III

APPLICATION TO DIAGNOSIS, DEBUGGING AND UPDATING OF LOGIC PROGRAMS WITH IMPLICIT AND EXPLICIT NEGATION

Carlos Viegas Damásio - Luís Moniz Pereira
AI Centre, Uninova and DCS
Universidade Nova de Lisboa
2825 Monte da Coperica, Portugal
e-mail: fdllmp@fcct.ul.pt

RESUMEN

Estudios recientes hacen uso de la programación lógica (LP) y, en particular, de la LP con negación explícita (programación lógica extendida-XLP) [22, 12, 13] para resolver y representar problemas de razonamiento no monótono [27, 26]. El propósito de este trabajo es ampliar de una manera unificada el alcance de las aplicaciones XLP al diagnóstico, depuración declarativa y actualización de bases de conocimiento. La potencia expresiva de XLP para hacer esto se logra permitiendo que haya programas con resultados contradictorios, que serán revisados por una semántica de extracción de contradicciones, la cual separa convenientemente aquellas suposiciones que conllevan alguna contradicción y las revisa.

La estructura del texto es la siguiente:

Aunque el título de este trabajo sugiere una orientación aplicada, nuestra presentación sería más bien incompleta con una descripción superficial de los fundamentos teóricos que lo soportan. En consecuencia, en la sección 2 introducimos el lenguaje definido por nosotros y usado en el resto del artículo -programas lógicos extendidos con dos tipos de negación-.

Después, en la sección 3, examinamos la semántica extendida de programación lógicas (WFSX). Al final de esta sección se habrá mostrado cómo asignar significado a una clase
amplia de programas lógicos extendidos, que serán usados entonces como nuestra representación del problema y como vehículo para resolver problemas.

En la sección 4, el lector puede encontrar los métodos de extracción de contradicciones bivaluadas y trivaluadas, que hemos definido y fundamentado con la semántica WFSX y que son las herramientas teóricas básicas usadas en las aplicaciones. Con esta sección se concluye la parte más formal del artículo.

En la sección 5 se muestra cómo usar los resultados previos para resolver problemas de diagnóstico general. Empezamos informando sobre un teorema principal que define el espectro de aplicabilidad de la extracción de contradicciones al diagnóstico. En esencia, hemos mostrado que podemos capturar un marco de trabajo unificado de las dos corrientes principales del diagnóstico basado en modelos: las aproximaciones basadas en consistencia y las aproximaciones abductivas. El método propuesto define una traducción de este marco a un lenguaje de la programación lógica extendida con restricciones de integridad. Esta sección se cierra con varios ejemplos de aplicaciones ilustrativas de nuestra aproximación al diagnóstico. Partes de esta sección aparecieron en [29].

Posteriormente, en la sección 6, mostramos como el depurador de los programas lógicos normales puede ser fructíferamente comprendido como un problema de extracción de contradicciones/diagnóstico. Describimos y analizamos aquellos dos aspectos, siendo el principal logro una transformación del programa que es capaz de identificar todos los conjuntos mínimos posibles de trabas que pueden explicar la conducta anormal de un programa erróneo. Una parte de esta sección apareció en [28].

Concluimos este artículo con una pequeña sección que exhibe cómo se puede usar la transformación del depurador anteriormente descrito en el problema de la actualización en las bases de datos deductivas, comparándolo con trabajos previos. Una parte de esta sección apareció en [30].

1. INTRODUCTION

Recent approaches make use of logic programming (LP), and in particular LP with explicit negation (extended logic programming-XLP) [22, 12, 13], to solve and represent nonmonotonic reasoning problems [27, 26]. The aim of the present work is to enlarge in an unified way the scope of XLP applications to diagnosis, to declarative debugging, and to knowledge base updates. The expressive power of XLP to do so is attained by allowing programs with contradictory results to be revised by a contradiction removal semantics which adequately withdraws assumptions that support some contradiction, and revises them.

First we elaborate on the work of [23, 25] on contradiction removal of extended logic programs (CRSX), so as to obtain not only three-valued revisions of assumptions (to the undefined truthvalue) but also two-valued ones. In the two-valued revision case, assumptions are changed into their complements instead.

Then we apply this theory to diagnosis. Because [5] unifies the abductive and consistency-based approaches to diagnosis for generality we present a methodology that transforms any diagnostic problem of [5] into an extended logic program, and solve it with our contradiction removal methods. Another unifying approach to diagnosis with logic programming [32] uses Generalised Stable Models [16]. The criticisms they voice of Console and Torasso’s approach do not carry over to our representation, ours having the advantage of a more expressive language: explicit negation as well as implicit negation (or negation by default).

In addition, we apply our theory to debugging, setting forth a method to debug normal Prolog programs, and showing that declarative debugging [21] can be envisaged as contradiction removal, and so providing a simple and clear solution to this problem. Furthermore, we show how diagnostic problems can be solved with contradiction removal applied to the artifact’s representation in logic plus observations. Declarative debugging can thus be used to diagnose blueprint specifications of artifacts.

Our final application concerns the problems of updating knowledge bases expressed by logic programs. We compare with previous work and show, as before, the superiority of the results obtained by our theoretical developments regarding the semantics of the extended logic programs and its attending contradiction removal techniques.

The structure of the text is as follows:

Although this work’s title suggests a more application oriented focus, we think that our presentation would be rather incomplete with a shallow description of the theoretical foundations supporting it. Therefore, in section 2 we introduce the language, defined by us and used in the rest of the article - logic programs extended with two kinds of negation.

Afterwards, in section 3, we review the extended logic program semantics (WFSX). Therefore, by the end of this section it has been shown how to assign meaning to a broad class of extended logic programs which will then be used as our problem representation and problem solving vehicle.

In section 4 the reader can find the 3-valued and 2-valued contradiction removal methods that we’ve defined and supported by the WFSX semantics, which are the basic theoretical tools used in the applications. With this section we conclude the more formal part of the paper.

Section 5 shows how to use the previous results to solve general diagnosis problems. We start by reporting a major theorem that defines the contradiction removal applicability
spectrum to diagnosis. In essence, we have shown that we can capture an unifying framework of the two main streams of model-based diagnosis: the consistency-based and abductive approaches. The proposed method defines a translation from this framework into the language of extended logic programming with integrity constraints. This section closes with several illustrative application examples of our approach to diagnosis. Parts of this section appeared in [29].

Subsequently, in section 6, we show how the debugging of normal logic programs can be fruitfully understood as a diagnosis/contradiction removal problem. We describe and analyse these two views, the main achievement being a program transformation that is able to identify all the possible minimal sets of bugs that can explain the abnormal behaviour of an erroneous program. Parts of this section appeared in [28].

We conclude this article with a small section that exhibits how the above debugging transformation can be used for the view update problem in deductive databases, and compare to previous work. Parts of this section appeared in [30].

We thank Jose Júlio Alferes for his important help in the carrying out part of the research reported here. We thank ESPRIT BR Compulog 2 (n°. 6810), and JNICT for their financial support, which made this work possible.

2. LANGUAGE

An atom of a given a first-order language \( L \) is an expression of the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate symbol of \( L \), and the \( t_i \)s are terms of \( L \). An objective literal is an atom \( A \) or its explicit negation \( \neg A \). We also use the symbol \( \rightarrow \) to denote complementary literals in the sense of explicit negation. Thus \( \rightarrow A = A \). Here, a literal is either an objective literal \( L \) or its default negation not \( L \). By not \( \{a_0, \ldots, a_m, \ldots\} \) we mean \( \{\neg a_0, \ldots, \neg a_m, \ldots\} \).

A term (resp. atom, objective literal, literal) is called ground if it does not contain variables. The set of all ground terms of \( L \) is called the Herbrand universe of \( L \). By the extended Herbrand base of \( L \), we mean the set of all ground objective literals of \( L \). For short use \( H \) to denote the extended Herbrand base of \( L \).

An extended logic program is a finite set of rules of the form:

\[
H \leftarrow L_1, \ldots, L_n \quad (n \geq 0)
\]

where \( H \) is an objective literal and each of the \( L_i \)s is a literal. In conformity with the standard convention we write rules of the form \( H \leftarrow \) also simply as \( H \).

A normal logic program is an extended logic program where each literal appearing in the body of a rule is either an atom or the default negation of an atom. A normal logic program \( P \) is called definite if none of its rules contains default literals.

By the extended Herbrand base \( H(P) \), we mean the language with alphabet consisting of all the constants, predicate and function symbols that explicitly appear in \( P \).

By grounded version of an extended logic program \( P \) we mean the (possibly infinite) set of ground rules obtained from \( P \) by substituting in all possible ways each of the variables in \( P \) by elements of its Herbrand universe. Thus, without loss of generality (cf. [33]), we coalesce an extended logic program \( P \) with its grounded version.

A program with integrity rules (or constraints) is a set of rules as defined above, plus a set of denials, or integrity rules, of the form:

\[
\bot \leftarrow A_1, \ldots, A_n, \text{not } B_1, \ldots, \text{not } B_m
\]

where \( A_1, \ldots, A_n, B_1, \ldots, B_m \) are objective literals, and \( n + m > 0 \). The symbol \( \bot \) stands for falsity.

3. WFSX OVERVIEW

In this section we briefly review the semantics WFSX for normal logic programs (i.e. with negation by default) extended with a second explicit, negation, which subsumes the well founded semantics [10] of normal programs. For more details the reader is referred to the article in this volume by José Júlio Alferes.

