A Logic Programming System for Non-monotonic Reasoning

José Júlio Alferes* Carlos Viegas Damásio† Luís Moniz Pereira†

Abstract

The evolution of Logic Programming semantics has included the introduction of a new explicit form of negation, beside the older implicit (or default) negation typical of Logic Programming. The richer language has been shown adequate for a spate of knowledge representation and reasoning forms.

The widespread use of such extended programs requires the definition of a correct top-down querying mechanism, much as for Prolog wrt. normal programs. One purpose of this paper is to present and exploit a SLDNF-like derivation procedure, SLX, for programs with explicit negation under well founded semantics (*WFSX*) and prove its soundness and completeness. (Its soundness wrt. the answer-sets semantics is also shown.) Our choice of *WFSX* as the base semantics is justified by the structural properties it enjoys, which are paramount for top-down query evaluation.

Of course, introducing explicit negation requires dealing with contradiction. Consequently, we allow for contradiction to appear, and show moreover how it can be removed by freely changing the truth-values of some subset of a set of predefined revisable literals. To achieve this, we have in this paper introduced a paraconsistent version of *WFSX*, $WFSX_p$, that allows contradictions, and for which our SLX top-down procedure is proven correct as well.

This procedure can be used to detect the existence of pairs of complementary literals in $WFSX_p$ simply by detecting the violation of integrity rules $\mathbf{f} \Leftarrow L, \neg L$ introduced for each L in the language of the program. Furthermore, integrity constraints of a more general form are allowed, whose violation can likewise be detected by SLX.

Removal of contradiction or integrity violation is accomplished by a variant of the SLX procedure which collects, in a formula, the alternative combinations of revisable literals' truth-values that ensure the said removal. The formulas, after simplification, can then be satisfied by a number of truth-values changes in the revisables, among "true", "false", and "undefined". A notion of minimal change is defined as well that establishes a closeness relation between a program and its revisions. Forthwith, the changes can be enforced by introducing or deleting program rules for the revisable literals.

To illustrate the usefulness and originality of our framework we applied it to obtain a novel logic programming approach, and results, in declarative debugging and model based diagnosis problems.

Keywords: Logic programming procedures; Logic programming semantics; Non-monotonic reasoning; Belief revision; Well founded semantics.

1 Introduction

For some time now, programming in logic has been shown to be a viable proposition. Since the mid-fifties, the desire to impart the computer with the ability to reason logically lead to the development of automated theorem proving, which took up the promise of giving logic to artificial intelligence. As a result of the effort to find simple and efficient theorem proving strategies, Horn clause programming under SLD resolution was discovered and implemented [16, 37].

*DMAT, Univ. Évora, 7000 Évora, Portugal and CRIA, Uninova, jja@fct.unl.pt

†CRIA, Uninova and DCS, U. Nova de Lisboa, 2825 Monte da Caparica, Portugal, {cd|lmp}@fct.unl.pt.

We all thank Esprit BR project Compulog 2 (no. 6810), and JNICT - Portugal for their support.

However, because Horn clauses admit only positive conclusions or facts, they give rise to a monotonic semantics, i.e. one by which previous conclusions are never questioned in spite of additional information, and thus the number of derived conclusions cannot decrease – hence the monotonicity. Also, nothing can be concluded false, except by assuming that which is not finitely proven true is false. But this condition prevents, by definition, the appearance of any and all contradictions.

Thus, although Horn clause programming augmented with the NOT operator (cf. Prolog) under the SLDNF derivation procedure [41] does allow negative conclusions, these are only drawn by default (or implicitly), just in case the corresponding positive conclusion is not forthcoming in a finite number of steps – hence the specific form of Closed World Assumption (CWA) [58] of the completion semantics given to such programs [15].

This form of negation is capable of dealing with incomplete information, by assuming false exactly what is not true in a finite manner. However, there remains the issue of non-terminating computations, even for finite programs. To deal with this and other problems of the completion semantics, a spate of semantics proposals were set forth, from the late eighties onwards, including the well-founded semantics (WFS) of [24], which deals semantically with non-terminating top-down computations, by assigning such computations the truth value of "false" or "undefined", and thereby giving semantics to every program. For this semantics several query evaluation procedures have been defined [7, 11, 13, 32, 47, 53, 54, 57, 61]. The well-founded semantics deals only with normal programs, i.e. those with just negation by default, and thus it provides no mechanism for explicitly declaring the falsity of literals. This can be a serious limitation.

The evolution of Logic Programming (LP) semantics has now included the introduction of a new explicit form of negation, beside the older implicit (or default) negation typical of Logic Programming. The richer language has been shown adequate for a diversity of knowledge representation and reasoning forms [9, 28, 48].

In fact, in recent years several authors (e.g. [27, 48, 66]) have stressed the importance of extending LP with a second kind of negation $\neg$, for use in deductive databases, knowledge representation, and non-monotonic reasoning (NMR). Different semantics for extended LPs with $\neg$-negation (ELP) have appeared (e.g. [3, 27, 45, 55, 65, 66]). The particular generalization for extended programs of WFS defined in [45], *WFSX*, which is taken as the base semantics in this article, provides an added qualitative representational expressivity that can capture a wide variety of logical reasoning forms, and serve as an instrument for programming them.

The two forms of negation, default and explicit, are not unrelated: our "coherence principle" stipulates that the latter entails the former. Default negation and the revision of believed assumptions in the face of contradiction, or integrity constraint violation, are the two non-monotonic reasoning mechanisms available in logic programming. Their use in combination with explicit negation adds on a qualitative representational expressivity, as we've shown in [48].

For the widespread of such uses of extended logic programs the definition of a correct top-down querying mechanism is required, much as for Prolog wrt. normal programs. Indeed, users normally wish to know the instances of a literal that belong to the semantics rather than the whole semantics. One purpose of this paper is to present and exploit a SLDNF-like derivation procedure, SLX, for programs with explicit negation under well founded semantics (*WFSX*), and prove its soundness and completeness. (Its soundness wrt. the answer-sets semantics is also shown.) Additionally, we have produced a Prolog interpreter for the procedure, and a pre-processor into Prolog based on it, which show the procedure's amenability to implementation. Because of lack of space these implementations are not described in the paper but are available on request.

The SLX derivation method is directly inspired on the semantic AND-tree characterization of *WFSX* in our paper [2], but does not presuppose it. In fact, that work can be seen as a preliminary step towards the definition of the procedure presented here, in which there was no concern with the actual construction of the trees (they were assumed to exist). The attribution of failure and success was there made "a posteriori", and assuming all the required trees simultaneously available. Here, not only do we define how the derivations are constructed in a top-down way, using an arbitrary selection rule, but also do we attribute success and failure of literals incrementally as the derivation develops. The characterization in [2] is a declarative semantics, and a first step towards the present one, which is procedural.

Our choice of *WFSX* as the base semantics is justified by the structural properties it possess, which are paramount for top-down query evaluation. Indeed, because of its properties, which other approaches do not fully enjoy, *WFSX* is a natural candidate to being the semantics of choice for logic programs extended with explicit negation. Namely, it exhibits the structural properties: well-foundedness, cumulativity, rationality, relevance, and partial evaluation. By well-foundedness we mean that it can be simply characterized (without any recourse to three-valued logic) by two iterative fixpoint operators. By cumulativity [20, 38], we refer to the efficiency related ability of using lemmas, i.e. the addition of lemmas does not change the semantics of a program. By rationality [20, 38], we refer to the ability to add the negation of a non-provable conclusion without changing the semantics. By relevance [21], we mean that the top-down evaluation of a literal's truth-value requires only the call-graph below it. By partial evaluation [21] we mean that the semantics of a partially evaluated program keeps to that of the original[1]. Additionally, it is amenable to both top-down and bottom-up procedures, and its complexity for finite DATALOG programs is polynomial[2].

Furthermore, *WFSX* is sound wrt. to the answer-sets semantics of [27], and it is a better aproximation to answer-sets than simply using WFS plus the renaming of $\neg$-literals (cf. proposition 3.1). Thus, our top-down method can be used as a sound one for answer-sets, that provides less incompleteness than others. These and other properties of *WFSX* are detailed and proven in [6]. Since *WFSX* coincides with WFS on normal programs, our method is applicable to it and, for ground programs, compares favourably with previous approaches [2].

To the best of our knowledge paper [62] is the only in the literature that addresses the topic of proof procedures for extended logic programs. The author uses the notion of conservative derivability [66] as the proof-theoretic semantics for extended programs. The paper provides a program transformation from such programs to normal ones. Then it is proved that Kunen semantics [39] applied to the transformed program is sound and complete wrt. conservative derivability. This approach has several problems mainly motivated by the interpretation of default negation as finite failure as recognized by the author. For instance, in the program $\{a \leftarrow a\}$ the literal $\neg a$ is false but a is undefined, contrary to the results obtained by answer sets and WFSX where *not* a is true. As a final remark, conservative derivability is not defined for programs with functor symbols. Therefore the approach is only applicable to extended programs without functor symbols. *WFSX* solves all these problems properly. This author, moreover, deals with contradiction by inhibiting the propagation of its consequences.

Of course, introducing explicit negation requires dealing in addition with veritable contradiction. Indeed, information is not only normally incomplete but contradictory as well. One consequence, not all (negation by) default assumptions can be made, but only those not partaking of contradiction. This amounts to the ancient and venerable logical principle of "reductio ad absurdum": *if an assumption leads to contradiction withdraw it.* One major contribution of our work is that of tackling this issue with some generality within our semantics of extended logic programs.

Like [62], we too allow for contradiction to appear, and show moreover how it can be removed by freely changing the truth-values of some subset of a set of pre-designated revisable literals. To achieve this, we start by introducing a paraconsistent version of *WFSX*, $WFSX_p$, that permits contradictions, and for which our SLX top-down procedure is proven correct as well.

This procedure can be used to detect the existence of pairs of complementary literals in $WFSX_p$ simply by detecting the violation of integrity rules $\mathbf{f} \Leftarrow L, \neg L$ introduced for each L in the language of the program. Furthermore, integrity constraints of a more general form are allowed (cf. section

[1] Stable model based approaches [25], such as answer-sets [27], enjoy neither cumulativity, nor rationality, nor relevance.

[2] Not so for stable model based approaches: there are no iterative top-down or bottom-up operators, and the complexity for computing the stable models of a program is co-NP-complete, even for DATALOG.

7.1), whose violation can likewise be detected by SLX.

Removal of contradiction or integrity violation is accomplished by a variant of the SLX procedure which collects, in a formula, the alternative combinations of revisable literal's truth-values that ensure the said removal. The formulas, after simplification, can then be satisfied by a number of truth-values changes in the revisables, among "true", "false", and "undefined". A notion of minimal change (improving on previous work) is defined as well that establishes a closeness relation between a program and its revisions. Forthwith, the changes to achieve a revision can be enforced by introducing or deleting program rules only for the revisable literals.

We do not address here the minimization of the formulas obtained: it is susceptible of simplification methods, generalized for the three-valued setting, once the formula is normalized. Incremental simplification methods are also possible.

Our procedures have been implemented and used to run a great variety of non-monotonic reasoning problems, solved using extended logic programming, many of which published in our aforementioned references. We've also introduced extensions to the logic programming language that allow for the representation of preference relations on the revisions [18]. This is essential for efficient and practical revision methods.

1.1 Structure of the paper

The remainder of this paper is structured in sections, as follows. Firstly, we argue the need for extended logic programs with two types of negation, default and explicit. Secondly, we produce an overview of the *WFSX* base semantics. Thirdly, we motivate and recap a sound and (theoretically) complete top-down procedure, SLX, for deciding whether a literal belongs to the well-founded model of *WFSX*.

To address the problem of revising contradiction we then introduce a paraconsistent version of *WFSX*. Then we show that the SLX procedure is also sound and complete with respect to it, so as to be able to detect for which pairs of complementary literals contradiction obtains. Subsequently, we elaborate on the SLX procedure, by employing a combination of sound and complete pruning rules wrt. to (at least) finite ground programs.

The following section addresses the issue of revising programs contradictory wrt. a set of integrity constraints. First, the language is extended to allow for a general form of integrity constraints. Next are introduced the notions of program state and revision, and of the closeness of revisions to the original program.

Afterwards, the reader can find two extended application examples, in the fields of declarative debugging and diagnosis, which show the utility of the general revision framework presented here. The novelty of the present approach lies in its formalization, and the use of the closeness relation with more general integrity constraints. For other applications to non-monotonic taxonomies, hypothetical and abductive reasoning, reasoning about actions, and more model based diagnosis and declarative debugging see [48, 50, 49, 51].

The next to last subsection overviews the issues involved in our implemented program revision and minimization of change procedures but for lack of space does not detail them.

In the last subsection we provide more detailed theoretical insights, justifying the choice of our particular closeness relation.

Finally, there follow the conclusions, a discussion, and mention of future work.

Appendix A contains the proof of correctness for SLX wrt. the paraconsistent *WFSX*. Appendix B contains proofs on distance and closeness between program states.

2 Extended logic programs

In this section we begin by reviewing the main motivations for introducing a second kind of negation in logic programs.

In normal logic programs the negative information is implicit, i.e. it is not possible to explicitly state falsity, and propositions are assumed false if there is no reason to believe they are true. This is what is wanted in some cases. For instance, in the classical example of a database that explicitly states flight connections, one wants to implicitly assume that the absence of a connection in the database means that no such connection exists.

However this is a serious limitation in other cases. As argued in [44, 64], explicit negative information plays an important rôle in natural discourse and common-sense reasoning. The representation of some problems in logic programming would be more natural if logic programs had some way of explicitly representing falsity. Consider for example the statement: *"Penguins do not fly"* One way of representing this statement within logic programming could be:

$$no\_fly(X) \leftarrow penguin(X)^3$$

But this representation does not capture the connection between the predicate $no\_fly(X)$ and the predication of flying. This becomes clearer if, additionally, we want to represent the statement: *"Birds fly"*. Clearly this statement can be represented by $fly(X) \leftarrow bird(X)$. But then no connection whatsoever exists between the predicates $no\_fly(X)$ and $fly(X)$. Intuitively one would like to have such an obvious connection established.

The importance of these connections grows if we think of negative information for representing exceptions to rules [35]. The first statement above can be seen as an exception to the general rule that normally birds fly. In this case we really want to establish the connection between flying and not flying.

Exceptions expressed by sentences with negative conclusions are also common in legislation [34, 36]. For example, consider the provisions for depriving British citizens of their citizenship:

> 40 - (1) Subject to the provisions of this section, the Secretary of State may by order deprive any British citizen to whom this subsection applies of his British citizenship if [...]
>
> (5) The Secretary of State shall not deprive a person of British citizenship under this section if [...]

Clearly, 40.1 has the logical form "P if Q" whereas 40.5 has the form "¬ P if R". Moreover, it is also clear that 40.5 is an exception to the rule of 40.1.

Above we argued for the need of having explicit negation in the heads of rules. But there are also reasons that compel us to believe explicit negation is needed in their bodies too. Consider the statement[4]:

"A school bus may cross railway tracks under the condition that there is no approaching train"

It would be wrong to express this statement by the rule $cross \leftarrow not\ train$. The problem is that this rule allows the bus to cross the tracks when there is no information about either the presence or the absence of a train. The situation is different if explicit negation is used: $cross \leftarrow \neg train$. Then the bus is only allowed to cross the tracks if the bus driver is sure that there is no approaching train. The difference between *not* p and $\neg p$ in a logic program is essential whenever we cannot assume that available positive information about p is complete, i.e. we cannot assume that the absence of information about p clearly denotes its falsity.

Moreover, the introduction of explicit negation in combination with the existing default negation allows for greater expressivity, say for representing statements like:

"If the driver is not sure that a train is not approaching then he should wait"

in a natural way: $wait \leftarrow not\ \neg train$.

Examples of such combinations also appear in legislation. For example consider the following article from "The British Nationality Act 1981" [29]:

[3] Or equivalently, as suggested in [26], $fly'(X) \leftarrow penguin(X)$.

[4] This example is due to John McCarthy, and was published for the first time in [27].

(2) A new–born infant who, after commencement, is found abandoned in the United Kingdom shall acquire British citizenship by section 1.2 if it is not shown that it is not the case that the person is born [...]

Clearly, conditions of the form "it is not shown that it is not the case that P" can be expressed naturally by $not\ \neg P$.

Another motivation for introducing explicit negation in logic programs relates to the symmetry between positive and negative information. This is of special importance when the negative information is easier to represent than the positive one. One can first represent it negatively, and then say that the positive information corresponds to its complement.

In order to make this clearer, take the following example [27]:

Example 2.1 Consider a graph description based on the predicate $arc(X, Y)$, which expresses that in the graph there is an arc from vertex X to vertex Y. Now suppose that we want to determine which vertices are terminals. Clearly, this is a case where the complement information is easier to represent, i.e. it is much easier to determine which vertices are not terminal. By using explicit negation in combination with negation by default, one can then easily say that terminal vertices are those which are not nonterminal:

$$\begin{array}{rcl} \neg terminal(X) & \leftarrow & arc(X,Y) \\ terminal(X) & \leftarrow & not\ \neg terminal(X) \end{array}$$

Finally, another important motivation for extending logic programming with explicit negation is to generalize the relationships between logic programs and non-monotonic reasoning formalisms. Such relationships, drawn for the most recent semantics of normal logic programs, have proven of extreme importance for both sides, giving them mutual benefits and clarifications.

Indeed, extended logic programming has been shown adequate for namely these forms of reasoning: incomplete information, contradiction handling, belief revision, default, abductive, counter-factual, and hypothetical. And it has been shown adequate for namely these knowledge representation forms: rules, default rules, constraints (denials), exceptions to defaults, preferences among defaults, hypothetical possibilities, and falsity, whether via explicit or default negation (see references in [9, 48]).

3 *WFSX* overview

In this section, for the sake of the paper's self-sufficiency, we recall the language of logic programs extended with explicit negation, or extended logic programs for short, and briefly review the *WFSX* semantics [45]. An extended program is a set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\ (0 \leq m \leq n)$$

where each L_i is an objective literal. An objective literal is either an atom A or its explicit negation $\neg A$. In the sequel, we also use $\neg$ to denote complementary literal wrt. the explicit negation, so that $\neg\neg A = A$. The set of all objective literals of a program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. The symbol *not* stands for negation by default[5]. $not\ L$ is called a default literal. Literals are either objective or default literals. By $not\ \{a_1, \ldots, a_n, \ldots\}$ we mean $\{not\ a_1, \ldots, not\ a_n, \ldots\}$. An interpretation of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $\mathcal{H}(P)$. Objective literals in T are said to be *true* in I, objective literals in F *false by default* in I, and in $\mathcal{H}(P) - I$ *undefined* in I.

WFSX follows from WFS for normal programs plus the coherence requirement relating the two forms of negation:
"For any objective literal L, if $\neg L$ is entailed by the semantics then $not\ L$ is also entailed".

