New Generation Computing, 9 (1991) 277-303
OHMSHA, LTD. and Springer-Verlag

Logical Diagnosis of $\mathcal{LDL}$ Programs

Oded SHMUELI*
Department of Computer Science,
Technion—Israel Institute of Technology,
Haifa, Israel, 32000.
Shalom TSUR
Microelectronics and Computer Technology Corporation,
3500 West Balcones Center Drive,
Austin, TX, 78759.

Received 12 November 1990
Revised manuscript received 21 June 1991

Abstract The debuggers of Ref. 11) and most of their derivatives are of the *meta-interpreter* type. The computation of the debugger tracks the computation of the program to be diagnosed at the level of procedure call. This is adequate if the intuitive understanding of the programmer is in terms of procedure calls; as is indeed the case in Prolog.

In $\mathcal{LDL}$ however, while the semantics of the language are described in a bottom-up, fixpoint model of computation,[8] the *actual* execution of a program is a complex sequence of low-level procedure calls determined (and optimized) by the compiler. Consequently, a trace of these procedure calls is of little use to the programmer. Further, one cannot "execute" an $\mathcal{LDL}$ program as if it was a Prolog program; the program may simply not terminate in its Prolog reading and several $\mathcal{LDL}$ constructs have no obvious Prolog counterparts.

We identify the origin of a fault in the $\mathcal{LDL}$ program by a top-down, query/subquery approach. The basic debugger, implemented in Prolog, is a shell program between the programmer and the $\mathcal{LDL}$ program: it poses queries and uses the results to drive the interaction with the user. It closely resembles the one presented in Ref. 11). The core of a more sophisticated debugger is presented as well.

Several concepts are introduced in order to quantify debugging abilities. One is that of a *generated interpretation*, in which the structureless intended interpretation of Ref. 11) is augmented with causality. Another is the (idealized) concept of a *reliable oracle*. We show that given an incorrect program and a reliable oracle which uses a generated interpretation, a cause

* Research partially supported by the Fund for Promotion of Research at the Technion.

for the fault will be found in finitely many steps. This result carries over to the more sophisticated debugger.

Keywords: Logic Programming, Debugging, Diagnosis, Deductive Databases

§1 Introduction

The term *diagnosis* is used in the context of programming to denote the process which is intended to explain the differences, if any, between the actual, observed behavior of a program during execution and its intended behavior. The intended behavior is either conveyed by a formal specification or, more often, it is left unspecified and manifests itself in the expectations of the programmer with respect to the program. In either case, program diagnosis is impossible without imparting this knowledge of the intended behavior to the diagnostic system— either in advance or, alternatively, during the human/system interaction that governs the process.

Traditionally, programs are written using procedural languages such as Fortran or C. The behavior (semantics) of such programs is described in terms of states, represented by the values of the declared program variables and data-structures, and the computation-induced transitions among states. This model of behavior is an abstraction of the real program behavior and the programmer uses it as a guidance in the debugging of his/her program. The tool-set at his/her disposal supports this model and typically includes means to either observe selected states (traps, checkpoints) or the tracing and display of selected variables during the sequence of state transitions.* The information, obtained by these tools, helps the programmer in inferring the changes that must be made to the present version of the program so as to modify its behavior to the intended behavior.

Databases are usually accessed either by ad-hoc queries or using DML statements (e.g. SQL) embedded in a conventional programming language (e.g. COBOL). So, for databases, debugging needs were met essentially by traditional techniques. Recently, there is a trend of increasing DML capabilities with predominantly declarative constructs (e.g. DATALOG,$\mathcal{LDL}$, NAIL!). Declarative languages i.e., languages that enable problem-statement in terms of *what* needs to be computed, as opposed to *how* it should be computed, provide the user with the same level of abstraction as the specifications, used to convey the intended behavior in the procedural programming case. Consequently, the model of computation that guides the programmer in the procedural case, is of no value in this context. In particular, there is no notion of a state** and even though the declarative specification may be compiled into procedural target

* We assume that the class of programs here contains only sequential ones. We will not elaborate on the class of concurrent programs in which clearly the model of computation is more complex.

** We will reintroduce this notion in the sequel when we discuss updates.

code, tracing the behavior at that level is of little value to the programmer.