An interpretation of an extended program \( P \) is denoted by \( T \cup \text{not } F \), where \( T \) and \( F \) are disjoint subsets of \( H(P) \). Objective literals in \( T \) are said to be true in \( I \), objective literals in \( F \) false by default in \( I \), and in \( H(P) \) 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 \( L \) must also be entailed'.

This requirement states that whenever some literal is explicitly false then it must be assumed false by default.

Because it is 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 fixpoints of Gelfond-Lifschitz \( \Gamma \)-like operators [11, 12]. The equivalence between both definitions is proven in [1]. We begin by recalling the definition of \( \Gamma \):

**Definition 3.1 (The \( \Gamma \)-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 \( P' \) is the program obtained from \( P' \) by removing all rules
containing a default literal not A such that A ∈ T, and by then removing all the remaining default literals from P.

Let J least model of T, then T is obtained from J by replacing the introduce atoms ¬A by ¬¬A.

To impose the coherence requirement we introduce:

**Definition 3.2 (Seminormal version of a program)** The seminormal version of a program P is the program P', obtained from P by adding to the (possibly empty) body of each rule L ← Body the default literal not ¬L, where ¬L is the complement of L wrt explicit negation.

Below we use Γ(S) to denote Γ(S), and Γ(∅(S)) to denote Γ(∅(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 = Γ(T)
2. T ⊆ Γ(T)

The partial stable model generated by T is the interpretation T ∪ not (¬(T) − Γ(T)).

In other words, partial stable models are determined by the fixpoints of Γ(T). Given a fixpoint T, objective literals in T are true in the PSM, objective literals not in Π(T) are false by default, and all the others are undefined. 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 X − Γ(T), and vice-versa). Moreover note how the usage of Γ(T) imposes coherence: if ¬L is true, i.e. it belongs to Γ(T), then in Γ(T), via semi-normality, all rules for L are removed and, consequently, L ∈ Γ(T), i.e. L is false by default.

**Example 3.1** Program P = {α, ¬α} has no partial stable models. Indeed, the only fixpoint of Γ(T) is {α, ¬α}, and {α, ¬α} ∉ Γ(T), {α, ¬α} = ∅. Thus it is not a PSM.

Programs without partial stable models are said contradictory. Now we simply define the semantics for non-contradictory programs.

**Theorem 3.1 (WSFS semantics)** Every non-contradictory program P has a least (wrt ⊆) 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 (Iα):

I₀ = ∅
I₁ = Γ(∅)
I₀ = ∪ {I₀ | α < δ} for limit ordinal δ

![Image](https://via.placeholder.com/150)

Application to diagnosis, debugging and updating of logic programs with implicit and explicit negation

Carlos Viegas Damásio - Luis Moniz Pereira

There exists a smallest ordinal λ for the sequence above, such that I₉ is the smallest fixpoint of Γ₁. Then WFM(P) = I₉ ∪ not (¬(P) − Γ₁). I₉

In this constructive definition literals obtained after an application of Γ₁ (i.e. in some I₉) are true in WFM(P), and literals not obtained after an application of Γ₁ (i.e. not in I₉, for some α) are false by default in WFM(P).

Note that, like the alternating fixpoint definition of WFS [38], this definition of WSFS also relies on the application of two anti-monotonic operators. However, unlike the definition of WFS, these operators are distinct.

4. CONTRADICTION REMOVAL

As we’ve seen before, WSFS is not defined for every program, i.e. some programs are contradictory and are given no meaning. While for some programs this seems reasonable (e.g. example 3.1), for others this can be too strong.

**Example 4.1** Consider the statements ‘Birds, not shown to be abnormal, fly’, ‘Tweety is a bird and does not fly’ and ‘Socrates is a man’ which are naturally expressed by the program:

fly(X) ← bird(Y), not abnormal(X)  
¬fly(tweety)  
bird(tweety)  
man(socrates)

WSFS assigns no semantics to this program. However, intuitively, we should at least be able to say that Socrates is a man and Tweety is a bird. It would also be reasonable to conclude that Tweety doesn’t fly, because the rule stating that it doesn’t fly, since it is a fact, makes a stronger statement than the one concluding it flies. The latter relies on accepting an assumption of non-normality, justified by the closed world assumption treatment of the negation as failure, and involving the abnormality predicate. Indeed, whenever an assumption supports a contradiction it seems logical to be able to take the assumption back in order to prevent it - ‘Reductio ad absurdum’, or ‘reasoning by contradiction’.

Other researchers have defined paraconsistent semantics for contradictory programs e.g. [6, 2, 17, 36, 39] and use them to formalize diverse forms of reasoning in contradictory databases. On the contrary, we only allow a program to run into contradiction in order to remove it.

To deal with the issue of contradiction brought about by closed world assumptions, rather than defining more sceptical semantics one can rely instead on a less sceptical semantics and accompany it with a revision process that restores consistency, whenever violation of integrity constrains occurs.

These very sceptical semantics model rational reasons who assume the program absolutely correct and so, whenever confronted with a Closed World Assumption (or
We take back reusable assumptions (i.e., assumptions on reusable literals) in a minimal way, and in all alternative ways of removing contradiction.

4.1. Three-valued contradiction removal

Before tackling the question of which assumptions to revise to abolish contradiction, we begin by showing how to impose in a program a revision that takes back some reusable assumption, identifying rules of a special form, which have the effect of prohibiting the falsity of an objective literal in models of a program. Such rules can prevent an objective literal being false, hence their name:

Definition 4.2 (Inhibition rule) The inhibition rule for not \( L \) is \( L \leftarrow \neg L \). Let \( IR(S) = (L \leftarrow \neg L) \), where \( S \) is a set of default literals.

These rules state that if \( \neg A \) is true then \( A \) is also true, and so a contradiction arises. Intuitively this is quite similar to the effect of integrity constraints of form \( \perp \leftarrow \neg A \). Technically the difference is that the removal of such a contradiction in the case of inhibition rules is dealt by \( WFSX \) itself, where in the case of those integrity constraints isn’t.

These rules allow, by adding them to a program, to force default literals in \( WFSX \) to become undefined. Note that changing the truth value of reusable literals from true to undefined is less committing than changing it to false.

To declaratively define the intended program revisions void of contradiction we start by first considering the resulting \( WFSX \)'s of all possible ways of revising a program \( P \) with inhibition rules, by taking back reusable assumptions, even if some revisions are still contradictory programs.

However, it might happen that several different revisions in fact correspond to the same, in the sense that they lead to the same consequences. For a more detailed discussion and solution to this problem the reader is referred to [1].

Definition 4.3 (Submodels of a program) A submodel of a (contradictory) program \( P \) with ICs, and reusable literals \( Rev \), is any pair \( <M, R> \) where \( R \) is a subset of \( Rev \) and \( M = WFM(P, IR(R)) \). In a submodel \( <M, R> \) we dub \( R \) the submodel revision, and \( M \) are the consequences of the submodel revision. A submodel is contradictory iff \( \perp \in M \) or \( M \) is contradictory.

Example 4.2 Consider \( P = (p \leftarrow \neg q; \neg p \leftarrow \neg r; a \leftarrow \neg b) \) with reusable literals \( Rev = (\neg q, \neg r, \neg b) \). Its submodels lattice is depicted in figure 1, where shadowed submodels are contradictory ones.
As we are interested in revising contradiction in a minimal way, we care about those submodels that are noncontradictory and among these, about those that are minimal in the sense of set inclusion.

**Definition 4.4 (Three-valued revision)** A submodel \(<M, R>\) is a three-valued revision of a program \(P\) iff it is noncontradictory.

**Definition 4.5 (Minimal noncontradictory submodel)** A three-valued revision \(<M, R>\) is a minimal noncontradictory submodel (MNS for short) or a minimal revision, of a program \(P\) iff there exists no other three-valued revision \(<M', R'>\) such that \(R' \subseteq R\).

By definition, each MNS of a program \(P\) reflects a revision of \(P\) \(P \cup RevRules\) that guarantees noncontradiction, and such that for any set of rules \(RevRules' \subseteq RevRules\), \(P \cup RevRules'\) is contradictory. In other words, each MNS reflects a revision of the program that restores consistency, and which adds a minimal set of inhibition rules for revisables.

It is clear that with these intended revisions some programs have no revision. This happens when contradiction has a basis on non-revisable literals.

**Example 4.3** Consider program \(P = \{a \leftarrow not \ b; \ b \leftarrow not \ c; \not a; \ c\}\) with revisable literals \(Rev = \{(not \ c)\}\). The only submodels of \(P\) are:

\(<WFM(P), \{\}\>\) and \(<WFM(P \cup \{c \leftarrow not \ c\}), \{not \ c\}\>\).

---

As both these submodels are contradictory \(P\) has no MNS, and thus no revisions. Note that if \(not \ b\) were revisable, the program would have a revision \(P \cup \{b \leftarrow not \ b\}\). If \(not \ b\) were absent from the first rule, \(P\) would have no revision no matter what are the revisables.

4.2. Two-valued contradiction removal

For most practical applications of contradiction removal techniques the three-valued revisions are too sceptical. To cope with this problem we define in this section a two-valued contradiction removal method. Instead of revising CWAs from true to undefined we change their truth-value to false. Contradiction removal is achieved by adding to the original program the complements\(^4\) of some revisable literals.

**Definition 4.6 (Revision facts)** The revision fact for \(not \ L\) in \(S\) is \(RF(S) = \{L \mid not \ L \in S\}\), where \(S\) a set of default literals.

These facts allows, by adding them to a program, to force default literals in \(WFSX\) to become false.

