$
$a \leftarrow not\ c, x, y, z \qquad b \leftarrow not\ a, x, y, z \qquad c \leftarrow not\ b, x, y, z$

6.2 Implementing the TR transformation

The XSB Prolog system[1] 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 [4] (standing for XSB Answer Set Programming), is included in XSB Prolog as a practical programming interface to Smodels [19], 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 [32]. 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.

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.

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

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 Abductive Reasoning with Logic Programs

Logic Programs have been used for a few decades now in knowledge representation and reasoning. Amongst the most common kinds of reasoning performed using them, we can find deduction, induction and abduction. When query answering, if we know that the underlying semantics is *relevant*, i.e. guarantees it is enough to use only the rules relevant to the query (those in its call-graph) to assess its truthfulness, then we need not compute a whole model in order to find an answer to our query: it suffices to use just the call-graph relevant part of the program. This way of top-down finding a solution to a query, dubbed "backward chaining", is possible only when the underlying semantics is *relevant* in the above sense, because the existence of a full model is guaranteed.

Currently, the standard 2-valued semantics used by the logic programming community is Stable Models [14]. It's properties are well known and there are efficient implementations (such as *DLV* and *SModels* [7, 19]). However, Stable Models (SMs) miss some important properties, both from the theoretical and practical perspectives: guarantee of model existence for every NLP, relevancy and cumulativity. Since SMs do not enjoy relevancy they cannot use just backward chaining for query answering. This means that it may incur in waste of computational resources, when extra time and memory are used to compute parts of the model which may be irrelevant to the query.

When performing abductive reasoning, we want to find, by need only (via backward chaining), one possible set of conditions (abductive literals of the program to be assumed either *true* or *false*) sufficient to entail our query. However, sometimes we also want to know which are (some of) the consequences (side-effects, so to speak) of such conditions. I.e., we want to know the truth value of some other literals, not part of the query's call graph, whose truth-value may be determined by the abductive conditions found. In some cases, we might be interested in knowing every possible side-effect — the truth-value of every literal in a complete model satisfying the query and ICs. In other situations though, our focus is only in some specific side-effects of abductions performed.

In our approach, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals within reserved construct $inspect/1$. It is advantageous, from a computational point of view, to be able to compute only the truth-value of the important side-effects instead of the whole model, so as not to waste precious time and computational resources. This is possible whenever the underlying semantics guarantees model existence, and enjoys relevance.

7.1 Abduction

Abduction, or inference to the best explanation, is a reasoning method whereby one chooses the hypotheses that would, if true, best explain the observed evidence. In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which have no rules. They can be considered *true* or *false* for the purpose of answering a query. Abduction in LPs ([2, 8, 11, 17, 18]) can naturally be used in a top-down query-oriented proof-procedure to find an (abductive) answer to a query, where abducibles are leafs in the call dependency graph. The Well-Founded Semantics (WFS) [13], which enjoys relevancy, allows for abductive query answering. We used it in the specific implementation described in section 8.2 based on ABDUAL [2]. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

Because they do not depend on any other literal in the program, abducibles can be modeled in a Logic Program system without specific abduction mechanisms by including for each abducible an even loop over default negation, e.g.,

$abducible \leftarrow not\ neg\_abducible. \qquad neg\_abducible \leftarrow not\ abducible.$

where $neg\_abducible$ is a new abducible atom, representing the (abducible) negation of the abducible. This way, under the SM semantics, a program may have models where some $abducible$ is *true* and another where it is *false*, i.e. $neg\_abducible$ is *true*. If there are n abducibles in the program, there will be 2^n models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, e.g. the one available in ABDUAL, both $abducible$ and $neg\_abducible$ remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in some Partial Stable Models.

Using the SM semantics abduction is done by guessing the truth-value of each abducible and providing the whole model and testing it for stability; whereas using the WFS with abduction, it can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles — i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query.

An additional situation, addressed in this paper, is when one wishes to only passively determine which abducibles would be sufficient to satisfy some goal but without actually abducing them, just consuming other goals' needed and produced abductions. The difference is subtle but of importance, and it requires a new construct. Its mechanism, of *inspecting without abducing*, can be conceived and implemented through *meta-abduction*, and is discussed in detail in the sequel.

7.2 Backward and Forward Chaining

Abductive query-answering is intrinsically a backward-chaining process, a top-down dependency-graph oriented proof-procedure. Finding the side-effects of a set of abductive assumptions may be conceptually envisaged as forward-chaining, as it consists of progressively deriving conclusions from the assumptions until the truth value of the chosen side-effect literals is determined.

