Debugging by Diagnosing Assumptions

Luís Moniz Pereira, Carlos Viegas Damásio, José Júlio Alferes

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

Abstract. We present a novel and uniform technique for normal logic program declarative error diagnosis. We lay down the foundations on a general approach to diagnosis using logic programming, and bring out the close relationship between debugging and fault–finding.
Diagnostic debugging can be enacted by contradiction removal methods. It relies on a simple program transformation to provide a contradiction removal approach to debugging, based on revising the assumptions about predicates' correctness and completeness.
The contradiction removal method is justified in turn in terms of well–founded semantics. The loop detection properties of well–founded semantics will allow in the future for a declarative treatment of otherwise endless derivations. The debugging of programs under well–founded semantics with explict negation is also foreseen.
Here, we apply our techniques to finite SLDNF derivations, whose meaning coincides with the well–founded model, for which our contradiction removal method and algorithm is sound and complete. Several examples illustrate the algorithm at work.

1 Introduction

It is clear that fault-finding or diagnosis is akeen 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 causes, 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

* We thank JNICT and Esprit BR project Compulog 2 (no 6810) for their support.

down the foundations on a general approach to diagnosis using logic programming. By 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 [4].

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 [4] 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 incorrections are diagnosed but not perfected, i.e. sufficient corrections are made to the program but only virtually. This process, of performing sufficing virtual corrections is at the kernel 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. Whenever 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 [8]. 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 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 very same methods can be applied to the updating of knowledge bases with integrity constraints represented as logic programs. By only partially transforming the program the user can express which predicates are liable to retraction of of rules and addition of facts. The iterative contradiction removal algorithm of [8] ensures that the minimal transactions thus obtained do satisfy the integrity constraints.

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 go a long 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 [11] and SLDNF–resolution [3] coincide in meaning. The debugging of programs under well–founded semantics with explicit negation [5] is also foreseen, where new and demanding problems are yet to be solved. The loop detection properties of well–founded semantics will allow for a declarative treatment of otherwise endless derivations.

In the subsequent section we clearly state to what programs our method is applicable. The next two sections briefly review, respectively, our work in two-valued contradiction removal and Lloyd's rendering of classical declarative error diagnosis. In section 5 we describe the diagnostic debugging theory and enounce its attending theorems. In section 6 we present the above mentioned program transformation. A debugging algorithm is also produced. Finally, the relationship between debugging and diagnosis is displayed via a simple logical circuit. Proofs are omitted for brevity but can be found in an extended version of this paper.

2 Scope of Application

Given a first order language *Lang*, a (normal) logic program is a set of rules of the form:

$$H \leftarrow B_1, \ldots, B_n, not\, C_1, \ldots, not\, C_m \quad (m \geq 0, n \geq 0)$$

where $H, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms in *Lang*. H is an alternative representation of rule $H \leftarrow$. Each logic program rule stands for all its ground instances wrt *Lang*. Given program P we denote by $\mathcal{H}_\mathcal{P}$ (or sometimes $\mathcal{H}$) its Herbrand base.

We here examine 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.

By considering only source programs, we guarantee that the well-founded model (WFM) is total[2] and equivalent to the model computed by SLDNF-Resolution. In [9] Przymusinsky showed that SLDNF-Resolution is sound wrt to well–founded semantics.

Well–founded semantics plays an important rôle in our approach to declarative debugging. Indeed, on the one hand, by considering only source programs, it is guaranteed that the WFM is total and equivalent to the model computed by SLDNF-Resolution. Thus, for these programs is indifferent to consider the WFM or Clark's completion semantics (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 WFS, to a transformed (object) programs require extending the class of normal logic programs with integrity rules of the form:

$$\perp \leftarrow B_1, \ldots, B_n, not\, C_1, \ldots, not\, C_m \quad (m \geq 0, n \geq 0)$$

where $B_1, \ldots, B_n,\ C_1, \ldots, C_m$ are atoms. Contradictory programs are those where $\perp$ belongs to the program model M_P. Programs with integrity rules are liable to be contradictory.

3 How to Revise your Assumptions

First we present a rephrased review of some results of [8], that will be of use in order to obtain a declarative error diagnosis theory of source programs by virtue of a contradiction removal process exercised over a transformed version of them.