**Definition 4.7 (Submodels of a program)** A submodel \(<M, R>\) of a (contradictory) program \(P\) with ICs, and revisable literals \(Rev\), is any pair \(<M, R>\) where \(R\) is a subset of \(Rev\) and \(M = WFM(P \cup RF(R))\). In a submodel \(<M, R>\) we dub \(R\) the submodel revision, and \(M\) are the consequences of the submodel revision. A submodel is contradictory iff \(\bot \in M\) or \(M\) is contradictory.

Similarly to the three-valued case we define two-valued and minimal revisions:

**Definition 4.8 (Two-valued revision)** A submodel \(<M, R_2>\) is a two-valued revision of a program \(P\) iff it is noncontradictory.

**Definition 4.9 (Minimal two-valued revisions)** A particular two-valued revision \(<M, R_2>\) is a minimal revision, of a program \(P\) iff there exists no other two-valued revision \(<M', R'_2>\), such that \(R' \subseteq R\).

For simplicity we'll use \(R\) to identify a two-valued revision. The other component is implicit.

**Example 4.4** Consider contradictory program \(P\):

\[
\begin{align*}
a & \leftarrow not \ b, \ not \ c \\
\not a & \leftarrow not \ d \\
c & \leftarrow e \\
\bot & \leftarrow b, \ not \ f
\end{align*}
\]

\(^4\) The complement of atom \(L\) is \(not \ L\), and of literal \(not \ not \ L\) is \(L\).
5. APPLICATION TO DIAGNOSIS

In this section we describe a general program transformation that translates diagnostic problems (DP), in the sense of [5], into logic programs with integrity rules. By revising this program we obtain the diagnostic problem’s minimal solutions, i.e. the diagnoses. The unifying approach of abductive and consistency-based diagnosis presented by these authors enables us to represent easily and solve a major class of diagnostic problems using two-valued contradiction removal. Similar work has been done [32] using Generalised Stable Models [16].

We start by making a short description of a diagnostic problem as defined in [5, 8].

A DP is a tuple consisting of a system description, inputs and observations. The system is modelled by a Horn theory describing the devices, their behaviours and relationships. In this diagnosis setting, each component of the system to be diagnosed has a description of its possible behaviours with the additional restriction that a given device can only be in a single mode of a set of possible ones. There is a mandatory mode in each component modelled, the correct mode, that describes correct device behaviour; the other mutually exclusive behaviour modes represent possible faulty behaviours.

Having this static model of the system we can submit to it a given set of inputs (contextual data) and compare the results obtained with the observations predicted by our conceptualized model. Following [5] the contextual data and observation part of the diagnostic problem are sets of parameters of the form parameter(value) with the restriction that a given parameter can only have one observed valued.

From these introductory definitions [5] present a general diagnosis framework unifying the consistency-based (c.f. [35, 8] and others) and abductive approaches (c.f. [31] and others). These authors translate the diagnostic problem into abduction problems where the abducibles are the behaviour modes of the various system components. From the observations of the system two sets are constructed: $\Psi^+$, the subset of the observations that must be explained, and $\Psi^- = \{ \neg f(X) : f(Y) \}$ is an observation, for each admissible value $X$ of parameter other than $Y$. A diagnosis is a minimal consistent set of abnormality hypotheses, with additional assumptions of correct behaviour of the other devices, that consistently explain some of the observed outputs: the program plus the hypotheses must derive (cover) all the observations in $\Psi^+$ consistent with $\Psi^-$. By varying the set $\Psi^+$ a spectrum of different types of diagnosis is obtained.

We show that it is always possible to compute the minimal solutions of a diagnostic problem by computing the minimal revising assumptions of a simple program transformation of the system model.

Example 5.1 Consider the following partial model of an engine, with only one component oil_cup, which has behaviour modes correct and holed [5]:

$$
\begin{align*}
oil\text{\_below\_car}\text{(present)} & \leftarrow \text{holed(oil\_cup)} \\
oil\text{\_level}\text{(low)} & \leftarrow \text{holed(oil\_cup)} \\
oil\text{\_level}\text{(normal)} & \leftarrow \text{correct(oil\_cup)} \\
enGINE\text{\_temperature}\text{(low)} & \leftarrow \text{oil\_level(low),engine(on)} \\
enGINE\text{\_temperature}\text{(normal)} & \leftarrow \text{oil\_level(normal),engine(on)}
\end{align*}
$$

An observation is made of the system, and it is known that the engine is on and that there is oil below the car. The authors study two abduction problems corresponding to this DP:

1. $\Psi^+ = \{ \text{oil\_below\_car(present)} \}$ and $\Psi^- = \{ \}$ (Poole’s view of a diagnostic problem [31]) with minimal solution $W_1 = \{ \text{holed(oil\_cup)} \}$.
2. $\Psi^+ = \Psi^- = \{ \}$ (De Kleer’s DP view [7]) with minimal solution $W_2 = \{ \}$.

To solve abduction problem 1 it is necessary to add the following rules:

$$
\begin{align*}
\bot & \leftarrow \neg \text{oil\_below\_car(present)} \\
\text{correct(oil\_cup)} & \leftarrow \neg \text{not(oil\_cup)} \\
\text{holed(oil\_cup)} & \leftarrow \text{not(oil\_cup)},\text{fault\_mode(oil\_cup, holed)}
\end{align*}
$$

The above program, as wanted, has a single two-valued minimal revision:

$$
\{ \text{not(oil\_cup)}, \text{fault\_mode(oil\_cup, holed)} \}
$$

To solve the second problem, the transformed program has the same rules of the program for problem $P$, except the integrity constraint-it is not necessary to cover any set of observations. The program thus obtained is non-contradictory having minimal revision $\{ \}$.

Next, we present the general program transformation which turns a diagnostic abduction problem into a contradiction removal problem.
Theorem 5.1 Given an abduction problem (AP) corresponding to a diagnostic problem, the minimal solutions of AP are the minimal revising assumptions of the modelling program plus contextual data and the following rules:
1. \( \bot \leftarrow \text{not obs}(v), \) for each \( \text{obs}(v) \in \Psi^* \).
2. \( \neg \text{obs}(v), \) for each \( \text{obs}(v) \in \Psi^* \).

and for each component \( c_i \) with distinct abnormality behaviour nodes \( b_j \) and \( b_k \):
3. \( \text{correct}(c_i) \leftarrow \text{not ab}(c_i) \).
4. \( b_j(c_i) \leftarrow \text{ab}(c_i), \text{fault_node}(c_i, b_j) \).
5. \( \bot \leftarrow \text{fault_node}(c_i, b_j), \text{fault_node}(c_i, b_k) \) for each \( b_j, b_k \) with revisables \( \text{fault_node}(c_i, b_j) \) and \( \text{ab}(c_i) \).

We don’t give a detailed proof of this result but take into consideration:
- Rule 1 ensures that, for each consistent set of assumptions \( \text{obs}(v) \in \Psi^* \) must be entailed by the program.
- Rule 2 guarantees the consistency of the sets of assumptions with \( \Psi^* \).
- Rules 4 and 5 deal and generate all the possible mutually exclusive behaviours of a given component.

Finally, in no revision there appears the literal \( \text{fault_node}(c, \text{correct}) \), thus guaranteeing that minimal revising assumptions are indeed minimal solutions to the DP.

The concept of declarative debugging, see section 6, can be used to aid in the development of logic programs and in particular to help the construction of behavioural models of devices. Firstly, a Prolog prototype or blueprint of the component is written and debugged using the methodology presented in that section. After the system is constructed, the diagnostic problems can be solved using contradiction removal as described above, in the correct blueprint.

In the rest of this section we’ll present several examples of diagnosis problems. Whenever possible, we'll try to write the logic programs as close as possible to the ones obtained by the previous program transformation. We start by a very simple example which shows how difficult the modelisation task can be.

Example 5.2 Consider the simple logic circuit of Fig. 2. We’ll present two models of the circuit. Both are correct for predicting the behaviour of the circuit, but only one can be used to perform correctly the diagnosis task.

The naive solution would model an or gate with the following program:

\[
\begin{align*}
\text{or\_gate}(G, 1, 1, 1) & \leftarrow \text{correct}(G) \\
\text{or\_gate}(G, 1, 0, 1) & \leftarrow \text{correct}(G) \\
\text{or\_gate}(G, 0, 1, 1) & \leftarrow \text{correct}(G) \\
\text{or\_gate}(G, 0, 0, 0) & \leftarrow \text{correct}(G) \\
\text{correct}(G) & \leftarrow \text{not ab}(G)
\end{align*}
\]

The topology of the circuit is captured by:

\[
\begin{align*}
\text{node}(e, E) & \leftarrow \text{node}(a, A), \text{node}(b, B), \text{or\_gate}(g_1, A, B, E) \\
\text{node}(f, F) & \leftarrow \text{node}(c, C), \text{node}(d, D), \text{or\_gate}(g_2, C, D, F) \\
\text{node}(g, G) & \leftarrow \text{node}(e, E), \text{node}(f, F), \text{or\_gate}(g_3, E, F, G)
\end{align*}
\]

Given the inputs, this program correctly predicts the outputs. But our main concern is diagnosis, and this program can not be used to do it!!! Suppose the situation where the value at nodes 'a', 'b', 'c' and 'd' is 1 and the output at node 'g' is 0. Obviously, we cannot explain this wrong output because we have no description of the behaviour of an or gate when it is abnormal, i.e. there are no fault-models. So we only require the consistency with the observed output \( \Psi^* = \{ \bot \} \) and \( \Psi = \{ \text{not node}(g, 1) \} \):

\[
\begin{align*}
\text{node}(a, 1) & \quad \text{node}(b, 1) & \quad \text{node}(c, 1) & \quad \text{node}(d, 1) & \quad \text{node}(g, 1)
\end{align*}
\]

If we apply the contradiction removal method, with the revisables being the \text{ab} literals, we obtain as minimal revisions: \{\text{ab}(g_1), \text{ab}(g_2), \text{ab}(g_3)\}.