[5] This designation has been used in the literature instead of the more operational *"negation as failure (to prove)"*.

Note that this requirement is obeyed by answer-sets, but not so by WFS when explicitly negated literals are simply replaced by new atoms (as in [55]). The requirement states that whenever some literal is explicitly declared as false then, for coherence, it must be assumed false by default too. In an epistemic view of logic program, coherence can be seen as an instance of the necessitation principle. That principle states that if something is known then it is believed [4]. Consequently, if something is known to be false (i.e. is explicitly false), then it is believed false (i.e. is false by default).

Example 3.1 Consider the program:

$$\begin{array}{rcl} married(mary,tom) & \leftarrow & not\ married(mary,peter) \\ married(mary,peter) & \leftarrow & not\ married(mary,tom) \\ \neg married(mary,tom) & & \end{array}$$

Because of coherence, with *WFSX* this program entails $not\ married(mary,tom)$ and, consequently, $married(mary,peter)$. This is the result of answer-sets as well.

By simply using WFS, with the renaming of explicitly negated literals, the program does not entail $married(mary,peter)$, which seems a counterintuive result.

Because more adequate for our purposes, here we present *WFSX* in a distinctly different manner with respect to its original definition. This presentation is based on alternating fix-points of Gelfond–Lifschitz Γ–like operators [25, 27]. The proof of equivalence between both definitions, as well as proofs of other results in this section, can be found in [6]. We begin by recalling the definition of Γ:

Definition 3.1 (The Γ–operator) *Let P be an extended program, I an interpretation, and let P' (resp. I') be obtained from P (resp. I) by denoting every literal $\neg A$ by a new atom, say $\neg\_A$. The GL–transformation $\frac{P'}{I'}$ is the program obtained from P' by removing all rules containing a default literal not A such that $A \in I'$, and by then removing all the remaining default literals from P. Let J be the least model of $\frac{P'}{I'}$. ΓI is obtained from J by replacing the introduced atoms $\neg\_A$ by $\neg A$.*

To impose the coherence requirement we introduce:

Definition 3.2 (Semi-normal version of a program) *The semi-normal version of a program P is the program P_s obtained from P by adding to the (possibly empty) Body of each rule $L \leftarrow Body$ the default literal not $\neg L$, where $\neg L$ is the complement of L wrt. explicit negation.*

Below we use $\Gamma(S)$ to denote $\Gamma_P(S)$, and $\Gamma_s(S)$ to denote $\Gamma_{P_s}(S)$.

Definition 3.3 (Partial stable model) *A set of objective literals T generates a partial stable model (PSM) of an extended program P iff: (1) $T = \Gamma\Gamma_s T$; and (2) $T \subseteq \Gamma_s T$. The partial stable model generated by T is the interpretation $T \cup not\ (\mathcal{H}(P) - \Gamma_s T)$.*

In other words, partial stable models are determined by the fix-points of $\Gamma\Gamma_s$. Given a fix-point T, objective literals in T are *true* in the PSM, objective literals not in $\Gamma_s T$ are *false by default*, and all others are *undefined*. Thus, objective literals in $\Gamma_s T$ are all the true or undefined ones. Note that condition (2) imposes that a literal cannot be both true and false by default (viz. if it belongs to T it does not belong to $\mathcal{H} - \Gamma_s T$, and vice-versa). Moreover note how the use of Γ_s imposes coherence: if $\neg L$ is true, i.e. it belongs to T, then in $\Gamma_s T$, via semi-normality, all rules for L are removed and, consequently, $L \notin \Gamma_s T$, i.e. L is false by default.

Example 3.2 Program $P = \{a; \quad \neg a\}$ has no partial stable models. Indeed, the only fix-point of $\Gamma\Gamma_s$ is $\{a, \neg a\}$, and $\{a, \neg a\} \not\subseteq \Gamma_s\{a, \neg a\} = \{\}$. Thus it is not a PSM.

Programs without partial stable models are said *contradictory*. In section 5 we elaborate further on the results obtained by using $\Gamma\Gamma_s$ on contradictory programs. Now we define the semantics for non-contradictory programs.

Theorem 3.1 (*WFSX* semantics) *Every non-contradictory program P has a least (wrt. $\subseteq$) partial stable model, the well-founded model of P ($WFM(P)$).*

To obtain an iterative "bottom-up" definition for $WFM(P)$ we define the following transfinite sequence $\{I_\alpha\}$:

$$\begin{array}{rcll} I_0 & = & \{\} & \\ I_{\alpha+1} & = & \Gamma\Gamma_s I_\alpha & \\ I_\delta & = & \bigcup\{I_\alpha \mid \alpha < \delta\} & \textit{for limit ordinal } \delta \end{array}$$

There exists a smallest ordinal λ for the sequence above, such that I_λ is the smallest fix-point of $\Gamma\Gamma_s$, and $WFM(P) = I_\lambda \cup not\ (\mathcal{H}(P) - \Gamma_s I_\lambda)$.

In this constructive definition literals obtained after an application of $\Gamma\Gamma_s$ (i.e. in some I_α) are true in $WFM(P)$, and literals not obtained after an application of Γ_s (i.e. not in $\Gamma_s I_\alpha$, for some α) are false by default in $WFM(P)$.

Note that, like the alternating fix-point definition of WFS [63], this definition of *WFSX* also relies on the application of two anti-monotonic operators. However, unlike the definition of WFS, these operators are distinct. As we'll see, this points to the definition of two kinds of derivation procedures, one reflecting applications of Γ and proving verity of objective literals, and the other reflecting applications of Γ_s and proving non-falsity of objective literals. The two become meshed when the proof of verity of *not* L is translated into the failure to prove the non-falsity of L.

Theorem 3.2 (Relation to WFS) *For normal logic programs (i.e. without explicit negation)* WFSX *coincides with the well–founded semantics of [24].*

Theorem 3.3 (Relation to Answer-sets) *Let P be an extended logic program with at least one answer-set. Every answer-set of P is also a PSM of P. Moreover, for any objective literal L:*

- *If $L \in WFM(P)$ then L belongs to all answer-sets of P.*
- *If not $L \in WFM(P)$ then L does not belong to any answer-set of P.*

For lack of space, here we do not elaborate more on the relation to answer-sets, and on how the use of semi-normality imposes coherence. However, it is interesting to note that also any other combination of the Γ–like operators used to define *WFSX* (i.e. the operators: $\Gamma_s\Gamma$, $\Gamma\Gamma$[6], and $\Gamma_s\Gamma_s$) gives semantics that are sound wrt. answer-sets, but which are not as close to the latter as *WFSX*. By "not as close" we mean that its least fix-points are subsets of the intersection of all answer-sets, smaller (wrt. set inclusion) than *WFSX* itself. Thus we say that the *WFSX* combination is the best approximation to answer-sets semantics, compared to the others.

Proposition 3.1 *Let P be a non-contradictory program. Then:*

- $lfp(\Gamma_s\Gamma) \subseteq lfp(\Gamma_s\Gamma_s) \subseteq lfp(\Gamma\Gamma_s)$
- $lfp(\Gamma_s\Gamma) \subseteq lfp(\Gamma\Gamma) \subseteq lfp(\Gamma\Gamma_s)$

4 SLX derivation procedure

In this section we review the definition of SLX[7], a top-down derivation procedure for logic programs extended with explicit negation under the well-founded semantics, first presented in [1], whose correctness wrt. *WFSX* is proven there.

The definition of SLX relies on SLDNF-like well known definitions of derivation, refutation, and failure. For the sake of simplicity, in [1] and also in this recap, the definition is only for ground (but possibly infinite) programs.

In order to informally motivate the definitions, we start with the simpler problem of programs without explicit negation. It is well known [11, 12, 13, 47, 61] that the main issues in the definition of top-down procedures for WFS are infinite positive recursion, and infinite recursion through

[6] This combination exactly corresponds to WFS with a renaming of explicitly negated literals, as in [55].

[7] Where X stands for eXtended programs, and SL stands for Selected Linear.

negation by default. The former gives rise to the truth value false (so that, for some L involved in the recursion, there should exist a refutation for *not* L, and no refutation for L), and the latter to the truth value undefined (so that both L and *not* L should have no refutation).

Apart from these additional problems, we mainly follow the ideas of SLDNF [15, 41] where refutations are derivations ending with the empty goal, derivations are constructed via resolution, and the negation as failure rule (stating that a selected *not* L is removed from the goal if L has no refutation and is the last goal of the derivation if L has one).

To solve the problem of positive recursion, as in SLS-resolution [54], we have introduced a failure rule for not necessarily finite derivations.

Example 4.1 Let $P = \{p \leftarrow p\}$. The derivations for p are $(\leftarrow p, \leftarrow p, \ldots)$ and every prefix of it. None of the prefixes of this infinite derivation is a refutation, because all of them end with a goal different from the empty goal. By stipulating that no infinite derivation is a refutation, there is no refutation for $\leftarrow p$, and so we have that $(\leftarrow \mathit{not}\ p, \leftarrow \Box)$ is a refutation.

For recursion through negation by default we want both L and *not* L to have no refutations, which seems to violate the negation as failure rule. In fact that rule in SLDNF states that if there is no refutation for L, *not* L should succeed (i.e. be removed from the goal), and if L has a refutation than *not* L should fail (i.e. be the last goal in the derivation).

This is so because SLDNF relies on a two-valued semantics, and so failure of verity means falsity and vice-versa. In WFS, it being a three-valued semantics, the same cannot apply. In fact, for WFS a failure of L simply means that L is not true, i.e. it can be false or undefined.

To solve this problem, others [53, 54, 57, 61] have defined derivation procedures that consider the extra status of "undefined" or "undetermined" assigned to goals. Literals involved in an infinite recursion through negation are assigned that status. As we shall see below, this approach does not easily generalize to extended programs (cf. remark 4.1).

Instead of considering an extra status, and for similarity with SLDNF, we've distinguished two kinds of derivations: SLX-T-derivations that prove verity in the WFM, and SLX-TU-derivations (where TU stands for "true or undefined") that prove non-falsity in the WFM. Now, for any L, the verity of *not* L succeeds iff there is no SLX-TU-refutation for L (i.e. L is false), and the non-falsity of *not* L succeeds iff there is no SLX-T-refutation for L (i.e. L is not true).

Having these two kinds of derivations, it becomes easy to define the derived goal of a goal G, even when the selected literal L is involved in recursion through negation by default. Indeed, since in this case and according to WFS, L is to be undefined, it must be failed (i.e. be the last goal in the derivation) if it occurs in a SLX-T-derivation, and refuted (i.e. be removed from the goal) if it occurs in a SLX-TU-derivation.

Example 4.2 Let $P = \{p \leftarrow \mathit{not}\ p\}$. In order to prove the verity of p we start by building a SLX-T-derivation for $\leftarrow p$. By resolving the goal with the program rule we obtain $\leftarrow p, \leftarrow \mathit{not}\ p$. Now, $\leftarrow \mathit{not}\ p$ is the last goal in the derivation if there is a SLX-TU-refutation for $\leftarrow p$, and is replaced by the empty goal otherwise.

So we start building a SLX-TU-derivation for $\leftarrow p$. Resolving the goal with the program rule we obtain $\leftarrow p, \leftarrow \mathit{not}\ p$, and so there is a recursion in *not* p through negation by default. Thus, as we are in a derivation for "true or undefined", *not* p is removed from the goal, and its derived goal is the empty goal. This gives a SLX-TU-refutation for $\leftarrow p$, which in turn determines $\leftarrow \mathit{not}\ p$ to be the last goal in the SLX-T-derivation above, making it a failed one. So p is undefined (T-derivations fail and there is a successful TU-derivation).

Though having two kinds of derivations, we do not duplicate the work. In fact, each derivation type can be simply seen as a status assigned to a usual derivation. One of the status fails literals involved in recursion through negation, and the other succeeds those literals.

Now we motivate the generalization of this procedure to deal with explicit negation in *WFSX*.

In a lot of points the treatment of extended programs is akin to that of normal ones, where instead of atoms we refer to objective literals, namely because, as expected, objective literals are treated exactly like atoms in WFS.

The main difference in the generalization to extended programs resides in the treatment of negation by default. In order to fulfill the coherence requirement there must be an additional way to refute a literal *not* L. In fact *not* L is true if $\neg L$ is true.

Example 4.3 Consider P:

$$\begin{array}{lllll} a & \leftarrow & not\ b & \neg a & \leftarrow \\ b & \leftarrow & not\ a & & \end{array}$$

whose WFM is $\{\neg a, b, not\ a, not\ \neg b\}$.

In order to prove the verity of b, one starts to build one SLX-T-derivation for goal $\leftarrow b$. By solving it with the only rule for b in P we obtain $\leftarrow b, \leftarrow not\ a$. Now, according to what was said above, one has to build SLX-TU-derivations for $\leftarrow a$.

Solving $\leftarrow a$ with the first program rule we obtain $\leftarrow a, \leftarrow not\ b$. Since this is a case of recursion through negation by default, and we are in a TU-derivation, the derived goal is $\leftarrow \Box$. Consequently there is no refutation for $\leftarrow b$. However, since $\neg a$ is true, by coherence $\leftarrow not\ a$ (and thus $\leftarrow b$ also) must be refuted, by the additional method of proof for default literals via coherence.

In SLX-T-derivations, in order to obtain coherence, we introduce an extra possibility of removing a *not* L from a goal:

> In a SLX-T-derivation, a selected literal *not* L is removed from a goal if there is no SLX-TU-refutation for L *or if there is one SLX-T-refutation for* $\neg L$.

Care must also be taken in non-falsity derivations because the coherence requirement can override undefinedness (cf. [45]):

Example 4.4 Consider P :

$$\begin{array}{llllll} a & \leftarrow & b & \neg b & \leftarrow & \\ b & \leftarrow & not\ c & c & \leftarrow & not\ c \end{array}$$

whose WFM is $\{\neg b, not\ a, not\ b, not\ \neg b, not\ \neg c\}$.

In trying to prove verity of *not* a one starts building SLX-TU-derivations for $\leftarrow a$ to check whether they fail. The only possible such derivation starts with $\leftarrow a, \leftarrow b, \leftarrow not\ c$. To determine the derived goal of *not* c one tries to find a SLX-T-refutation for $\leftarrow c$ which, as the reader can easily check, does not exist. Thus there is a SLX-TU-refutation for $\leftarrow a$ ($\leftarrow a, \leftarrow b, \leftarrow not\ c, \leftarrow \Box$), so that there is no SLX-T-refutation for $\leftarrow not\ a$.

However, *not* a belongs to the WFM of P. Note that this problem occurs because there is a SLX-TU-refutation for $\leftarrow b$ which should not exist since b is false. Indeed the falsity of b in the WFM is imposed by the coherence principle, since $\neg b$ is true. Note that in the derivations above $\neg b$ is not even used.

In fact, in the SLX-TU-derivation for $\leftarrow a$, the goal $\leftarrow b$ should be the last one, because $\neg b$ is true. In general:

> In a SLX-TU-derivation resolution can only be applied to a selected objective literal L *if there is no SLX-T-refutation for* $\neg L$.

Remark 4.1 *Note that coherence intervenes differently in T and TU-derivations. This is a consequence of having two distinct anti-monotonic operators in the definition of the WFM.*

At this point it is easier to understand the claim we made before that the usage of an undefined status, instead of having two different kinds of derivation, hampers the generalization to extended programs. In fact, in normal programs one can simply replace the two kinds of derivations by the addition of an undefined status because the single difference between the two derivation types is in the success or failure of literals involved in infinite recursion through default negation. However, in extended programs there are other differences, namely in what regards the applications of the coherence principle, which make the distinction crucial.

The formalization of the intuitions presented above yields the following definitions of derivations and refutations:

Definition 4.1 (SLX refutation and failure) *A SLX-T-refutation (resp. SLX-TU-refutation) for G in P is a finite SLX-T-derivation (resp. SLX-TU-derivation) which ends in the empty goal ($\leftarrow \Box$).*

A SLX-T-derivation (resp. SLX-TU-derivation) for G in P is a SLX-T-failure iff it is not a refutation, i.e. it is infinite or it ends with a goal other than the empty goal.

Definition 4.2 (SLX-T-derivation) *Let P be an extended program, and R an arbitrary but fixed computational rule. A SLX-T-derivation $G_0, G_1, \ldots$ for G in P via R is defined as follows: $G_0 = G$. Let G_i be $\leftarrow L_1, \ldots, L_n$ and suppose that R selects the literal L_k $(1 \leq k \leq n)$. Then:*

- *if L_k is an objective literal, and the input rule is $L_k \leftarrow B_1, \ldots, B_m$, the derived goal is $\leftarrow L_1, \ldots, L_{k-1}, B_1, \ldots, B_m, L_{k+1}, \ldots L_n$.*
- *if L_k is not A then, if there is a SLX-T-refutation for $\neg A$ in P or there is no SLX-TU-refutation for A in P, the derived goal is $\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots L_n$.*
- *otherwise G_i is the last goal in the derivation*

Definition 4.3 (SLX-TU-derivation) *Let P be an extended program, and R an arbitrary but fixed computational rule. A SLX-TU-derivation $G_0, G_1, \ldots$ for G in P via R is defined as follows: $G_0 = G$. Let G_i be $\leftarrow L_1, \ldots, L_n$ and suppose that R selects the literal L_k $(1 \leq k \leq n)$. Then:*

- *if L_k is an objective literal then*
  - *if there exists a SLX-T-refutation for $\neg L_k$ then G_i is the last goal in the derivation.*
  - *otherwise, if the input rule is $L_k \leftarrow B_1, \ldots, B_m$ the derived goal is:*
    $$\leftarrow L_1, \ldots, L_{k-1}, B_1, \ldots, B_m, L_{k+1}, \ldots L_n$$
  - *if there is no rule for L_k then G_i is the last goal in the derivation.*
- *if L_k is not A then:*
  - *if there is a SLX-T-refutation for $\leftarrow A$ in P then G_i is the last goal in the derivation.*
  - *if all SLX-T-derivations for $\leftarrow A$ are SLX-T-failures then the derived goal is:*
    $$\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots L_n.$$
  - *due to infinite recursion through default negation, it might happen that the previous points are not enough to determine the derived goal. In such a case, by definition, the derived goal is also $\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots L_n$.*

The soundness of this procedure wrt. to both *WFSX* and the answer-sets semantics, as well as its theoretical completeness[8] wrt. *WFSX* were proven in [1]. In this section we simply present the theorems. Their proofs follow immediately from the proofs of theorems 5.5 and 5.6 presented in appendix A.

