Diagnosis and Debugging as Contradiction Removal in Logic Programs

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 apply to normal logic programs with integrity rules a contradiction removal approach, and use it to uniformly treat diagnosis and debugging, and as a matter of fact envisage programs as artifacts and fault-finding as debugging. Our originality resides in applying to such programs the principle that if an assumption leads to contradiction then it should be revised: assumptions are **not A** literals with no rules for A; contradiction is violation of an integrity rule; and revision consists in assuming A instead. Since revised assumptions may introduce fresh contradictions the revision process must be iterated. To do so we've devised an algorithm which is sound and complete.
Our use of normal logic programs extends that of Horn programs made by Konolige, and so adds expressiveness to the causal part of his framework. Non-abnormalities are assumed rather than abduced, and are revised only if they result in contradiction; simple logic programming techniques achieve it.

Keywords: Diagnosis, Debugging, Non-monotonic Reasoning, Logic Programming

1 Introduction

There is evidence that non-monotonic reasoning problems can be solved with recourse to contradiction removal techniques in logic programs extended with a second (explicit) negation [10, 11, 12]. Here we adapt to normal logic programs with integrity rules the contradiction removal approach, and use it to uniformly treat diagnosis and debugging, in fact envisaging programs as artifacts, and fault-finding as debugging.

Our originality resides in applying to normal logic programs the principle that if an assumption leads to a contradiction then it should be revised: assumptions are **not A** literals with no rules for A; contradiction is violation of an integrity rule; and revision consists in assuming A instead. Since revised assumptions may introduce fresh contradictions, the revision process must be iterated.

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

We also set forth a novel simple method to debug logic programs [9], showing how debugging can be envisaged as contradiction removal. Thus, exactly the same technique can be employed to first debug the blueprint specification of an artifact, and used next to perform diagnoses with the debugged specification.

Because [16] relies on Generalised Stable Models [7] to define diagnoses, they are at odds to produce an algorithm to compute minimal ones, as they themselves acknowledge (p. 520). Instead of requiring a new semantics we view diagnoses as an iterative program update process.

Our use of normal logic programs extends that of Horn programs made by [8], and so adds expressiveness to the causal part of his framework. Non-abnormalities are assumed rather than abduced, and are revised only if they result in contradiction; simple logic programming techniques achieve it.

All examples and algorithms were implemented and tested using Prolog.

2 Revising Contradictory Logic Programs

A logic program is a normal logic program plus integrity rules. A logic program is a set of rules and integrity rules having the form

$$H \leftarrow B_1, \ldots, B_n, not\, C_1, \ldots, not\, C_m \quad (m \geq 0, n \geq 0)^2$$

where $H, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms (or positive literals), and in integrity rules H is the symbol $\perp$ (*contradiction*). **not L** is called a default or negative literal. Literals are either positive or default ones. A set of rules stands for all its ground instances. Contradictory programs are those where $\perp$ belongs to the program model M_P. Programs with integrity rules are liable to be contradictory.

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.

To remove contradiction the first issue is defining which default literals **not A**, true by CWA, may be **revised** to false, i.e. by adding A. Contradiction removal is achieved by adding to the original program the complements of some revisables.

Definition 1 (Revisables). The revisable literals of a program P are a subset of $Rev(P)$, the set of all default literals **not A** with no rules for A in P.

Definition 2 (Positive assumptions of a program). A set S of positive literals is a set of positive assumptions of program P iff

$$\forall L \in S \Rightarrow not\, L \in Rev(P)$$

[2] When $n = m = 0$, H is an alternative representation for rule $H \leftarrow$.

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 (Revising assumptions of a program). A set of positive assumptions S of P is a set of revising assumptions (or a **revision**) iff $\perp \notin Con(Rev(P, S))$[3]

Next we identify which subsets of the revisables support contradiction via some integrity rule. The revision of elements from each subset, i.e. adding the corresponding positive assumptions, can eliminate the introduction of contradiction (i.e. $\perp$) via that rule, by withdrawing the support given to the rule body by such CWA elements. But the revision wrt those assumptions may also introduce fresh contradiction via some other integrity rule. If that is the case then the revision process is simply iterated. Finally, we are interested in the minimal revisions after iteration, i.e. the minimal sets of revising assumptions.