Intuitively, the first two diagnoses are incorrect. For instance, consider the diagnosis \{\text{ab}(g_1)\}. In this situation gate 3 still has an input node with logical value 1, therefore its output should be also 1. The problem is that in the program above an 'or' gate to give its output must have both inputs determined, i.e. the absorption property of these gates is not correctly modeled. An alternative and correct description of this circuit is given below:

\[
\begin{align*}
\text{or\_gate}(G, 11, 12, 1) & \leftarrow \text{node}(11, 1), \text{correct}(G) \\
\text{or\_gate}(G, 11, 12, 1) & \leftarrow \text{node}(12, 1), \text{correct}(G) \\
\text{or\_gate}(G, 11, 12, 0) & \leftarrow \text{node}(11, 0), \text{node}(12, 0), \text{correct}(G) \\
\text{correct}(G) & \leftarrow \text{not ab}(G)
\end{align*}
\]

The connection’s representation part is slightly simplified:

\[
\begin{align*}
\text{node}(e, E) & \leftarrow \text{or\_gate}(g_1, a, b, E) \\
\text{node}(f, F) & \leftarrow \text{or\_gate}(g_2, c, d, F) \\
\text{node}(g, G) & \leftarrow \text{or\_gate}(g_3, e, f, G)
\end{align*}
\]

Figure 2. The three or problem.
Now, with the same set of inputs and constraints we obtain the expected diagnosis:

\[(ab(g_1), \overline{ab(g_3)}) \quad \overline{ab(g_3)}\]

Finally, notice that using this new model it is also not possible to explain the output of gate g_3. If we set \[\Psi^+ = \{\text{node}(g, 0)\}\] and \[\Psi^- = \{\overline{\text{node}(g, 1)}\}\], translated according to theorem 5.1 to:

\[\bot \leftarrow \overline{\text{node}}(g, 0) \quad \overline{\text{node}}(g, 1)\]

This new program (plus the input and circuit description) is contradictory, i.e., there is no two-valued revision.

Other solution is given to the previous problem is described in the next example: we maintain the wrong model of the gates add a particular fault model to it. Besides, the example will exemplify in a concrete situation the distinction between three-valued revision and two-valued revision.

**Example 5.3** Consider the circuit of figure 5.3, with inputs \[a = 0, \quad b = 1, \quad c = 1, \quad d = 1, \quad h = 1\]

\[\begin{array}{c}
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\quad \# \quad \# \\
\end{array}\]

\[\begin{array}{c}
a \quad b \\
\quad g_1 \quad e \\
g_2 \quad f \\
\quad g_1 \quad g_2 \\
h \quad k \\
\end{array}\]

Figure 3. Logic circuit of example 4.3.

and (incorrect) output 0. Its behavioural model is:

\[
\begin{align*}
\text{and}_\text{gate}(G, 1, 1, 1) & \leftarrow \text{correct}(G) \\
\text{and}_\text{gate}(G, 1, 1, 0) & \leftarrow \text{abnormal}(G) \\
\text{and}_\text{gate}(G, 0, 1, 0) & \leftarrow \text{correct}(G) \\
\text{and}_\text{gate}(G, 0, 1, 1) & \leftarrow \text{abnormal}(G) \\
\text{and}_\text{gate}(G, 1, 0, 0) & \leftarrow \text{correct}(G) \\
\text{and}_\text{gate}(G, 1, 0, 1) & \leftarrow \text{abnormal}(G) \\
\text{and}_\text{gate}(G, 0, 0, 0) & \leftarrow \text{correct}(G) \\
\text{and}_\text{gate}(G, 0, 0, 1) & \leftarrow \text{abnormal}(G)
\end{align*}
\]

And a similar set of rules for \text{or} gates, as in example 5.2. According to the program transformation two auxiliary rules are needed:

\[
\begin{align*}
\text{correct}(G) & \leftarrow \overline{\text{ab}}(G) \\
\text{abnormal}(G) & \leftarrow \text{ab}(G)
\end{align*}
\]

and the description of the circuit and its connections:

\[
\begin{align*}
\text{node}(a, 0) & \quad \text{node}(b, 1) \quad \text{node}(c, 1) \quad \text{node}(d, 1) \quad \text{node}(h, 1)
\end{align*}
\]

The minimal solutions to this problem are highlighted in figure 4. The two-valued minimal revisions \{ab(g_1), ab(g_2), ab(g_3), ab(g_4)\} are the minimal solutions to the diagnosis problem. The above representation does not suffer from the problems of the example 5.2. This is due to the fact that when an abnormality assumption is made the gate's fault-model become 'active', an output value is produced which can be used by other gates in the circuit. Notice that this program is able to explain the outputs: if an integrity rule enforcing that the output at node 'g' should be 0 is added to the program then the minimal revisions are the same as before.

\[
\begin{align*}
\text{node}(c, 1) & \quad \text{node}(d, 1) \quad \text{node}(e, 1) \quad \text{node}(f, 1) \quad \text{node}(g, 1) \quad \text{node}(h, 1) \quad \text{node}(i, 1) \quad \text{node}(j, 1) \quad \text{node}(k, 1) \quad \text{node}(l, 1) \quad \text{node}(m, 1) \quad \text{node}(n, 1) \quad \text{node}(o, 1) \quad \text{node}(p, 1) \quad \text{node}(q, 1) \quad \text{node}(r, 1) \quad \text{node}(s, 1) \quad \text{node}(t, 1) \quad \text{node}(u, 1) \quad \text{node}(v, 1) \quad \text{node}(w, 1) \quad \text{node}(x, 1) \quad \text{node}(y, 1) \quad \text{node}(z, 1)
\end{align*}
\]

Figure 4. Diagnoses of example 4.3.

If instead of two-valued contradiction removal the three-valued one is used four (with two intuitively incorrect) single-fault diagnoses are found: \{ab(g_1), ab(g_2), ab(g_3)\} and \{ab(g_4)\} Remember that these literals are revisied to undefined, blocking the propagation of values from inputs to outputs. This short example shows again that the naive model of logical gates is not adequate for diagnosis. More differences between three-valued and two-valued contradiction will be drawn in the next example.

In example 5.4 we'll show how to represent and reason with fault-models in the diagnosis task.
Example 5.4 Consider the situation in figure 5, where two inverters are connected in series.

![Two inverters circuit](image)

**Figure 5. Two inverters circuit.**

This particular situation can be represented by the program below:

\[
\begin{align*}
inv(T, G, I, 1) & \leftarrow node(T, I, 0), not \ ab(G) \quad 1 \\
inv(T, G, I, 0) & \leftarrow node(T, I, 1), not \ ab(G) \quad 2 \\
node(T, B, B) & \leftarrow inv(T, g1, a, B) \quad 3 \\
node(T, C, c) & \leftarrow inv(T, g2, b, C) \quad 4 \\
node(0, a, 0) & \quad 5 \\
not \ node(0, c, 0) & \quad 6
\end{align*}
\]

Rules 1-2 model normal inverter behaviour, where correct has been replaced by not \(ab\). Rules 3-4 specify the circuit topology. Rule 5 establishes the input as 0. Rule 6 specifies the observed output should not be 0 (consistency-based approach). The extra parameter \(T\) in all rules is a time-stamp that lets us encode multiple observations. For the time being suppose that the previous observation was made at time 0. The revisables are, as usual, the \(ab\) literals.

Revising this program, using either contradiction removal methods, these minimal revisions are obtained: \{\(ab(g1)\)\} and \{\(ab(g2)\)\}:

Now, trying to explain the output, via integrity rule \(\bot \leftarrow not \ node(0, c, 1)\), the program is contradictory and non-revisable. It is necessary to add a fault-model to the program:

\[
\begin{align*}
inv(T, G, I, 0) & \leftarrow fault_node(G, s0) \quad 7 \\
inv(T, G, I, 1) & \leftarrow fault_node(G, s1) \quad 8 \\
inv(T, G, I, V) & \leftarrow node(T, I, V), fault_node(G, sh) \quad 9 \\
\bot & \leftarrow fault_node(G, M1), fault_node(G, M2), M1 \neq M2 \quad 10
\end{align*}
\]

Rules 7-9 model three fault modes: one expresses the output is stuck at 0, the other that it is stuck at 1, whatever the input may be, and the other that the output is shorted with the input. According to rule 10 the three fault modes are mutually exclusive. If a pure consistency-based diagnosis is performed the revisions are the same as before. Whereas, the observed output can be explained:

\[
\bot \leftarrow not \ node(0, c, 1) \quad 11
\]

The program consisting of rules 1-11 is revisable with minimal diagnosis (with either of the contradiction removal techniques):

\[
\{\(ab(g1)\), fault_node(g1, s0)\}, \{\(ab(g2)\), fault_node(g2, s1)\}
\]

Regardless of the minimal revisions being the same with both methods, they have different consequences. The two-valued approach really explains the output, i.e. \(node(0, c, 1)\) is entailed by any of the revised programs. The three-valued method doesn’t it satisfies the constraints by (indirectly) undefining the literals \(node(0, c, 0)\) and \(node(0, c, 1)\). The distinct effects will be clear soon.