Theorem 4.1 (Soundness of SLX) *Let P be a non-contradictory extended logic program, L an arbitrary literal from P. If there is an SLX-T-refutation for $\leftarrow L$ in P then $L \in WFM(P)$.*

Corollary 4.1 (Soundness wrt. answer-sets) *Let P be an extended logic program with at least one answer-set, and L an arbitrary objective literal from P. If there is an SLX-T-refutation for $\leftarrow L$ in P then L belongs to all answer-sets of P. If there is an SLX-T-refutation for $\leftarrow$ not L in P then there is no answer-set of P with L.*

Theorem 4.2 (Theoretical completeness of SLX) *Let P be a non-contradictory program, and L an arbitrary literal from P. If $L \in WFM(P)$ then there exists a SLX-T-refutation for $\leftarrow L$ in P.*

[8] In practice completeness cannot be achieved because in general the WFM is not computable [24]. However, in theory, and with the possible need of constructing more than ω derivations, completeness is obtained.

5 Detecting Contradictions

As mentioned in the introduction, being able to "run" non-contradictory programs is not our only goal. We're also interested in detecting occurrences of contradiction in order to remove it.

To remove a contradiction, as we shall see in detail in section 7, we need first to detect it and, more importantly, we need to know what are the "reasons" or "causes" that lead to contradiction, i.e. what are the assumptions made that are responsible for contradiction by supporting it. But both the definition of *WFSX* and the correctness results for SLX do not serve these purposes. Note that the WFM is only defined for non-contradictory programs, and correctness is only proven for those programs.

It can be argued that the definition of *WFSX* is good enough for detecting contradiction, since it is possible to know whether a program has PSMs. In [1], we've shown that SLX can detect contradiction: if a program P is contradictory then there exists a $L \in \mathcal{H}$ for which there are SLX-T-refutations for both $\leftarrow L$ and $\leftarrow \neg L$. But these results are still weak for the purpose of using the semantics, as well as SLX, to determine how a contradiction is to be removed. They do not guide us to the "causes" for the contradiction.

To be able to determine such "causes", in this section we first generalize *WFSX* in order to define a paraconsistent WFM for contradictory programs, and then show that the correctness of SLX can be lifted to this paraconsistent case.

The main idea of the paraconsistent *WFSX* (or $WFSX_p$ for short) is to calculate, always in keeping with coherence, all consequences of the program, even those leading to contradiction, as well as those arising from contradiction[9]. The following example provides an intuitive preview of what we intend to capture:

Example 5.1 Consider program P :

$$\begin{array}{rlcrlc} a \leftarrow & not\ b & \text{(i)} & d \leftarrow & not\ a & \text{(iii)} \\ \neg a \leftarrow & not\ c & \text{(ii)} & e \leftarrow & not\ \neg a & \text{(iv)} \end{array}$$

1. $not\ b$ and $not\ c$ hold since there are no rules for either b or c.
2. $\neg a$ and a hold from 1 and rules (i) and (ii).
3. $not\ a$ and $not\ \neg a$ hold from 2 and the coherence principle relating the two negations.
4. d and e hold from 3 and rules (iii) and (iv).
5. $not\ d$ and $not\ e$ hold from 2 and rules (iii) and (iv), as they are the only rules for d and e.
6. $not\ \neg d$ and $not\ \neg e$ hold from 4 and the coherence principle.

The whole set of literal consequences is then:

$$\{not\ b, not\ c, \neg a, a, not\ a, not\ \neg a, d, e, not\ d, not\ e, not\ \neg d, not\ \neg e\}.$$

Here we do not argue that $WFSX_p$ is in fact a *"semantics"*. Instead we view $WFSX_p$ as a tool needed for detecting contradiction and its causes, when *WFSX* is not able to assign meaning to a program. This tool is then used as the basis for removing contradiction where possible. The semantics assigned to the original is the *WFSX* of the program obtained after removing contradiction.

Since the purpose of $WFSX_p$ is to detect contradiction on the basis of *WFSX*, every principle that allows derivation of literals in *WFSX* must also be enforced in $WFSX_p$. This is why coherence is still kept in the paraconsistent case.

WFSX is not defined for contradictory programs because such programs have no PSMs. By definition of PSM, a program has none if either it has no fix-points of $\Gamma\Gamma_s$ or if all fix-points T of $\Gamma\Gamma_s$ do not comply with $T \subseteq \Gamma_s T$. The next theorem shows that the first case is impossible, i.e. all programs (contradictory or otherwise) have fix-points of $\Gamma\Gamma_s$.

[9] This kind of paraconsistent reasoning is called *liberal reasoning* in [67].

Theorem 5.1 *The operator $\Gamma\Gamma_s$ is monotonic, for arbitrary sets of literals.*

Proof: We have to prove that for arbitrary sets of objective literals A and B: $A \subseteq B \Rightarrow \Gamma\Gamma_s A \subseteq \Gamma\Gamma_s B$. To do so we begin by proving that both Γ and Γ_s are anti-monotonic:

Assume that $A \subseteq B$. By definition of GL-transformation, it is clear that $\frac{P}{B} \subseteq \frac{P}{A}$ (resp. $\frac{P_s}{B} \subseteq \frac{P_s}{A}$). Since these transformed programs are definite, and Horn clause logic is monotonic, it directly follows that $\Gamma B \subseteq \Gamma A$ (resp. $\Gamma_s B \subseteq \Gamma_s A$).

Thus, if $A \subseteq B$, by anti-monotonicity of Γ_s, $\Gamma_s B \subseteq \Gamma_s A$, and by anti-monotonicity of Γ, $\Gamma\Gamma_s A \subseteq \Gamma\Gamma_s B$. $\diamondsuit$

Consequently every program has a least fix-point of $\Gamma\Gamma_s$. If, for some program P, the least fix-point of $\Gamma\Gamma_s$ complies with condition (2) of definition 3.3 then the program is non-contradictory and the least fix-point is the WFM. Otherwise the program is contradictory. Moreover, the test for knowing whether a program is contradictory, given its least fix-point of $\Gamma\Gamma_s$, can be simplified to: "the program is non-contradictory iff its least fix-point of $\Gamma\Gamma_s$ has no pair of $\neg$-complementary literals" (cf. theorem 5.3 below).

Lemma 5.2 *Let S be any non-contradictory set of literals, i.e. such that $\not\exists L \in \mathcal{H},\ \{L, \neg L\} \subseteq S$. Then, $S \cap \Gamma S \subseteq \Gamma_s S$.*

Proof: Let L be any literal such that $L \in S \cap \Gamma S$, i.e. $L \in S \wedge L \in \Gamma S$. Given that, by hypothesis, S is non-contradictory, then $\neg L \notin S$. Thus $\neg L \notin S \wedge L \in \Gamma S$, which, by definition of Γ_s, clearly implies that $L \in \Gamma_s S$. $\diamondsuit$

Theorem 5.3 *Let T be the least fix-point of $\Gamma\Gamma_s$ for a program P. Then:*

$$T \not\subseteq \Gamma_s T \quad iff \quad \exists L \in \mathcal{H},\ \{L, \neg L\} \subseteq T$$

Proof:

$\Leftarrow$ If $\exists L \in \mathcal{H},\ \{L, \neg L\} \subseteq T$ then when calculating $\Gamma_s T$ all rules for both L and $\neg L$ are deleted. Thus neither L nor $\neg L$ belong to $\Gamma_s T$, and so $T \not\subseteq \Gamma_s T$.

$\Rightarrow$ We prove that, if $T = lfp(\Gamma\Gamma_s)$ and $\not\exists L \in \mathcal{H},\ \{L, \neg L\} \subseteq T$, then $T \subseteq \Gamma_s T$.

We start by showing that with the above assumptions, $T \cap \Gamma_s T$ is a post-fixed point of $\Gamma\Gamma_s$, i.e. $\Gamma\Gamma_s\,(T \cap \Gamma_s T) \subseteq T \cap \Gamma_s T$. This inclusion is proven in two steps: (1) $\Gamma\Gamma_s(T \cap \Gamma_s T) \subseteq T$, and (2) $\Gamma\Gamma_s(T \cap \Gamma_s T) \cap T \subseteq \Gamma_s T$.

1. Trivially $T \cap \Gamma_s T \subseteq T$. By monotonicity of $\Gamma\Gamma_s$, this implies that $\Gamma\Gamma_s(T \cap \Gamma_s T) \subseteq \Gamma\Gamma_s T = T$.
2. Trivially $T \cap \Gamma_s T \subseteq \Gamma_s T$. By anti-monotonicity of Γ, this implies $T = \Gamma\Gamma_s T \subseteq \Gamma(T \cap \Gamma_s T)$, i.e. $\forall L,\ L \in T \Rightarrow L \in \Gamma(T \cap \Gamma_s T)$.
   Given that by hypothesis $\not\exists L \in \mathcal{H},\ \{L, \neg L\} \subseteq T$, if $L \in T$ then $\neg L \notin T$, and since $T \cap \Gamma_s T \subseteq T$, it follows that $\neg L \notin T \cap \Gamma_s T$. Thus:
   $$\forall L,\ L \in T \quad \Rightarrow \quad (\neg L \notin T \cap \Gamma_s T \ \wedge\ L \in \Gamma(T \cap \Gamma_s T))$$
   So, similarly to the proof of lemma 5.2: $\forall L,\ L \in T \quad \Rightarrow \quad L \in \Gamma_s(T \cap \Gamma_s T)$, i.e. $T \subseteq \Gamma_s(T \cap \Gamma_s T)$.
   Again by anti-monotonicity of Γ, $\Gamma\Gamma_s(T \cap \Gamma_s T) \subseteq \Gamma T$.
   This implies that $\Gamma\Gamma_s(T \cap \Gamma_s T) \cap T \subseteq T \cap \Gamma T$, and by lemma 5.2:
   $$\Gamma\Gamma_s(T \cap \Gamma_s T) \cap T \subseteq T \cap \Gamma T \subseteq \Gamma_s T$$

Since $T \cap \Gamma_s T$ is a post-fixed point of $\Gamma\Gamma_s$, and $\Gamma\Gamma_s$ is monotonic, then[10] $lfp(\Gamma\Gamma_s) \subseteq T \cap \Gamma_s T$. Given that, by hypothesis, $T = lfp(\Gamma\Gamma_s)$, $T \subseteq T \cap \Gamma_s T$, and so, by the properties of intersection, $T \subseteq \Gamma_s T$. $\diamondsuit$

[10] Recall that $lfp(F) = glb\{x |\ F(x) \leq x\}$, where F is any monotonic operator in a complete lattice.

So, if one is interested only in the WFM, the condition $T \subseteq \Gamma_s T$ can be replaced by testing whether T has $\neg$-complementary literals. As noted in page 7, the former condition guarantees that literals cannot be both true and false by default. By removing that condition this guarantee is no longer valid. But this is precisely what we want in the definition of the paraconsistent *WFSX* (note how both a and *not* a belong to the desired semantics in example 5.1).

All which is explained above points towards a definition of $WFSX_p$ where the construction of the WFM given by the least fix-point of $\Gamma\Gamma_s$ is kept, condition $T \subseteq \Gamma_s T$ is removed, and where contradictory programs are those that contain a pair of complementary literals in the WFM:

Definition 5.1 (Paraconsistent *WFSX*) *Let P be an extended program whose least fix-point of $\Gamma\Gamma_s$ is T. Then, the paraconsistent well-founded model of P is $WFM_p(P) = T \cup not\ (\mathcal{H} - \Gamma_s T)$.*

Theorem 5.4 (Generalization of *WFSX*) *Let P be such that $WFM_p(P) = T \cup not\ F$. P is non-contradictory iff for no objective literal L, $\{L, \neg L\} \subseteq T$. Moreover, if P is non-contradictory then $WFM_p(P) = WFM(P)$.*

Does this definition fulfill the objectives we put to ourselves in the beginning of this section? In other words, does $WFSX_p$ comply with coherence? And does it calculate all consequences of the program, even those leading to contradiction, as well as those arising from contradiction?

The answer to both these questions is yes. The fact that $WFSX_p$ complies with coherence is easy to check. The same argument used for non-contradictory programs can also be used here: for a model $T \cup not\ F$, if $\neg L$ belongs to T, then in $\Gamma_s T$, via semi-normality, all rules for L are removed and, consequently, $L \notin \Gamma_s T$, i.e. $L \in F$.

Checking the verity of the answer to the second question is not so immediate. What we wish to guarantee is that when a literal L is both true and false, we do consider both the consequences of it being true, as well as those of it being false. Let's first look at the following property of the sequence for calculating the least fix-point of $\Gamma\Gamma_s$ of a program.

Proposition 5.1 *Let $\{I_\alpha\}$ be the sequence for calculating the least fix-point T of $\Gamma\Gamma_s$ in program P, let I_β be an element of that sequence, and L be an arbitrary literal from P. Then:*

- *if $L \notin \Gamma_s I_\beta$ then not $L \in WFM_p(P)$;*
- *if $L \in \Gamma\Gamma_s I_\beta$ then $L \in WFM_p(P)$.*

Proof (sketch): The proof of the second item is trivial given the monotonicity of $\Gamma\Gamma_s$. To prove the first item, it is enough to prove that the sequence obtained by iterating $\Gamma_s\Gamma$ starting with $\Gamma_s\{\}$ is decreasing. This result is proven in [6]. For brevity, and given that the proof is rather long, we do not present it here. $\diamondsuit$

With this property one can envisage the applications of Γ_s as determining verity of default literals, assuming that the *not* Ls are false for all Ls already determined true; and the applications of Γ as determining verity of objective literals, assuming that the *not* Ls are true for all Ls already determined false.

Consequently, to guarantee that when a literal L is both true and false all the consequences that should follow are obtained, one has to guarantee that, when applying Γ_s, both L and $\neg L$ always belong to its argument so as to maximize default literals; and that, when applying Γ, neither L nor $\neg L$ ever belong to its argument so as to maximize objective literals as well. The next proposition does just that:

Proposition 5.2 *Let P be a contradictory program whose least fix-point of $\Gamma\Gamma_s$ is T, and such that $\{L, \neg L\} \subseteq T$. Then there exists an element I_α of the sequence for calculating T such that $\{L, \neg L\} \subseteq I_\alpha$. Moreover, for every ordinal $\beta \geq \alpha$, $\{L, \neg L\} \subseteq I_\beta$, $L \notin \Gamma_s I_\beta$, and $\neg L \notin \Gamma_s I_\beta$.*

Proof (sketch): The existence of I_α is guaranteed by definition of the sequence. The first consequence follows immediately from the monotonicity of $\Gamma\Gamma_s$. The second and third follow from the first one and the semi-normality in Γ_s. $\diamondsuit$

Example 5.2 The sequence for the least fix-point of $\Gamma\Gamma_s$ of program P of example 5.1 is:

$$\begin{array}{ll} I_0 = \{\} & \\ I_1 = \Gamma\Gamma_s\{\} & = \Gamma\{a,\neg a,d,e\} = \{a,\neg a\} \\ I_2 = \Gamma\Gamma_s\{a,\neg a\} & = \Gamma\{\} \qquad\qquad = \{a,\neg a,d,e\} \\ I_3 = \Gamma\Gamma_s\{a,\neg a,d,e\} & = \Gamma\{\} \qquad\qquad = \{a,\neg a,d,e\} \ = \ I_2 \end{array}$$

Thus $WFM_p(P) = \{not\ b, not\ c, \neg a, a, not\ a, not\ \neg a, d, e, not\ d, not\ e, not\ \neg d, not\ \neg e\}$.

Note that $not\ d$ is obtained in I_1 (i.e. $d \notin \Gamma_s I_1$) because $a \in I_1$, and that d is obtained in I_2 (i.e. $d \in \Gamma_s I_2$) because $a \notin \Gamma_s I_1$.

Having defined $WFSX_p$ for contradictory programs, in order to implement it, we wish to define a "top-down" derivation procedure for it. The next theorems[11] show that a new derivation procedure is not needed because SLX is also correct wrt. the paraconsistent *WFSX*.

Theorem 5.5 (Soundness of SLX) *Let P be an extended logic program, L an arbitrary literal from P. If there is an SLX-T-refutation for $\leftarrow L$ in P then $L \in WFM_p(P)$.*

Theorem 5.6 (Theoretical completeness of SLX) *Let P be an extended program, and L an arbitrary literal from P. If $L \in WFM_p(P)$ then there exists a SLX-T-refutation for $\leftarrow L$ in P.*

6 On the Implementation of SLX

Although sound and complete for the paraconsistent *WFSX*, SLX is not effective (even for finite ground programs). In fact, and because it furnishes no mechanism for detecting loops, termination is not guaranteed. Completeness here is only ideal completeness. In order to provide an effective implementation of SLX we have first to tackle the issue of guaranteeing termination.

As for the WFS of normal programs, *WFSX* and $WFSX_p$ too are in general not computable. Thus, it is not possible to guarantee termination in the general case. In this section we modify SLX such that termination is guaranteed (at least) for finite ground programs[12].

This modified SLX procedure can be easily implemented via a Prolog meta-interpreter[13]. Due to its SLDNF-resemblance, it has also been rather easy to implement a pre-processor that compiles $WFSX_p$ programs into Prolog, using a format corresponding to the specialization of the interpreter rules, plus a small number of general "built-in" predicates. Lack of space prevents us from presenting these implementations here[14].

6.1 Guaranteeing Termination

To guarantee termination (at least) for finite ground programs, in this section we introduce rules that prune SLX-derivations, and eliminate both cyclic positive recursion and cyclic recursion through negation by default (hereafter simply called cyclic negative recursion).

To detect both kinds of cyclic recursions we use two kinds of ancestors:

- *Local ancestors* are assigned to literals in the goals of a derivation, and are used for detecting cyclic positive recursion. For the purpose of including local ancestors, we replace literals in goals by pairs $L_i : S_i$, where L_i is a literal and S_i is the set of its local ancestors.
- *Global ancestors* are assigned to derivations, and are used to detect cyclic negative recursion.

[11] Proofs are presented in appendix A.

[12] The technique we're about to define also guarantees termination for allowed bounded-term non-ground programs. The discussion of guaranteeing termination for these cases is, however, beyond the scope of this paper.

[13] This can be simply done by mimicking the definition of SLX-derivations with ancestors, considering a left-most selection rule.

[14] The code is available on request.

Intuitively, and if one thinks of a derivation as expanding an AND-tree, the local ancestor of a literal occurrence are the literals appearing in the path from the root of the tree to that occurrence.

Global ancestors of a subsidiary derivation are the local ancestors of the literal L that invoked it, plus the ancestor goal of the derivation in which L appears. The top-goal derivation has no global ancestors. Moreover we divide global ancestors into two sets: global T-ancestors and global TU-ancestors. Global T-ancestors (resp. TU-ancestors) are those that were introduced in a SLX-T-derivation (resp. SLX-TU-derivation).

To deal with the non-termination problem of cyclic positive recursion it suffices to guarantee that no such infinite derivations are generated. That can be achieved if no selected literal belonging to its set of local ancestor is ever expanded. This leads to the following pruning rule:

1. Let G_i be a goal in a SLX-derivation (either T or TU), and let L_k be the literal selected by R. If L_k belongs to its local ancestors then G_i is the last goal in the derivation.