First we define support:

Definition 5 (Support set of a literal). Support sets of any literal L true in the model M_P of a normal program P, are denoted by $SS(L)$, always exist and are obtained as follows:

1. If L is a positive literal, 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 $B_i \in \mathcal{B}$. i.e. there are as many $SS(L)$ as ways of making true some rule body for L.
2. If L is a default literal *not* A:
   (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[5] 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 a SS of the complement of every chosen literal. i.e. 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 supports of $\perp$:

Definition 6 (Assumption set of $\perp$ wrt revisables). An assumption set of $\perp$ wrt revisables R is any set $AS(\perp, R) = SS(\perp) \cap R$, for some $SS(\perp)$.

We define a spectrum of possible revisions with the known notion of hitting set:

Definition 7 (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.

---

[3] A literal $L \notin Con(P)$ iff $P \not\models L$

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

[5] The complement of a positive literal L is *not* L, and of a default literal *not* L is L.

We revise programs by revising the literals of candidate removal sets:

Definition 8 (Candidate removal set). A candidate removal set of P wrt revisables R is a minimal hitting set of the collection of all assumption sets $AS(\bot, R)$.

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

Based on the above, we have devised an iterative algorithm to compute the minimal sets of revising assumptions 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 candidate removal sets.[6]

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, 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 out other contradictory programs obtained by it); (3) the new program is contradictory but revisable and this very same algorithm is iterated until we finitely attain one of the two other cases.

The sets of assumptions used to obtain the revised non-contradictory programs are the minimal revising sets of assumptions of the original program. The algorithm can terminate after executing only one step when the program is either non-contradictory or contradictory and non-revisable, i.e., it has no $CRSs$. For a precise description of the algorithm the reader is refered to [15]. It can be shown this algorithm is NP-Hard [3].

3 Application to Declarative Debugging

In this section we apply contradiction removal to perform debugging of terminating normal logic programs. Besides looping there are only two other kinds of error, cf. [9]: wrong solutions and missing solutions.

3.1 Debugging Wrong Solutions

Consider the following buggy program P:

```
a(1)                 b(2)  c(1,X)
a(X)←b(X),c(Y,Y)     b(3)  c(2,2)
```

[6] [17] gives an "algorithm" for computing minimal diagnoses, called DIAGNOSE (with a bug corrected in [4]). DIAGNOSE can be used to compute $CRSs$, needing only the definition of the function Tp refered there. Our Tp was built from a top-down derivation procedure adapted from [14, 13].

As you can check, goal $a(2)$ succeeds in the above program. Suppose now that $a(2)$ should not be a conclusion of P, so that $a(2)$ is a wrong solution. What are the minimal causes of this bug? There are three. First, the obvious one, the second rule for a has a bug; the second is that $b(2)$ should not hold in P; and, finally, that neither $c(1, X)$ nor $c(2, 2)$ should hold in P.

This type of error (and its causes) is easily detected using contradiction removal by means of a simple transformation applied to the original program:

– Add default literal $not\ ab(i, [X_1, X_2, \ldots, X_n])$ to the body of each i-th rule of P, where n is its arity and $X_1, X_2, \ldots, X_n$ its head arguments.

Applying this to P we get program P_1:

```
   a(1) ←not ab(1,[1])        a(X) ←b(X), c(Y,Y), not ab(2,[X])
   b(2) ←not ab(3,[2])        b(3) ←not ab(4,[3])
c(1,X)←not ab(5,[1,X])   c(2,2)←not ab(6,[2,2])
```

Now, if we have wrong solution $p(X_1, X_2, \ldots, X_n)$ in P just add to P_1 integrity rule $\perp \leftarrow p(X_1, X_2, \ldots, X_n)$, and revise it to find the possible causes of the wrong solution, using as revisables the ab literals.