Suppose now that an additional experiment is made at time 1, by setting the input to 1 and observing output 1. This test is modeled by the rules:

\[
\begin{align*}
node(0, a, 1) & \quad 12 \\
\bot & \quad 13 \\
\bot & \leftarrow not \ node(1, c, 1) \quad 14
\end{align*}
\]

With the third-valued contradiction removal method the minimal diagnoses are the same as before, whereas with the two-valued one they are:

\[
\{\(ab(g1)\), fault_node(g1, s0)\}, \{\(ab(g2)\), fault_node(g2, s1)\}, \{\(ab(g1)\), fault_node(g1, s1)\}
\]

Next, a typical and problematic problem is presented and correctly (and easily) solved.

Example 5.5 [37]

Three bulbs are set in parallel with a source via connecting wires and a switch, as specified in the first three rules (where \(ok\) is used instead of correct). Normality is assumed by default in the rule for \(ok\). The two integrity rules enforce that the switch is always either open or closed. Since both cannot be assumed simultaneously, this program has two minimal revisions, with \(ab\), open, closed being the revisables: one obtained by revising the CWA on open (i.e. adding open); the other by revising the CWA on closed (i.e. adding closed). In the first open, not on(b1), not on(b2), not on(b3) are true in the model; in the second closed, on(b1), on(b2), on(b3) do.

![Three bulbs diagram](image)
Further integrity rules specify observed behaviour to be explained. For instance, to explain that bulb 1 is on it is only necessary to add $\bot \leftarrow \text{not on}(b_1)$ to obtain the single, intuitive, minimal revision [closed].

Suppose instead we wish to explain that bulb 2 is off (i.e. not on). Adding $\bot \leftarrow \text{on}(b_2)$, five minimal revisions explain it, four of which express faults:

$$\begin{align*}
\{\text{closed, not ab}(s)\} & \quad \{\text{closed, not ab}(w_1)\} \\
\{\text{closed, not ab}(b_2)\} & \quad \{\text{closed, not ab}(w_2)\} \\
\text{open} &
\end{align*}$$

Adding now both integrity rules, only two of the previous revisions remain: both with the switch closed, but one stating that bulb 2 is abnormal and the other that wire 2 is.

Finally, we show a more extensive example due to [4].

**Example 5.6 [4]**

Causal nets are a general representation schema used to describe possibly incomplete causal knowledge, in particular to represent the faulty behaviour of a system. Consider the (simple) causal model of a car engine in figure 6. A causal net is formed by nodes and arcs connecting nodes. There are (at least) three types of nodes:

- **Initial Cause** nodes - represent the deep causes of the faulty behaviour. It is the initial perturbations are not directly observable;

- **State** nodes - describe partial states of the modeled system;

- **Finding** nodes - observable manifestations of the system.

There are two kinds of arcs: **causal arcs** that represent cause/effect relationships and **has manifestations** arcs connecting states with their observable manifestations. These arcs can be labeled by a MAY tag, stating some sort of incompleteness in the model.

This formalism can be easily translated to logic programs:

$$\begin{align*}
lubric\_oil\_burning & \leftarrow piston\_rings\_used \\
stack\_smoke & \leftarrow lubric\_oil\_burning \\
irreg\_oil\_consumpt & \leftarrow lubric\_oil\_burning \\
oil\_loss & \leftarrow oil\_cup\_holed \\
oil\_below\_car & \leftarrow oil\_loss, may(oil\_below\_car, oil\_loss) \\
oil\_lack & \leftarrow oil\_loss \\
oil\_lack & \leftarrow irreg\_oil\_consumpt \\
coolant\_evaporation & \leftarrow high\_engine\_temp \\
vapour & \leftarrow coolant\_evaporation \\
power\_decrease & \leftarrow high\_engine\_temp, may(power\_dec, high\_eng\_temp) \\
lack\_of\_accel & \leftarrow power\_decrease \\
melted\_pistons & \leftarrow coolant\_evaporation, may(melted\_pistons, cool\_evap) \\
smoke\_from\_eng & \leftarrow melted\_pistons \\
\end{align*}$$

Figure 6. Causal model in a mechanical domain.
If the findings ‘dirty spark plugs’, ‘lack of acceleration’, ‘temperature indicator is red’ and ‘vapour’ are observed the following integrity rules are added to the program:

\[ \downarrow \leftrightarrow \text{not dirty\_spark\_plugs} \quad \downarrow \leftrightarrow \text{not vapour} \]

\[ \downarrow \leftrightarrow \text{not lack\_of\_accel} \quad \downarrow \leftrightarrow \text{not temp\_indic\_red} \]

By revising the program, with the revisables being the initial cause nodes and may literals, the minimal revisions are:

\[
\begin{align*}
& \{ \text{old\_spark\_plugs, may(irreg\_ignition, spark\_plugs\_used\_up)} \\
& \quad \text{piston\_rings\_used, may(power\_decrease, irreg\_ignition)} \} \\
& \{ \text{oil\_cup\_holed, may(irreg\_ignition, spark\_plugs\_used\_up)} \\
& \quad \text{old\_spark\_plugs, may(power\_decrease, irreg\_ignition)} \} \\
\{ \text{oil\_cup\_holed, old\_spark\_plugs, may(power\_decrease, high\_engine\_temp)} \}
\end{align*}
\]

\[
\begin{align*}
& \{ \text{old\_spark\_plugs, piston\_rings\_used, may(power\_decrease, high\_engine\_temp)} \}
\end{align*}
\]

6. DEBUGGING

It is clear that fault-finding or diagnosis is akin to debugging. In the context of logic, both arise as a confrontation between theory and model. Whereas in debugging one confronts an erroneous theory, in the form of a set of clauses, with models in the form of input/output pairs, in diagnosis one confronts a perfect theory (a set of rules acting as a blueprint or specification for some artifact) with the imperfect input/output behaviour of the artifact (which, if it were not faulty, would behave in accordance with a theory model).

What is common to both is the mismatch. The same techniques used in debugging to pinpoint faulty rules can equally be used to find the clauses, in a perfect blueprint, which are at odds with artifact behaviour. Then, by means of the correspondence from the blueprint’s modelization to the artifact’s subcomponents whose i/o behaviour they emulate, the faulty ones can be spotted.

Declarative debugging then is essentially a diagnosis task, but until now its relationship to diagnosis was unclear or inexistent. We present a novel and uniform technique for normal logic program declarative error diagnosis by laying down the foundations on a general approach to diagnosis using logic programming. In so doing the debugging activity becomes clarified, thereby gaining a more intuitive appeal and generality.

This new view may beneficially enhance the cross-fertilization between the diagnosis and debugging fields. Additionally, we operationalize the debugging process via a contradiction removal (or abductive) approach to the problem. The ideas of this work extend in several ways the ones of [21].

A program can be thought of as a theory whose logical consequences engender its actual input/output behaviour. Whereas the program’s intended input/output behaviour is postulated by the theory’s purported models, i.e. the truths the theory supposedly accounts for.

The object of the debugging exercise is to pinpoint erroneous or missing axioms, from erroneous or missing derived truths, so as to account for each discrepancy between a theory and its models. The classical declarative debugging theory [21] assumes that these models are completely known via an omniscient entity or ‘oracle’. In a more general setting, that our theory accounts for, these models may be only partially known and the lacking information might not be (easily) obtainable. By hypothesizing the incorrect and missing axioms that are compatible with the given information, possible corrections are diagnosed but not perfected, i.e. sufficient corrections are made to the program but only virtually. This process of performing sufficient virtual corrections is the crux of our method.

From the whole set of possible diagnoses we argue that the set of minimal ones is the expected and intuitive desired result of the debugging process. When the intended interpretation (model) is entirely known, then a unique minimal diagnosis exists which identifies the bugs in the program. However, in the presence of incomplete information, the set of minimal diagnoses corresponds to all conceivable minimal sets of bugs; these are exactly the ones compatible with the missing information; in other words, compatible with all the imaginable oracle answer sequences that would complete the information about the intended model. It is guaranteed one of these sets pinpoints bugs that justify the disparities observed between program behaviours and user expectations. Mark that if only one minimal diagnosis is obtained then at least part of the bugs in the program were sieved, but more may persist.

Diagnostic debugging can be enacted by the contradiction removal methods introduced in section 4.2 [29]. Indeed, a simple program transformation affords a contradiction removal approach to debugging, on the basis of revising the assumptions about predicates’ correctness and completeness, just for those predicates and goals that support buggy behaviour. We shall see this transformation has an effect similar to that of turning the program into an artifact specification with equivalent behaviour, whose predicates model the components, each with associated abnormality and fault-mode literals. When faced with the disparities between the expected and observed behaviour, the transformed program generates, by using contradiction removal methods, all possible virtual corrections of the original program. This is due to a one-to-one mapping between the (minimal) diagnoses of the original program and the (minimal) revisions of the transformed one.
These ideas on how debugging and fault-finding relate are new, the attractiveness of the approach being its basis on logic programs. In the same vein that one can obtain a general debugger for normal logic programs, irrespective of the program domain, one can aim at constructing a general fault-finding procedure whatever the faulty artifact may be, just as long as it can be modelled by logic programs not confined to being normal logic programs, but including more expressive extensions such as explicit negation.