Clearly other means are needed. These are the topic of this paper. We will confine ourselves to means which are of use with logic programming languages and will demonstrate how they could be used within $\mathcal{LDL}$. The most interesting feature of the diagnosis of these languages is that the same logic, that is used by the programmer to specify the problem, can be used in its diagnosis. Thus, the debugging process is shifted thus from the intuitive at the procedural level, to the system-directed at the declarative level. We will demonstrate how the system utilizes the program to elicit additional information from the programmer and eventually leads him/her to a correct program.

The subject of algorithmic (or logical) debugging originated with Ref. 11). There have been many attempts at improving and extending Shapiro's work, a non-exclusive list is: [3,4,5,6,9,10] In this paper we concentrate on two aspects. The first contribution is in formalizing and structuring the user's mental picture so that guarantees can be given as to the convergence of the debugging process. The second contribution is in treating the subject of debugging a compiled language. The results reported here first appeared in Ref. 12).

The paper is organized as follows. Section 2 formalizes concepts relevant to debugging $\mathcal{LDL}$ programs. Section 3 presents a basic debugger. Section 4 treats debugging of updating programs. Section 5 presents a more sophisticated debugger. We conclude in Section 6.

§2 Debugging Declarative Languages

2.1 Formalization

We will assume a basic familiarity with the logic programming paradigm,[7] and mainly introduce some additional definitions. Specifically, we shall assume familiarity with the concepts of *term*, *atom*, *literal*, *expression*, *interpretation*, *Herbrand interpretation*, *clause*, *Horn clause*(*or rule*), *clause head*, *clause body*, *substitution*, *instance*, *unification*. A *fact* is a ground atom, i.e. one containing no variables. An expression e is *satisfiable* in an interpretation I if there exists a ground instance e' of e which is true w.r.t. the interpretation. An expression is *valid* in an interpretation I if all ground instances of the expression are true w.r.t. I; so, for a fact f and I an Herbrand interpretation, f is valid in I iff it is in I and $\neg f$ is valid in I iff it is not in I.

The set of predicate symbols is partitioned into *EDB* and *IDB* predicate symbols. The *EDB* symbols are those of *base relations* and the *IDB* symbols are those of *derived relations*. The idea is that the interpretation (or *extension*) of base relations is given explicitly as a set of unit clauses which are part of the program, while that of derived relations are to be computed using (non-unit) program clauses. By convention, *EDB* predicate symbols may only appear in clause bodies or in unit clauses, no generality is lost by this requirement. Furthermore, all predicate symbols in unit clauses (i.e. bodyless Horn clauses) must be in *EDB*. We shall allow unit clauses with variables, which denote infinite *EDB* relations.

We will assume a basic familiarity with the $\mathcal{LDL}$ language; for an exposition of the $\mathcal{LDL}$ language features the reader is referred to Ref. 8), and for the formal description of its semantics to Ref. 1). $\mathcal{LDL}$ is based on Horn clauses. Its repertoire includes predicate symbols, (free) function symbols, variables and constants. So far an $\mathcal{LDL}$ program is a definite program as defined in Ref. 7)—both in syntax and semantics. However, the language has additional constructs that set it apart from the definite programs. First is the use of (stratified) negation. Second, $\mathcal{LDL}$'s universe includes sets, and is therefore quite different than the traditional Herbrand universe; for this paper, however, we shall assume that the $\mathcal{LDL}$ universe is identical to the Herbrand universe. Finally, there are constructs in $\mathcal{LDL}$ that futher diverge from definite programs. Facilities are provided for *grouping* elements into sets. Facilities are also provided for *updates*, i.e. changing the extensions of (base) predicates.

An $\mathcal{LDL}$ *program* is a finite set of clauses obeying the restrictions concerning *IDB* and *EDB* predicate symbols. An $\mathcal{LDL}$ *query* (or simply, *query*) is a conjunction of literals. A *query form* is a query representing a family of queries. It only uses predicate symbols and the constants **b** and **f**. It specifies the predicate symbols of literals of the query and for each one it says whether the actual queries have this argument ground (**b**) or not (**f**).