To treat cyclic negative recursion, tests over the global ancestors are necessary. It is easily shown that any form of this recursion reduces to one of four combination cases, depending on the cycle occurring between the two possible derivation types. In SLX-TU-derivations the selected literal is removed from the goal, and in SLX-T-derivations the goal is the last in the derivation. Moreover, all these combinations can be reduced to just one:

Lemma 6.1 (Reduction of negative cycles) *All cyclic negative recursions can be detected in SLX-T-derivations by looking only at its global T-ancestors.*

The same does not hold for any other combination case, i.e. there are cycles that are only detectable with the test of lemma 6.1 (cf. [2]). Lemma 6.1 yields pruning rule 2:

2. Let G_i be a goal in a SLX-T-derivation, and let L_k be the literal selected by R. If L_k belongs to the set of global T-ancestors then G_i is the last goal in the derivation.

Theorem 6.2 (Elimination of cyclic recursion for WFSX) *Pruning rules 1 and 2 are necessary and sufficient for guaranteeing that all positive and negative cyclic recursions are eliminated.*

We now embed these two pruning rules in the definitions of SLX-derivation (refutations and failures remain as before). Note that the pruning rules do not make use of TU-ancestors. So they will not be considered in the definitions:

Definition 6.1 (SLX-T-derivation with ancestors) *Let P be an extended program, and R an arbitrary but fixed computational rule. A SLX-T-derivation $G_0, G_1, \ldots$ for G in P via R, with T-ancestors ST is defined as follows: $G_0 = G : \{\}$. Let G_i be $\leftarrow L_1 : S_1, \ldots, L_n : S_n$ and suppose that R selects $L_k : S_k$ $(1 \leq k \leq n)$. Then:*

- *if L_k is an objective literal, $L_k \notin S_k \cup ST$, and the input rule is $L_k \leftarrow B_1, \ldots, B_m$, the derived goal is $\leftarrow L_1 : S_1, \ldots, L_{k-1} : S_{k-1}, B_1 : S', \ldots, B_m : S', L_{k+1} : S_{k+1}, \ldots L_n : S_n$ where $S' = S_k \cup \{L_k\}$.*
- *if L_k is not A then, if there is a SLX-T-refutation for $\leftarrow \neg A : \{\}$ in P with T-ancestors $ST \cup S_k$, or there is no SLX-TU-refutation for $\leftarrow A : \{\}$ in P with the same ancestors, the derived goal is $\leftarrow L_1 : S_1, \ldots, L_{k-1} : S_{k-1}, L_{k+1} : S_{k+1}, \ldots L_n : S_n$.*
- *otherwise G_i is the last goal in the derivation.*

Definition 6.2 (SLX-TU-derivation with ancestors) *Let P be an extended program, and R an arbitrary but fixed computational rule. A SLX-TU-derivation $G_0, G_1, \ldots$ for G in P via R, with T-ancestors ST is defined as follows: $G_0 = G : \{\}$. Let G_i be $\leftarrow L_1, \ldots, L_n$ and suppose that R selects $L_k : S_k$ $(1 \leq k \leq n)$. Then:*

- *if L_k is an objective literal then*

– *if $L_k \in S_k$ or there is no rule for L_k then G_i is the last goal in the derivation.*
– *else if there exists a SLX-T-refutation for $\leftarrow \neg L_k : \{\}$ with T-ancestors ST then G_i is the last goal in the derivation.*
– *otherwise, if the input rule is $L_k \leftarrow B_1, \ldots, B_m$ the derived goal is:*

$$\leftarrow L_1 : S_1, \ldots, L_{k-1} : S_{k-1}, B_1 : S', \ldots, B_m : S', L_{k+1} : S_{k+1}, \ldots L_n : S_n$$

where $S' = S_k \cup \{L_k\}$.

- *if L_k is not A then:*
  – *if there is a SLX-T-refutation for $\leftarrow A : \{\}$ in P with T-ancestors ST then G_i is the last goal in the derivation.*
  – *otherwise the derived goal is $\leftarrow L_1 : S_1, \ldots, L_{k-1} : S_{k-1}, L_{k+1} : S_{k+1}, \ldots L_n : S_n$.*

Theorem 6.3 (Correctness for SLX with ancestors) *Let P be an extended program. Then:*

- *If $L \in WFM_p(P)$ then there is a SLX-T-refutation for $\leftarrow L : \{\}$ with empty T-ancestors. Moreover, all the subsidiary derivations needed in the refutation are finite, and in finite number.*
- *If $L \notin WFM_p(P)$ then all SLX-T-derivations for $\leftarrow L : \{\}$ with empty T-ancestors are finite and end with a goal different from $\leftarrow \Box$. Moreover, all the subsidiary derivations needed are finite, and in finite number.*

Proof: Follows from theorems 6.2, 5.5, and 5.6. $\diamondsuit$

7 Revision Framework

In the previous sections we presented both a paraconsistent version of *WFSX* and a corresponding top-down querying procedure (able therefore to detect contradiction in programs). We are now ready to foster the stance, adopted in the introduction, that if a program is contradictory its revision is in order.

By revision of a contradictory program P we mean a non-contradictory program P' obtained from P without introducing new predicates. One question immediately arises: what changes are allowed to P in order to obtain P'?

Clearly, any modification must result from changing directly the truth value of some, considered basic, literals. Thus the above question hinges on the following one: for which literals, the revisable ones, should we allow explicit modification in their truth values?

As argued in our previous work [46, 50], the modification should be based solely on those literals that depend on no other literals, and by introducing or removing rules for such basic literals only. This is the stance we take here. Thus, for the purpose of defining revision, we begin by describing a framework that explicitly divides the program into a pair called a program state: one element contains the subprogram that cannot be changed; the other contains the subprogram that can be changed in a predefined way, namely by adding rules of a simple fixed form for the revisable literals only. The latter part is such that it contains rules only for literals that depend on no other. A program state is changed (to another state) simply by modifying the latter part. In order to formalize the notion of program states, we begin by defining open programs. These are "open" in the sense that the part which can be changed is not yet fixed in truth value. The changes in truth-value of the other, non revisable, literals will be indirectly brought about, via the program rules, by the changes in the open part of the program, through the presence, in rules, of revisable literals.

Until now we've considered contradiction as coming simply from the nonexistence of a consistent *WFSX*. In general, however there can be other reasons to consider a program state unacceptable, due to additional integrity knowledge about the intended program semantics. To encompass these situations, we now extend our language to deal with general integrity constraints of the form:

$$L_1 \vee L_2 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge L_{n+2} \wedge \ldots \wedge L_{n+m} \qquad (n+m \geq 0)$$

where each L_i is a literal, and $\Leftarrow$ stands for classical implication.

A program is contradictory iff its *WFSX* is contradictory, or violates some integrity constraint. By theorem 5.3, the first proviso can be adroitly removed by adding the constraint $\Leftarrow L, \neg L$, for every literal L in the program.

In our previous work, we've always considered separately the two cases of removing contradiction by changing the truth value of revisable literals either into undefined [5, 46], or into true [50]. Note that, in the absence of integrity constraints, contradiction can always be removed by changing the truth value of literals into undefined only. However, in order to apply our contradiction removal approach to diagnosis, for instance, we also felt the need for stronger revisions, where the truth value of literals can be changed into true. Indeed, whenever normality of some component could not be assumed, we wanted that its abnormality be posited, in order to derive any available information via known fault models.

In the present work, as already mentioned above, we will allow revision by introducing or removing rules for revisable literals, such that we can change their truth values from any value to any other value, in a mixed way. By relying on the above more general integrity constraints, a mixture of literal revisions to undefined or to the default complement value can be obtained. If no constraint requires otherwise, the truth value of literals is (knowledge minimally) changed into undefined; if one desires a literal, say $ab(C)$ in the diagnosis setting, to have solely truth-values false and true then we may simply introduce the constraint $ab(C) \vee not\ ab(C) \Leftarrow \mathbf{t}$.

Example 7.4.1 below shows how this mixed revision can be used to perform three-valued declarative debugging, i.e. declarative debugging of programs under a three-valued semantics. We also show the use and utility of this admixture in the diagnosis setting, in subsection 7.4.2.

The structure of the remainder of this section is as follows: we begin by extending the language with the more general integrity constraints above. Then we present the framework of open programs and program states, and in the next subsection we define the declarative semantics of revision[15], by means of minimal changes to a current contradictory program state.

7.1 Integrity Constraints

As argued by Reiter in [60], the basic utility of integrity constraints is that only some program (or database) states are considered acceptable, and the constraints are meant to enforce these acceptable states by filtering out all other ones. Integrity constraints can be of two types:

Static The enforcement of these constraints depends only on the current state of the database, independently of any prior state.

Dynamic These depend on two or more program states. Reiter gives as example that employee salaries can never decrease.

It is not a purpose of this article address the evolution of a program in time, and thus only static constraints are considered. Reiter shows that the classical accounts of integrity constraints in first-order knowledge bases [33, 42, 59] (namely those of consistency with or entailment by the database) do not capture our intuitions. The author argues that a program expresses knowledge about the external world and the integrity constraints address this epistemic state, i.e. what the program knows. These problems do not carry over to our setting because we assume the database

[15] The semantics is defined at present only for finite ground programs.

has a single three-valued model (the program state) and therefore the epistemic and theoremhood views coincide. We'd have to address these problems if we allow disjunction in the heads of rules.

Due to the fact that we are considering a three-valued setting, the approaches of two-valued revision or abductive frameworks (e.g [19, 22, 23, 30]) do not immediately carry over.

Let's start by presenting our integrity constraint language:

Definition 7.1 (Integrity Constraints) *An integrity constraint of an extended logic program P has the following normal form, where each L_i is a literal belonging to the language of P:*

$$L_1 \vee L_2 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge L_{n+2} \wedge \ldots \wedge L_{n+m} \qquad (n+m \geq 0)$$

To an empty consequent (head) we associate the symbol $\mathbf{f}$ *and to an empty antecedent (body) the symbol* $\mathbf{t}$. *An integrity theory is a set of integrity constraints, standing for their conjunction.*

Notice that the literals appearing in the constraints can be objective or default ones. According to the theoremhood view of integrity constraint checking [59, 42] we define constraint satisfaction in the following manner:

Definition 7.2 (Constraint Satisfaction) *Given a set of literals I[16], a ground integrity constraint*

$$L_1 \vee L_2 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge L_{n+2} \wedge \ldots \wedge L_{n+m} \qquad (n+m \geq 0)$$

is violated by I iff every literal $L_{n+1}, \ldots, L_{n+m} \in I$ and none of the literals $L_1, \ldots, L_n \in I$. Otherwise, the constraint is satisfied by I.

Equivalently, an integrity constraint is satisfied iff the following (classical) implication is satisfied:

$$L_1 \in I \vee \ldots \vee L_n \in I \Leftarrow L_{n+1} \in I \wedge \ldots \wedge L_{n+m} \in I \quad (n+m \geq 0)$$

For simplicity, we restrict ourselves to constraints with a finite number of literals from finite interpretations. In the general case (countable sets of literals) it is necessary to extend the setting to first-order logic.

Except from that, we assume nothing about the underlying interpretations because we'd like to be as general as possible. Enforcing some condition on them has to be by an explicit statement in the integrity theory.

It is only necessary to consider, as our basic building blocks, constraints of the form $L \Leftarrow \mathbf{t}$ and $\mathbf{f} \Leftarrow L$, where L is a literal, to be able to "program" more elaborate conditions. By combining these primitive constraints using the connectives '$\wedge$', '$\vee$' and '$\Leftarrow$' we can express any property about some interpretation I. Subsequently, any such condition can be easily rewritten into integrity theory form by using the properties of propositional classical logic, i.e. translated into a conjunction of simple classical implications in the normal form "disjunction if conjunction". To impose a given value condition on a specific literal in I use table 1, where $L = \mathbf{f}$ (resp. $\mathbf{u}$, $\mathbf{t}$) we mean that L is false (resp. undefined, true), to obtain the corresponding integrity constraint theory.

Condition	Integrity Theory	Condition	Integrity Theory
$L = \mathbf{f}$	$not\ L \Leftarrow \mathbf{t}$	$L \neq \mathbf{f}$	$\mathbf{f} \Leftarrow not\ L$
$L = \mathbf{u}$	$(\mathbf{f} \Leftarrow L) \wedge (\mathbf{f} \Leftarrow not\ L)$	$L \neq \mathbf{u}$	$L \vee not\ L \Leftarrow \mathbf{t}$
$L = \mathbf{t}$	$L \Leftarrow \mathbf{t}$	$L \neq \mathbf{t}$	$\mathbf{f} \Leftarrow L$

Table 1: Basic conditions on I

[16]Remark that the notion of constraint satisfaction refers to an arbitrary set of literals I, be it contradictory or otherwise.

Example 7.1 Imagine the condition "if a is undefined and b is false then c is true" must be verified by a given interpretation. We lookup to table 1 and formalize it as:

$$(c \Leftarrow \mathbf{t}) \Leftarrow [(\mathbf{f} \Leftarrow a) \wedge (\mathbf{f} \Leftarrow not\ a) \wedge (not\ b \Leftarrow \mathbf{t})]$$

By means of the classical (meta-)interpretation of integrity constraints we obtain:

$$\begin{gathered}(c \in I \Leftarrow \mathbf{t}) \Leftarrow [(\mathbf{f} \Leftarrow a \in I) \wedge (\mathbf{f} \Leftarrow not\ a \in I) \wedge (not\ b \in I \Leftarrow \mathbf{t})] \\ \equiv \\ c \in I \Leftarrow a \notin I \wedge not\ a \notin I \wedge not\ b \in I\end{gathered}$$

Moving the negated conjuncts to the consequent of the last implication, we produce:

$$c \in I \vee a \in I \vee not\ a \in I \Leftarrow not\ b \in I$$

which is nothing more than the integrity constraint theory:

$$\{c \vee a \vee not\ a \Leftarrow not\ b\} \tag{1}$$

The reader can check any interpretation obeys the enounced condition iff it makes (1) true. □

Special integrity theories enforce some well-known basic principles. Suppose that for all ground objective literals L at least one of the conditions below is present in the integrity theory.

$not\ L \Leftarrow \neg L$	$(c1)$	$\mathbf{f} \Leftarrow L \wedge not\ L$	$(c3)$	$L \vee not\ L \Leftarrow \mathbf{t}$	$(c5)$
$\neg L \Leftarrow not\ L$	$(c2)$	$\mathbf{f} \Leftarrow L \wedge \neg L$	$(c4)$	$L \vee \neg L \Leftarrow \mathbf{t}$	$(c6)$

Condition *c1* is nothing more than the coherence requirement on interpretations. Condition *c2* expresses a type of Closed World Assumption, i.e. if L can be assumed false then it is explicitly false; in combination with *c1* they make explicit negation and default negation equivalent. Conditions *c3* and *c4* enforce non-contradiction of the interpretations (notice that if coherence is imposed then *c4* reduces to *c3*). Non-undefinedness of literals is ensured by *c5* or *c6* (condition *c5* is stronger than *c6* in the presence of the coherence principle.).

In the following we'll mainly make use of the explicit contradiction avoidance condition (*c4*), because the meaning of our programs will be given by the paraconsistent well-founded model, which already enforces coherence and *c3*.

7.2 Open Programs

Next we formally define the framework of open programs and program states mentioned at the beginning of this section.

The truth-varying basic beliefs are represented in an open program by a set of objective literals (the open literals) whose truth-values are not determined beforehand. The fixed part of an open program is an arbitrary extended logic program, which derives additional knowledge from the open literals once their value is known. Integrity constraints prune out unintended combinations of literals. Formally:

Definition 7.3 (Open Program) *An open program is a triple $\langle P, ICs, O_p\rangle$. The first component, P, is an extended logic program, the second one a set of integrity constraints as in definition 7.1, and the third a set of objective literals $O_p \subseteq \mathcal{H}(P)$ such that $L \in O_p$ iff $\neg L \in O_p$. O_p is the set of open literals. Additionally, there are no rules in P for any of the open literals.*

Given an open program, a state for it is established by introducing, for each open literal, a rule defining its truth value.

Definition 7.4 (Program State) *A program state of a given open program $\langle P_{Fix}, ICs, O_p\rangle$ is a tuple $\langle P_{Fix}, P_{Var}, ICs, O_p\rangle$. For each literal $L \in O_p$ the variable part, P_{Var}, contains just one of the rules $L \leftarrow \mathbf{t}$ or $L \leftarrow \mathbf{u}$ or $L \leftarrow \mathbf{f}$. Moreover, if $L \leftarrow \mathbf{t} \in P_{Var}$ then $\neg L \leftarrow \mathbf{f} \in P_{Var}$ (or if $\neg L \leftarrow \mathbf{t} \in P_{Var}$ then $L \leftarrow \mathbf{f} \in P_{Var}$). No other rules belong to P_{Var}.*

By definiton, the symbols $\mathbf{t}$ and *not* $\mathbf{f}$ belong to all models. Neither $\mathbf{u}$ nor *not* $\mathbf{u}$ belong to any model. The meaning of these symbols can be "programmed" with fact $\mathbf{t}$, the rule $\mathbf{u} \leftarrow not\ \mathbf{u}$ and the absence of any fact or rule for $\mathbf{f}$. In other words, by adding the rule $L \leftarrow \mathbf{t}$ (resp. $L \leftarrow \mathbf{u}$, $L \leftarrow \mathbf{f}$) literal L is assigned the truth-value *true* (resp. *undefined, false*).

Example 7.2 The program P:

$$\begin{array}{rclcrcl} a & \leftarrow & not\ c & \quad & c & \leftarrow & not\ d \\ \neg a & \leftarrow & not\ b & & d & & \end{array}$$

is contradictory. To revise it we first characterize it by the state $\sigma = \langle P_{Fix}, P_{Var}, ICs, O_p \rangle$, where:

$$\begin{array}{ll} P_{Fix} & = \{a \leftarrow not\ c;\ \neg a \leftarrow not\ b;\ c \leftarrow not\ d\} \\ P_{Var} & = \{b \leftarrow \mathbf{f};\ \neg b \leftarrow \mathbf{f};\ d \leftarrow \mathbf{t};\ \neg d \leftarrow \mathbf{f}\} \\ ICs & = \{\mathbf{f} \Leftarrow L, \neg L \mid L \in \mathcal{H}\} \\ O_p & = \{b, \neg b, d, \neg d\} \end{array}$$

To remove the contradiction we change the rules in P_{Var}.

A program state can be thought to correspond to an agent's set of beliefs about the "world". The basic beliefs are represented by the variable part of the program state. The fixed part allows the agent to express how to extract derived beliefs from the basic ones. The integrity theory states, directly or indirectly, which combinations of (possibly contradictory) beliefs are sustainable by the agent.