Example 1. Consider $P = \{a \leftarrow not\, b; \perp \leftarrow a\}$. Since we have no rules for b, $not\, b$ is true by Closed World Assumption (CWA) on b. Hence, by the second rule, we have a contradiction. We argue the CWA may not be held of atom b as it leads to contradiction.

The first issue in contradiction removal is to define exactly which negative literals $not\, A$ alone, true by CWA, may be *revised* to false by adding A. Contradiction removal is achieved by adding to the original program the complements[3] of some revisable literals.

Definition 1 (Revisables). The revisables of a program P are a subset of $NoRules(P)$, the set of literals of the form $not\, A$, with no rules for A in P.

[2] A well-founded model is total iff all literals are either true or false in it.

[3] The complement of atom L is $not\, L$, and of literal $not\, L$ is L.

Definition 2 (Positive assumptions of a program). A set A of atoms is a set of positive assumptions of program P if for every $a \in A$, $not\ a$ is a revisable.

Definition 3 (Revised program wrt positive assumptions). Let A be a set of positive assumptions of P. The revised P wrt A, $Rev(P, A)$, is the program $P \cup A$.

Definition 4 (Violation of an integrity rule by a program). An integrity rule $\perp \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ is violated by program P iff

$$P \models B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m.$$

Definition 5 (Revision of a set of integrity rules). A program revision to satisfy a set of integrity rules S in P is a (possibly empty) set of positive assumptions A s.t. $Rev(P, A)$ does not violate any integrity rule in S.

Definition 6 (Revision of a program). A set of positive assumptions A is a **revision** of P iff $Rev(P, A) \not\models \perp$, i.e. A is a revision of the set of all integrity rules of P.

Example 2. Consider contradictory program P:

a $\leftarrow$not b, not c.	$\perp\leftarrow$a, a$'$.
a$'\leftarrow$not d.	$\perp\leftarrow$b.
c $\leftarrow$e.	$\perp\leftarrow$d, not f.

Intuitively literals $not\ b$, $not\ d$ and $not\ e$ are true by CWA, entailing a and a', and thus $\perp$ via violation of the integrity rule $\perp \leftarrow a, a'$.

The revisions of the above program are $\{e\}$, $\{d, f\}$, $\{e, f\}$ and $\{d, e, f\}$. The minimal ones are $\{e\}$ and $\{d, f\}$.

Next we identify which subsets of the revisables support contradiction via violation of some of the integrity rules, in the sense that the revision of elements from each such subsets (i.e. adding the corresponding positive assumptions) definitely eliminates the introduction of $\perp$ via those integrity rules that were being violated. But the revision may introduce fresh contradiction now via distinct, newly violated,integrity rules. If that is the case, then the revision process is just simply iterated. We are interested only in the overall minimal revisions.

To do so, we first define support in the WFM; informally a support set of literal L is the set of nodes in a derivation for L :

Definition 7 (Support set of a literal). Support sets of literal L true in the well–founded model M_P of a program P are denoted by $SS(L)$, always exist, and are obtained as follows:

1. When L is an atom, then for each rule $L \leftarrow \mathcal{B}$[4] in P such that $\mathcal{B}$ is true in M_P, each $SS(L)$ is formed by the union of $\{L\}$ with some $SS(B_i)$ for each $B_i \in \mathcal{B}$. There are as many $SS(L)$ as ways of making true some rule body for L.

[4] $H \leftarrow \mathcal{B}$ is alternative rule notation, where set $\mathcal{B}$ is the set of literals in its body.

2. When L is a *not* A literal:
   (a) if no rules exist for A in P then the single support set of L is $\{not\ A\}$.
   (b) if rules for A exist in P then choose from each rule with non-empty body a single literal whose complement is true in M_P. For each such multiple choice there are several $SS(not\ A)$, each formed by the union of $\{not\ A\}$ with an SS of the complement of every literal of a multiple choice. There are as many $SS(not\ A)$ as minimal ways of making false all rule bodies for A.

The revisables on which contradiction rests are those in some support of $\perp$:

Definition 8 (Assumption set of $\perp$ wrt revisables). An assumption set of $\perp$ wrt revisables R is any set $AS(\perp, R) = SS(\perp) \cap R$, of negative literals, for some $SS(\perp)$.