Since $a(2)$ is a wrong solution, by adding $\perp \leftarrow a(2)$ to P_1 we obtain the minimal revisions $\{ab(2, [2])\}$, $\{ab(3, [2])\}$ and $\{ab(5, [1, 1]), ab(6, [2, 2])\}$, as expected.

3.2 Debugging Missing Solutions

Suppose now the program should not finitely fail on some goal but does so. This is the missing solution problem. Say, for instance, $a(4)$ should succeed in program P above. Which are the minimal sets of rules that added to P make $a(4)$ succeed ? There are two minimal solutions: either add rule $a(4)$ or rule $b(4)$.

To find this type of bug it suffices to add for each predicate p with arity n the rule to P_1:

– $p(X_1, X_2, \ldots, X_n) \leftarrow missing(p(X_1, X_2, \ldots, X_n))$.

All that's necessary to state that if some predicate q has a missing solution $q(X_1, X_2, \ldots, X_n)$ then a contradiction arises, is to add to the program the integrity rule $\perp \leftarrow not\ q(X_1, X_2, \ldots, X_n)$, and revise the transformed program using as revisables $not\ missing(A)$, for all atoms A. The transformed program obtained is P plus the rules:

```
  a(X) ←missing( a(X) )
  b(X) ←missing( b(X) )
c(X,Y)←missing( c(X,Y) )
```

If we want to find the possible causes of the missing solution to $a(4)$ then we add the integrity rule $\perp \leftarrow not\ a(4)$ and obtain, as expected, the two minimal revisions $\{missing(a(4))\}$ and $\{missing(b(4))\}$.

In the case of definite programs only one at a time of the two previous transformations suffices to detect the possible causes of an error. In the case of normal logic programs both transformations must be applied simultaneously in order to achieve complete detection of the both types of error, as the type of error revisables which are relevant change from *ab* to *missing* and vice-versa, at each body rule *not* (cf [9]):

Example 2. Consider program

$$a \leftarrow \text{not } b \quad a \leftarrow c \quad b \leftarrow$$

The two transformations result in:

$$\begin{array}{lll} a \leftarrow \text{not } b, \text{ not } ab(1) & a \leftarrow c, \text{ not } ab(2) & b \leftarrow \text{not } ab(3) \\ a \leftarrow \text{missing}(a) & b \leftarrow \text{missing}(b) & c \leftarrow \text{missing}(c) \end{array}$$

When a is considered a missing solution the integrity rule $\bot \leftarrow not\, a$ is added to the transformed program, thereby generating contradiction. The minimal revisions of the program are:

$$\{missing(a)\}, \{missing(c)\}, \{ab(3)\}$$

If just the missing solution transformation were applied then the third minimal, intuitive, revision would be left out.

4 Updating Knowledge Bases

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

As stated in [5, 6] 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 problem 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. 2) and the first-order formula to a single ground literal. The method can be generalised as in [6], with possible floundering problems, 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 retractions and assertions.

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

Definition 9. The transformation T_{upd} 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 $H \leftarrow B_1, \ldots, B_n, not\, C_1, \ldots, not\, C_m$ in P the literal $not\, retract\_inst((H \leftarrow B_1, \ldots, B_n, not\, C_1, \ldots, not\, C_m))$.
- Add the rule $p(X_1, X_2, \ldots, X_n) \leftarrow assert\_inst(p(X_1, X_2, \ldots, X_n))$ for each predicate p with arity n in the language of P.