Let q be a single literal query. For a set S of facts define the *projection of* S *on* q, denoted $\pi_q(S)$, to be that subset of S of facts that are unifiable with q. Let P be an $\mathcal{LDL}$ program. The semantics of $\mathcal{LDL}$ defines the *model of* P, $M(P)$, which is obtained by repeated bottom-up derivations using the rule-set. The answers computed by a program P for query q are defined by the $\mathcal{LDL}$ semantics as $\pi_q(M(P))$. The answers are facts for the predicate symbol of q.

If negation is used in an $\mathcal{LDL}$ program then the program need be *layered* (or *stratified*). The layering scheme is a prescription for one of the possibly many minimal models of the program. Layering means partitioning the predicate symbols of a progam P into disjoint sets, called *layers*, and a total order (from lowest to highest) is defined on the layers. If predicate symbol p appears in a clause whose head predicate symbol is q, then q must be in a higher layer than p or in the same layer as p. Furthermore, if the appearance of p is negated then q must be at a higher layer. $M(P)$ is built one layer at a time, from lowest to highest. In each layer, facts for the predicate symbols in this layer are produced from P by repeated bottom-up "firing" of clauses defining the layer's predicates, facts produced for lower layers are taken as if they were *EDB* facts. In this paper we employ the following convention, the first layer of a program includes all, and only, unit clauses whose head predicate symbol is in *EDB*.

An $\mathcal{LDL}$ compiler is *correct* for a class C of (*program*, *query*) pairs if for all such pairs the run-time code computes, within finite time, exactly the set of answers that is prescribed by the semantics of the language. In this paper, the class C of interest contains those programs and queries for query forms that are certified by the compiler as *safe*, i.e. termination (and therefore finiteness of answer) has been certified by the $\mathcal{LDL}$ compiler. We assume that $\mathcal{LDL}$'s compiler is correct for C. This implies that the above certification is correct.

2.2 Overview of the Debugging Process

In the classical reading, the truth value of a ground atom of a program P is determined by P's *interpretation*. This notion is insufficient however to capture the programmer's intuition with respect to the required outcome of a computation—it lacks an element of *causality* that would capture the additional knowledge the programmer has, regarding the origin, or reason for the existence, of any particular fact in the expected outcome. The programmer reasons that a given fact is in the expected outcome either because it is a basic (i.e. should be an *EDB*) fact, or because it is derived, in a number of steps, from one or more of the basic facts. A partial ordering of facts in the interpretation is thus implied and is required to properly capture the programmer's intuition.

An *intended interpretation* is the one the user has in mind for his/her program; it usually embodies the user's intuition. We will denote the intended interpretation by I and use it to define the two central notions relevant to the process of diagnosis: those of a *wrong fact* i.e., a fact in $M(P)$ but not in I, and a *missing fact* i.e., a fact in I but not in $M(P)$. Wrong facts and missing facts are the manifestation of errors that are observed by the programmer. Their causes stem from either a *missing basic fact* or a *wrong clause* in the program P. We will define these concepts in the sequel. The *diagnostic system* or synonymously, the *debugger*, is a program that accepts as input (1) a program P and (2) a missing or a wrong fact. Upon input, the debugger,

(1) Drives an interactive dialogue, by occasionally prompting the programmer for information about the intended model of the program.
(2) Provides a conclusion in the form of the identification of either a missing basic fact or a wrong clause in the program.

The diagnosis proceeds on a fact-by-fact basis.* For each instance of a missing or wrong fact submitted, a conclusion is provided. Therefore, the correctness of a program after diagnosis and the application of the corrections is not absolute but is relative to the facts that were submitted for diagnosis.

The event of a wrong fact in the answer is diagnosed using a proof-tree. The user directs the diagnostic system "down" the proof-tree until an incorrect clause is discovered. The process is as follows. Suppose that a wrong fact was produced by a clause, whose body literals were satisfied. If all the body literals were correctly satisfied, yet the conclusion should not have been derived, the clause is labeled as wrong. Otherwise, one of these body literals, say b, must have

* Observe that a single "programming error" may result in both missing and wrong facts; it may be discovered by treating either.

been wrongly satisfied; otherwise, the conclusion would have been correct. Thus, the user must identify b and the process recurs on it. The responsibility of producing and maintaining these proof-trees is relegated to the $\mathcal{LDL}$ compiler. Technically, this is not a difficult problem.