Definition 9 (Hitting set). A hitting set of a collection of sets C is a set formed by the union of one non-empty subset from each $S \in C$. A hitting set is minimal iff no proper subset is a hitting set for C. If $\{\} \in C$ then C has no hitting sets.

We revise programs (resp. minimally) by revising the literals of (resp. minimally) hitting sets:

Definition 10 (Contradiction removal set). A contradiction removal set of P wrt revisables R is a minimal hitting set of the collection of all assumption sets $AS(\perp, R)$ supporting $\perp$.

A program is not revisable if $\perp$ has a support set without revisable literals.

Based on the above, we have devised an iterative algorithm to compute the minimal revisions of a program P wrt to revisables R, and shown its soundness and completeness for finite R. The algorithm is a repeated application of an algorithm to compute contradiction removal sets.[5]

The algorithm starts by finding out the $CRSs$ of the original program plus the empty set of positive assumptions (assuming the original program is revisable, otherwise the algorithm stops after the first step). To each CRS there corresponds a set of positive assumptions obtained by taking the complement of their elements. The algorithm then adds, non-deterministically and one at a time, each of these sets of assumptions to the original program. One of three cases occurs: (1) the program thus obtained is non-contradictory and we are in the presence of one minimal revising set of assumptions; (2) the new program is contradictory and non-revisable (and this fact is recorded by the algorithm to prune away other contradictory programs obtained by it); (3) the new program

[5] [10] gives an "algorithm" for computing minimal diagnosis, called DIAGNOSE (with a bug corrected in [2]). DIAGNOSE can be used to compute $CRSs$, needing only the redefinition of the function Tp refered there. Our Tp can be built from a top-down derivation procedure adapted from [7, 6].

is contradictory but revisable, and this very same algorithm is iterated until we finitely attain one of the two other cases. For the formal description see Alg. 1.

The sets of assumptions employed to obtain the revised non-contradictory programs are the minimal revisions of the original program. The algorithm can terminate after executing only one step when the program is either non-contradictory, or contradictory but non-revisable. It can be shown this algorithm is (at least) NP-complete [1].

Algorithm 1 (Minimal revisions of a program).

Input: A possibly contradictory program P and a set R of revisables.

$AS_0 := \{\{\}\}$
$Cs := \{\}$
$i := 0$
repeat
    $AS_{i+1} := \{\}$
    for each $A \in AS_i$
        if $\neg\exists C \in Cs : C \subseteq A$
            if $Rev(P, A) \models \bot$
                if $Rev(P, A)$ is revisable
                    for each $CRS_j(R)$ of $P \cup A$
                        Let $NAs := A \cup not\ CRS_j(R)$
                        $AS_{i+1} := AS_{i+1} \cup \{NAs\}$
                    endfor
                else
                    $Cs := MinimalSetsOf(Cs \cup \{A\})$
                endif
            else
                $AS_{i+1} := AS_{i+1} \cup \{A\}$
            endif
        endif
    endfor
    $AS_{i+1} := MinimalSetsOf(AS_{i+1})$
    $i := i + 1$
until $AS_i = AS_{i-1}$.

Output: AS_i, the collection of all minimal revisions of P wrt R.

Example 2 (cont). The integrity rule $\bot \leftarrow a, a'$ is violated. By adding to P any of the sets of assumptions $\{b\}$, $\{d\}$, or $\{e\}$, this rule becomes satisfied.

Program $Rev(P, \{e\})$ is non-contradictory: thus $\{e\}$ is a revision of P. But $Rev(P, \{b\})$ and $Rev(P, \{d\})$ still entail $\bot$, respectively violating integrity rules $\bot \leftarrow b$ and $\bot \leftarrow d, not\ f$. In $Rev(P, \{b\})$ integrity rule $\bot \leftarrow b$ cannot be satisfied:

$\{b\}$ is not a revision. In $Rev(P, \{d\})$ integrity rule can be satisfied by adding to $\{d\}$ the assumption f, to obtain also the revision $\{d, f\}$ (cf. Fig. 1).

