

Model-Based Diagnosis Meets Error Diagnosis in Logic Programs*

Luca Console
Dipartimento di
Matematica e Informatica
Università di Udine
Via Zanon 6
I-33100 Udine, Italy

Gerhard Friedrich
Christian Doppler Laboratory
for Expert Systems
Technische Universität Wien
Paniglgasse 16
A-1040 Vienna, Austria

Daniele Theseider Dupré
Dipartimento di Informatica
Università di Torino
Corso Svizzera 185
I-10149 Torino, Italy

Abstract

Model-based diagnosis aims at providing a general theory of diagnosis. In this paper we propose the use of model-based diagnosis techniques in software diagnosis rather than in diagnosing physical systems. In particular, we define the concept of "model-based" diagnosis of logic programs and a preference relation among diagnoses. Based on the concept of leading diagnoses (according to the preference order), we show that bug localization by model-based diagnosis stops where traditional approaches would still query the user. We then present an algorithm for computing diagnoses and we briefly discuss its correctness and completeness. We finally compare the definition to other approaches to logic program debugging.

1 Introduction

Program debugging is one of the most critical activities in software development and many efforts have been made in order to build tools that support such a task (see e.g. [Sommerville, 1989]). In particular, a lot of attention has been paid in the logic programming community to the design of automatic declarative error diagnosers (after Shapiro's seminal work [1983]). Such diagnosers are declarative in the sense that they do not need any understanding of the computational behavior of the program but they need to know only the intended interpretation of the program (the set of answers that the program should compute). Information on such an intended behavior is obtained querying an oracle, usually corresponding to the user. Several procedures for diagnosing logic programs have been developed [Shapiro, 1983; Lloyd, 1987; Naish, 1992], dealing with two types of bugs: wrong answers, i.e. undesired consequences that are derived by the program, and missing answers, i.e. desired consequences that are not derived. Moreover, search strategies are adopted in order to minimize the number of queries to the oracle.

*This work has been partially supported on the Italian part by MURST 40% and on the Austrian part by the Austrian Industries.

Model-Based Diagnosis (MBD) concentrates on the problem of isolating faults of physical systems, starting from a model of the structure (components and their connections) and function (or behavior) of the system and a set of observations indicating an abnormal behavior [Hamscher *et al.*, 1992]. In particular, a system is faulty if the observed behavior of the system contradicts the behavior predicted by assuming that all of its components are correct. Making assumptions about faulty components of the modeled system allows explaining the deviation of the observed behavior from the predicted one; thus the faulty components can be isolated (see figure 1). Moreover, one of the goals of MBD is to select the observation that best discriminates among a set of competing leading diagnoses (using information-theoretic concepts). Formal characterizations of the diagnostic task have been proposed, namely the consistency-based [de Kleer and Williams, 1987; Reiter, 1987; Friedrich and Nejd, 1990; de Kleer *et al.*, 1992] and the abductive one [Poole, 1989; Console *et al.*, 1989; Console and Torasso, 1991].

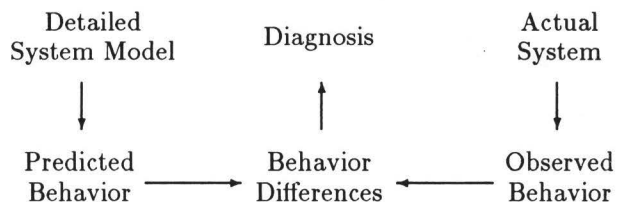


Figure 1: Model-Based Diagnosis

The goal of this paper is to use MBD techniques to diagnose logic programs (and suggest corrections). Such a use of MBD will cause conceptual shifts with respect to both traditional MBD and declarative error diagnosis. With respect to the former, the main problem is to identify what plays the roles of the model and the observations. At a first glance, it would be intuitive that the specification of the program plays the role of the model of the system while the program to be debugged plays the role of the actual system, i.e. it provides observations. However, since we do not want to restrict our approach by the assumption of having a formal, complete specification for the procedures in the program (this is not realistic since providing it could be at least as difficult as writing the correct program), we cannot use specifica-

tions as models. Thus, in our approach the program to be diagnosed (whose structure and behavior is known in detail) is technically used in the same way as the model is used in MBD; the clauses of the program are regarded as components which may be correct or abnormal. In contrast, the role of the observations is technically played by the information available on the user's model of the intended behavior of the program.