The event of a missing fact occurs because either there is no rule in the program such that its head unifies with the missing fact, or if such a clause exists, its body can not be satisfied in $M(P)$, following the unification. In the latter case, there are two possibilities. If the clause instance is satisfied in the intended interpretation, subject to causality constraints, then there is a body literal which is satisfied in I but is not in $M(P)$, such a literal b is identified and the reason for b being missing is searched for, recursively. In case all clause instances fail to be satisfied in I, subject to causality constraints, a cause has been identified.

As mentioned, negation in $\mathcal{LDL}$ is handled using the idea of layering which means that if $\neg q$ (negative q) appears in the body of a rule with head predicate p then q is defined only in terms of predicates that belong to layers lower than that of p. Now, during the navigation down the proof-tree to diagnose a wrong fact, a negated predicate, say $\neg q$, may be visited. To claim that $\neg q$ is wrong is equivalent to claiming that q should have been produced in a lower layer i.e., it is treated as missing in a lower layer. Likewise, if $\neg q$ is missing, the reason may be because q is erroneously derived in a lower layer and hence, it becomes a wrong fact in the layer of q. Therefore, in diagnosing stratified $\mathcal{LDL}$ programs there is an interplay between the procedures required to analyze wrong and missing facts. This presents no great difficulty. This interplay is well-known in similar dubugging contexts, see Ref. 6).

2.3 The User's Mental Picture

We introduced the notion of an intended interpretation and noted that we use this notion as an embodiment of the users' intuition with respect to the program. The exact reason as to why the user would consider a fact to belong to his/her intended interpretation falls within the realm of the human psyche and does not follow from the definition, nor is it dependent on the program P. The purpose of this concept is, however, to provide a formal guarantee, that *if* the user interacts with the system in accordance with this definition, i.e., the user "plays by the rules", then convergence to the origin of his reported problem, in the program P, can be guaranteed. In this section and the next we elaborate on various interpretations that can be used to formalize this reasoning process and settle on one version that we assume the user to follow when he/she debugs an $\mathcal{LDL}$ program.

If p is a fact then $\neg p$ is a *negative fact*, we shall also refer to a fact as a *positive fact*; a *general fact* is a positive or a negative fact. The user thinks of a set of general facts. Each of these general facts may be justified by the user in terms of other facts. We will use the notation $p \rightarrow q$ to denote that the user uses general fact p to justify general fact q. The relation $\rightarrow$ is called the *justification* *relation*.

This model of reasoning would imply a directed graph (V, E) in which the nodes in V denote facts, positive or negative, and the edges in E denote justifications. So, there is an edge (p, q) in E from general fact p directed to general fact q iff $p \rightarrow q$ is in the justification relation. This could possibly include circular justifications. This framework is too general however for our purposes. Additional restrictions are assumed to allow effective debugging. The additional restrictions that we need are as follows:

(1) The set of general facts the user thinks of is consistent, i.e. for no fact p in that set is $\neg p$ also in the set, and vice versa.
(2) The justification relation is acyclic. This prevents circular reasoning.
(3) Negative facts can only be used for a justification but cannot be justified themselves—they are given. In other words, an edge of the type $\neg p \rightarrow q$ is admissible but $p \rightarrow \neg q$ or $\neg p \rightarrow \neg q$ is not.
(4) In addition to restriction (2) we assume that facts are graded, or categorized, as more basic or less basic, and that there are no arcs from a less basic fact to a more basic fact. We impose a *layering scheme* on the set of general facts. The scheme is such that the lower the layer the more basic are facts in the layer. The number of layers is *finite* while the number of facts in each layer is potentially infinite. Furthermore, for all predicate symbols p, the facts for p must be of the same layer.
(5) Each fact is justified by a finite number of facts i.e., the in-degree to a node representing a fact is finite (it may also be zero, i.e. no incoming edges at all).
(6) Each descending justification chain from a fact to one if its justifying facts, to one of its justifying facts etc., is finite.

We will refer to a structure that meets the restrictions mentioned here as an *acceptable structure*.

2.4 Generated Interpretations

A generated interpretation is an acceptable structure which is constructed in a particular fashion, as detailed below. This "construction" is done in the user's mind and there may be no program which can manufacture it. So, a generated interpretation is another, more refined, abstraction as to what kind of structures users may have in mind.

First consider a single layer generated interpretation for a set of predicate symbols of interest; intuitively, the predicate symbols of a program P without negation (or a single layer in a program P).