Note that it is trivial to make a fixed part rule dependent on a belief expressed by a variable part literal. For instance, as we'll see, in a diagnosis setting the variable part contains the non-abnormality and the normal and faulty behaviour modes. The fixed part describes the behaviour of components in correct or faulty modes, system topology, and the predicted "outcomes". The integrity theory can express some meta-knowledge about the system, e.g. that a component cannot be abnormal and correct simultaneously, that a "node" cannot have two "values", etc...

The variable component determines an interpretation, and vice-versa, as shown in table 2 for a pair of open literals $\{L, \neg L\}$. The coherence requirement is imposed by the additional proviso in definition 7.4 and also guarantees non-contradiction in P_{Var}. By convention, when writing the variable part we omit all rules of the form $L \leftarrow \mathbf{f}$, since the effect is the same.

Interpretation	Variable Part	Interpretation	Variable Part
$\{not\ L, not\ \neg L\}$	$\{L \leftarrow \mathbf{f}, \neg L \leftarrow \mathbf{f}\}$	$\{\}$	$\{L \leftarrow \mathbf{u}, \neg L \leftarrow \mathbf{u}\}$
$\{not\ L\}$	$\{L \leftarrow \mathbf{f}, \neg L \leftarrow \mathbf{u}\}$	$\{not\ L, \neg L\}$	$\{L \leftarrow \mathbf{f}, \neg L \leftarrow \mathbf{t}\}$
$\{not\ \neg L\}$	$\{L \leftarrow \mathbf{u}, \neg L \leftarrow \mathbf{f}\}$	$\{L, not\ \neg L\}$	$\{L \leftarrow \mathbf{t}, \neg L \leftarrow \mathbf{f}\}$

Table 2: Correspondence between interpretations and the variable part

To each program state we associate two interpretations. Let Ω denote an open program $\langle P_{Fix}, ICs, O_p \rangle$ and σ a program state $\langle P_{Fix}, P_{Var}, ICs, O_p \rangle$ of Ω. The model of Ω determined by σ, and denoted by $\mathcal{M}(\sigma)$, is $WFSX_p(P_{Fix} \cup P_{Var})$. The basic beliefs of σ, denoted by $\mathcal{B}(\sigma)$, are captured by the $WFSX_p$, i.e. $\mathcal{B}(\sigma) = \mathcal{M}(\sigma) \cup \mathcal{H}(P_{Var})$.

Definition 7.5 (Contradictory Program State) *A program state σ is contradictory iff $P_{Fix} \cup P_{Var}$ is contradictory or there exists at least one integrity constraint in σ which is violated by $\mathcal{M}(\sigma)$. Otherwise it is non-contradictory.*

Clearly, as argued at the beginning of this section, the sentence "$P_{Fix} \cup P_{Var}$ is contradictory" can be removed from definition 7.5 if ICs of the form $\mathbf{f} \Leftarrow L, \neg L$ are introduced for all L:

Proposition 7.1 *Let σ be a state of an open program $\langle P_{Fix}, ICs, O_p\rangle$ such that:*

$$ICs \supseteq \{\mathbf{f} \Leftarrow L, \neg L \mid L \in \mathcal{H}(P_{Fix})\}$$

Then, σ is contradictory iff there exists at least one integrity constraint in ICs which is violated by $\mathcal{M}(\sigma)$.

Example 7.2 (cont.) For the program state σ in example 7.2:

$$\begin{array}{ll} \mathcal{B}(\sigma) & = \{not\ b, not\ \neg b, d, not\ \neg d\} \\ \mathcal{M}(\sigma) & = \mathcal{B}(\sigma) \cup \{not\ c, not\ \neg c, a, \neg a, not\ a, not\ \neg a\} \end{array}$$

This state is contradictory because it violates the constraint $\mathbf{f} \Leftarrow a, \neg a$.

Note that open programs can be seen as a three-valued abductive framework for *WFSX*, along the lines of Generalized Stable Models [30] extension of Stable Model semantics [25].

7.3 Minimal Change

Contradictory states correspond to a situation where the basic beliefs of an agent are non-satisfactory. In such a situation the agent has to change its basic or revisable beliefs, so as to "reach" a new non-contradictory state. However this change is not arbitrary.

Because the agent starts by holding an (intrinsic) epistemic preference for the initial program state, intuitively it should change its beliefs in a minimal way with respect to it. The new program states (i.e. the new states of belief) should be as close as possible to the initial set of beliefs. The first issue is how to define this notion of "closeness" between two program states.

Given that notion of closeness we can define minimal revisions of a program state. We will only consider revisions of a single program state, thereof obtaining several new program states[17]. Thus, with the approach of this section, iteration of revisions is only possible after the agent epistemically chooses a unique program state from the set of revisions of another program state, and again starts from a single state, taken as the preferred one.

Our notion of closeness is an extension, to the three-valued case, of Winslett's difference between a pair of models of the Possible Model Approach (PMA) [68]. The difference between models is defined as the facts upon which a pair of models disagree. In the PMA the models which have minimal difference, i.e. minimal change, in the sense of set-inclusion are preferred.

Because we are considering a three-valued setting, a refined notion of difference is necessary. What is distinctive in our approach is that if a literal is not undefined then we prefer to revise it first to undefined before revising to the opposite truth-value; if it is undefined then we revise it either to true or false. This is mainly motivated by the truth and knowledge orders among literals[18]: moving from *true* to *false* or vice-versa, in either ordering, is always through *undefined*.

Definition 7.6 (Difference between interpretations) *Let $\mathcal{I}$ be the set of all coherent sets of literals wrt. to a finite language and $I_1, I_2 \in \mathcal{I}$ such that $I_1 = T_1 \cup not\ F_1$ and $I_2 = T_2 \cup not\ F_2$. The difference between interpretations I_1 and I_2 is defined by:*

$$Diff(I_1, I_2) = \left\{ \begin{array}{l} (F_2 - F_1) \times \{\mathbf{f}\} \cup \\ (F_1 - F_2) \times \{\mathbf{u}\} \cup \\ (T_1 - T_2) \times \{\mathbf{u}\} \cup \\ (T_2 - T_1) \times \{\mathbf{t}\} \end{array} \right.$$

Literals labeled with $\mathbf{t}$ (resp. $\mathbf{f}$) are those that change their truth-value to $\mathbf{t}$ (resp. $\mathbf{f}$), when going from I_1 to I_2. Literals labeled with $\mathbf{u}$ are those that change their original truth-value either to $\mathbf{u}$ or to their complementary truth-value. Some of these sets might not be disjoint.

[17] In order to iterate revisions we should also consider revisions of sets of program states. We leave that to future work.

[18] In the truth ordering $\mathbf{f} < \mathbf{u} < \mathbf{t}$, and in the knowledge one $\mathbf{u} < \mathbf{f}$ and $\mathbf{u} < \mathbf{t}$.

Example 7.3 Let $I_1 = \{a\} \cup not\ \{b, c, d\}$ and $I_2 = b, e \cup not\ d, f$, then

$$Diff(I_1, I_2) = \{(a, \mathbf{t}), (b, \mathbf{t}), (b, \mathbf{u}), (c, \mathbf{u}), (e, \mathbf{t}), (f, \mathbf{f})\}$$

For a more detailed, motivated, and theoretically based approach to the previous definition see subsection 7.6.

Definition 7.7 (Closeness relation) *Let $\mathcal{I}$ be the set of all interpretations wrt. a finite language, and M, A and $B \in \mathcal{I}$. We say that A is closer to M than B iff $Diff(M, A) \subset Diff(M, B)$.*

The closeness relation is defined between interpretations, not between program states. When program states are considered, one of the following two interpretations could be used: either the model determined by a program state, $\mathcal{M}(\sigma)$, or its associated beliefs, $\mathcal{B}(\sigma)$. We assume that what changes are the agent's basic beliefs, and that other changes of belief are just consequences of basic belief changes. Any belief can readily be made dependent on a basic one (c.f. 7.4.1). So, when facing contradiction, the agent should minimally change its basic beliefs, not necessarily what it extracts from those beliefs (i.e. their consequences). This has close connections with the belief base revision methods in the literature (see [43] and references therein).

Definition 7.8 (Revision of a program state) *Let σ be the initial program state of an open program Ω_P. A revision of σ is an element τ of Σ_P (the set of non-contradictory program states) such that:*

$$\forall_{\delta \in \Sigma_P}\ Diff(\mathcal{B}(\sigma), \mathcal{B}(\delta)) \subseteq Diff(\mathcal{B}(\sigma), \mathcal{B}(\tau)) \ \Rightarrow\ \delta = \tau$$

Notice that revision is defined here in terms of a single initial program state. Immediate extensions to more general cases can be defined as in [14]. There the authors define "local" and "global" belief revision semantics corresponding to, respectively, the notions of update and revision operators of [31]. In a local semantics one computes the revisions of each initial program state and the revision of the initial set is the set union of all these revisions. In a global semantics the preferred revisions are the ones with minimal difference with respect to the whole set of initial program states. For more details and comparisons see [14, 31].

Example 7.4 Consider the following interpretations corresponding to the variable part of four program states $\sigma_0, \ldots, \sigma_3$ of an open program:

$$\begin{array}{ll} I_0 = \{a\} \cup not\ \{\neg a, b, \neg c\} & I_1 = \{a, c\} \cup not\ \{\neg a, b, \neg c\} \\ I_2 = \{a, d\} \cup not\ \{\neg a, \neg c, \neg d\} & I_3 = \{a, c, d\} \cup not\ \{\neg a, \neg c, \neg d\} \end{array}$$

Suppose the agent believes in I_0 and that the corresponding program state (σ_0) is contradictory. Additionally, I_1, I_2 and I_3 are the unique beliefs that are consistent with the agent's current knowledge. Given the above interpretations I_1, I_2 and I_3, we've the following "differences" to I_0:

$$\begin{array}{l} Diff(I_0, I_1) = \{(c, \mathbf{t})\} \\ Diff(I_0, I_2) = \{(b, \mathbf{u}), (d, \mathbf{t}), (\neg d, \mathbf{f})\} \\ Diff(I_0, I_3) = \{(b, \mathbf{u}), (c, \mathbf{t}), (d, \mathbf{t}), (\neg d, \mathbf{f})\} \end{array}$$

The revisions of the initial program state σ_0 are $\{\sigma_1, \sigma_2\}$ because $Diff(I_0, I_1) \subseteq Diff(I_0, I_3)$ and $Diff(I_0, I_2) \subseteq Diff(I_0, I_3)$.

Example 7.2 (cont.) The revisions of state σ are the two states resulting from, in σ, replacing P_{Var} by either P^1_{Var} or P^2_{Var}, where:

$$\begin{array}{l} P^1_{Var} = \{b \leftarrow \mathbf{u};\ \neg b \leftarrow \mathbf{f};\ d \leftarrow \mathbf{t};\ \neg d \leftarrow \mathbf{f}\} \\ P^2_{Var} = \{b \leftarrow \mathbf{f};\ \neg b \leftarrow \mathbf{f};\ d \leftarrow \mathbf{u};\ \neg d \leftarrow \mathbf{f}\} \end{array}$$

In other words, contradiction is removed by either undefining b or d.

Suppose now that one is just interested in revisions of d to its default complement (i.e. revisions of d to undefined are not allowed). To that end we need only add the constraint $d \vee not\ d \Leftarrow \mathbf{t}$.

With this constraint, the revisions are the two states resulting from replacing P_{Var} by either P^1_{Var} or P^3_{Var}, where:

$$P^3_{Var} = \{b \leftarrow \mathbf{f};\ \neg b \leftarrow \mathbf{f};\ d \leftarrow \mathbf{f};\ \neg d \leftarrow \mathbf{f}\}$$

i.e., the contradiction is removed by either undefining b or by making d false. □

We end this subsection by showing the of ours relationship to Winslett's closeness relation. We show that when we have two-valued interpretations our definition of $Diff$ is equivalent to Winslett's. Noting that given two arbitrary sets X and Y the sets $X - Y$ and $Y - X$ are disjoint, the condition $Diff(M, B) \subseteq Diff(M, A)$ in theorem 7.5 is equivalent to:

$$\begin{array}{rcll} (T_B - T_M) \cup (T_M - T_B) & \subseteq & (T_A - T_M) \cup (T_M - T_A) & \text{and} \\ (F_B - F_M) \cup (F_M - F_B) & \subseteq & (F_A - F_M) \cup (F_M - F_A) & \end{array}$$

If only two-valued interpretations are considered then:

$$\begin{array}{rcll} (T_B - T_M) \cup (T_M - T_B) & = & (F_B - F_M) \cup (F_M - F_B) & \text{and} \\ (T_A - T_M) \cup (T_M - T_A) & = & (F_A - F_M) \cup (F_M - F_A) & \end{array}$$

and therefore $Diff(M, B) \subseteq Diff(M, A)$ iff $(T_B - T_M) \cup (T_M - T_B) \subseteq (T_A - T_M) \cup (T_M - T_A)$. This corresponds to Winslett's [68] notion of closeness between two interpretations (possible worlds).

Also note that if our initial program state has all literals false, minimality by set inclusion is obtained. This shows the relationship to our two-valued contradiction removal methods of [50].

7.4 Application Examples

We now present two examples of application of the revision techniques. First, we show for the first time how to achieve three-valued declarative debugging, and next the benefits of using mixed two- and three-valued revisions in model based diagnosis applications.

7.4.1 Declarative Debugging Example

Consider the following logic program:

$$\begin{array}{llll} a \leftarrow not\ b & \qquad & b \leftarrow c \\ c \leftarrow b, d & \qquad & c \leftarrow not\ c \end{array}$$

whose well-founded model is $\{not\ d, not\ \neg a, not\ \neg b, not\ \neg c, not\ \neg d\}$

In order to perform debugging we transform this program into an open one in a straightforward way, by explicitly assuming each rule not incorrect and that each predicate covers intended calls to it. The open literals are the inc (for *incorrect*), and unc (for *uncovered* literals [40] and their explicit complements (cf. [49] for a detailed justification of this program rendering, and non-propositional examples). The fixed part is:

$$\begin{array}{llll} a \leftarrow not\ b, not\ inc(a/1) & \qquad & a \leftarrow unc(a) \\ b \leftarrow c, not\ inc(b/1) & \qquad & b \leftarrow unc(b) \\ c \leftarrow b, d, not\ inc(c/1) & \qquad & c \leftarrow unc(c) \\ c \leftarrow not\ c, not\ inc(c/2) & \qquad & d \leftarrow unc(d) \\ & & \\ \neg a \leftarrow unc(\neg a) & \qquad & \neg b \leftarrow unc(\neg b) \\ \neg c \leftarrow unc(\neg d) & \qquad & \neg c \leftarrow unc(\neg d) \end{array}$$

Suppose now the intended model of the program is $\{b, d\} \cup not\ \{c, \neg a, \neg b, \neg c, \neg d\}$. Our integrity theory expresses this information as:

$$
\begin{array}{lll}
\mathbf{f} \Leftarrow not\ a & not\ c \Leftarrow \mathbf{t} & not\ \neg a \Leftarrow \mathbf{t} \\
\mathbf{f} \Leftarrow a & d \Leftarrow \mathbf{t} & not\ \neg b \Leftarrow \mathbf{t} \\
b \Leftarrow \mathbf{t} & not\ \neg c \Leftarrow \mathbf{t} & not\ \neg d \Leftarrow \mathbf{t}
\end{array}
$$

$$
\mathbf{f} \Leftarrow a \wedge \neg a \qquad \mathbf{f} \Leftarrow b \wedge \neg b \qquad \mathbf{f} \Leftarrow c \wedge \neg c \qquad \mathbf{f} \Leftarrow d \wedge \neg d
$$

The number of possible program states of this open program is exactly 2176782336!!! We consider our initial set of beliefs to have all open literals false, making the initial program state equivalent to the original program. This associated program state is contradictory, and therefore we must perform a revision to regain consistency. We obtain a unique revision with variable part:

$$
P_{Var} = \{unc(a) \leftarrow \mathbf{u}, unc(b) \leftarrow \mathbf{t}, inc(c/1) \leftarrow \mathbf{t}, inc(c/2) \leftarrow \mathbf{t}, unc(d) \leftarrow \mathbf{t}\} \tag{2}
$$

The remaining literals in P_{Var} are all false. The revision "returns" the bugs of the initial program. The diagnosis is that there are rules missing for literals b and d, to make them true as desired; then a rule for keeping a undefined is lacking given the newly imposed truth of b; and both rules for c are incorrect.

The existence of a single revision is justified by the fact that the intended model is completely known. When this happens debugging ensures that all bugs become known. If only part of the intended model is known or specified then several revisions may exist. For instance, if the truth-value of literal c is not given then two minimal revisions exist:

$$
\begin{array}{l}
P^1_{Var} = \{unc(a) \leftarrow \mathbf{u}, unc(b) \leftarrow \mathbf{t}, unc(d) \leftarrow \mathbf{t}\} \\
P^2_{Var} = \{unc(a) \leftarrow \mathbf{u}, unc(c) \leftarrow \mathbf{t}, unc(d) \leftarrow \mathbf{t}\}
\end{array}
$$

Notice that the first revision is a subset of (2). Intuitively, the above two revisions cover all the possible buggy situations dependent on the truth-value of c.

7.4.2 Model Based Diagnosis Example

Consider the anomalous situation of the four inverter circuit of figure 1:

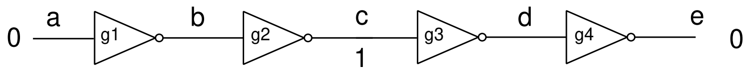

Figure 1: Four inverter circuit

The normal and abnormal behaviour of the inverter gate is modeled by the rules below. We assume that our first two inverter gates, of type 1, have two known modes of erroneous behaviour, either the output is always "0" (mode "stuck at 0") or is always "1" (mode "stuck at 1"). It is also implicit that any abnormality is permanent and not intermitent. The second type of gate, to which belong the other two, has no specified model of faulty behaviour. We begin by defining the fixed program part:

$$
\begin{array}{rcl}
inv1(G, I, 1) & \leftarrow & not\ ab(G), node(I, 0) \\
inv1(G, I, 0) & \leftarrow & not\ ab(G), node(I, 1) \\
inv2(G, I, 1) & \leftarrow & not\ ab(G), node(I, 0) \\
inv2(G, I, 0) & \leftarrow & not\ ab(G), node(I, 1) \\
\\
inv1(G, \_, 0) & \leftarrow & ab(G), s\_at\_0(G) \\
inv1(G, \_, 1) & \leftarrow & ab(G), s\_at\_1(G) \\
\\
s\_at\_0(G) & \leftarrow & fault\_mode(G, s0) \\
s\_at\_1(G) & \leftarrow & fault\_mode(G, s1)
\end{array}
$$

The value at a node is either predicted or observed:

$$\begin{array}{rcl} node(N,V) & \leftarrow & predicted(N,V) \\ node(N,V) & \leftarrow & observed(N,V) \end{array}$$

The connections among components and nodes are described by:

$$\begin{array}{rclrcl} predicted(b,B) & \leftarrow & inv1(g1,a,B) & predicted(c,C) & \leftarrow & inv1(g2,b,C) \\ predicted(d,D) & \leftarrow & inv2(g3,c,D) & predicted(e,E) & \leftarrow & inv2(g4,d,E) \end{array}$$

The first integrity rule below ensures that the fault modes are exclusive, i.e. to an abnormal gate at most one fault mode can be assigned simultaneously. The second integrity rule expresses that, for each node, a single value can be predicted or observed at any one time.

$$\begin{array}{rcl} \mathbf{f} & \Leftarrow & fault\_mode(G,S_1) \wedge fault\_mode(G,S_2) \wedge S_1 \neq S_2 \\ \mathbf{f} & \Leftarrow & node(N,V_1) \wedge node(N,V_2) \wedge V_1 \neq V_2 \end{array}$$

Our open literals are of the form $ab(G)$ and $fault\_mode(G,M)$ and their explicit complements. In the initial program state all open literals are taken to be false. Suppose we make the first measurements in nodes a and c and obtain a 0 and a 1, respectively. The values of the nodes are described in the program by the new facts $observed(a,0)$ and $observed(c,1)$. Now the program is in a contradictory program state (because it predicts for node c value "0"), and therefore should be revised. There are just two revisions according to wheter either $ab(g1)$ or $ab(g2)$ have value *undefined*. Thus the possible causes of malfunction are due to gate 1 or gate 2.