The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions of a model are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Worse, there may be many alternative

models satisfying an abductive query (and the ICs) whose differences just repose on irrelevant conclusions. So unnecessary computation of irrelevant conclusions can be compounded by the need to discard irrelevant alternative complete models too.

A more intelligent solution would be afforded by selective forward-chaining, where the user would be allowed to specify those conclusions she is focused on, and only those would be computed in forward-chaining fashion. Combining backward-chaining with selective forward-chaining would allow for a greater precision in specifying what we wish to know, and improve efficiency altogether. In the sequel we show how such a selective forward chaining from a set of abductive hypotheses can be replaced by backward chaining from the focused on conclusions — the inspection points — by virtue of a controlled form of abduction which, never performing extra abductions, just checks for abducibles assumed elsewhere.

8 Inspection Points

Meta-abduction is used in *abduction inhibited inspection*. Intuitively, when an abducible is considered under mere inspection, meta-abduction abduces only the intention to *a posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted, but elsewhere. In practice, when we want to meta-abduce some abducible 'x', we abduce a literal '$consume(x)$' (or '$abduced(x)$'), which represents the intention that 'x' is eventually abduced elsewhere in the process of finding an abductive solution. The check is performed after a complete abductive answer to the top query is found. Operationally, 'x' will already have been or will be later abduced as part of the ongoing solution to the top goal.

Example 9. **Relevant and irrelevant side-effects.** Consider this logic program where $drink\_water$ and $drink\_beer$ are abducibles.

$\leftarrow thirsty, not\ drink.$ % This is an Integrity Constraint
$wet\_glass \leftarrow use\_glass.$ $use\_glass \leftarrow drink.$
$drink \leftarrow drink\_water.$ $drink \leftarrow drink\_beer.$
$thirsty.$ $drunk \leftarrow drink\_beer.$
$unsafe\_drive \leftarrow inspect(drunk).$

Suppose we want to satisfy the Integrity Constraint, and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. In this case, full forward-chaining or computation of whole models is a waste of time, because we are interested only in a subset of the program's literals. What we need is a selective ersatz forward chaining mechanism, an inspection tool which permits to check the truth value of given literals as a consequence of the abductions made to satisfy a given query plus any Integrity Constraints.

Moreover, in this example, if we may simply want to know the side-effects of the possible actions in order to decide (to drive or not to drive) **after** we know which side-effects are true. In such case, we do not want to the IC $\leftarrow not\ unsafe\_drive$ because that would always impose $not\ drink\_beer$. We want to allow all possible solutions for the single IC $\leftarrow thirsty, not\ drink$ and then check the side-effects of each abductive solution.

Example 10. **Police and Tear Gas Issue.** Consider this other NLP, where '$tear\_gas$', '$fire$', and '$water\_cannon$' are the only abducibles. Notice that $inspect$ is applied to calls.

$\leftarrow police, riot, not\ contain.$	% this is an Integrity Constraint
$contain \leftarrow tear\_gas.$	$contain \leftarrow water\_cannon.$
$smoke \leftarrow fire.$	$smoke \leftarrow inspect(tear\_gas).$
$police.$	$riot.$

Notice the two rules for '$smoke$'. The first states that one explanation for smoke is fire, when assuming the hypothesis '$fire$'. The second states '$tear\_gas$' is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, '$fire$' is a much more plausible explanation for '$smoke$' and, therefore, in order to let the explanation for '$smoke$' be '$tear\_gas$', there must be a plausible reason — imposed by some other likely phenomenon. This is represented by $inspect(tear\_gas)$ instead of simply '$tear\_gas$'. The '$inspect$' construct disallows regular abduction — only allowing meta-abduction — to be performed whilst trying to solve '$tear\_gas$'. I.e., if we take tear gas as an abductive solution for smoke, this rule imposes that the step where we abduce '$tear\_gas$' is performed elsewhere, not under the derivation tree for '$smoke$'. Thus, '$tear\_gas$' is an *inspection point*. The IC, because there is '$police$' and a '$riot$', forces '$contain$' to be *true*, and hence, '$tear\_gas$' or '$water\_cannon$' or both, must be abduced. '$smoke$' is only explained if, at the end of the day, '$tear\_gas$' is abduced to enact containment. Abductive solutions should be plausible, and '$smoke$' is plausibly explained by '$tear\_gas$' if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Plausibility is an important concept in science, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue. In this example, another way of viewing the need for the new mechanism embodied by the $inspect$ predicate is to consider we have 2 agents: one is a police officer and has the possibility of abducing (corresponding to actually throwing) $tear\_gas$; the other agent is a civilian who, obviously, does not have the possibility of abducing (throwing) $tear\_gas$. For the police officer agent, having the $smoke \leftarrow inspect(tear\_gas)$ rule, with the $inspect$ is unnecessary: the agent knows that $tear\_gas$ is the explanation for smoke because it was himself who abduced (threw) $tear\_gas$; but for the civilian agent the $inspect$ in the $smoke \leftarrow inspect(tear\_gas)$ rule is absolutely indispensable, since he cannot abduce $tear\_gas$ and therefore cannot know, without $inspect$ing, if that is the real explanation for $smoke$.