It is assumed predicate symbols *retract_inst* and *assert_inst* don't belong to the language of P. The revisables 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 $\bot \leftarrow not\, A$ is added to $T_{\text{upd}}(P)$. 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 $\bot \leftarrow A$ instead. With this method the resulting transactions are more "intuitive" than the ones obtained by [6]:

Example 3 [6]. Consider the following logic program and the request to make pleasant(fred) a logical consequence of it (insertion problem):

pleasant(X)←not old(X), likes_fun(X)
pleasant(X)←sports_person(X), loves_nature(X)
sports_person(X)←swimmer(X)
sports_person(X)←not sedentary(X)
old(X)←age(X,Y), Y > 55
swimmer(fred)
age(fred,60)

The transactions returned by Guessoum and LLoyd's method are:

1. { assert(pleasant(fred)) }
2. { assert(likes_fun(fred)),retract((old(X) ← age(X,Y),Y>55)) }
3. { assert(likes_fun(fred)),retract(age(fred,60)) }
4. { assert(sports_person(fred)),assert(loves_nature(fred)) }
5. { assert(swimmer(fred)),assert(loves_nature(fred)) }
6. { assert(loves_nature(fred)) }

Notice that transactions 4 and 5 are asserting facts (sports_person(fred), resp. swimmer(fred)) that are already conclusions of the program ! 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. { assert_inst(pleasant(fred)) }
2. { assert_inst(likes_fun(fred)),retract_inst((old(fred)←age(fred,60),60>55)) }

3. { assert_inst(likes_fun(fred)),retract_inst(age(fred,60)) }
4. { assert_inst(loves_nature(fred)) }

If the second transistion is added to the program then it is not necessary to remove the rule old(X)←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))[7].

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 [5] 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 don't obey the integrity constraint theory:

Example 4 [5]. Consider the definite logic program from where $r(a)$ must not be a logical consequence of it (the deletion problem):

r(X)←p(X)	p(a)
r(X)←p(X),q(X)	q(a)

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

$$T_1 = \{retract(p(a))\} \text{ and } T_2 = \{retract(p(a)), retract(q(a))\}$$

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

With our method we firstly apply T_{upd} to the program obtaining (notice how the integrity constraint theory is coded):

r(X)←p(X), not retract_inst((r(X) ← p(X)))
r(X)←p(X), q(X), not retract_inst((r(X) ← p(X), q(X)))
p(a)←not retract_inst(p(a))
q(a)←not retract_inst(q(a))

p(X)←assert_inst(p(X))
q(X)←assert_inst(q(X))
r(X)←assert_inst(r(X))

$\perp$ ←not p(X), q(X)

[7] 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.

The request to delete $r(a)$ is converted in the integrity rule $\perp \leftarrow r(a)$ which is added to the previous program. As the reader can check, this program is contradictory. By computing its minimal revisions, the minimal transactions that *satisfy* the integrity theory are obtained:

1. $\{retract\_inst(p(a)), retract\_inst(q(a))\}$
2. $\{retract\_inst(r(a) \leftarrow p(a)), retract\_inst((r(a) \leftarrow p(a), q(a)))\}$
3. $\{retract\_inst(q(a)), retract\_inst((r(a) \leftarrow p(a)))\}$

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

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

5 Application to Diagnosis

In this section we show diagnostic problems in the sense of [1] can be expressed in normal logic programs with integrity rules. By revising the program to remove contradiction 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 contradiction removal. Similar work has been done by [16] using Generalised Stable Models [7].

We start by making a short description of a diagnostic problem as defined in [1, 2]. A **DP** is a triple consisting of a system description, inputs and observations. The system is modeled 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 model, 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 [1] 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.

With these introductory definitions, [1] present a general diagnosis framework unifying the consistency-based and abductive approaches. 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 **DP** two sets are constructed: Ψ^+, 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 f 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 Ψ^+ consistent with Ψ^-. By varying the set Ψ^+ a spectrum of different types of diagnosis is obtained.