Given a set of assertions about the intended behavior of the program (a set of expected answers and a set of undesired ones), the goal is to determine the bugs by comparing the actual program behavior with such assertions. In the same way as a system is faulty when assuming that all its components are correct is in contradiction with the observed behavior, a program is faulty when assuming that all its clauses are correct is in contradiction with the assertions on the intended behavior. The goal is to remove such a discrepancy, isolating a set of clauses which are not correct and must be fixed. In particular, this amounts to determining a set of clauses to be removed and, possibly, a set of clauses to be added such that the resulting modified program is in accordance with the assertions (see figure 2)¹.

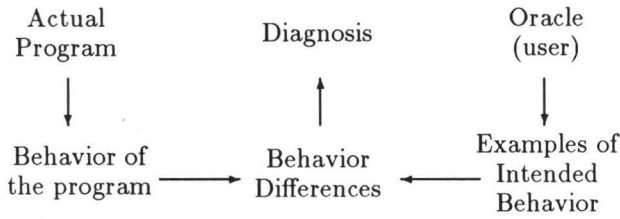


Figure 2: Program Diagnosis

Given a partial specification of the intended behavior, our approach characterizes the alternative ways of modifying the program in order to match such a specification. This is a conceptual shift with respect to declarative debugging whose aim is at querying the user until a fault (one at a time) is isolated.

The approach we present has some advantages with respect to other error diagnosers. We look at the program and at all the 'observations' globally and thus we are able to characterize the whole set of diagnoses (possibly involving multiple faults), ranking them according to some preference criteria, rather than looking at one execution tree for one wrong or missing clause instance at a time. This allows us to devise an algorithm for computing the diagnoses in increasing order of preference. Moreover, the use of MBD techniques can provide a more focused interaction with the user. For example, if one diagnosis is much more likely than all the others, we simply output it without asking new information. More generally, we can include in our approach techniques similar to those used in MBD to ask information that can discriminate as much as possible among competing leading diagnoses rather than asking information about the predicates in the considered AND-OR execution tree.

Section 2 presents a motivating example; section 3 provides definitions of diagnoses and preference orderings

¹This is something more than traditional MBD, involving also some form of repair [Friedrich *et al.*, 1992].

among them; section 4 presents an algorithm for computing diagnoses; section 5 introduces a further example; section 6 compares our approach to related ones.

2 Motivating Example

The following example motivates the use of MBD techniques for localizing errors in logic programs.

Example 1 Consider the following program (constants, functions and predicates are in lower case, variables are in upper case and are universally quantified):

```

(c0)      p(0)
(c1)      q1(X) ← q2(X)
...
(cn-1)    qn-1(X) ← qn(X)
(cn)      qn(X) ← p(X)
(cn+1)    r1(X) ← r2(X)
...
(c2n-1)    rn-1(X) ← rn(X)
(c2n)      rn(X) ← p(X)
(c2n+1)    s1(X) ← s2(X)
...
(c3n-1)    sn-1(X) ← sn(X)
(c3n)      sn(X) ← p(X)
  
```

Assume that the user queries $q_1(X)$, $r_1(X)$, $s_1(X)$ and the program returns $X = 0$ as the only answer. In contrast, the intended behavior of the program is the following: neither $q_1(0)$ nor $r_1(0)$ nor $s_1(0)$ should be entailed by the program, while $q_1(1)$, $r_1(1)$, $s_1(1)$ should be a valid answer to the query (and it is not returned even on backtracking). In this case all the declarative diagnosers mentioned in the introduction (in particular, Shapiro's one which aims at minimizing the number of queries to the user needed to isolate a fault) would query the user about the validity of at least $O(\log(n))$ facts.

The approach based on MBD we will describe does not need to query the user at all in order to identify the most likely diagnosis. In fact, one possibility is that the clause $p(0)$ is faulty (and should be replaced by $p(1)$); this corresponds to a single fault in MBD. Of course, there are other possible solutions which do not include $p(0)$ but any of them has to involve at least three clauses (e.g. c_1, c_{n+1}, c_{2n+1}), i.e. it corresponds to a triple fault. Thus we can conclude immediately that $p(0)$ is the most likely diagnosis, i.e. only this clause has to be fixed. We succeed in doing so because we look at the whole space of solutions rather than considering one execution tree and trying to isolate one fault at a time.

3 Definitions

In this section we provide the definitions for error diagnosis in logic programs.