A *generated interpretation* is built in *stages*. In each stage a set of facts is added. At stage 0 a set S_0 containing *basic facts* is created; this set may be infinite. A fact added at stage i is said to be i-generated. If s is i-generated and t is j-generated, $i < j$, then s is generated *prior* to t and t is generated *following*

s. The number of stages may be finite or infinite. A fact may be added only once. Clearly, if s is i-generated then it is not j-generated, unless $i = j$.

At each stage, finitely or infinitely many facts may be added. Finally, for *all* facts q for predicate symbols in the set of interest which are not added in any stage, add $\neg q$ to stage 0. (When presenting generated interpretations, this addition is implicit and will not be shown.)

There is an additional structure which is added to the interpretation, namely, a justification relation. Each fact f in a stage may be *connected* via justification edges to a finite number of facts at lower stages (which justify f). By construction, a generated interpretation satisfies the six requirements for being an acceptable structure. (The set of facts is consistent, an acyclic $\rightarrow$ relation, no edge enters a negative fact, there is a single layer, the number of edges entering a fact is finite, the stages limit the length of descending chains.)

Now consider a multi-layer generated interpretation, intuitively, for a program P with negation. Such a generated interpretation is formed from a collection of generated interpretations, one for each layer. With respect to edges, a fact in (some stage of) layer k may also be connected to finitely many facts whose predicate symbols are in layers lower than k. All facts in layers lower than layer k are considered of a prior generation with respect to the facts of layer k. By construction, the result is an acceptable structure.

Notationally, a generated interpretation is a 5-tuple $GI = (k, I, L, S, E)$ where k is the number of layers, I is the set of positive facts, L is a function from I to $\{1, ..., k\}$, S is a function that assigns to each fact in I a stage number (a non-negative integer), and E is the set of edges. When we talk about a generated interpretation as an interpretation we shall refer to its I component which is a subset of the Herbrand base. While the user reasoning process that conceptually produces a generated interpretation resembles the process of bottom-up repeated rule-applications, there is not necessarily a program underlying this process—the programmer's intuition may be wrong and a program for which I is a model may not even exist.

For programs that may use negation we make the following assumption: the layers in the program and those in the definition of an acceptable structure coincide. This assumption means that the user has defined the layers in his program in accordance with the layering scheme in the generated interpretation. Also, recall our assumption about layered programs that the first layer (i.e. layer 1) of a program is composed exclusively of *EDB* predicate symbols.

Figure 1 graphically shows an instance of a generated interpretation. The small horizontal lines denote stages within the layers, the circles on line segments denote specific facts in the layers, and the arrows denote the causal connections. We maintain that most intended interpretations are generated ones; this is not a formal claim and hence can not be proved.

Let GI be a generated interpretation. Let P be a layered program.

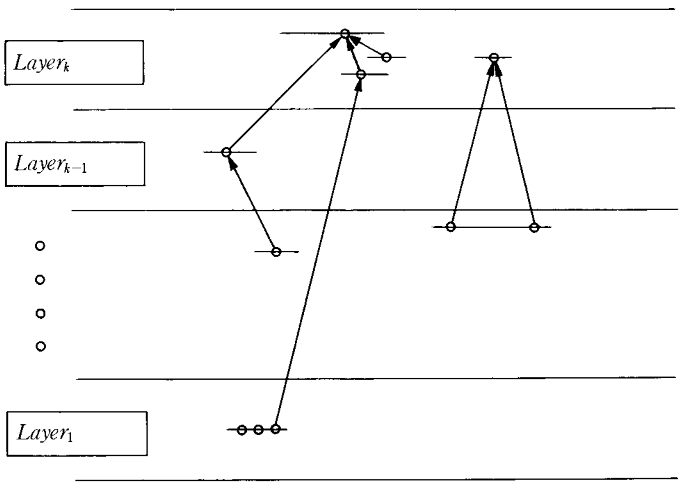

Fig. 1 A Generated Interpretation.

Definition 2.1 (Missing Atom, Missing Fact)
An atom s is a *missing atom* (*ma*) w.r.t. GI in program P if

- It is satisfiable in I, i.e. there is an instance $s' \in I$ of s.
- It is missing in $M(P)$; i.e. no instance of this atom is in $M(P)$.