Example 11. **Nuclear Power Plant Decision Problem.** This example was extracted from [31] and adapted to our current designs, and its abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff will dust the power plant on cleaning days, but only if there is no alarm sounding. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant imme-

diately! Abducible literals are $cleaning\_day$, $temp\_rise$ and $faulty\_alarm$.

$$\begin{array}{ll} dust & \leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\ sound\_alarm & \leftarrow temp\_rise \\ sound\_alarm & \leftarrow faulty\_alarm \\ evacuate & \leftarrow sound\_alarm \\ & \leftarrow not\ cleaning\_day \end{array}$$

Satisfying the unique IC imposes $cleaning\_day$ *true* and gives us three minimal abductive solutions: $S_1 = \{dust, cleaning\_day\}$,
$S_2 = \{cleaning\_day, sound\_alarm, temp\_rise, evacuate\}$, and
$S_3 = \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\}$. If we pose the query $?-not\ dust$ we want to know what could justify the cleaners dusting not to occur given that it is a cleaning day (enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove $not\ dust$. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressible by adding the IC $\leftarrow not\ sound\_alarm$, which would then abduce one or both of those two abducibles as plausible explanations. The $inspect/1$ in the body of the rule for $dust$ prevents any abduction below $sound\_alarm$ to be made just to make $not\ dust$ true. One other possibility would be for two observations, coded by ICs $\leftarrow not\ temp\_rise$ or $\leftarrow not\ faulty\_alarm$, to be present in order for $not\ dust$ to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries: $?-evacuate$ and $?-inspect(evacuate)$, respectively.

8.1 Declarative Semantics of Inspection Points

A simple transformation maps programs with inspection points into programs without them. Mark that the Stable Models of the transformed program where each $abducible(X)$ is matched by the abducible X (X being a literal a or its default negation $not\ a$) clearly correspond to the intended procedural meanings ascribed to the inspection points of the original program.

Definition 11. ***Transforming Inspection Points.*** *Let P be a program containing rules whose body possibly contains inspection points. The program $\Pi(P)$ consists of:*

1. *all the rules obtained by the rules in P by systematically replacing:*
   - *$inspect(not\ L)$ with $not\ inspect(L)$;*
   - *$inspect(a)$ or $inspect(abduced(a))$ with $abduced(a)$ if a is an abducible, and keeping $inspect(L)$ otherwise.*
2. *for every rule $A \leftarrow L_1, \ldots, L_t$ in P, the additional rule:*
   $inspect(A) \leftarrow L_1^{'}, \ldots, L_t^{'}$ *where for every* $1 \leq i \leq t$*:*

$$L_i^{'} = \begin{cases} abduced(L_i) & \text{if } L_i \text{ is an abducible} \\ inspect(X) & \text{if } L_i \text{ is } inspect(X) \\ inspect(L_i) & \text{otherwise} \end{cases}$$

The semantics of the $inspect$ predicate is given by the generated rules for $inspect$

Example 12. **Transforming a Program P with Nested Inspection Levels.**

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d & y \leftarrow inspect(not\ a) \\ z \leftarrow d & y \leftarrow b, inspect(not\ z), c \end{array}$$

Then, $\Pi(P)$ is:

$$\begin{array}{ll} x & \leftarrow a, inspect(y), b, c, not\ d \\ inspect(x) & \leftarrow abduced(a), inspect(y), abduced(b), abduced(c), not\ abduced(d) \\ y & \leftarrow not\ inspect(a) \\ y & \leftarrow b, not\ inspect(z), c \\ inspect(y) & \leftarrow not\ abduced(a) \\ inspect(y) & \leftarrow abduced(b), not\ inspect(z), abduced(c) \\ z & \leftarrow d \\ inspect(z) & \leftarrow abduced(d) \end{array}$$

The abductive stable model of $\Pi(P)$ respecting the inspection points is:
$\{x, a, b, c, abduced(a), abduced(b), abduced(c), inspect(y)\}$.