If an additional observation is made at node e, modeled by adding the fact $observed(e,0)$, two more revisions are obtained (from the initial variable program state) with either $ab(g3)$ and $ab(g4)$ *undefined*.

Till now we've only imposed consistency with the observations, and so revising the ab literals is sufficient to regain consistency. If we further want a revision to explain how the circuit model can predict every observed wrong output, a new integrity constraint is added to the open program:

$$predicted(N,V) \Leftarrow observed(N,V) \quad (I1)$$

But now any state of the program is contradictory because inverters of type 2 cannot have thier wrong outputs explained: their fault models are missing (in the general case fault models may be incomplete). If we only enforce the condition above on node c, i.e. replacing (I1) by

$$predicted(c,V) \Leftarrow observed(c,V)$$

the existing revisions are[19]:

$$\begin{array}{c} \{ab(g1) \leftarrow \mathbf{t}, fault\_mode(g1,s0) \leftarrow \mathbf{t}, ab(g3) \leftarrow \mathbf{u}\} \\ \{ab(g1) \leftarrow \mathbf{t}, fault\_mode(g1,s0) \leftarrow \mathbf{t}, ab(g4) \leftarrow \mathbf{u}\} \\ \{ab(g2) \leftarrow \mathbf{t}, fault\_mode(g2,s1) \leftarrow \mathbf{t}, ab(g3) \leftarrow \mathbf{u}\} \\ \{ab(g2) \leftarrow \mathbf{t}, fault\_mode(g2,s1) \leftarrow \mathbf{t}, ab(g4) \leftarrow \mathbf{u}\} \end{array}$$

Note that these revisions, containing undefined literals, can be seen as approximations to the two–valued case. Hence the use of the "undefined" value, in order to make it possible to work with incomplete models. Also mark that to impose that literals be predicted is computationally more expensive, due to the necessity of using alternative fault models, which in real applications can be very complex.

Another use of a mixed two- and three-valued revision is that we might want to delve into the fault model of one part of the system so as to explain observations, whilst simply requiring consistency from another part in relation to which we don't want to delve deeper. One can thereby decrease the computational effort required for such a more focused analysis.

[19] Recall that all other literals are false.

In general there may exist an exponential number of revisions, which is inherent to the complexity of the type of problem being solved. In order to cope with this complexity, more powerful techniques like preferences, abstractions and refinement strategies are needed [18]. The use of additional observations and measurements also improve the diagnoses obtained. These avenues of research are still being pursued at present.

7.5 Computing Revisions

We have devised and implemented an algorithm to compute revisions[20], based on the SLX procedure with ancestors. The algorithm uses two functions, insert and delete, each of which returns a first-order sub-formula. The syntax is $del(L, Cx, AnsL, AnsG)$ and $ins(L, Cx, AnsL, AnsG)$.

The topmost first-order formula expresses the conditions on the open literals which guarantee a non-contradictory state. The basic conditions are tests on the verity and on the non-falsity of open literals, used to construct the formula. Finding these is achieved by a top-down procedure in all respects similar to SLX plus that it collects all the relevant conditions on open literals, which are then simplified.

The deletion function is defined in terms of the insertion one as $del(L, Cx, AnsL, AnsG) = \sim ins(L, Cx, AnsL, AnsG)$. The first argument of ins is a literal, the second a flag of the form t or tu, and the other two the sets of ancestors necessary to guarantee termination of cyclic derivations.

The insertion function determines the necessary and sufficient conditions to enforce a literal true (if $Cx = t$) or to enforce its non-falsity (if $Cx = tu$). The deletion function in turn guarantees non-verity of a literal (if $Cx = t$) or its falsity (if $Cx = tu$). Consider again example 7.2:

Example 7.2 (cont.) Take the constraint $\mathbf{f} \Leftarrow a \wedge \neg a$. The first necessary step for finding the conditions on open literals which satisfy the integrity constraint is to reformulate it using the deletion and insertion functions. Namely the constraint is satisfied iff $del(a, t, \{\}, \{\}) \vee del(\neg a, t, \{\}, \{\})$, i.e. if a or $\neg a$ are not true. By applying the revision algorithm we obtain the following formula:

$$d \notin \mathcal{I} \vee (\neg b \notin \mathcal{I} \wedge not\ b \notin \mathcal{I})$$

Note that the two revisions in page 23 are the ones that satisfy this condition.

The above does not describe a complete algorithm to compute closest revisions. It only provide the first step (albeit the most important) of revision generation. To have a full revision system it is necessary to equip it with a method for formula minimization, so as to get only closest revisions.

7.6 Distance and Closeness Relations

We present in this section our theoretical results justifying the use of closeness relation definition 7.7 to guide the revision process. It is not essential for the sequel. We first define a natural metric between program states, obtained from the classic and Fitting orderings among three-valued interpretations. Afterwards, we point some problems of a closeness relation based on minimal distance according to the said metric, and provide an alternative definition of closeness. We characterize it and present some of its properties.

The classic and Fitting orderings among three-valued interpretations are the generalization to sets of literals of the truth and knowledge orders among literals. Roughly, the classic ordering says that interpretations with less degree of truth are preferred, and the Fitting ordering amounts to prefer minimizing information (i.e. maximizing the degree of undefinedness). If not explicitly stated, we assume interpretations are always given with respect to some language $\mathcal{L}$. Formally, we have:

Definition 7.9 (Classical and Fitting Orderings) *If $I_1 = T_1 \cup not\ F_1$ and $I_2 = T_2 \cup not\ F_2$ are two interpretations, then we say that:*

- $I_1 \preceq_C I_2$ *iff $T_1 \subseteq T_2$ and $F_2 \subseteq F_1$ (classic "truth" ordering);*

[20] For lack of space these are not completely described here, but the implementation is available on request.

- *$I_1 \preceq_F I_2$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$ (Fitting "knowledge" ordering);*

Using these two partial orders among interpretations we proceed to define our "distance unit." Intuitively, given a finite partial order, the nearest elements to an arbitrary element of the poset of interpretations (i.e. at a minimum distance) are its cover and covering elements (wrt. to the given partial order).

Definition 7.10 (Nearest elements) *Let $(\mathcal{M}, \preceq)$ be a finite poset. The set of nearest elements of a given $A \in \mathcal{M}$, denoted by $A^{\odot}_{\preceq}$, is defined by:*

$$A^{\odot}_{\preceq} =_{def} \{B \in \mathcal{M} \mid (B \prec A \wedge \nexists_C \ B \prec C \prec A) \vee (A \prec B \wedge \nexists_C \ A \prec C \prec B)\}$$

Now, we can use the classic or Fitting orderings to determine the nearest interpretations ($I^{\odot}_{\preceq_C}$ and $I^{\odot}_{\preceq_F}$) to some interpretation I. The following result strongly supports our notion of nearest elements. The two partial orders (and also their dual relations) have the same set of nearest elements:

Theorem 7.1 *Let $I = T \cup not\ F$ be a finite three-valued interpretation wrt. to some language $\mathcal{L}$[21]:*

$$\begin{array}{lcl} I^{\odot}_{\preceq_C} = I^{\odot}_{\preceq_F} & = & \{ \ \ \{ (T \cup \{L\}) \cup not\ F \mid L \notin F \wedge L \notin T \wedge \neg L \in F \ \} \cup \\ & & \quad \{ T \cup not\ (F \cup \{L\}) \mid L \notin F \wedge L \notin T \ \} \cup \\ & & \quad \{ (T - \{L\}) \cup not\ F \mid L \in T \ \} \cup \\ & & \quad \{ T \cup not\ (F - \{L\}) \mid L \in F \wedge \neg L \notin T \ \} \mid L \in \mathcal{L}\} \end{array}$$

Given this characterization of nearest elements, we can next define the distance between two interpretations by counting the minimal number of steps necessary to go from one interpretation to the other. A "step" consists in moving from one element to one of its nearest ones. Before we do that we must enforce another condition on the poset (besides being finite): the Hasse diagram of the poset must be connected. Posets obeying the previous condition are said connected. Intuitively, this ensures it is always possible to go from one place to another in the poset. Notice that a poset having a bottom or a top element automatically verifies the "connectivity" condition. The definition of distance follows directly from the above characterization of nearest elements:

Definition 7.11 (Distance) *Let $(\mathcal{M}, \preceq)$ be a finite connected poset. The elements reachable from $x \in \mathcal{M}$ in n steps are defined recursively as follows:*

$$\begin{array}{l} \left(x^{\odot}_{\preceq}\right) \uparrow^{0} = \{x\} \\ \left(x^{\odot}_{\preceq}\right) \uparrow^{(n+1)} = \bigcup \left\{ y^{\odot}_{\preceq} \mid y \in \left(x^{\odot}_{\preceq}\right) \uparrow^{n} \right\} \end{array}$$

The distance between two elements $x, y \in \mathcal{M}$ is given by:

$$d_{\preceq}(x, y) = \min\ i \mid \left\{ y \in \left(x^{\odot}_{\preceq}\right) \uparrow^{i} \right\}$$

The next theorem shows that this notion of distance has the required properties:

Theorem 7.2 *Let $(\mathcal{M}, \preceq)$ be a connected poset; then $(\mathcal{M}, d_{\preceq})$ is a metric space, i.e., $d_{\preceq}$ satisfies, for any $x, y, z \in \mathcal{M}$, the following properties required of a distance:*

d1) $d_{\preceq}(x, x) = 0$;

d2) *If* $x \neq y$ *then* $d_{\preceq}(x, y) > 0$;

d3) $d_{\preceq}(x, y) = d_{\preceq}(y, x)$;

[21] Proof of this and the main theorems in this section are presented in appendix B.

d4) $d_{\preceq}(x,y) \leq d_{\preceq}(x,z) + d_{\preceq}(z,y)$.

Now we can use the above results to characterize the distance between interpretations when the classic or Fitting orderings are used to "sort" them. Note that this result is also valid if the duals of the former orderings are used, since the nearest worlds are the same for the four partial orders among interpretations.

Theorem 7.3 *Let $\mathcal{I}$ be the set of all three-valued coherent interpretations wrt. to a finite language and $x, y \in \mathcal{I}$ such that $x = T_1 \cup not\ F_1$ and $y = T_2 \cup not\ F_2$. Then $d_{\preceq_C}(x,y) = d_{\preceq_F}(x,y) = dist(x,y)$, where*

$$dist(x,y) = \#(T_1 - T_2) + \#(T_2 - T_1) + \#(F_1 - F_2) + \#(F_2 - F_1)$$

i.e. $dist(x,y) = \#Diff(x,y)$.

Definition 7.12 (Distance between program states) *Let σ_1 and σ_2 be two program states of an open program Ω_P. The distance between σ_1 and σ_2, denoted by $dstate(\sigma_1,\sigma_2)$ is equal to $dist(\mathcal{B}(\sigma_1), \mathcal{B}(\sigma_2))$.*

When we restrict to two-valued interpretations, our definition of distance is tantamount to extension of Dalal's [17]. Dalal measures distance between worlds (i.e. models) by counting the number of propositions that differ in their truth-values. It can be easily seen that $dist(x,y) = 4 \times d_{\mathrm{Dalal}}(x,y)$. This results in part from the fact that to change a literal to its complement truth-value costs 2. But when we change an objective literal to its opposite truth-value coherence imposes that we should also change its explicit negation complement, which again costs 2. But in general our distance function is more refined than Dalal's, because of the extra truth-value and the explicitly negated literals.

Given this distance relation between states, we could now define our notion of "closeness" to a contradictory state like in [17], by selecting as close those states that are at a minimum distance from the current one. It is known from the literature [14] that the use of this closeness relation as the basis for state revision preference has some problems, exemplified in the following, relative to accumulated distances of successive state changes.

Example 7.5 Consider the following two-valued interpretations, corresponding to the variable part of four program states $\sigma_0 \ldots \sigma_3$ of an open program:

$$\begin{array}{ll} I_0 = \{\neg a, \neg b, \neg c\} \cup not\ \{a,b,c\} & I_1 = \{\neg a, \neg b, c\} \cup not\ \{a,b,\neg c\} \\ I_2 = \{a,b,\neg c\} \cup not\ \{\neg a,\neg b,c\} & I_3 = \{a,b,c\} \cup not\ \{\neg a,\neg b,\neg c\} \end{array}$$

Suppose the agent believes in I_0 initially and that the corresponding program state (σ_0) is contradictory. Additionally, I_1, I_2 and I_3 are the unique beliefs that are consistent with the agent's current knowledge. If minimal distance is used the agent will change its beliefs to σ_1 because $dist(I_0,I_1) = 4$, $dist(I_0,I_2) = 8$ and $dist(I_0,I_3) = 12$. Later, imagine the agent finds out that σ_1 too is not sustainable because of, say, some introduced integrity constraint. Therefore, it will change again its beliefs to σ_3 because $dist(I_1,I_2) = 12$ and $dist(I_1,I_3) = 8$. The agent will have changed to a farther "world", σ_3, with respect to the initial one σ_0 (see figure 2). This example functions also as a counter-argument to the use of minimal distance even with Dalal's definitions.

Since the change from one state to another is spurred by the violation of a program's integrity theory, it is important to guarantee that the order in which the integrity constraints are considered is immaterial or commutative with regard to obtaining the same set of equally close final revised states [22]

Because the diagram contains all states, any edge can be implicitly expressed by an integrity constraint and vice-versa. The successive satisfaction of integrity constraints can be seen as a path such that any rules added for open literals must be kept along the way, to ensure continued satisfaction of the already introduced constraints.

[22] In terms of the AGM [52] revision operator this means that the natural property $K \dot{+} (\phi \wedge \psi) = (K \dot{+} \phi) \dot{+} \psi = (K \dot{+} \psi) \dot{+} \phi$ is not verified.

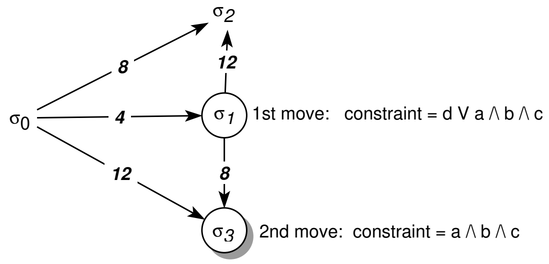

Figure 2: Minimum distance counter-argument example

Thus, on account of these problems with prefering revised states at a minimal distance, which is sensitive to program change order, we need another notion of closeness. In the example, intuitively σ_2 should also be at the same degree of closeness from σ_0 as σ_1 so that it is possible to go there, whatever the order the ICs are taken in.

Suppose we start from an initial state σ and add an arbitrary integrity constraint. The integrity constraint determines univocally a set of program states $\mathcal{C}$. If an additional constraint is enforced a subset $\mathcal{C}_\infty$ of $\mathcal{C}$ would be obtained. Firstly, the set of closest elements of $\mathcal{C}$ must be such that it contains the elements at minimal distance of σ. Furthermore, it should contain those elements in $\mathcal{C}$ guaranteeing that, if we change to $\mathcal{C}_\infty$, then for every state σ_1 at minimal distance of the initial program state σ there exists a state in $\mathcal{C}$ at minimal distance of σ_1. Therefore we don't loose any program state irrespective of the order of treatment of constraints. This motivates the definition:

Definition 7.13 (Closeness within $\mathcal{C}$) *Given a finite metric space $(\mathcal{M}, d_\preceq)$, let $M \in \mathcal{M}$ and $\mathcal{C} \subseteq \mathcal{M}$.*

$$Close_{d_\preceq}(M, \mathcal{C}) = \{A \in \mathcal{C} \mid \forall_{B \in \mathcal{C}} (d_\preceq(M, A) = d_\preceq(M, B) + d_\preceq(B, A) \Rightarrow A = B)\}$$

A geometric analogy is given in figure 3. To find a closest element A we draw a line between the initial element and an arbitrary element B of $\mathcal{C}$. The closest element will be the one nearer to the initial program state M.

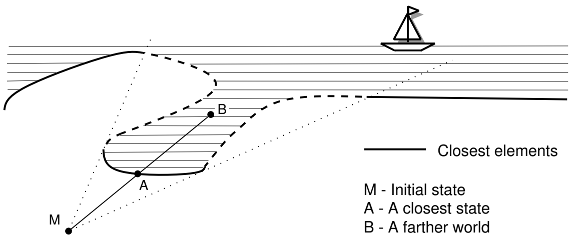

Figure 3: Closeness relation

The following property ensures what we want. Roughly, it says that all original distances of elements in $\mathcal{C}$ are "preserved" in the set of closest elements.

Theorem 7.4 *Given a finite metric space $(\mathcal{M}, d_\preceq)$, let $M \in \mathcal{M}$ and $\mathcal{C} \subseteq \mathcal{M}$, then:*

$$\forall_{D \in \mathcal{C}}\ \exists_{A \in Close_{d_\preceq}(M, \mathcal{C})}\ d_\preceq(M, D) = d_\preceq(M, A) + d_\preceq(A, D)$$

Obviously, the metric space considered in *Close* above is the set of all program states with metric *dstate*. Now we proceed to equivalently characterize the set of revisions in terms of *Diff*. The extension to program states follows immediately.