Theorem 1. *Given an abduction problem corresponding to a diagnostic problem, its minimal solutions are the minimal revising assumptions of the modeling program plus contextual data, and the following rules:*

1. $\perp \leftarrow not\ obs(v)$ *for each* $obs(v) \in \Psi^+$
2. $\perp \leftarrow obs(v)$ *for each* $obs(v) \in \Psi^-$

and for each component c_i *with distinct abnormal behaviour modes* b_j *and* b_k*:*

3. $correct(c_i) \leftarrow not\ ab(c_i)$
4. $b_j(ci) \leftarrow ab(c_i), fault\_mode(c_i, b_j)$
5. $\perp \leftarrow fault\_mode(c_i, b_j), fault\_mode(c_i, b_k)$ *for each* b_j*,* b_k

with revisables $fault\_mode(c_i, b_j)$ *and* $ab(c_i)$*.*

We don't give here a proof of this result but take into consideration that:

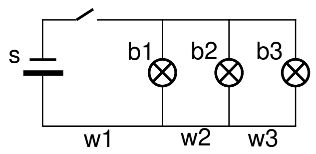
- Rule 1 ensures that in each consistent set of assumptions $obs(v) \in \Psi^+$ must be entailed by the program
- Rule 2 guarantees the consistency of the sets of assumptions with Ψ^-
- Rules 4 and 5 deal with and generate all the possible mutually exclusive behaviours of a component

Finally, note that in no revision there appears the literal $fault\_mode(c, correct)$, thus guaranteeing that minimal revising assumptions are indeed minimal solutions to the diagnostic problem.

5.1 Examples

Example 5 [8].

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.



on(b1) ← closed, ok(s), ok(w1), ok(b1) $\qquad$ ⊥ ← not open, not closed
on(b2) ← closed, ok(s), ok(w1), ok(w2), ok(b2) $\qquad$ ⊥ ← open, closed
on(b3) ← closed, ok(s), ok(w1), ok(w2), ok(w3), ok(b3) $\qquad$ ok(X) ← not ab(X)

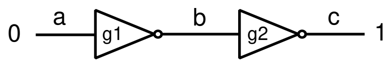
Further integrity rules specify observed behaviour to be explained. For instance, to explain that bulb 1 is on it is only necessary to add $\perp \leftarrow not\ on(b1)$ to obtain the single, intuitive, minimal revision $\{closed\}$.

Suppose instead we wish to explain that bulb 2 is off (i.e. not on). Adding $\perp \leftarrow on(b2)$, five minimal revisions explain it, four of which express faults:

$$\{closed, not\ ab(s)\} \quad \{closed, not\ ab(w_1)\}$$
$$\{closed, not\ ab(b_2)\} \quad \{closed, not\ ab(w_2)\}$$
$$\{open\}$$

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.

Example 6. Two inverters are connected in series. Rules 1-2 model normal inverter behaviour, where *correct* has been replaced by *not ab*. 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. The revisables are *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]

[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.

```
inv(G,I,1) ← node(I,0), not ab(G)                          1
inv(G,I,0) ← node(I,1), not ab(G)                          2
inv(G,I,0) ← node(I,_), fault_mode(G,s0)                   3
inv(G,I,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
```

The following expected minimal revisions are produced by our algorithm:

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

To see this, note that the above program entails $\perp$. In order to find the minimal revisions we apply our iterative algorithm. In the first iteration, the sets of positive assumptions:

$$A_1 = \{ab(g1), fault\_mode(g1, s0)\} \; A_2 = \{ab(g2), fault\_mode(g2, s1)\}$$
$$A_3 = \{ab(g2), fault\_mode(g1, s0)\} \; A_4 = \{ab(g1), fault\_mode(g2, s1)\}$$

are generated from the complements of the candidate removal sets. $Rev(P, A_3)$ and $Rev(P, A_4)$ are contradictory. In the second iteration new sets of positive assumptions are computed:

$$A_5 = \{ab(g2), fault\_mode(g1, s0), fault\_mode(g2, s1)\}$$
$$A_6 = \{ab(g1), fault\_mode(g1, s0), fault\_mode(g2, s1)\}$$
$$A_7 = \{ab(g1), fault\_mode(g1, s1), fault\_mode(g2, s1)\}$$

The set of positive assumptions A_5 is originated by $Rev(P, A_3)$ and the last two by $Rev(P, A_4)$. $Rev(P, A_7)$ is still contradictory and a further iteration is required, producing $A_8 = \{ab(g1), ab(g2), fault\_mode(g1, s1), fault\_mode(g2, s1)\}$. The algorithm thus computes the sets of revising assumptions A_1, A_2, A_5, A_6 and A_8, with A_1 and A_2 being the minimal ones, i.e. the minimal revisions of the original program.