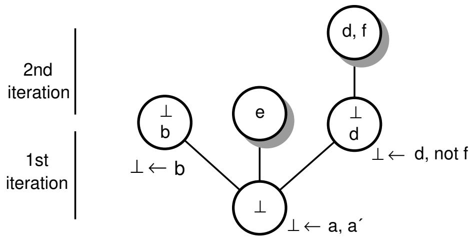

Fig. 1. Revision of example 2.

In [8] we apply these techniques to solve a rather general class of diagnosis problems.

4 Declarative Error Diagnosis

Next we present the classical theory of declarative error diagnosis, following mainly [4], in order to proceed to a different view of the issue.

It would be desireable that a program gave all and only the correct answers to a user's queries. Usually a program contains some bugs that must be corrected before it can produce the required behaviour.

Let the meaning of a logic program P be given by the normal Herbrand models for $comp(P)$, Clark's completion of P. Let the ultimate goal of a program be for its meaning to respect the user's intended interpretation of the program.

Definition 11 (Intended interpretation [4]). Let P be a program. An intended interpretation for P is a normal Herbrand interpretation for $comp(P)$.

Definition 12 (Program correct wrt intended interpretation [4]). A logic program P is correct wrt to an intended interpretation I_M iff I_M is a model for $comp(P)$.

Errors in a terminating logic program manifest themselves through two kinds of symptoms (we deliberately ignore here the question of loop detection).

Definition 13 (Symptoms). Let P be a logic program, I_M its intended interpretation, and A an atom in the Herbrand base of P.

– if $P \vdash_{SLDNF} A$ and $A \notin I_M$ then A is a *wrong solution* for P wrt I_M.

– if $P \not\vdash_{SLDNF} A$ and $A \in I_M$ then A is a *missing solution* for P wrt I_M.

Of course, if there is a missing or a wrong solution then the program is not correct wrt its intended interpretation, and therefore there exists in it some bug requiring correction. In [4] two kinds of errors are identified: uncovered atoms and incorrect clause instances. As we deal with ground programs only, we prefer to designate as incorrect rules the latter type of error.

Definition 14 (Uncovered atom). Let P be a program and I_M its intended interpretation. An atom A is an uncovered atom for P wrt I_M iff $A \in I_M$ but for no rule $A \leftarrow W$ in P, $I_M \models W$.

Definition 15 (Incorrect rule). Let P be a program and I_M its intended interpretation. A rule $A \leftarrow W$ is incorrect for P wrt I_M iff $A \notin I_M$ and $I_M \models W$.

Theorem 1 (Two types of bug only [4]). *Let P be a program and I_M its intended interpretation. P is incorrect wrt I_M iff there is an uncovered atom for P wrt to I_M or there is an incorrect rule for P wrt to I_M.*

Thus, if there is a missing or a wrong solution there is, at least, an uncovered atom or an incorrect rule for P.

Example 3. Let P be the (source) program with model $\{not\, a, b, not\, c\}$:

$$\mathrm{a} \leftarrow \mathrm{not\ b.} \quad \mathrm{b} \leftarrow \mathrm{not\ c.}$$

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.

5 What is Diagnostic Debugging?

We now know, from the previous section (cf. theorem 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 16 (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[6] are removed, and a fact A for each uncovered atom A is added to the program, then the model of the new transformed program is the intended interpretation of the original one.

As justified in section 2, our approach uses the well–founded semantics to identify the model of programs.

Theorem 2. *Let P be a source program and I_M its intended interpretation. If $WFM(P) \neq I_M$, and*

$$\begin{aligned} Unc &= \{\quad A \quad : \ A \ is\ an\ uncovered\ atom\ for\ P\ wrt\ I_M\} \\ InR &= \{A \leftarrow B\!: \ A \leftarrow B\ is\ incorrect\ for\ P\ wrt\ I_M\} \end{aligned}$$

then $WFM((P - InR) \cup Unc) = I_M$.

Example 4. Consider the source program P

$$\mathrm{a{\leftarrow}not\ b.}\quad \mathrm{b{\leftarrow}not\ c.}$$

The $WFM(P)$ 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,

$$\mathrm{b{\leftarrow}not\ c.}\quad \mathrm{c.}$$

obtained by applying the transformation above, is I_M.