Definition 1 (Error diagnosis problem) An error diagnosis problem is a triple $\langle P, S^+, S^- \rangle$ where:

- P is a definite logic program;
- S^+ and S^- are sets of queries, where variables in S^+ are intended as universally quantified and variables in S^- are intended as existentially quantified.

The program P has an intended interpretation I (a Herbrand model); S^+ and S^- can be seen as approximations of I . In most practical cases, in fact, we cannot assume to have a complete specification available, since providing it would not be realistic². In particular, in our specification S^+ (S^-) is a set of queries that are true (false) in the intended interpretation. In other words, S^+ (S^-) can be seen as a set of positive (negative) examples about the intended behavior of P . Depending on the power of the available deduction system, P , S^+ and S^- can be made more expressive.

In the declarative diagnosis literature, a distinction between wrong and missing answers is made. In our framework, a wrong answer corresponds to a member of S^- which is entailed by the program, while a missing answer corresponds to a member of S^+ which is not entailed. Notice that S^+ may also contain correct answers (formulae that are true in the intended interpretation and are indeed entailed by the program) while S^- can contain formulae that are false in the intended interpretation and are not entailed by the program.

In example 1, P is the set of clauses $\{c_0, \dots, c_{3n}\}$, S^+ is $\{q_1(1), r_1(1), s_1(1)\}$ and S^- is $\{q_1(0), r_1(0), s_1(0)\}$.

Solving an error diagnosis problem corresponds to identifying a set of clauses to be removed and a set of clauses to be added in order to restore consistency with respect to S^- and to derive all the desired answers in S^+ . Changing a clause can be seen as removing and adding. Note that S^- and S^+ correspond to consistency and provability constraints in deductive databases.

Definition 2 (Diagnosis) A pair $\langle C, E \rangle$ of sets of clauses, where $C \subseteq P$, is a **diagnosis** if and only if:

- $(P \setminus C) \cup E \models a$ for all $a \in S^+$
- $(P \setminus C) \cup E \not\models a$ for all $a \in S^-$

The set C of clauses to be removed is the “core” of a diagnosis, while the choice of E , given a C , is highly unconstrained. However, the following property holds.

Property 1 Given an error diagnosis problem and a set C of clauses, then the following are equivalent:

1. There exists an E such that $\langle C, E \rangle$ is a diagnosis;
2. $\langle C, S^+ \rangle$ is a diagnosis.

We could thus be satisfied with providing as solution a set C of clauses such that $\langle C, S^+ \rangle$ is a diagnosis (according to definition 2). On the other hand, we believe that it can be important to provide as a solution an E which is more informative for the user, giving a hint on how the program could be corrected. Notice that this is more than diagnosis, in the sense that it provides also a rough form of repair. However, in order to do this in a constrained way, it would be important to have a canonical syntactic form in which E has to be expressed (we shall return to that later). It is important to stress that we do not aim at synthesizing the correct program but only at checking whether it is possible to change the program to

²The declarative diagnosers mentioned in the introduction also assume that there is such an intended interpretation, but they do not assume it is specified in advance (information about it can be asked to an oracle when needed).

entail S^+ and at giving some constraints on how the program could be changed. Our goal is thus very different from those of automatic program synthesis and, specifically, from “inductive logic programming” (originated in [Shapiro, 1983] and developed e.g. in [Muggleton, 1991; Quinlan, 1991]).

In practice, it can be useful to introduce some preference criteria among alternative solutions. Several criteria are possible. A reasonable one is based on the number of clauses to be removed or added, which corresponds to the number of faulty components in MBD if we assume that all components have the same probability of failing (and failing is less probable than being correct).

Definition 3 (Ordering among diagnoses) $\langle C, E \rangle < \langle C', E' \rangle$ iff $|C| < |C'|$ or $(|C| = |C'|$ and the number of predicates in the heads of clauses in E is less than the number of predicates in the heads of clauses in E').

Definition 4 (Preferred diagnosis) A diagnosis is preferred iff it is minimal with respect to $<$.

Intuitively, we want to remove as few clauses as possible and add clauses for as less predicates as possible.

Another possible ordering could be based on preferring diagnoses where the sum of the number of clauses to be removed and the number of predicates for which clauses are added is minimum.

In the ordering defined above, for a given C and a given set of predicates, there can be several E s defining these predicates such that the diagnoses given by C and each one of such E s are preferred diagnoses.