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

Now the following expected minimal revisions are produced:

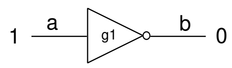
$$\{ab(g1), fault\_mode(g1, s0)\}\ \{ab(g2), missing(g2, 1)\}$$

The above fault model "completion" is a general technique for explaining all observations, with the advantage, with respect to [8]'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 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 a predicate must actually ensure some predication about goals for it (eg. 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 *not* $ab(G, R, HeadArguments)$ instead of *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 *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 numbering 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 7. The (buggy) model of an inverter gate below entails $node(b, 0)$, and also (wrongly) $node(b, 1)$, when its input is 1.



inv(G,I,0)←node(I,1), not ab(G)
inv(G,I,1)←node(I,1), not ab(G) % bug: node(I,0)
node(b,V)←inv(g1,a,V)
node(a,1)

After the debugging transformation:

$$\begin{array}{l}
\mathrm{inv(G,I,0)} \leftarrow \mathrm{node(I,1),not\ ab(G,1,[G,I,0])} \\
\mathrm{inv(G,I,1)} \leftarrow \mathrm{node(I,1),not\ ab(G,2,[G,I,1])} \\
\mathrm{node(b,V)} \leftarrow \mathrm{inv(g1,a,V),\ not\ ab(3,[b,V])} \\
\mathrm{node(a,1)} \leftarrow \mathrm{not\ ab(4,[a,V])}
\end{array}$$

Now, adding to it $\perp \leftarrow node(b,1)$, and revising the now contradictory program the following minimal revisions are obtained:

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

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.

References

1. L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7:133–141, 1991.
2. J. de Kleer and B.C. Williams. Diagnosis with behavioral modes. In *Proc. IJCAI'89*, pages 1329–1330, 1989.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman and Co., 1979.
4. 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.
5. A. Guessoum and J. W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.
6. A. Guessoum and J. W. Lloyd. Updating knowledge bases II. *New Generation Computing*, 10(1):73–100, 1991.
7. A. C. Kakas and P. Mancarella. Generalised stable models: A semantics for abduction. In *Proc. ECAI'90*, pages 401–405, 1990.
8. K. Konolige. Using default and causal reasoning in diagnosis. In C. Rich B. Nebel and W. Swartout, editors, *Proc. KR'92*, pages 509–520. Morgan Kaufmann, 1992.
9. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
10. L. M. Pereira, J. J. Alferes, and J.N. Aparício. Contradiction removal within well founded semantics. In W. Marek A. Nerode and V.S. Subrahmanian, editors, *Proc. Logic Programming and NonMonotonic Reasoning'91*, pages 105–119. MIT press, 1991.
11. L. M. Pereira, J. J. Alferes, and J.N. Aparício. Contradiction removal semantics with explicit negation. In *Proc. Applied Logic Conf.*, Amsterdam, 1992. ILLC.
12. L. M. Pereira, J. J. Alferes, and J.N. Aparício. Logic programming for nonmonotonic reasoning. In *Proc. Applied Logic Conf.*, Amsterdam, 1992. ILLC.
13. 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.
14. L. M. Pereira, J.N. Aparício, and J. J. Alferes. Derivation procedures for extended stable models. In *Proc. IJCAI-91*. Morgan Kaufmann, 1991.

15. 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*, pages 316–330. MIT Press, 1993.
16. C. Preist and K. Eshghi. Consistency-based and abductive diagnoses as generalised stable models. In *Proc. Fifth Generation Computer Systems'92*. ICOT, 1992.
17. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–96, 1987.