Theorem 7.5 *Let $\mathcal{I}$ be the set of all interpretations wrt. to a finite language, $M \in \mathcal{I}$ and $\mathcal{C} \subseteq I$. Then,*

$$A \in Close_{dist}(M, \mathcal{C}) \equiv \forall_{B \in \mathcal{C}}\ Diff(M, B) \subseteq Diff(M, A) \ \Rightarrow\ A = B$$

The above theorems states that the interpretations closer to M are the ones with minimal difference in the sense of set inclusion, as we've defined. Therefore, the revisions of a program state are those program states for which the beliefs differ minimally (i.e. with minimal $Diff$) from the current ones. When $\mathcal{C} = \{\mathcal{B}\left(()\tau\right) : \tau \in \Sigma_P\}$ and $M = \mathcal{B}\left(()\sigma\right)$ we obtain definition 7.8.

Finally, we should point out that the minimum distance approach has the advantage of being computationally less expensive than the adopted closeness relation. In fact, the revision with minimum distance approach is polynomially bounded, in contrast to the exponential complexity of the minimal $Diff$ revision method.

8 Conclusions, Discussion, and Future work

The widespread use of the richer representation language of extended logic programs and its applications requires the definition of a correct top-down querying mechanism, much as for Prolog wrt. to normal programs. In this paper we've presented and exploited a SLDNF-like derivation procedure, SLX, for extended programs under *WFSX* and proven its soundness and completeness.

The introduction of explicit negation required us to deal with contradiction. We showed how it can be removed by freely changing the truth-values of some subset of a set of predefined revisable literals. To achieve this, we first introduced a paraconsistent version of *WFSX*, $WFSX_p$, that allows contradictions to appear, and for which our SLX top-down querying procedure was proven correct as well.

SLX was then used to detect the existence of pairs of complementary literals in $WFSX_p$ simply by detecting the violation of integrity rules $\mathbf{f} \Leftarrow L, \neg L$ introduced for each L in the language of the program. Integrity constraints of a general form were allowed, whose violation is likewise detected by SLX.

Contradiction removal is accomplished by a variant of SLX which collects, in a formula, the alternative combinations of revisable literals' truth-values that ensure the said removal. A notion of minimal change was defined as well, which establishes a closeness relation between a program and its revisions. Changes are enforced by introducing or deleting program rules for the revisable literals only.

To illustrate the usefulness and originality of our framework we applied it to obtain a novel logic programming approach, and results, to declarative debugging and model based diagnosis.

One point of discussion that might be raised is that SLX is applicable only to (infinite) ground programs. Moreover, for these programs, the computational complexity of SLX can be greater than that of the iterative bottom-up definition of *WFSX*. So why would one prefer SLX?

The reasons for prefering SLX to the bottom-up definition of *WFSX* are tantamount to those for usually prefering Prolog to bottom-up procedures of normal programs. Note that the problem of complexity also occurs when comparing, in the ground case, the complexity of SLD resolution to that of the T_P bottom-up operator.

Moreover, specially in non-ground programs, users normally wish to know the instances of a literal that belong to the semantics rather than the whole semantics.

But SLX, as it is, can also be applied to non-ground programs, provided they are allowed and bounded-term. Moreover, we intend to extend SLX to deal with (non-allowed) non-ground programs.

A straightforward generalization method for non-ground programs would be to proceed as usual in the expansion of goals with rule variants, and keeping the test for inclusion in the ancestors lists. However it has two problems: first, as shown in [8], this loop detection method does not guarantee termination in the nonground case, even for (non-allowed) term-bounded programs; second, the procedure flounders on non-ground default literals.

To guarantee termination for non-ground term-bounded programs, we intend to introduce tabulation methods into SLX[23]. This will also decrease the computational complexity of the procedure.

Such a modification differs significantly from existing procedures for WFS that use tabulation (e.g. [10, 12]). On the one hand, SLX is applicable to programs with explicit negation whilst others aren't; on the other hand, even for normal programs, SLX does not need a status distinct from successful and failed, as it is based on two types of proof, whilst others include a status unknown as well, because based on a single proof type, which complicates the procedure. Another subject of future work is that of introducing in SLX constructive negation techniques for solving the floundering problem.

One further point of discussion is whether there might be some program transformation, from ELPs into normal programs, such that available top-down procedures for WFS could be then applied to obtain an equivalent semantics to that of *WFSX*, where if the program is contradictory this must be detected. We have one such transformation, but even so, the very fact that the resulting program has a specific form makes any general WFS procedure less than optimal because the specifity would not be exploited (in particular, programs double in size and derivations are repeated). In contradistinction, our SLX procedure takes into account the WFSX particulars. In any case, in our paper [2], we compare our approach to other WFS top-down procedures, for the case of normal programs.

Another possible generalization of this work is to introduce program rules with disjunctive heads. Over our semantics several approaches to disjunction in program rules might be constructed. We have not yet adopted any one approach because the ongoing research on disjunction for logic programs is still stabilizing, though we favour one similar to that of [56]. One problem is that none of the proposals to this date include explicit negation as we define it. Another is that contradiction removal methods when disjunction is involved have yet to be devised and given a semantics. We are working towards a satisfactory solution to these issues. Till one is found it would premature to incorporate fully fledged disjunction. For the moment though, our integrity constraints can capture the intended effect of many uses of disjunction.

Finally, it is our intention to proceed with the study of more efficient implementations of SLX and revision procedures. Indeed, this is the subject of a national project in cooperation with LIACC in Porto, within which we will implement our procedures at abstract machine level using sidetracking (i.e. deterministic call priority), parallelism, and low-level support for tabulation.

References

[1] J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *ILPS*. MIT Press, 1994. To appear.

[2] J. J. Alferes, C. V. Damásio, and L. M. Pereira. Top-down query evaluation for well-founded semantics with explicit negation. In A. Cohn, editor, *Proc. ECAI'94*, pages 140–144. Morgan Kaufmann, 1994.

[3] J. J. Alferes and L. M. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 574–588. MIT Press, 1992.

[4] J. J. Alferes and L. M. Pereira. Belief, provability, and logic programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA'94*, volume 838 of *LNAI*, pages 106–121. Springer-Verlag, 1994.

[5] J. J. Alferes and L. M. Pereira. Contradiction: when avoidance equal removal. In R. Dyckhoff, editor, *4th Int. Ws. on Extensions of LP*, volume 798 of *LNAI*. Springer-Verlag, 1994.

[6] José Júlio Alferes. *Semantics of Logic Programs with Explicit Negation.* PhD thesis, Universidade Nova de Lisboa, October 1993.

[7] K. Apt and R. Bol. Logic programming and negation: a survey. *J. Logic Programming*, 19/20:9–72, 1994.

[23] Note that, being cumulative, *WFSX* is amenable to such methods.

[8] K. Apt, R. Bol, and J. Klop. On the safe termination of Prolog programs. In Levi and Marteli, editors, *Proc. ICLP'89*, pages 353–368. MIT Press, 1989.

[9] C. Baral and M. Gelfond. Logic programming and knowledge representation. *J. Logic Programming*, 19/20:73–148, 1994.

[10] N. Bidoit and P. Legay. Well!: An evaluation procedure for all logic programs. In *Int. Conf. on Database Technology*, pages 335–348, 1990.

[11] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *Proc. ILPS'93*. MIT Press, 1993.

[12] W. Chen and D. S. Warren. A goal–oriented approach to computing well–founded semantics. In K. Apt, editor, *Int.Joint Conf. on Logic Programming*, pages 589–603. MIT Press, 1992.

[13] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *PODS'93*, 1993.

[14] T. S.-C. Chou and M. Winslett. A Model-Based Belief Revision System. *Journal of Automated Reasoning*, 12(2):157–208, April 1994.

[15] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[16] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un systéme de communication homme–machine en français. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix–Marseille II, 1973.

[17] M. Dalal. Investigations into a theory of knowledge base revision: Preliminary report. In *7th AAAI*, pages 475–479, 1988.

[18] C. V. Damásio, W. Nejdl, and L. M. Pereira. REVISE: An extended logic programming system for revising knowledge bases. In *KR'94*. Morgan Kaufmann, 1994.

[19] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 686–700. MIT Press, 1992.

[20] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 166–180. MIT Press, 1991.

[21] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan Kaufmann, 1992.

[22] P. M. Dung. Negation as hypotheses: An abductive framework for logic programming. In K. Furukawa, editor, *8th Int. Conf. on LP*, pages 3–17. MIT Press, 1991.

[23] K. Eshghi and R. Kowalski. Abduction compared with negation by failure. In *6th Int. Conf. on LP*. MIT Press, 1989.

[24] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[25] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th Int. Conf. on LP*, pages 1070–1080. MIT Press, 1988.

[26] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In M. Reinfrank, J. de Kleer, M. Ginsberg, and E. Sandewall, editors, *2nd Int. Ws.on NMR*, pages 74–99. LNAI 346, Springer–Verlag, 1989.

[27] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 579–597. MIT Press, 1990.

[28] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 559–573. MIT Press, 1992.

[29] HMSO. *British Nationality Act*. Her Majesty's Stationery Office, 1981.

[30] A. C. Kakas and P. Mancarella. Generalized stable models: A semantics for abduction. In *Proc. ECAI'90*, pages 401–405, 1990.

[31] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base revising it. In J. A. Allen et al, editor, *2nd KRR*, pages 387–394. Morgan Kaufmann, 1991.

[32] D. B. Kemp, P. J. Stuckey, and D. Srivastava. Query Restricted Bottom–up Evaluation of Normal Logic Programs. In *Proc. JICSLP'92*, pages 288–302. MIT Press, 1992.

[33] R. Kowalski. *Logic for Problem Solving.* Noth Holland, 1978.

[34] R. Kowalski. The treatment of negation in logic programs for representing legislation. In *2nd Int. Conf. on AI and Law*, pages 11–15, 1989.

[35] R. Kowalski. Problems and promises of computational logic. In John Lloyd, editor, *Computational Logic*, pages 1–36. Basic Research Series, Springer–Verlag, 1990.

[36] R. Kowalski. Legislation as logic programs. In *Logic Programming in Action*, pages 203–230. Springer–Verlag, 1992.

[37] R. Kowalski and D. Khuener. Linear resolution with selection function. *Artificial Intelligence*, 5:227–260, 1971.

[38] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.

[39] K. Kunen. Negation in logic programming. *Journal of LP*, 4:289–308, 1987.

[40] J. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2), 1987.

[41] J. Lloyd. *Foundations of Logic Programming.* Springer–Verlag, 1987.

[42] J. Lloyd and R. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2:93–109, 1985.

[43] B. Nebel. Syntax based approaches to belief revision. In P. Gärdenfors, editor, *Belief Revision*, pages 52–88. Cambridge University Press, 1992.

[44] D. Pearce and G. Wagner. Reasoning with negative information I: Strong negation in logic programs. In L. Haaparanta, M. Kusch, and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*, pages 430–453. Acta Philosophica Fennica 49, 1990.

[45] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons, 1992.

[46] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 105–119. MIT Press, 1991.

[47] L. M. Pereira, J. N. Aparício, and J. J. Alferes. A derivation procedure for extended stable models. In *Int. Joint Conf. on AI.* Morgan Kaufmann, 1991.

[48] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of Logic Programming. Special issue on Nonmonotonic reasoning*, 17(2, 3 & 4):227–263, 1993.

[49] L. M. Pereira, C. Damásio, and J. J. Alferes. Debugging by diagnosing assumptions. In P. A. Fritzson, editor, *1st Int. Ws. on Automatic Algorithmic Debugging, AADEBUG'93*, number 749 in LNCS, pages 58–74. Springer–Verlag, 1993.

[50] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 316–330. MIT Press, 1993.

[51] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal in logic programs. In L. Damas and M. Filgueiras, editors, *6th Portuguese AI Conf.* Springer–Verlag, 1993.

[52] P.Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States.* The MIT Press, 1988.

[53] H. Przymusinska, T. C. Przymusinski, and H. Seki. Soundness and completeness of partial deductions for well–founded semantics. In A. Voronkov, editor, *Proc. of the Int. Conf. on Automated Reasoning.* LNAI 624, 1992.

[54] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *8th Symp. on Principles of Database Systems.* ACM SIGACT-SIGMOD, 1989.

[55] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 459–477. MIT Press, 1990.

[56] T. Przymusinski. Static semantics for normal and disjunctive programs. Technical report, Dep. of Computer Science, Univ. of California at Riverside, 1993.

[57] T. Przymusinski and D.S. Warren. Well–founded semantics: Theory and implementation. 1992.

[58] R. Reiter. On closed–world data bases. In H. Gallaire and J. Minker, editors, *Logic and DataBases*, pages 55–76. Plenum Press, 1978.

[59] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie and J. Mylopoulos, editors, *On Conceptual Modelling*, pages 191–233. Springer–Verlag, 1984.

[60] R. Reiter. On asking what a database knows. In John Lloyd, editor, *Computational Logic*, pages 96–113. Basic Research Series, Springer–Verlag, 1990.

[61] K. A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13:1–22, 1992.

[62] F. Teusink. A proof procedure for extended logic programs. In *Proc. ILPS'93*. MIT Press, 1993.

[63] A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of the Symposium on Principles of Database Systems*, pages 1–10. ACM SIGACT-SIGMOD, 1989.

[64] G. Wagner. A database needs two kinds of negation. In B. Thalheim, J. Demetrovics, and H-D. Gerhardt, editors, *Mathematical Foundations of Database Systems*, pages 357–371. LNCS 495, Springer–Verlag, 1991.

[65] G. Wagner. Logic programming with strong negation and innexact predicates. *J. of Logic and Computation*, 1(6):835–861, 1991.

[66] G. Wagner. Neutralization and preeemption in extended logic programs. Technical report, Freien Universitat Berlin, 1993.

[67] G. Wagner. Reasoning with inconsistency in extended deductive databases. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 300–315. MIT Press, 1993.

[68] M. Winslett. Reasoning about action using a possible model approach. In *7th AAAI*, pages 89–93, 1988.

A Proofs of SLX correctness

We begin by assigning ranks to derivations. The proofs of correctness essentially rely on two lemmas proven by transfinite induction on the rank of derivations. In order to trim the proof we begin by making some simplifications in the definitions of derivations:

In definition 4.2 of SLX-T-derivation one possible way of removing a selected default literal *not* A from a goal is to find a SLX-T-refutation for $\leftarrow \neg A$. However this case is redundant. Note that the other case for removing *not* A is when there is no SLX-TU-refutation for $\leftarrow A$. But definition 4.3 states that in a SLX-TU-derivation, if there is a SLX-T-refutation for the explicit complement of a selected objective literal then the goal is the last in the derivation. Thus, if there is a SLX-T-refutation for $\leftarrow \neg A$, the only SLX-TU-derivation for $\leftarrow A$ is this single goal and is a failure, and so, even when not considering the first possibility, *not* A is nevertheless removed from the goal. Consequently, in definition 4.2 the case $L_k = not\ A$ can be simplified to:

– *if there is no SLX-TU-refutation for* A *in* P *then the derived goal is*

$$\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots L_n$$

Now let's look at the cases for a selected objective literal L_k in definition 4.3. Clearly the first one corresponds to introducing *not* $\neg L_k$ in the derived goal. This is so because if there is a SLX-T-refutation for $\leftarrow \neg L$ the derivation will become a failure (and this is equivalent to the first case), and if there is no such refutation it is simply removed (and this is equivalent to the second case). Consequently, in definition 4.3 we remove the first case for a selected objective literal, keep the third, and modify the second to[24]:

– *if the input rule is* $L_k \leftarrow B_1, \ldots, B_m$ *the derived goal is*

$$\leftarrow L_1, \ldots, L_{k-1}, not\ \neg L_k, B_1, \ldots, B_m, L_{k+1}, \ldots L_n$$

Now we assign ranks to these simplified derivations. As the proofs shall show, we do not need to assign a rank neither to SLX-T-failures nor to SLX-TU-refutations. These do not contribute towards proving literals that belong to the WFM[25].

[24] Note how this exactly corresponds to using a semi-normal version of the program in SLX-TU-derivations.

[25] This is tantamount to having no need to assign a rank to undetermined nodes in [61].

Intuitively, the rank of a SLX-T-refutation reflects the depth of "calls" of subsidiary trees that are considered in the refutation. Its definition, below, can be seen as first assigning to each literal removed from a goal an associated rank. When removing an objective literal no subsidiary tree is considered, and so the rank is not affected. The empty goal has rank 0. When removing a default literal, the depth of subsidiary trees that has to be considered is the maximum (more precisely, the least upper bound for the infinite case) of the depth of all SLX-TU-failures[26]. The depth needed for finally removing all literals from a goal is the maximum of the ranks associated with each of the literals in the goal.

Definition A.1 (Rank of a SLX-T-refutation) *The rank of a SLX-T-refutation is the rank of its first goal. Ranks of goals in the refutation are defined as follows:*

- *The rank of* $\leftarrow \Box$ *is 0.*
- *Let* G_i *be a goal in a refutation whose next selected literal is objective. The rank of* G_i *is the rank of* G_{i+1}.
- *Let* G_i *be a goal in a refutation whose next selected literal is a default one, not* L, *and let* α *be the least ordinal upper bound (i.e. maximum in the finite case) of the ranks of the SLX-TU-failures for* $\leftarrow L$[27]. *The rank of* G_i *is the maximum of* α *and the rank of* G_{i+1}.

Ranks of SLX-TU-failures reflect the depth of "calls" that is needed to fail the derivation of subsidiary trees. Note that the failure of a derivation is uniquely determined by the last goal in the derivation, and more precisely by its selected literal. If that literal is objective then no subsidiary tree is needed to fail it, and thus its rank is 0. For failing a default literal *not* L one has to find a SLX-T-refutation for $\leftarrow L$. Several might exist, but it is enough to consider the one with minimum depth. Moreover, in this case one has to increment the rank, since the default literal *not* L was failed, and caused an extra "call". Note that, for SLX-T-refutations this increment is not considered. The issue of incrementing the rank only for one kind of derivations is tantamount to that of considering the increment of levels of I_is in the sequence for constructing the WFM only after the application of the two operators, Γ and Γ_s.

Definition A.2 (Rank of a SLX-TU-failure) *An infinite SLX-TU-failure has rank 0. The rank of a finite SLX-TU-failure is the rank of its last goal. Let* G_n *be the last goal of the derivation, and let* L_k *be its selected literal:*

- *if* L_k *is an objective literal then the rank is 0.*
- *if* L_k *is a default literal, not* A, *then the rank is* $\alpha + 1$, *where* α *is the minimum of the ranks of all SLX-T-refutations for* $\leftarrow A$.

The following lemma is used in the proofs of correctness. This lemma relates the existence of sequences where some default literals are removed to the Γ operator by which some default literals are removed from the body of rules:

Lemma A.1 *Let* I *be an interpretation, and let* $(\leftarrow L), G_1, \ldots$ *be a sequence of goals constructed as per definition 4.3 (resp. definition 4.2), except that selected default literals not* L_k *such that* $L_k \notin I$ *are immediately removed from goals. Then,* $L \in \Gamma_s I$ *(resp.* $L \in \Gamma I$*) iff the sequence is finite and ends with the empty goal.*

Proof (sketch): Here we omit the proof for $L \in \Gamma I$ with definition 4.2, which is similar.