In order not to distinguish among such solutions, we would like to have a compact way to represent all of them. This could be done limiting E to clauses in the language of P and S^+ (this means that we do not “invent” auxiliary predicates) and of the form:

$$p_i(X) \leftarrow X \neq a_1, \dots, X \neq a_n$$

such that they give the strongest definition of p_i which is consistent with S^- (i.e. the exceptions are minimized). We shall call these clauses “exception clauses” and we shall say that each a_j is an “exception” for p_i . This is the canonical form advocated above for providing “repair” information to the user. In fact, clauses in this form provide the user with a constraint on how the actual definition of p_i should be: the clause for p_i provides an upper bound for the extension of p_i which is consistent with the program and the ‘observations’.

Diagnoses where E is a set of exception clauses will be denoted as **diagnoses in exception form**. It is worth noting that limiting to such a syntactic form is not restrictive, since the following property holds:

Property 2 For a fixed C , if there is a definite program E in a language L such that $\langle C, E \rangle$ is a diagnosis, then for some set E' of exception clauses in the same language L , $\langle C, E' \rangle$ is a diagnosis.

In practice, diagnoses in exception form can be built just in case the exceptions for each one of the predicates in E' can be expressed in a finite way (we shall return to that in section 4).

Example 1 (contd.). We have that

$$\langle \{c_0\}, \{p(X) \leftarrow X \neq 0\} \rangle$$

is the only preferred diagnosis in exception form. The next diagnoses in the order which are not supersets of the previous one are

$$\langle \{c_0\}, \{q_i(X) \leftarrow X \neq 0, r_j(X) \leftarrow X \neq 0, s_k(X) \leftarrow X \neq 0\} \text{ for any } 1 \leq i, j, k \leq n. \rangle$$

The use of the exception form allows us to provide an upper bound for the extension of the predicates for which a new definition is added to the program. It could be interesting to provide also a lower bound for such an extension. This corresponds to specifying what is strictly needed in order to entail S^+ . Providing both the upper and lower bound can help the user in fixing the program. In the example above, the lower bound for the definition of p for the preferred diagnosis is $p(1)$. In fact, when c_0 is removed, a definition for p must be added in order to entail S^+ . In order to do so consistently with S^- , such a definition must at least entail $p(1)$ and at most any instance of p other than $p(0)$.

4 Computing Diagnoses

In this section we first outline a non-deterministic, general algorithm for computing the diagnoses in exception form (on backtracking on the non-deterministic choices). We also discuss how such an algorithm should be modified in order to provide both upper and lower bounds for the extension of the predicates for which new definitions are added. Finally, we briefly discuss how such an algorithm should be modified in order to generate the solutions in increasing order of preference.

Let us first introduce a couple of definitions.

Definition 5 (Most General Exception Clause)
Given a predicate p and a set H of atoms, the Most General Exception Clause (MGE) for p consistent with H is the clause:

$$p(\bar{X}) \leftarrow \bigwedge_{p(\bar{t}) \in H} \bar{X} \neq \bar{t}$$

where $\bar{X}$ is a tuple of new variables. In case there is no $p(\bar{t}) \in H$, the MGE for p is the clause $p(\bar{X})$.

Definition 6 (Universal Abductive Explanation)
Given a background theory T and a set A of assumable predicates, a Universal Abductive Explanation of a set S of formulae is a set E of formulae of the form $\forall \bar{X} p(\bar{X})$, where $p \in A$, such that $T \cup E \models S$.

The algorithm is as follows:

Input: P, S^+, S^- .

Output: A diagnosis $\langle C, E \rangle$.

Procedure:

1. Find a set C' of clauses in P such that $(P \setminus C') \cup S^+ \not\models a$ for all $a \in S^-$. This can be done using the traditional techniques of consistency-based diagnosis [Reiter, 1987], that is, generating "conflicts", which in our case are sets of clauses in P used to derive some element of S^- from $P \cup S^+$; then, computing a hitting set C' of such conflicts, i.e. a set of clauses that contains at least an element from each conflict and then cuts every derivation of an element

$a \in S^-$. The reason why we use also S^+ to generate conflicts is a way of generating only those C' s for which S^+ can be consistently added, i.e. given property 1, those for which an E exists. This means that the algorithm can be stopped after this step, if one is not interested in determining E .