However we must still go some way until this ultimate goal can be achieved. The current method applies only to a particular class of normal logic programs where the well-founded model [10] and SLDNF-resolution [20] coincide in meaning. The debugging of programs under wellfounded semantics with explicit negation [1, 22] is also foreseen, where new and demanding problems are yet to be solved. On the positive side, the loop detection properties of well-founded semantics will allow for a declarative treatment of otherwise endless derivations.

We examine here the problem of declarative error diagnosis, or debugging, for the class of normal logic programs, where SLDNF-Resolution can be used to finitely compute all the logic consequences of these programs, i.e. SLDNF-Resolution gives the complete meaning of the program. In the sequel we designate this particular class of programs as source programs.

Well-founded semantics plays this important role in our approach to declarative debugging. By considering only source programs, we guarantee that the well-founded model (WFM) is total5 and equivalent to the model computed by SLDNF-Resolution. In [34], Przymusinski showed that SLDNF-Resolution is sound wrt to well-founded semantics. Thus, for these programs it is indifferent to consider the WFM or Clark’s completion semantics [3] (which characterizes SLDNF).

On the other hand, we intend to further develop this approach, and then deal with the issue of debugging of programs under WFS. By using WFS, loop problems are avoided. Conceivably, we could so debug symptoms in loop-free parts of a normal program under SLDNF, even if some other parts of it have loops.

Last, but not least, the basis of our declarative debugging proposal consists in applying a contradiction removal method we’ve defined for programs under WFSX.

6.1. Declarative Error Diagnosis

Next we present the classical theory of declarative error diagnosis, following mainly [21], in order to proceed to a different view of the issue.

---

5 A well-founded model is total iff all literals are either true or false in it.
Example 6.1 Let $P$ be the (source) program with model $\{ not a, b, not c \}$:

$$a \leftarrow not b \quad b \leftarrow not c$$

Suppose the intended interpretation of $P$ is $I_M = \{ not a, not b, c \}$, i.e. $b$ is a wrong solution, and $c$ a missing solution for $P$ wrt $I_M$. The reader can check, $c$ is an uncovered atom for $P$ wrt $I_M$, and $a \leftarrow not b$ is an incorrect rule for $P$ wrt $I_M$.

6.2. What is Diagnostic Debugging?

We now know, from the previous section (cf. theorem 6.1), that if there is a missing or a wrong solution then there is, at least, an uncovered atom or an incorrect rule for $P$. In classical declarative error diagnosis the complete intended interpretation is always known from the start. Next we characterize the situation where only partial knowledge of the intended interpretation is available but, if possible or wanted, extra information can be obtained. To formalise this debugging activity we introduce two entities: the user and the oracle.

Definition 6.6 (User and Oracle) Let $P$ be a source program and $I_M$ the intended interpretation for $P$. The user is identified with the limited knowledge of the intended model that he has, i.e. a set $U \subseteq I_M$. The oracle is an entity that knows everything, that is, knows the whole intended interpretation $I_M$.

By definition, the user and the oracle share some knowledge and the user is not allowed to make mistakes nor the oracle to lie. The user has a diagnosis problem and poses the queries and the oracle helps the user: it knows the answers to all possible questions. The user may coincide with the oracle as a special case.

Our approach is mainly motivated by the following obvious theorem: if the incorrect rules of a program are removed, and a fact $A$ for each uncovered atom is added to the program, then the model of the new transformed program is the intended interpretation of the original one.

As justified in the section introduction, our approach uses the well-founded semantics to identify the model of programs.

Theorem 6.2 Let $P$ be a source program and $I_M$ its intended interpretation. If $WFMP) \neq I_M$, and

$$Unc = \{ A : A \text{ is an uncovered atom for } P \text{ wrt } I_M \}$$

$$InR = \{ A \leftarrow B : A \leftarrow B \text{ is incorrect for } P \text{ wrt } I_M \}$$

then $WFMP(\text{P-InR} \cup Unc) = I_M$.

Example 6.2 Consider the source program $P$

$$a \leftarrow not b \quad b \leftarrow not c$$

The $WFMP(P)$ is $\{ not a, b, not c \}$. If $I_M = \{ not a, not b, c \}$ is the intended interpretation, then $c$ is an uncovered atom for $P$ wrt $I_M$, and $a \leftarrow not b$ is an incorrect rule for $P$ wrt $I_M$.

The WFM of the new program,

$$b \leftarrow not c \quad c$$

obtained by applying the transformation above, is $I_M$.

Definition 6.7 (Diagnosis) Let $P$ be a source program, $U$ a set of literals of the language of $P$, and $D$ the pair $< Unc, InR >$ where $Unc \subseteq Hp, InR \subseteq P$. $D$ is a diagnosis for $U$ wrt $P$ iff

$$U \subseteq WFMP(P-InR) \cup Unc$$

Example 6.2 (cont.) The diagnoses for $U = \{ not a, c \}$ wrt $P$ are:

$D_1 = \{ b, c \}, \{ \} \quad D_4 = \{ c \}, \{ a \leftarrow not b \}$

$D_2 = \{ b, c \}, \{ a \leftarrow not b \} \quad D_5 = \{ c \}, \{ a \leftarrow not b, b \leftarrow not c \}$

$D_3 = \{ b, c \}, \{ b \leftarrow not c \} \quad D_6 = \{ b, c \}, \{ a \leftarrow not b, b \leftarrow not c \}$

Each one of these diagnoses can be viewed as a virtual correction of the program. For example, $D_1$ can be viewed as stating that if the program is corrected so that $b$ and $c$ become true, by adding them as facts say, then the literals in $U$ also become true. Another possibility is to set $c$ true and correct the first rule of the original program. This possibility is reflected by $D_3$.

However some of these diagnoses are redundant: for instance in $D_6$ there is no reason to consider the second rule wrong; doing so is redundant.

This is even more serious in the case of $D_3$. There, the atom $b$ is considered uncovered and all rules for $b$ are considered wrong.

Definition 6.8 (Minimal Diagnosis) Let $P$ be a source program and $U$ a set of literals. Given two diagnoses $D_1 = \langle Unc_1, InR_1 \rangle$ and $D_2 = \langle Unc_2, InR_2 \rangle$ for $U$ wrt $P$ we say that $D_1 \leq D_2$ if $Unc_1 \cup InR_1 \subseteq Unc_2 \cup InR_2$.

$D$ is a minimal diagnosis for $U$ wrt $P$ if there is no diagnosis $D_1$ for $U$ wrt $P$ such that $D_1 \leq D$. $\{ \}, \{ \}$ is called the empty diagnosis.

Example 6.2 (cont.) The minimal diagnoses for $U = \{ not a, c \}$ wrt $P$ are $D_1$ and $D_5$ above.
Obviously, if the subset of the intended interpretation given by the user is already a consequence of the program, we expect empty to be the only minimal diagnosis: i.e., based on that information no bug is found:

**Theorem 6.3** Let \( P \) be a source program, and \( U \) a set of literals. Then \( U \subseteq \text{WFM}(P) \) iff the only minimal diagnosis for \( U \) wrt \( P \) is empty.

A property of source programs is that if the set \( U \) of user provided literals is the complete intended interpretation (the case when the user knowledge coincides with oracle's), a unique minimal diagnosis exists. In this case the minimal diagnosis uniquely identifies all the errors in the program and provides one correction to all the bugs.

**Theorem 6.4** Let \( P \) be a source program and \( I_M \) its intended interpretation. Then diagnosis \( D = <\text{Unc}, \text{InR}> \) is the unique minimal diagnosis for \( I_M \) wrt \( P \) where

\[
\text{Unc} = \{ \ A : A \text{ is an uncovered atom for } P \text{ wrt } I_M \}
\]

\[
\text{InR} = \{ A \leftarrow B : A \text{ is incorrect for } P \text{ wrt } I_M \}
\]

The next lemma helps us show important properties of minimal diagnosis:

**Lemma 6.5** Let \( P \) be a source program, and \( U_1 \) and \( U_2 \) sets of literals. If \( U_1 \subseteq U_2 \) and if there are minimal diagnosis for \( U_1 \) and \( U_2 \) wrt \( P \) then there is a minimal diagnosis for \( U_1 \text{ wrt } P \) contained in a minimal diagnosis for \( U_2 \text{ wrt } P \).

Let us suppose the set \( U \) provided by the user is a proper subset of the intended interpretation. Then it is expectable that the errors are not immediately detected, in the sense that several minimal diagnoses may exist. The next theorem guarantees that at least one of the minimal diagnoses finds an error of the program.

**Theorem 6.6** Let \( P \) be a source program, \( I_M \) its intended interpretation, and \( U \) a set of literals. If \( U \subseteq I_M \) and if there are minimal diagnosis for \( U \text{ wrt } P \) then there is a minimal diagnosis \((\text{Unc}, \text{InR})\) for \( U \text{ wrt } P \) such that for every \( A \in \text{Unc} \), \( A \) is an uncovered atom for \( P \text{ wrt } I_M \), and for every rule \( A \leftarrow B \in \text{InR}, A \leftarrow B \) is incorrect for \( P \text{ wrt } I_M \).

As a special case, even giving the complete intended interpretation, if one single minimal diagnosis exists then it identifies at least one error.

**Corollary 6.1** Let \( P \) be a source program, \( I_M \) its intended interpretation, and \( U \) a set of literals. If there is a unique minimal diagnosis \((\text{Unc}, \text{InR})\) for \( U \text{ wrt } P \) then for every \( A \in \text{Unc} \), \( A \) is an uncovered atom for \( P \text{ wrt } I_M \), and for every rule \( A \leftarrow B \in \text{InR}, A \leftarrow B \) is incorrect for \( P \text{ wrt } I_M \).