The user may have only limited knowledge of the intended interpretation.

Definition 17 (Diagnosis). Let P be a source program, U a set of literals of the language of P, and D the pair $\langle Unc, InR\rangle$ where $Unc \subseteq \mathcal{H}_{\mathcal{P}}$, $InR \subseteq P$.

D is a diagnosis for U wrt P iff

$$U \subseteq WFM((P - InR) \cup Unc).$$

Example 4 (cont). The diagnoses for $U = \{not\, a, c\}$ wrt P are:

$$\begin{array}{ll} D_1 = \langle\{b,c\}, \{\}\rangle & D_5 = \langle\{c\}, \{a \leftarrow not\, b\}\rangle \\ D_2 = \langle\{b,c\}, \{a \leftarrow not\, b\}\rangle & D_6 = \langle\{c\}, \{a \leftarrow not\, b;\ b \leftarrow not\, c\}\rangle \\ D_3 = \langle\{b,c\}, \{b \leftarrow not\, c\}\rangle & \\ D_4 = \langle\{b,c\}, \{a \leftarrow not\, b;\ b \leftarrow not\, c\}\rangle & \end{array}$$

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

[6] In this section program means source program, unless stated otherwise.

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_5.

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

D is a minimal diagnosis for U wrt P iff there is no diagnosis D_1 for U wrt P such that $D_1 \preceq D$. $\langle \{\}, \{\} \rangle$ is called the empty diagnosis.

Example 4 (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 3. *Let P be a source program, and U a set of literals. Then $U \subseteq WFM(P)$ 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 4. *Let P be a source program and I_M its intended interpretation. Then $D = \langle Unc, InR \rangle$, where*

$$\begin{aligned} Unc &= \{\quad A \quad : A \text{ is an uncovered atom for } P \text{ wrt } I_M\} \\ InR &= \{A \leftarrow B \colon\ A \leftarrow B \text{ is incorrect for } P \text{ wrt } I_M\} \end{aligned}$$

is the unique minimal diagnosis for I_M wrt P.

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

Lemma 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 wrt P contained in a minimal diagnosis for U_2 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 imediately 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. *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 wrt P then there is a minimal diagnosis $\langle Unc, InR \rangle$ for U wrt P such that for every $A \in Unc$, A is an uncovered atom for P wrt I_M, and for every rule $A \leftarrow B \in InR$, $A \leftarrow B$ is incorrect for P 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 7. *Let P be a source program, I_M its intended interpretation, and U a set of literals. If there is a unique minimal diagnosis $\langle Unc, InR \rangle$ for U wrt P then for every $A \in Unc$, A is an uncovered atom for P wrt I_M, and for every rule $A \leftarrow B \in InR$, $A \leftarrow B$ is incorrect for P wrt I_M.*

In a process of debugging, when several minimal diagnoses exists, 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 4).

Example 4 (cont). As mentioned above, minimal diagnoses for $U = \{not\, a, c\}$ wrt P are $D_1 = \langle \{b, c\}, \{\} \rangle$ and $D_5 = \langle \{c\}, \{a \leftarrow not\, b\} \rangle$.

By theorem 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_5, c is uncovered and rule $a \leftarrow not\, b$ 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 $not\, b$ and the only minimal diagnosis is D_5.

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

6 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 19. The transformation Υ 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 $H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ in P the literal $not\ incorrect(H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m)$.
- Add the rule $p(X_1, X_2, \ldots, X_n) \leftarrow uncovered(p(X_1, X_2, \ldots, X_n))$ 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 $not\ incorrect(\ldots)$ and $uncovered(\ldots)$ are, respectively, true and false in the transformed program. The next theorem captures this intuitive result.