2. Determine, given C' , an E which allows the entailment of the elements in S^+ . This can be done as follows.

Compute a universal abductive explanation E' of S^+ using $P \setminus C'$ as a background theory and all the predicates as assumable; we generate *universal* explanations because we want to generate E in exception form. Now the exceptions have to be generated to avoid that some element of S^- follows from the modified program. In order to do that, compute the set of conflicts:

$D = \{ \{A_1, \dots, A_m\} \text{ such that there exists a derivation of some } p \in S^- \text{ from } (P \setminus C') \cup E' \text{ and } \{A_1, \dots, A_m\} \text{ is the set of the heads of the instances of the clauses used in the derivation} \}$

Choose a hitting set H of D , removing subsumed atoms. Let $p_1, \dots, p_n$ be the predicates occurring in some element of H . H characterizes a possible set of exceptions to be imposed on the predicates $p_1, \dots, p_n$ in order not to entail elements of S^- .

For each p_i , let C_i be the set of clauses in $(P \setminus C') \cup E'$ defining p_i , and let e_i be the MGEC of p_i consistent with H . In order to modify the program, we have to add $e_1, \dots, e_n$ and to remove the existing clauses $C'' = C_1 \cup \dots \cup C_n$ for $p_1, \dots, p_n$. However, we have to check that the program modified in this way actually entails S^+ . If

$$(((P \setminus C') \cup E') \setminus C'') \cup \{e_1, \dots, e_n\} \models S^+$$

then output

$$\langle C' \cup (C'' \cap P), (E' \setminus C'') \cup \{e_1, \dots, e_n\} \rangle$$

In the output we take the intersection between C'' and P in order to list only clauses that were in the original program.

In case we want to associate with each of the diagnoses computed with the above algorithm also a lower bound on the extension of the predicates for which new definitions are added, we proceed as follows. In step 2 we compute both an abductive explanation E'' of S^+ and the corresponding universal abductive explanation E' ; as regards the latter we perform the steps discussed above; as regards the former, we consider the set D'' of conflicts produced by E'' and we choose a hitting set H'' of D'' ; for each predicate p'_i occurring in H'' , instead of removing the existing definitions adding the most general extension, we simply add the exceptions mentioned by H'' to the body of each existing clause for p'_i ; this requires that the clauses are first put in "general form" [Clark, 1978], i.e. in a form such that the clause $r(\bar{t}) \leftarrow L_1, \dots, L_n$ is transformed into $r(\bar{X}) \leftarrow \bar{X} = \bar{t}, L_1, \dots, L_n$ where $\bar{X}$ is a tuple of distinct variables not occurring in the clause. **Example 1 (contd.).** Let us now show how the algorithm described above computes some of the diagnoses. The conflicts in step 1 are the following:

$$\begin{aligned} &\{c_0, c_1, \dots, c_n\} \\ &\{c_0, c_{n+1}, \dots, c_{2n}\} \\ &\{c_0, c_{2n+1}, \dots, c_{3n}\} \end{aligned}$$

A hitting set is $\{c_0\}$ (there are of course several other hitting sets). A universal abductive explanation that can be computed in step 2 is $p(X)$. In this case we have

$$D = \{ \{p(0), q_1(0), \dots, q_n(0)\}, \{p(0), r_1(0), \dots, r_n(0)\}, \{p(0), s_1(0), \dots, s_n(0)\} \}$$

A hitting set of D is $H_1 = \{p(0)\}$. From it we generate the following MGEC for p :

$$p(X) \leftarrow X \neq 0$$

This gives the preferred diagnosis reported in section 3. In order to compute the lower bound for the definition of p , we can consider the abductive explanation $p(1)$ from which we generate no conflict, thus getting $p(1)$ as the clause to be added.

Other hitting sets of D are $\{q_i(0), r_j(0), s_k(0)\}$ for any $1 \leq i, j, k \leq n$. Each of them provides one of the other solutions mentioned in section 3.

It should be clear that the algorithm is correct according to definition 2: the solution is built in such a way that it is consistent with S^- and the algorithm explicitly checks that S^+ is entailed by the modified program. Moreover, the algorithm can generate all the diagnoses in exception form, provided that there are no infinite execution trees for the goals in S^+ and S^- . In fact, in the first step we can generate (using techniques like those in [Reiter, 1987; Greiner *et al.*, 1989]) all sets of clauses that, if removed, restore consistency, while in the second step we can consider all abductive explanations; removing the candidates that do not entail S^+ at the end of step 2 does not lead to losing any solution, since in step 2 we consider not only minimal abductive explanations and thus, if we exclude one candidate E' because it does not lead to covering S^+ and there is some superset of E' which leads to covering S^+ , we consider also such a redundant explanation and the solutions it generates.