In a process of debugging, when several minimal diagnoses exist, queries should be posed to the oracle in order to enlarge the subset of the intended interpretation provided, and thus refine the diagnoses. Such a process must be iterated until a single minimal diagnosis is found. This eventually happens, albeit when the whole intended interpretation is given (cf theorem 6.4).

**Example 6.2** As mentioned above, the two minimal diagnoses for \( U = \{ \text{not } a, c \} \) wrt \( P \) are \( D_1 = \{ \{ b, c \}, \{ \} \} \) and \( D_3 = \{ \{ c \}, \{ a \leftarrow \text{not } b \} \} \).

By theorem 6.6, at least one of these diagnoses contains errors. In \( D_1 \), \( b \) and \( c \) are uncovered. Thus, if this is the error, not only literals in \( U \) are true but also \( b \). In \( D_3 \), \( c \) is uncovered and rule \( a \leftarrow \text{not } b \) is incorrect. Thus, if this is the error, \( b \) is false.

By asking about the truthfulness of \( b \) one can, in fact, identify the error: e.g. should the answer to such query be yes the set \( U \) is augmented with \( b \) and the only minimal diagnosis is \( D_1 \); should the answer be no \( U \) is augmented with \( \text{not } b \) and the only minimal diagnosis is \( D_3 \).

The issue of identifying disambiguating oracle queries is dealt with in the next section.

In all the results above we have assumed the existence of at least one minimal diagnosis. This is guaranteed because:

**Theorem 6.7** Let \( P \) be a source program, \( I_M \) its intended interpretation, and \( U \) a finite set of literals. If \( \{ U \subseteq I_M \text{ and } U \not\subseteq \text{WFM}(P) \} \) then there is a non-empty minimal diagnosis \( \langle \text{Unc}, \text{InR} \rangle \) for \( U \text{ wrt } P \) such that, for every \( A \in \text{Unc}, A \) is an uncovered atom for \( P \text{ wrt } I_M \), and for every rule \( A \leftarrow B \in \text{InR}, A \leftarrow B \) is incorrect for \( P \text{ wrt } I_M \).

### 6.3. Diagnosis as Revision of Program Assumptions

In this section we show that minimal diagnosis are minimal revisions of a simple transformed program obtained from the original source one. Let's start with the program transformation and some results regarding it.

**Definition 6.9** The transformation \( T \) that maps a source program \( P \) into a source program \( P' \) is obtained by applying to \( P \) the following two operations:

- Add to the body of each rule \( H \leftarrow B_1, \ldots, B_n \not\in C_{i-1}, \ldots, C_{n-1} \) in \( P \) the default literal not incorrect \((H \leftarrow B_1, \ldots, B_n \not\in C_{i-1}, \ldots, C_{n-1} \).
- Add the rule \( p(X_1, X_2, \ldots, X_n) \leftarrow \text{uncovered}(p(X_1, X_2, \ldots, X_n)) \) for each predicate \( p \) with arity \( n \) in the language of \( P \).

It is assumed predicate symbols incorrect and uncovered don't belong to the language of \( P \).
It can be easily shown that the above transformation preserves the truths of $P$: the literals $\text{not incorrect(\ldots)}$ and $\text{uncovered(\ldots)}$ are, respectively, true and false in the transformed program. The next theorem captures this intuitive result.

**Theorem 6.8** Let $P$ be a source program. If $L$ is a literal with predicate symbol distinct from incorrect and uncovered then $L \in \text{WFM}(P)$ iff $L \in \text{WFM}(Y(P))$.

**Example 6.2 (cont.)** By applying transformation $Y$ to $P$ we get

\[
\begin{align*}
a & \rightarrow \text{not b, not incorrect(a \leftarrow \text{not b})} \\
b & \rightarrow \text{not c, not incorrect(b \leftarrow \text{not c})} \\
c & \rightarrow \text{uncovered(c)}
\end{align*}
\]

The reader can check that the WFM of $Y(P)$ is

\[
\{ \text{not a, b, not c, not uncovered(a), not uncovered(b), not uncovered(c),} \\
\text{not incorrect(a \leftarrow \text{not b}), not incorrect(b \leftarrow \text{not c})} \}
\]

A user can employ this transformed program in the same way he did with the original source program, with no change in program behaviour. If he detects an abnormal behaviour of the program, in order to debug the program he then just explicitly states what answers he expects.

**Definition 6.10 (Debugging transformation)** Let $P$ be a source program and $U$ a set of user provided literals. The debugging transformation $Y_{\text{debug}}(P, U)$ converts the source program $P$ into an object program $P'$. $P'$ is obtained by adding to $Y(P)$ the integrity rules $\bot \leftarrow \text{not a}$ for each atom $a \in U$, and $\bot \leftarrow \text{a}$ for each literal not $a \in U$.

Our main result is the following theorem, which links minimal diagnosis for a given set of literals wrt a source program with minimal revisions of the object program obtained by applying the debugging transformation.

**Theorem 6.9** Let $P$ be a source program and $U$ a set of literals from the language of $P$. The pair $\langle \text{Unc, InR} \rangle$ is a diagnosis for $U$ wrt $P$ iff

\[
\{ \text{uncovered(A): A \in \text{Unc}} \} \cup \{ \text{incorrect(A \leftarrow B): A \leftarrow B \in \text{InR}} \}
\]

is a revision of $Y_{\text{debug}}(P, U)$, where not incorrect(\ldots) and not uncovered(\ldots) are all the revisable literals.

The proof is trivial and it is based on the facts that adding a positive assumption incorrect has an effect similar to removing the rule from the program, and adding a positive assumption uncovered makes $A$ true in the revised program. The integrity rules in $Y_{\text{debug}}(P, U)$ guarantee that the literals in $U$ are 'explained'.

**Algorithm 6.1 (Debugging of a source program)**

1. Transformation $Y(P)$ is applied to the program.
2. The user detects the symptoms and their respective integrity rules are inserted.
3. The minimal diagnosis are computed. If there is only one, one error or more are found and reported. Stop.

---

We conjecture that termination occurs, in the worst case, after the first time the oracle is consulted, i.e. the algorithm stops either the first or second time it executes this step.
4. The disambiguating queries are generated and the oracle consulted.

5. Its answers are added in the form of integrity rules.


**Example 6.2 (cont.)** After applying Υ to P the user detects that $b$ is a wrong solution. He causes the integrity rule $\bot \leftarrow b$ be added to $\mathcal{YP}$ and provokes a program revision to compute the possible explanations of this bug. He obtains two minimal revisions: $\{\text{uncovered}(c)\}$ and $\{\text{incorrect}(b \leftarrow \neg c)\}$.

Now, if desired, the oracle is questioned:

- What is the truth value of $c$ in the intended interpretation? Answer: true.

Then the user (or the oracle...) adds to the program the integrity rule $\bot \leftarrow \neg c$ and revises the program. The unique minimal revision is $\{\text{uncovered}(c)\}$ and the bug is found.

The user now detects that solution $a$ is wrong. Then he adds the integrity rule $\bot \leftarrow a$ too and obtains the only minimal revision, that detects all the errors.

$\{\text{incorrect}(a \leftarrow \neg b), \text{uncovered}(c)\}$

**Example 6.3** Consider the slight variation of example 5.4:

\begin{align*}
\text{inv}(T, G, l, 1) & \leftarrow \text{node}(T, l, 0, \text{not}(ab(G))) & 1 \\
\text{inv}(T, G, l, 0) & \leftarrow \text{node}(T, l, 1, \text{not}(ab(G))) & 2 \\
\text{node}(T, b, B) & \leftarrow \text{inv}(T, g1, a, B) & 3 \\
\text{node}(T, c, C) & \leftarrow \text{inv}(T, g2, b, C) & 4 \\
\text{node}(0, a, 0) & \leftarrow \text{node}(0, c, 0) & 5 \\
\neg \text{node}(0, c, 0) & \leftarrow \text{fault}(G, s0) & 6 \\
\text{inv}(T, G, l, V) & \leftarrow \text{node}(T, l, \_), V \neq 0, \text{missing}(G, V) & 7 \\
\bot & \leftarrow \text{fault}(G, M1), \text{fault}(G, M2), M1 \neq M2 & 8 \\
\bot & \leftarrow \neg \text{node}(0, c, 1) & 9
\end{align*}

We made the fault model partial by, withdrawing rules 8 and 9. So that we can still explain all observations, we “complete” the fault model by introducing rule 12, which expresses that in the presence of input to the inverter, and if the value to be explained is not equal to 0 (since that is explained by rule 7), then there is a missing fault mode for value $V$. Of course, missing has to be considered a revisable too. Now the following expected minimal revisions are produced:

$\{ab(g1), \text{fault}(g1, s0)\} \{ab(g2), \text{missing}(g2, 1)\}$

The above fault model ‘completion’ is a general technique for explaining all observations, with the advantage, with respect to [18]’s lenient explanations, that missing fault modes are actually reported. In fact, we are simply debugging the fault model according to the methods of the previous section: we’ve added a rule that detects and provides desired solutions not found by the normal rules, just as in debugging. But also solutions not explained by other fault rules: hence the $V \neq 0$ condition. The debugging equivalent of the latter would be adding a rule to ‘explain’ that a bug (i.e. fault mode) has already been detected (though not corrected). Furthermore, the reason $\text{node}(l, \_)$ is included in 12 is that there is a missing fault mode only if the inverter actually receives input. The analogous situation in debugging would be that of requiring that a predicate must actually ensure some predication about goals for it (e.g. type checking) before it is deemed incomplete.