Theorem 9. *Let P be a source program. If L is a literal with predicate symbol distinct from incorrect and uncovered then $L \in WFM(P)$ iff $L \in WFM(\Upsilon(P))$.*

Example 4 (cont). By applying transformation Υ to P we get

```
a←not b, not incorrect(a ← not b) .  a←uncovered(a).
b←not c, not incorrect(b ← not c) .  b←uncovered(b).
                                     c←uncovered(c).
```

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

$$\{not\ a, b, not\ c, not\ uncovered(a), not\ uncovered(b), not\ uncovered(c), \\ not\ incorrect(a \leftarrow not\ b), not\ incorrect(b \leftarrow not\ c) \quad \}$$

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 20 (Debugging transformation). Let P be a source program and U a set of user provided literals. The debugging transformation $\Upsilon_{debug}(P, U)$ converts the source program P into an object program P'. P' is obtained by adding to $\Upsilon(P)$ the integrity rules $\bot \leftarrow not\ a$ for each atom $a \in U$, and $\bot \leftarrow a$ 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 to a source program with minimal revisions of the object program obtained by applying the debugging transformation.

Theorem 10. *Let P be a source program and U a set of literals from the language of P. The pair $\langle Unc, InR\rangle$ is a diagnosis for U wrt P iff*

$$\{uncovered(A) : A \in Unc\} \cup \{incorrect(A \leftarrow B) : A \leftarrow B \in InR\}$$

is a revision of $\Upsilon_{debug}(P, U)$, where the revisables are all the $not\ incorrect(\ldots)$ and $not\ uncovered(\ldots)$ 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 $uncovered(A)$ makes A true in the revised program. The integrity rules in $\Upsilon_{debug}(P, U)$ guarantee that the literals in U are "explained".

For finite U, algorithm 1 can be used to compute the minimal diagnosis for the buggy source program.

Theorem 11 (Correctness). *Let P be a source program, I_M its intended interpretation, and U a finite set of literals. Algorithm 1 is sound and complete wrt the minimal revisions of $\Upsilon_{debug}(P, U)$, using as revisables all the not $incorrect(\_)$ and not $uncovered(\_)$ literals.*

Corollary 12. *Let P be a source program, I_M its intended interpretation, and U a finite set of literals. If $U \subseteq I_M$ and $U \not\subseteq WFM(P)$ then there is a non-empty minimal revision R of $\Upsilon_{debug}(P, U)$, using as revisables all the not $incorrect(\_)$ and not $uncovered(\_)$ literals, such that for every $uncovered(A) \in R$, A is an uncovered atom for P wrt I_M, and for every $incorrect(A \leftarrow B) \in R$, $A \leftarrow B$ is incorrect for P wrt I_M.*

From all minimal revisions a set of questions of the form "What is the truth value of $<AN\ ATOM>$ in the intended interpretation ?" can be compiled. The oracle answers to these questions identify the errors in the program.

Definition 21 (Disambiguating queries). Let $D = \langle Unc, InR \rangle$ be a diagnosis for finite set of literals U wrt to the source program P, I_M its intended interpretation, and let (the set of atoms)

$$Query = (Unc \cup Atom_{InR}) - U$$

where $Atom_{InR}$ is the set of all atoms appearing in rules of InR.

The set of disambiguating queries of D is:

$$\{What\ is\ the\ truth\ value\ of\ A\ in\ I_M? \mid A \in Query\}$$

The set of disambiguating queries of a set of diagnoses is the union of that for each diagnosis.

Now the answers to the disambiguating questions to the set of all diagnoses, given by the oracle, can be added to the current knowledge of the user, i.e. atoms answered true are added to U, and for atoms answered false their complements are instead. The minimal diagnoses of the debugging transformation with the new set U are then computed and finer information about the errors is produced. This process of generating minimal diagnoses, and of answering the disambiguating queries posed by these diagnoses can be iterated until only one final minimal diagnosis is obtained:

Algorithm 2 (Debugging of a source program).

1. Transformation $\Upsilon(P)$ 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[7].
4. The disambiguating queries are generated and the oracle consulted.
5. Its answers are added in the form of integrity rules.
6. Goto 3.

Example 4 (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 $\Upsilon(P)$ and provokes a program revision to compute the possible explanations of this bug. He obtains two minimal revisions: $\{uncovered(c)\}$ and $\{incorrect(b \leftarrow not\ c)\}$.

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 $\bot \leftarrow not\ c$ and revises the program. The unique minimal revision is $\{uncovered(c)\}$ 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.

$$\{incorrect(a \leftarrow not\ b), uncovered(c)\}$$

7 Debugging via Fault Finding and vice–versa

Here we illustrate with an example the close relationship between debugging and fault finding.