If $L \in \Gamma_s I$ then, as per the definition of Γ_s, there must exist a finite set of rules in $\frac{P_s}{I}$ such that L belongs to its least model. According to the definition of $\frac{P_s}{I}$ and of semi-normal program, there is a finite set of rules in P such that for each default literal *not* L in their bodies $L \notin I$, and for each such rule with head H, $\neg H \notin I$. Let P^* be the subset of P formed by those rules.

[26] Note that for removing a default literal all SLX-TU-failures must be considered. This is the reason behind "maximum".

[27] Note that, since we are in a SLX-T-refutation, all SLX-TU-derivations for $\leftarrow L$ are failures.

The only default literals to be considered by definition 4.3 will be those in the bodies, plus the default negations of $\neg$-complements of the heads of rules used in the derivation. So, given the completeness of SL-resolution[28] [41], and the fact that all these introduced literals are not in I (as shown above), a sequence of goals considering only the rules in the finite P^* exists and ends in the empty goal. Thus the least model of $\frac{P^*_s}{I}$ contains L. $\diamondsuit$

Lemma A.2 *Let P be an extended logic program, L an objective literal, and $\{I_\alpha\}$ be the sequence constructed for the WFM_p of P, as per theorem 3.1. In that case:*

1. *if there is a SLX-T-refutation for $\leftarrow L$ in P with rank $< i$ then $L \in I_i$.*
2. *if all SLX-TU-derivations for $\leftarrow L$ in P are failures with rank $\leq i$ then $L \notin \Gamma_s I_i$.*

Proof: By transfinite induction on i:

i is a limit ordinal δ: The case where $\delta = 0$ is trivial.

For $\delta \neq 0$, for point 1, assume that there is a SLX-T-refutation for $\leftarrow L$ with rank $< \delta$. Thus, there is a $\alpha < \delta$ for which such a refutation exists with rank $< \alpha$. Then, $\exists_{\alpha<\delta} L \in I_\alpha$. Thus, $L \in \bigcup_{\alpha<\delta} I_\alpha$, i.e. $L \in I_\delta$.

For point 2 the proof is similar, and omitted here for brevity.

Induction step: Assume points 1 and 2 of the lemma hold for some i. We now prove that point 1 also holds for $i+1$ (the proof for point 2 is similar, and it too is omitted for brevity).

If there is a SLX-T-refutation for $\leftarrow L$ with rank $< i+1$ then, by definition of ranks for these refutations, all subsidiary derivations for default literals $not\ L_j$ in the refutation are failed and of rank $< i+1$ (and thus $\leq i$) and are simply removed. So, given point 2, $\forall j, L_j \notin \Gamma_s I_i$.

From lemma A.1, by tacking there the interpretation $I = \Gamma_s I_i$, and by removing all $not\ L_j$ literals, it follows that $L \in \Gamma\Gamma_s I_i$, i.e. $L \in I_{i+1}$.$\diamondsuit$

Proof of theorem 5.5: If L is an objective literal, then the result follows immediately from lemma A.2, and the monotonicity of $\Gamma\Gamma_s$ (theorem 5.1).

Let $L = not\ A$. If there is a SLX-T-refutation for $\leftarrow not\ A$ with rank i then, by definition of SLX-T-refutation, all SLX-TU-derivations for $\leftarrow A$ are failures of rank $\leq i$. By point 2 of lemma A.2, $A \notin \Gamma_s I_i$.

Let M be the least fix-point of $\Gamma\Gamma_s$. Given that $\Gamma\Gamma_s$ is monotonic, $I_i \subseteq M$, i.e. for any objective literal A, $A \in I_i \Rightarrow A \in M$. By anti-monotonicity of Γ_s, $A \in \Gamma_s M \Rightarrow A \in \Gamma_s I_i$. Thus, since $A \notin \Gamma_s I_i$, $A \notin \Gamma_s M$ i.e., by definition of the WFM_p, $not\ A \in WFM_p(P)$. $\diamondsuit$

Now we prove theoretical completeness of SLX. To do so we begin by proving a lemma that, like lemma A.1, relates sequences with the Γ operator. Then we prove completeness for objective literals by transfinite induction on the ranks for a particular class of computation rules. Finally we lift this restriction, and prove completeness also for default literals.

Lemma A.3 *Let I be an interpretation, and L an objective literal. If $L \notin \Gamma_s I$ (resp. $L \notin \Gamma I$) then each possible sequence of goals starting with $\leftarrow L$ and constructed as per definition 4.3 (resp. definition 4.2), except that selected default literals not L_k such that $L_k \notin I$ are immediately removed from goals, is either: infinite; ends with a goal where the selected literal is objective; ends with a goal where the selected literal is not A and $A \in I$.*

Proof: Similar to the proof of lemma A.1. $\diamondsuit$

Lemma A.4 *Let P be an extended logic program, L an objective literal, and $\{I_\alpha\}$ be the sequence constructed for the WFM_p of P. Then, there exists a selection rule R such that:*

[28] Note that for definite programs both T and TU derivations reduce to SL-derivation.

1. *if* $L \in I_i$ *then there is a SLX-T-refutation for* $\leftarrow L$ *in* P *with rank* $< i$.
2. *if* $L \notin \Gamma_s I_i$ *then all SLX-TU-derivations for* $\leftarrow L$ *in* P *are failures with rank* $\leq i$.

Proof: Let R be a selection rule that begins by selecting all objective literals, and then default ones subject to that it selects a *not* L before a *not* L' if there is a j in the sequence of the $\{I_\alpha\}$ such that $L \notin \Gamma_s I_j$ and $L' \in \Gamma_s I_j$.

By transfinite induction on i:

i is a limit ordinal δ: The case where $\delta = 0$ is trivial. For point 1 and $\delta \neq 0$, the proof is similar to the one presented in lemma A.2 when $i = \delta$.

For point 2 and $\delta \neq 0$, assume that $L \notin \Gamma_s I_\delta$. By lemma A.3, making the I in that lemma equal to I_δ, each SLX-TU-derivation for $\leftarrow L$ is either:

- infinite, and in this case a failure of rank 0.
- ends with a goal where the selected literal is objective, i.e. a failure of rank 0.
- ends with a goal where the selected literal is *not* A and $A \in I_\delta$. In this case, and given that point 1 is already proven for $i = \delta$, there is a SLX-T-refutation for $\leftarrow A$ with rank $< \alpha$ such that $\alpha < \delta$. Thus, and according to the definition of ranks, the rank of this derivation is $\leq \delta$.

Note that, by considering the special selection rule R in the sequences mentioned in lemma A.3, these become indeed equal to derivations, where the *not* L_k such that $L_k \notin I_\delta$ are never selected.

Induction step: Assume points 1 and 2 of the lemma hold. We begin by proving that point 1 also holds for $i + 1$.

Assume that $L \in \Gamma\Gamma_s I_i$. By lemma A.1, there exists a sequence ending with the empty goal, constructed as per definition 4.2, except that selected default literals *not* L_k such that $L_k \notin \Gamma_s I_i$ are immediately removed from goals. By point 2, for any L_k, all SLX-TU-derivations for $\leftarrow L_k$ are failures with rank $\leq i$. Therefore the sequence is a refutation. Moreover its rank is $\leq i$ and thus also $< i$. This proves point 1.

Now we prove that point 2 also holds for $i + 1$. Assume that $L \notin \Gamma_s I_{i+1}$. By lemma A.3, considering the I in that lemma equal to I_{i+1}, each SLX-TU-derivation for $\leftarrow L$ is either:

- infinite, and in this case a failure of rank 0.
- ends with a goal where the selected literal is objective, i.e. a failure of rank 0.
- ends with a goal where the selected literal is *not* A and $A \in I_{i+1}$. In this case, and given that point 1 is already proven, there is a SLX-T-refutation for $\leftarrow A$ with rank $< i + 1$. Thus, and according to the definition of ranks, the rank of this derivation is $< i + 2$, i.e. $\leq i + 1$.

The argument for saying that the sequences of lemma A.3 are derivation is similar to the one used above for limit ordinals.$\diamondsuit$

Note that, in the proof of point 1 above, we never use the special selection rule R. Thus, for SLX-T-derivations an arbitrary selection rule can be used.

Moreover, in point 2, the only usage of R is to guarantee that the rank of all SLX-TU-failures is indeed $\leq i$. This is needed for proving the lemma by induction. However, it is clear that if by using R all SLX-TU-derivations are failures, although with a possibly greater rank, the same happens with an arbitrary selection rule[29]. This is why there is no need to consider the special selection rule in theorem 5.6.

Proof of theorem 5.6: If L is an objective literal the proof follows from lemma A.4.

[29] Note that literals involved in infinite recursion through negation do not give rise to SLX-TU-failures.

Let $L = not\ A$. By definition of WFM_p, there exists an ordinal λ such that I_λ is the least fix-point of $\Gamma\Gamma_s$. Thus, again by definition of WFM_p, $A \notin \Gamma_s I_\lambda$, and by point 2 of lemma A.4 all SLX-TU-derivations for $\leftarrow A$ in P are failures. Consequently, the SLX-T-derivation consisting of the single goal $\leftarrow not\ A$ is a refutation. $\Diamond$

B Proofs on distance and closeness

Proof of theorem 7.1: We prove one same case for both orderings. The other three cases are similar. Let $I = T \cup not\ F$, $J = T_1 \cup not\ F_1$ and $K = T_2 \cup not\ F_2$ such that $I \prec_F J \wedge \neg\exists_K\ I \prec_F K \prec_F J$, in other words:

$$\begin{array}{rl} & T \subseteq T_1 \wedge F \subseteq F_1 \wedge (T \neq T_1 \vee F \neq F_1) \wedge \\ \neg\exists_{T_2,F_2} & [T \subseteq T_2 \wedge F \subseteq F_2 \wedge (T \neq T_2 \vee F \neq F_2) \wedge \\ & T_2 \subseteq T_1 \wedge F_2 \subseteq F_1 \wedge (T_2 \neq T_1 \vee F \neq F_2)\,] \end{array}$$

Thus, J is a nearest element of I. Suppose $T \neq T_1$ and $F \neq F_1$ then $T_2 = T$ and $F_2 = F_1$ falsifies the condition above, thus $T = T_1$ or $F = F_1$. Consider the case where $T \neq T_1$, $F = F_1$ and $T \subseteq T_1$. In order to satisfy the above condition T_1, must be equal to $T \cup \{L\}$ such that $L \notin T$. If T_1 has more than one element than T, there is a K that makes the condition above false (namely the one obtained by adding a single element to the true literals in I). In order to ensure that J is an interpretation, T and F should be disjoint, and thus L must not belong to F (because $L \in T_1$), and should obey coherence, and therefore $\neg L$ must be in F. We have proven that the interpretations $\{\ T \cup \{L\} \cup not\ F \mid L \notin F \wedge L \notin T \wedge \neg L \in F\ \}$ are nearest elements of I.

We now proceed to show the same case for the classical ordering among interpretations. Consider again I, J and K, such that $I \prec_C J \wedge \neg\exists_K\ I \prec_C K \prec_C J$, so:

$$\begin{array}{rl} & T \subseteq T_1 \wedge F_1 \subseteq F \wedge (T \neq T_1 \vee F \neq F_1) \wedge \\ \neg\exists_{T_2,F_2} & [T \subseteq T_2 \wedge F_2 \subseteq F \wedge (T \neq T_2 \vee F \neq F_2) \wedge \\ & T_2 \subseteq T_1 \wedge F_1 \subseteq F_2 \wedge (T_2 \neq T_1 \vee F \neq F_2)\,] \end{array}$$

Thus, J is a nearest element of I and similarly, $T = T_1$ or $F = F_1$. Consider the case where $T \neq T_1$, $F = F_1$ and $T \subseteq T_1$. For the same reasons, T_1 must be equal to $T \cup \{L\}$, such that $L \notin T$. Disjointness and coherence must also be enforced and therefore $I^{\odot}_{\preceq_C} = I^{\odot}_{\preceq_F}$. $\Diamond$

Proof of theorem 7.3: The proof is by induction on $dist(x, y)$:

Base case: If $dist(x, y) = 0$ then $x = y$ by *(d1)* and therefore $T_1 = T_2$ and $F_1 = F_2$ and thus $\#Diff(x, y) = 0$.

Induction Step: Suppose $dist(x, y) = n + 1$ then there exist a $z = T_3 \cup not\ F_3$ such that $dist(x, z) = n$ and $y \in z^{\odot}_{\preceq_F}$. By induction: $dist(x, z) = \#(T_1 - T_3) + \#(T_3 - T_1) + \#(F_1 - F_3) + \#(F_3 - F_1)$.
But if $y \in z^{\odot}_{\preceq_F}$ either

1. $y = T_3 \cup \{L\} \cup not\ F_3$ and $L \notin F_3 \wedge L \notin T_3 \wedge \neg L \in F_3$;
2. $y = T_3 \cup not\ (F_3 \cup \{L\})$ and $L \notin F_3 \wedge L \notin T_3$;
3. $y = (T_3 - \{L\}) \cup not\ F_3$ and $L \in T_3$;
4. $y = T_3 \cup not\ (F_3 - \{L\})$ and $L \in F_3 \wedge \neg L \notin T_3$.

We prove only case 1, the others are similar. If $y = T_3 \cup \{L\} \cup not\ F_3$ and $L \notin F_3 \wedge L \notin T_3 \wedge \neg L \in F_3$ then $dist(x, y) = \#[T_1 - (T_3 \cup \{L\})] + \#[(T_3 \cup \{L\}) - T_1] + \#(F_1 - F_3) + \#(F_3 - F_1)$. Then two cases can occur: $L \in T_1$ or $L \notin T_1$. The former leads to contradiction, since for $L \in T_1$ then $\#[T_1 - (T_3 \cup \{L\})] = \#(T_1 - T_3) - 1$ and $\#[(T_3 \cup \{L\}) - T_1] = \#(T_3 - T_1) + 1$ and the sum above is equal to $dist(x, z) = n$, and by induction $dist(x, y) = n$. Contradiction.

If $L \notin T_1$ then $\#[T_1 - (T_3 \cup \{L\})] = \#(T_1 - T_3)$ and $\#[(T_3 \cup \{L\}) - T_1] = \#(T_3 - T_1) + 1$, therefore $dist(x, y) = dist(x, z) + 1 = n + 1$.

The set $Diff(x,y)$ contains the difference of y to x. If a literal changes from either **t** or **f** to undefined then it appears in $Diff(x,y)$ labeled with **u**. If it changes from false to true (resp. true to false) it appears labeled both with **u** and **t** (resp. **u** and **f**). The intuition is that changing a literal to undefined (or from undefined to false or true) costs one "unit". A change from false to true (or from true to false) costs 2 because it is necessary to change the literal from false to undefined and then from undefined to true. Therefore, the distance between x and y is given by the cardinality of $Diff(x,y)$. $\diamondsuit$

Proof of theorem 7.4: By induction on $d_{\preceq}(M,D)$:

Base case: If $d_{\preceq}(M,D) = 0$ then $M = A = D$ by *(d1)*. So $M \in Close_{d_{\preceq}}(\mathcal{C},M)$, since $\forall_{B\in\mathcal{C}} d_{\preceq}(M,M) = d_{\preceq}(M,B) + d_{\preceq}(B,M)$ implies $D = M$ ($d_{\preceq}$ is a metric).

Induction Step: If $D \in Closed_{d_{\preceq}}(\mathcal{C},M)$ then the result follows trivially. Consider $D \notin Closed_{d_{\preceq}}(\mathcal{C},M)$, i.e.

$$\exists_{B\in\mathcal{C}} \ : \ d_{\preceq}(M,D) = d_{\preceq}(M,B) + d_{\preceq}(B,D) \wedge d_{\preceq}(B,D) \geq 0$$

Thus, by induction: $\exists_A \ : \ A \in Closed_{d_{\preceq}}(\mathcal{C},M) \wedge d_{\preceq}(M,B) = d_{\preceq}(M,A) + d_{\preceq}(A,B)$.
Therefore, $d_{\preceq}(M,D) = d_{\preceq}(M,A) + d_{\preceq}(A,B) + d_{\preceq}(B,D)$.
But by *d4)* we have $d_{\preceq}(B,D) \leq d_{\preceq}(A,B) + d_{\preceq}(B,D)$ and therefore

$$d_{\preceq}(M,D) \geq d_{\preceq}(M,A) + d_{\preceq}(A,D)$$

Again by *(d4)*, $d_{\preceq}(M,D) \leq d_{\preceq}(M,A) + d_{\preceq}(A,D)$ and so $d_{\preceq}(M,D) = d_{\preceq}(M,A) + d_{\preceq}(A,D)$.$\diamondsuit$

Proof of theorem 7.5: It is enough to prove that: $dist(M,A) = dist(M,B) + dist(B,A)$ iff $Diff(M,B) \subseteq Diff(M,A)$. Here we only present the only if proof.
If $Diff(M,B) \subseteq Diff(M,A)$ then:

$$\begin{array}{l} T_M - T_B \subseteq T_M - T_A \text{ and } T_B - T_M \subseteq T_A - T_M \\ F_M - F_B \subseteq F_M - F_A \text{ and } F_B - F_M \subseteq F_A - F_M \end{array}$$

We first show that $T_M - T_A \subseteq (T_M - T_B) \cup (T_B - T_A)$, since

$$\begin{array}{l} T_M \cap \overline{T_A} \cap (T_M \cap \overline{T_B} \cup T_B \cap \overline{T_A}) \equiv \\ (T_M \cap \overline{T_A} \cap \overline{T_B}) \cup (T_M \cap \overline{T_A} \cap T_B) \equiv \\ T_M \cap \overline{T_A} \cap (\overline{T_B} \cup T_B) \equiv T_M - T_A \end{array}$$

Because $T_B - T_M \subseteq T_A - T_M$ then $T_M \cup T_B \subseteq T_M \cup T_A$, and thus $T_M \cap \overline{T_A} \cup T_B \cap \overline{T_A} \subseteq T_M \cap \overline{T_A}$, i.e. $T_B - T_A \subseteq T_M - T_A$. Using $T_M - T_B \subseteq T_M - T_A$ then we conclude $(T_M - T_B) \cup (T_B - T_A) \subseteq T_M - T_A$. With similar proofs, we conclude:

$$\begin{array}{l} T_M - T_A = (T_M - T_B) \cup (T_B - T_A) \\ T_A - T_M = (T_B - T_M) \cup (T_A - T_B) \\ F_M - F_A = (F_M - F_B) \cup (F_B - F_A) \\ F_A - F_M = (F_B - F_M) \cup (F_A - F_B) \end{array}$$

From these equalities and the fact that the right-hand sides are all disjoint the result follows. $\diamondsuit$