The analogy with debugging allows us to debug artifact specifications. Indeed, it suffices to employ the techniques of the previous section. By adding $\text{not}(ab(G, R, \text{HeadArguments})$ instead of $\text{not}(ab(G))$ in rules, where $R$ is the rule number, revisions will now inform us of which rules possibly produce wrong solutions that would explain bugs. Of course, we now need to add $\text{not}(ab(G, R))$ to all other rules, but during diagnosis they will not interfere if we restrict the revisables to just those with the appropriate rule numbers. With regard to missing solutions, we’ve seen in the previous paragraph that it would be enough to add an extra rule for each predicate. Moreover the same rule numberning technique is also applicable.

We now come full circle and may rightly envisage a program as just another artifact, to which diagnostic problems, concepts, and solutions, can profitably apply:

**Example 6.4** The (buggy) model of an inverter gate below entails $\text{node}(b, 0)$, and also (wrongly) $\text{node}(b, 1)$, when its input is 1.

\[
\begin{array}{ccc}
1 & a & b \\
\end{array}
\]

\[
\text{inv}(G, l, 0) \leftarrow \text{node}(l, 1, \text{not}(ab(G))) \\
\text{inv}(G, l, 1) \leftarrow \text{node}(l, 1, \text{not}(ab(G))) \% \text{bug: node}(l,0) \\
\text{node}(b, V) \leftarrow \text{inv}(g1, a, V) \\
\text{node}(a, 1)
\]

After the debugging transformation:

\[
\begin{align*}
\text{inv}(G, l, 0) & \leftarrow \text{node}(l, 1, \text{not}(ab(G, 1, \{G, l, 0\}))) \\
\text{inv}(G, l, 1) & \leftarrow \text{node}(l, 1, \text{not}(ab(G, 2, \{G, l, 1\}))) \\
\text{node}(b, V) & \leftarrow \text{inv}(g1, a, V), \text{not}(ab(3, \{0, V\})) \\
\text{node}(a, 1) & \leftarrow \text{not}(ab(4, \{a, V\}))
\end{align*}
\]
Now, adding to it $\perp \leftarrow \text{node}(b, 1)$, and revising the now contradictory program the following minimal revisions are obtained:

$$\{ab(g1, 2; [g1, a, 1])\} \quad \{ab(3, [b, 1])\} \quad \{ab(4, [a, 1])\}$$

The minimal revision $\{ab(g1, 2; [g1, a, 1])\}$ states that either the inverter model is correct and therefore gate 1 is behaving abnormally or that rule 2 has a bug.

7. UPDATING KNOWLEDGE BASES

In this section we exhibit a program transformation to solve the problem of updating knowledge bases. Recall that a logic program stands for all its ground instances.

As stated in [14, 15] the problem of updating knowledge bases is a generalisation of the view update problem of relational databases. Given a knowledge base, represented by a logic program, an integrity constraint theory and a first order formula the updating program consists in updating the program such that:

- It continues to satisfy the integrity constraint theory;
- When the existential closure of the first-order formula is not (resp., is) a logical consequence of the program then, after the update, it becomes (resp., no longer) so.

Here, we restrict the integrity constraint theory to sets of integrity rules (c.f. Sect. 4.2) and the first-order formula to a single ground literal. The method can be generalised as in [15], in order to cope with first-order formulae.

We assume there are just two primitive ways of updating a program: retracting a rule (or fact) from the program or asserting a fact. A transaction is a set of such retraction and assertions.

Next, we define a program transformation in all respects similar to the one used to perform declarative debugging:

**Definition 7.1** The transformation $\gamma$ that maps a logic program $P$ into a logic program $P'$ is obtained by applying to $P$ the following two operations:

- Add to the body of each rule $H \leftarrow B_1, \ldots, B_m; \text{not } C_1, \ldots, \text{not } C_n$ in $P$ the default literal not retract_inst$(H \leftarrow B_1, \ldots, B_m; \text{not } C_1, \ldots, \text{not } C_n)$.
- Add the rule $p(X_1, X_2, \ldots, X_n) \leftarrow \text{assert} \_ \text{inst}(p(X_1, X_2, \ldots, X_n))$ for each predicate $p$ with arity $n$ in the language of $P$.

It is assumed the predicate symbols retract_inst and assert_inst don't belong to the language of $P$. The rewrites of the program $P'$ are the retract_inst and assert_inst literals.

If an atom $A$ is to be inserted in the database $P$, then the integrity rule $\perp \leftarrow \text{not } A$ is added to $\gamma(P)$. The minimal revisions of the latter program and integrity rule are the minimal transactions ensuring that $A$ is a logical consequence of $P$. If an atom $A$ is to be deleted, then add the integrity rule $\perp \leftarrow A$ instead. With this method the resulting transactions are more 'intuitive' than the ones obtained by [15]:

**Example 7.1** [15] Consider the following logic program and the request to make pleasant(fred) a logical consequence of it (insertion problem):

- pleasant(X) $\leftarrow$ not old(X), likes_fun(X)
- pleasant(X) $\leftarrow$ sports_person(X), loves_nature(X)
- sports_person(X) $\leftarrow$ swimmer(X)
- sports_person(X) $\leftarrow$ not sedentary(X)
- old(X) $\leftarrow$ age(X, Y), Y > 55
- swimmer(fred)
- age(fred, 60)

The transactions returned by Guessoun and Lloyd's method are:

1. $\{\text{assert(pleasant(fred))}\}$
2. $\{\text{assert(likes_fun(fred)), retract((old(X) $\leftarrow$ age(X, Y), Y > 55))}\}$
3. $\{\text{assert(likes_fun(fred)), retract(age(fred,60)))}\}$
4. $\{\text{assert(sports_person(fred)), assert(love_nature(fred)))}\}$
5. $\{\text{assert(swimmer(fred)), assert(love_nature(fred)))\}$
6. $\{\text{assert(love_nature(fred)))\}$

Notice that transactions 4 and 5 asserts facts (sports_person(fred), resp. swimmer(fred)) that are already conclusions of the program $\gamma$. Also remark that in transaction 2 the whole rule is being retracted from the program, rather than just the appropriate instance. On the contrary, our method returns the transactions:

1. $\{\text{assert_inst(pleasant(fred))}\}$
2. $\{\text{assert_inst(likes_fun(fred)), retract_inst((old(fred) $\leftarrow$ age(fred,60),60>55))}\}$
3. $\{\text{assert_inst(likes_fun(fred)), retract_inst(age(fred,60)))\}$
4. $\{\text{assert_inst(love_nature(fred)))\}$
If the second transition is added to the program then it is not necessary to remove the rule \( \text{old}(X) \leftarrow \text{age}(X, Y), Y > 55 \) from it. Only an instance of the rule is virtually retracted via assertion of the fact retract\_inst\_age(fred,60)).

Another advantage of our technique is that the user can express which predicates are liable to retraction of rules and addition of facts by only partially transforming the program, i.e. by selecting to which rules the not retract is added, or to which predicates the second rule in the transformation is applied.

In [14] is argued that the updating procedures should desirably return minimal transactions, capturing the sense of making 'least' changes to the program. These authors point out a situation where minimal transactions do not obey the integrity constraint theory:

**Example 7.2** [14] Consider the definite logic program from where \( r(a) \) must not be a logical consequence of it (the deletion problem):

\[
\begin{align*}
   r(X) & \leftarrow p(X) & \quad p(a) \\
   r(X) & \leftarrow p(X), q(X) & \quad q(a)
\end{align*}
\]

and the integrity constraint theory \( \forall x (p(x) \leftarrow q(x)) \). Two of the possible transactions that delete \( r(a) \) are:

\[
T_1 = \{ \text{retract}(p(a)) \} \quad \text{and} \quad T_2 = \{ \text{retract}(p(a)), \text{retract}(q(a)) \}
\]

Transaction \( T_1 \) is minimal but the updated program does not satisfy the integrity constraints theory. On the contrary, the updated program using \( T_2 \) does satisfy the integrity constraint theory.

With our method, we first apply \( T \) to the program, obtaining (notice how the integrity constraint theory is coded):

\[
\begin{align*}
   r(X) & \leftarrow p(X), \text{not retract\_inst}(r(X) \leftarrow p(X)) \\
   r(X) & \leftarrow p(X), q(X), \text{not retract\_inst}(r(X) \leftarrow p(X), q(X)) \\
   p(a) & \leftarrow \text{not retract\_inst}(p(a)) \\
   q(a) & \leftarrow \text{not retract\_inst}(q(a)) \\
   p(X) & \leftarrow \text{assert\_inst}(p(X)) \\
   q(X) & \leftarrow \text{assert\_inst}(q(X)) \\
   r(X) & \leftarrow \text{assert\_inst}(r(X)) \\
   \perp & \leftarrow \text{not}(p(X), q(X))
\end{align*}
\]

Remark that transaction \( T_1 \) is not a minimal revision of the previous program.

Due to the uniformity of the method, i.e. both insert and delete requests are translated to integrity rules, the iterative contradiction removal algorithm ensures that the minimal transactions obtained, when enacted, do satisfy the integrity constraints.

---

* It may be argued that we obtain this result because we consider only ground instances. In fact, we have devised a sound implementation of the contradiction removal algorithm that is capable of dealing with non-ground logic programs such as this one. For the above example the transactions obtained are the ones listed.
REFERENCES


[34] 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.