An implementation of the algorithm should be able to generate the diagnoses in the order of the preference relation (without necessarily generating all of them). In case we limit the algorithm to step 1 (producing a C for which an E exists), this can be done easily: in step 1 we generate hitting sets in order of increasing cardinality. Considering also step 2 is more complex since the explanation E' may lead to removing further clauses and thus C and E cannot be minimized independently. Given a preferred candidate C'_1 from step 1 (a candidate removing less clauses than the others), in step 2 we generate explanations in the order of increasing cardinality of the set of predicates involved and we build the solutions. Each E' may lead to removing further clauses, so before outputting a solution, we have to guarantee that we cannot generate better ones by using some other set C'_2 removing more clauses than C'_1 but such that, after the second step, leads to removing less clauses. In particular, we have to wait until we consider candidates C'_i that, in the first step, lead to removing more clauses than the final solution generated from C'_1 . Similar problems arise with both orderings discussed in section 3.

It is worth noting that MBD techniques to generate preferred diagnoses in an efficient way (e.g. [de Kleer,

1991]) can be transferred to the above algorithm (this, however, is outside the scope of this paper).

5 A further example

In this section we briefly discuss an example showing further advantages of our approach, namely the fact that we can isolate all bugs at once.

Example 2 Consider the following diagnosis problem:

$$\begin{aligned} P &= \{ (c_0) p(1) \\ &\quad (c_1) q(X) \leftarrow p(X) \\ &\quad (c_2) r(X) \leftarrow p(X) \} \\ S^+ &= \{ r(1) \} \quad S^- = \{ q(1) \} \end{aligned}$$

In the first step of our algorithm we determine two minimal sets of clauses that, if removed, restore consistency: $C'_1 = \{c_1\}$ and $C'_2 = \{c_0\}$ (notice that these two candidates can be discriminated by asking whether $p(1)$ is in the intended interpretation). By considering step 2, the first diagnoses that we generate which are not supersets of previous ones are the following:

$$\begin{aligned} C_1 &= \{c_1\}, E_1 = \emptyset \\ C_2 &= \{c_0\}, E_2 = \{ r(X) \} \\ C_3 &= \{c_0, c_1\}, E_3 = \{ p(X), q(X) \leftarrow X \neq 1 \} \end{aligned}$$

Suppose that the definition of S^+ and S^- arises from the following interaction with the user: first he/she runs the query $r(X)$ (or $r(1)$) and labels as correct the answer $X = 1$ (or *yes*); then he/she runs the query $q(X)$ (or $q(1)$) and labels as incorrect the answer $X = 1$ (or *yes*).

In the diagnosers in [Shapiro, 1983; Lloyd, 1987; Naish, 1992] this would correspond to searching a bug in the computation tree of $q(1)$. This will cause asking to the oracle whether $p(1)$ is in the intended interpretation. If we assume that the answer is "no", c_0 is output as the wrong clause instance. The fact that $r(1)$ must be derived from the program is not considered; it would be necessary to re-run previous successful queries after the program has been modified, in order to check whether they are still correctly answered, and, if this is not true, re-run the diagnoser. On the other hand, we consider all 'observations' at once and thus a diagnosis directly takes into account all of them and isolates all bugs at once.

6 Comparison with related work

It is interesting to compare our approach with the declarative diagnosers in the literature [Shapiro, 1983; Lloyd, 1987; Naish, 1992]. First of all, it is important to consider that the goals we are pursuing are different. We are interested in characterizing the diagnosis space and in designing a procedure which can compute the leading diagnoses according to some preference order. Each diagnosis may include multiple faults and explains all the misbehaviors of the program. On the other hand, most of the research on declarative diagnosis concentrated on defining procedures for determining one bug at a time using an oracle for asking information about the intended behavior of the program at hand. In our terminology, such diagnosers look at one conflict (for a wrong answer) or one explanation (for a missing answer) at a time, trying to isolate one fault from such a conflict

(or explanation); then repeat the same process again on another wrong or missing answer. Moreover, they assume that the oracle is able to answer any question on the considered execution tree.