Note that for each $abduced(a)$ the corresponding a is in the model.

8.2 Inspection Points Implementation

Once again, we based our inspection point practical implementation on a formally defined, XSB-implemented, true and tried abduction system — ABDUAL [2]. ABDUAL lays the foundations for efficiently computing queries over ground three-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models.

The query processing technique in ABDUAL relies on a mixture of program transformation and tabled evaluation. A transformation removes default negative literals (by making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal O and its set of rules R, a dual set of rules whose conclusions $not\ (O)$ are true if and only if O is false in R. Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [33] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

ABDUAL is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible to an ongoing list of current abductions, unless the negation of the abducible was added before to the lists failing in order to ensure abduction consistency. Meta-abduction is implemented adroitly by

means of a reserved predicate, '$inspect/1$' taking some literal L as argument, which engages the abduction mechanism to try and discharge any meta-abductions performed under L by matching with the corresponding abducibles, adopted elsewhere outside any '$inspect/1$' call. The approach taken can easily be at least partially adopted by other abductive systems, as we had the occasion to check with the authors [6]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation above, which is reported below.

Procedurally, in the ABDUAL implementation, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already adopted, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found. At inspection-point-top-down-proof-mode, whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query (unless of course the negation of the abducible has already been adopted by then, allowing for immediate failure at that search node.) That is, one (meta -) abduces the checking of some abducible A, and the check consists in confirming that A is part of the abductive solution by matching it with the object of the check. According to our method, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals subject to inspection mode, with the reserved construct '$inspect/1$'.

9 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.

In the context of abductive logic programs, we presented a new mechanism of inspecting literals that can check for side-effects of top-down driven 2-valued abductive solutions, enabled by a form of meta-abduction, and have provided for it a declarative semantics that relies on the relevance property of Layer Supported Model semantics. We implemented this inspection mechanism within the Abdual [2] meta-interpreter, as well as within the XSB-XASP system. Its solution relies on abduction itself, making it thus generally adoptable by other abductive frameworks.

We have also 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.

9.1 Coda

Prolog has been till recently the most accepted means to codify and execute logic programs, and a useful tool for research and application development in logic program-

ming. Several stable implementations were developed and refined over the years, with plenty of working solutions to pragmatic issues, ranging from efficiency and portability to explorations of language extensions. XSB-Prolog is one of the most sophisticated, powerful, efficient and versatile, focusing on execution efficiency and interaction with external systems, implementing logic program evaluation following the Well-Founded Semantics (WFS).

The XASP interface [4, 5] (XSB Answer Set Programming), part of XSB-Prolog, is a practical programming interface to Smodels[19]. XASP allows to compute the models of a Normal LP, and also to effectively combine 3- with 2-valued reasoning, to get the best of both worlds. This is achieved by using Smodels to compute the stable models of the residual program, one that results from a query evaluation in XSB using tabling[32]. The residual program is formed by delay lists, sets of undefined literals for which XSB could not find a complete proof, due to mutual loops over default negation in a set, as detected by the tabling mechanism. This method allows obtaining 2-valued models, by completion of the 3-valued ones of XSB. The integration maintains the relevance property for queries over our programs, something Stable Model semantics does not enjoy. In Stable Models, by its very definition, it is necessary to compute whole models of a given program. With the XSB implementation framework one may sidestep this issue, using XASP to compute the query relevant residual program on demand. Only the resulting residual program is sent to Smodels for computation of its stable models, and the result returned to XSB.

Having defined a more general 2-valued semantics for NLPs much remains to be explored, in the way of properties, complexity, comparisons, implementations, extensions and applications, 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 7 and 8. 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.

That the LSM semantics includes the SM semantics and that it always exists and admits top-down querying is a novelty making us look anew at 2-valued semantics use in KRR. LSMs' implementation, because of its relevance property, can avoid the need to compute whole models and all models, and hence SM's apodictic need for complete groundness and the difficulties it begets for problem representation. 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 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.