If s is ground then s is a *missing fact* (*mf*). □

We now define the notion of a *missing basic fact* (*mbf*) w.r.t. GI that identifies a basic deficiency of P w.r.t. I. Namely, a fact f, $f \in I - M(P)$ such that f could not be derived in P, where the derivation uses I as EDB facts and respects the causality manifested by generations, layering, and edges.

Definition 2.2 (Missing Basic Atom, Missing Basic Fact)
An atom s is a *missing basic atom* (*mba*) w.r.t. GI in program P if

- It is a missing atom.
- There is no ground instance of a rule in P, such that its head is an instance of s and its body is satisfied in I in such a way that the head instance is some generated fact in I, and each body literal instance is some general fact, of a lower stage or a lower layer, which is connected to s via an edge in E.

If s is ground then s is a *missing basic fact* (*mbf*). □

Definition 2.3 (Incorrect Clause)
A clause r in P is *incorrect* w.r.t. GI if it is not satisfied in I (as a logic formula).
–

Definition 2.4 (Correct Program, Incorrect Program)
A program P is *correct* w.r.t. GI if $M(P) = I$; otherwise P is *incorrect*. □

Consider a program P consisting of the clause $a \leftarrow a$, the intended generated interpretation GI has a single layer containing a single fact, a, and no justification edges. $M(P)$ is apparently empty. Then, P has a mbf, namely a. This is because a is in $I - M(P)$ and the (only) clause $a \leftarrow a$ cannot be properly satisfied in I. Indeed, the clause head, namely a, is in I, but its body literal, again a, is not connected to a via a justification edge. Now consider a program P' consisting of the clause $b \leftarrow$, the intended interpretation is empty and $M(P) = \{b\}$. Then, P' has an incorrect (unit) clause, namely $b \leftarrow$, because it is not satisfied in I due to b's absence. The above two cases illustrate simple instances generalized by the following theorem.

Theorem 2.1
P is incorrect w.r.t. GI iff either P has an incorrect clause or a missing basic fact.

Proof
($\leftarrow$) We consider the two cases:

Case (i): Let r be an incorrect clause instance for a clause in P; thus r is not satisfied in I. However, since $M(P)$ is a model for P, r is satisfied in $M(P)$. This implies $M(P) \neq I$.
Case (ii): Let s be a mbf; this implis $M(P) \neq I$.
In both cases P is incorrect by definition.

($\rightarrow$) By definition, $M(P) \neq I$. The proof is by induction on the number of layers, e, in P.

Basis(*Major Induction*): $e = 1$.
So, the program only has unit clauses.

Case 1: there is a fact $s \in M(P) - I$. As s is an instance of a unit clause in P, this unit clause is incorrect since it is not satisfied in I.
Case 2: there is a fact $s \in I - M(P)$. Since $s \in I - M(P)$ it cannot unify with any unit clause of P. Since $e = 1$ there are only unit clauses. Thus, this fact is a mbf as it can not unify with any rule head in P.

Induction(*Major Induction*): $e > 1$.

Case 1: there is a fact $s \in M(P) - I$. Consider a partial proof-tree T for s in which each internal node is a positive fact together with a rule in P used to derive it, and the fact and its children comprise a ground instance of the rule. The root is s. Each leaf is either an *EDB* fact or of the form $(\neg p)$ where p is a fact. Use $fact(v)$ to denote the ground literal in node v. Let v be a node in T such that $fact(v)$ is not satisfied in I but all of v's children are satisfied in I. The existence of v follows by (minor) induction on the height of T. If v is an internal

node then the rule labeling v is incorrect. If v is a leaf and positive then $fact(v)$, which it represents, is incorrect; this fact is an instance of a unit clause in P, this unit clause is incorrect. If v is a leaf and negative, say $(\neg q)$, then q is a fact which is not in $M(P)$ but is in I. Let z be the layer of P where the relation for the predicate symbol of q is defined, let $z = 0$ if the relation is not defined in any layer of P. If $z = 0$ then P has a mbf. Otherwise, consider P' which is P minus rules defining predicates in layers higher than z. Since P' is layered and $z < e$, and using the induction hypothesis, it follows that either P' has an incorrect clause or a missing basic fact. In either case P has an incorrect clause or a missing basic fact.
Case 2: there is a fact $s \in I - M(P)$. If (the predicate symbol of) s belongs to a lower layer than e, say l, then, by induction, we are done. So, let s be a k-generated general fact of this layer (e). We prove by (minor) induction on k that P has either a mbf w.r.t. GI or an incorrect clause.