However, it is interesting to show how the solution(s) they compute are included in the space characterized by definition 2. We shall analyze this point informally, using as references the characterizations provided in [Naish, 1992], where two different types of bugs are considered (as in most declarative diagnosers):

- *Incorrect clause instances*, defined by the following Prolog clause:

$$\text{bug}(A \leftarrow B) \text{ :- clause}(A, B), \text{unsatisfiable}(A), \\ \text{valid}(B).$$

(“valid” and “unsatisfiable” are calls to an oracle; an atom is valid if it is in the intended interpretation, it is unsatisfiable if it is not in the intended interpretation).

- *Uncovered atoms*, defined by the following Prolog clause:

$$\text{bug}(\text{atom}(A)) \text{ :- valid}(A), \\ \text{all } B \text{ not}(\text{clause}(A, B), \text{valid}(B)).$$

where “all” can be read as a universal quantifier. A is uncovered if it is valid but there is no clause in the program with a valid body that implies A .

In the practical versions of the diagnosers, such types of bugs are considered only if they actually lead to a wrong or missing answer. In our approach we consider globally all the pieces of information in S^+ and S^- (which can include wrong and missing answers as well as correct answers and incorrect answers not computed by the program). Given a clause c_i having an incorrect instance, if c_i is actually used to derive a wrong answer (and thus is recognized as a bug by the practical versions of the diagnosers after a number of queries to the oracle), it is clear that c_i belongs to the first component of some diagnosis according to our definition (since removing c_i – and, possibly, other clauses – is a way of removing the inconsistency). Moreover, given an uncovered atom, if it actually leads to some missing answer (i.e. the atom is such that if it were derivable, some missing answer would not be missing), then in our definition a clause implying such an atom will be part of the second component of some diagnosis.

7 Conclusions

Software is tremendously more erroneous than hardware and thus enhancing software diagnosis is very important. This paper proposes the use of MBD in such a task (in particular, error diagnosis in logic programs) and concentrates on the core of the diagnostic process, defining the concept of model-based diagnosis of logic programs and an algorithm for computing diagnoses. Other techniques developed in the MBD community can be transferred to software diagnosis; for example: use of fault probabilities based on metrics or user models, information theoretic techniques for observation selection, focusing techniques to improve the algorithm, hierarchical

decomposition as well as the integration of diagnosis and repair.

References

- [Clark, 1978] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
- [Console and Torasso, 1991] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.
- [Console et al., 1989] L. Console, D. Thesieder Dupré, and P. Torasso. A theory of diagnosis for incomplete causal models. In *Proc. 11th IJCAI*, pages 1311–1317, Detroit, 1989.
- [de Kleer and Williams, 1987] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
- [de Kleer et al., 1992] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2–3):197–222, 1992.
- [de Kleer, 1991] J. de Kleer. Focusing on probable diagnoses. In *Proc. AAAI 91*, pages 842–848, Anaheim, CA, 1991.
- [Friedrich and Nejd, 1990] Gerhard Friedrich and Wolfgang Nejd. MOMO — Model-based diagnosis for everybody. In *Proc. IEEE Conf. on AI Applications*, Santa Barbara, 1990.
- [Friedrich et al., 1992] G. Friedrich, G. Gottlob, and W. Nejd. Formalizing the repair process. In *Proc. 10th ECAI*, pages 709–713, Vienna, 1992.
- [Greiner et al., 1989] R. Greiner, B.A. Smith, and R.W. Wilkerson. A correction to the algorithm in Reiter’s theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.
- [Hamscher et al., 1992] W. Hamscher, L. Console, and J. de Kleer. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.
- [Lloyd, 1987] J.W. Lloyd. Declarative error diagnosis. *New Gen. Computing*, 10:133–154, 1987.
- [Muggleton, 1991] S. Muggleton. Inductive logic programming. *New Gen. Computing*, 8(4):295–318, 1991.
- [Naish, 1992] L. Naish. Declarative diagnosis of missing answers. *New Gen. Computing*, 10:225–285, 1992.
- [Poole, 1989] D. Poole. Normality and faults in logic-based diagnosis. In *Proc. 11th IJCAI*, pages 1304–1310, Detroit, 1989.
- [Quinlan, 1991] J. R. Quinlan. Determinate literals in inductive logic programming. In *Proc. 12th IJCAI*, pages 746–750, Sydney, 1991.
- [Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.
- [Shapiro, 1983] E. Shapiro. *Algorithmic Program Debugging*. MIT-Press, 1983.
- [Sommerville, 1989] I. Sommerville. *Software Engineering*. Addison-Wesley, 1989.