One topic of future work consists in defining partial model schemas, that can provide answers to queries in terms of abstract non-ground model schemas encompassing several instances of ground partial models. Abstract partial models, instead of ground ones, may be produced directly by the residual, a subject for further investigation.

Yet another topic of future exploration is the definition of a Well-Founded Layer Supported Model (WFLSM). Conceivably, the WFLSM would be defined as a partial model that, at each layer, is the intersection of the all LSMs. Floating conclusions are consequently disallowed. Incidental to this topic is the relationship of the WFLSM to O-semantics [26], it being readily apparent the former extends the latter.

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

10 Acknowledgements

We thank Marco Alberti, José Júlio Alferes, Pierangelo Dell'Acqua, Robert Kowalski, Juan Carlos Nieves, Mauricio Osorio, and David Warren for their comments.

References

1. J. J. Alferes, P. M. Dung, and L. M. Pereira. Scenario semantics of extended logic programs. In *LPNMR*, pages 334–348. MIT Press, 1993.
2. J.J. Alferes, L.M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *TPLP*, 4(4):383–428, July 2004.
3. K.R. Apt and H.A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundam. Inform.*, 14(3):339–343, 1991.
4. L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*. http://xsb.sourceforge.net/packages/xasp.pdf.
5. L.F. Castro and D.S. Warren. An environment for the exploration of non monotonic logic programs. In A. Kusalik, editor, *Procs. WLPE'01*, 2001.
6. H. Christiansen and V. Dahl. Hyprolog: A new logic programming language with assumptions and abduction. In *ICLP*, pages 159–173, 2005.
7. 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.
8. M. Denecker and D. De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In Apt, editor, *Procs. ICLP'92*, pages 686–700, Washington, USA, 1992. The MIT Press.
9. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I, II. *Fundamenta Informaticae*, XXII(3):227–255, 257–288, 1995.
10. J. Dix, G. Gottlob, and W. Marek. Reducing disjunctive to non-disjunctive semantics by shifting operations. *Fundamenta Informaticae*, 28:87–100, 1996.
11. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177, 1997.
12. F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.
13. 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.

14. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
15. M. Gelfond, H. Przymusinska, V. Lifschitz, and M. Truszczynski. Disjunctive defaults. In *KR-91*, pages 230–237, 1991.
16. P. Hitzler and M. Wendt. A uniform approach to logic programming semantics. *TPLP*, 5(1-2):93–121, 2005.
17. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.
18. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in AI and LP*, volume 5, pages 235–324. Oxford University Press, 1998.
19. 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, vol. 1265, pp. 420–429, 1997.
20. J. C. Nieves, M. Osorio, and C. Zepeda. Expressing extension-based semantics based on stratified minimal models. In *Logic, Language, Information and Computation*, volume 5514 of *LNCS*, pages 305–319. Springer, 2009.
21. L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In G. Dias et al., editor, *Progress in AI*, volume 3808 of *LNCS*, pages 29–42. Springer, 2005.
22. L. M. Pereira and A. M. Pinto. Approved models for normal logic programs. In N. Dershowitz and A. Voronkov, editors, *Procs. LPAR'07*, LPAR - LNAI, Yerevan, Armenia, October 2007. Springer.
23. L. M. Pereira and A. M. Pinto. Inspection points and meta-abduction in logic programs. In *INAP'09*, LNCS. Springer, 2009.
24. L. M. Pereira and A. M. Pinto. Layer supported models of logic programs. In *Procs. 10th LPNMR*, LNCS, pages 450–456. Springer, September 2009. Long version at http://centria.di.fct.unl.pt/∼lmp/publications/online-papers/LSMs.pdf.
25. L. M. Pereira and A. M. Pinto. Stable model implementation of layer supported models by program transformation. In *INAP'09*, LNCS. Springer, 2009.
26. L.M. Pereira, J.J. Alferes, and J.N. Aparício. Adding closed world assumptions to well-founded semantics. *TCS*, 122(1-2):49–68, 1994.
27. 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.
28. T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21. ACM Press, 1989.
29. T. C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.
30. T.C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
31. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In *AI*IA*, pages 49–60, 1999.
32. T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.
33. T. Swift and D. S. Warren. An abstract machine for slg resolution: Definite programs. In *Symp. on Logic Programming*, pages 633–652, 1994.
34. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.