Example 5. Consider the circuit of figure 2.

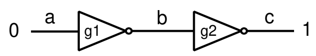

Fig. 2. A two inverters circuit.

We model them with the program:

[7] 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.

```
inv(G,I,1) ← node(I,0), not ab(G).                                1
inv(G,I,0) ← node(I,1), not ab(G).                                2
inv(G,_,0) ← node(I,_), fault_mode(G,s0).                         3
inv(G,_,1) ← node(I,_), fault_mode(G,s1).                         4
node(b,B) ← inv( g1, a, B ).                                      6
node(c,C) ← inv( g2, b, C ).                                      7
⊥          ← fault_mode( G, s0 ), fault_mode( G, s1 ).            8
node(a,0).                                                        9
⊥          ← node(c,0).                                           10
⊥          ← not node(c,1).                                       11
```

Rules 1-2 model normal inverter behaviour. Rules 6-7 specify the circuit topology. Rules 3-4 model two fault modes: one expresses the output is stuck at 0, and the other that it is stuck at 1, whatever the input may be (although it must exist). According to rule 8 the two fault modes are mutually exclusive. Rule 9 establishes the input as 0. Rule 11 specifies the observed output is 1, and that it must be explained (i.e. proved) on pain of contradiction. Rule 10 specifies that the expected output 0 does not obtain, and so is not to be proven. Let the revisables be ab and $fault\_mode$.

Explanation is to be provided by finding the revisions of revisables that added to the program avoid contradiction. Indeed, contradiction ensues if all CWAs are assumed.[8]

The following expected minimal revisions are produced by algorithm 1:

$$\{ab(g1), fault\_mode(g1, s0)\}\ \{ab(g2), fault\_mode(g2, s1)\}$$

Consider next that we make the fault model only partial by, withdrawing rule 4. So that we can still explain all observations, we “complete” the fault model by introducing rule 5 below, 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 3), then there is a missing fault mode for value V. Of course, *missing* has to be considered a revisable too.

```
inv(G,_,V) ←node(I,_), not equal(V,0), missing(G,V).   5
equal(V,V).                                            12
```

Now the following expected minimal revisions are produced:

$$\{ab(g1), fault\_mode(g1, s0)\}\ \{ab(g2), missing(g2, s1)\}$$

The above fault model “completion” is a general technique for explaining all observations reporting the missing fault modes. In fact, we are simply debugging the fault model according to the methods above: we've added a rule

[8] Comments: In rules 3-4, not ab(G) is absent (but could be added) because we presuppose, for simplicity, that the revision of a *fault_mode* to true implicitly signals an abnormality. Rule 8 is not strictly necessary since fault model rules 3-4 already make the fault modes incompatible.

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 not equal(V,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 $node(I, \_)$ is included in 5 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 some 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 in fault finding. Indeed, it suffices to employ the techniques of the previous section. By adding $not\, incorrect(\ldots, G)$ instead of $not\ ab(G)$ in rules, revisions will now inform us of which rules possibly produce wrong solutions that would explain bugs. Of course, we now need to add $not\, incorrect(\ldots, G)$ to all other rules, but during diagnosis they will not interfere if we restrict the revisables to just those for the original rules. With regard to uncovered atoms, we've seen in the previous paragraph that it would be enough to add an extra rule for each predicate.

References

1. M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman and Co., 1979.
2. R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41:79–88, 1989.
3. J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer–Verlag, 1984.
4. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
5. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *Proc. ECAI'92*, pages 102–106. John Wiley & Sons, 1992.
6. L. M. Pereira, J. J. Alferes, and C. Damásio. The sidetracking principle applied to well founded semantics. In *Proc. Simpósio Brasileiro de Inteligência Artificial SBIA'92*, pages 229–242, 1992.
7. L. M. Pereira, J.N. Aparício, and J. J. Alferes. Derivation procedures for extended stable models. In *Proc. IJCAI-91.* Morgan Kaufmann, 1991.
8. 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 Logic Programming and NonMonotonic Reasoning.* MIT Press, 1993.
9. 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.
10. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–96, 1987.
11. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Jouranel of ACM*, pages 221–230, 1990.