Basis(*Minor Induction*): $k = 0$.
If there is no (ground) instance r' of a rule r in P, such that its head is s and its body is satisfied in I (in the usual logical sense) where each body literal of r' is some fact in I, generated prior to s, which is connected to s via an edge in E, then s is a mbf.

Suppose such a (ground) instance r' exists. Since $k = 0$, there are no body literals from this layer (e). If the body is empty then r' is a unit clause and therefore $s \in M(P)$; contradiction.
Hence the body of r' must consist solely of (i) positive literals $q \in I$, where q is defined at a lower layer, or (ii) negative literals $(\neg q)$ such that $q \notin I$. It can not be that all positive body literals in this rule instance are in $M(P)$ and all negative literals $(\neg q)$ are such that $q \notin M(P)$, for this would imply that s is in $M(P)$ as well.

Case (i): there exists $s' \notin M(P)$ a (ground) positive literal in this rule instance which is in I. Clearly, s' comes from a lower layer. Then, by major induction, there is a mbf for P' w.r.t. GI or an incorrect clause in P' where P' is P minus all clauses in layers above the layer defining the predicate of q. In either case so does P.
Case (ii): there exists, in this rule instance, a negative literal $(\neg q)$ such that $q \in M(P)$. Since $q \notin I$, $q \in M(P') - I$ where P' is P minus all clauses in layers above the layer defining the predicate of q. By (major) induction, P' has either a mbf w.r.t. GI or an incorrect clause, in either case so does P.

Induction (*Minor*): $k > 0$.
If no rule head in P unifies with s we are done, because s is a mbf. Suppose there is a (ground) instance of a rule in P such that its head is s and its body is satisfied in I with positive facts which are either generated prior to s or at a layer lower than e, and some negative ground literals which are satisfied in I (with the

edge restrictions of E). It can not be that all positive body literals in this rule instance are in $M(P)$ and all negative literals ($\neg q$) are such that $q \notin M(P)$, for this would imply that s is in $M(P)$ as well.

Case (i): there exists $s' \notin M(P)$ a (ground) positive literal in this rule instance which is in I. First suppose that s' is in the same layer as s. Since s' is j-generated for some $j < k$, it follows by minor induction that there is a mbf for P w.r.t. GI or an incorrect clause in P.
Now suppose s' comes from a lower layer. Then, by major induction, there is a mbf for P' w.r.t. GI or an incorrect clause in P' where P' is P minus all clauses in layers above the layer defining the predicate of q. In either case so does P.
Case (ii): there exists, in this rule instance, a negative literal ($\neg q$) such that $q \in M(P)$. Since $q \notin I$, $q \in M(P') - I$ where P' is P minus all clauses in layers above the layer defining the predicate of q. By (major) induction, P' has either a mbf w.r.t. GI or an incorrect clause, in either case so does P. □

It should be noted that the above theorem is analogous to Proposition 3 in Ref. 6).

2.5 Debugger-User Interaction

We introduced the notion of a diagnostic system and mentioned that it prompts the user for information, that will be used to converge to a conclusion. The user, or alternatively a specification of the program that interacts with the debugger, acts thus as an *oracle*, knowledgeable about the generated interpretation. During interaction, the user must meet the following requirements:

- When asked about the validity of a general fact or is asked to satisfy an atom p, the user responds truthfully, i.e., consistent with the generated interpretation GI.
- When asked by the debugger to justify a fact f, by satisfying a rule body that would have generated f, the user responds by using only general facts, belonging to stages (or layers) lower than that of f and which are edge connected to f.

Note that these conditions are, again, part of the idealized "rules of the game" that, formally guarantee the convergence of the debugging process. In practice, the user receives considerably more assistance from the system and is thus not expected to possess all of the knowledge that these conditions would imply.

Any oracle that meets these requirements is called a *reliable oracle*. Example 2.1 demonstrates these concepts.

Example 2.1
Consider the program P:

```
        p(X) ← w(X, Y), q(Y).
```

```
w(1, 2).
q(2).
```

Assume the generated interpretation GI to be a two layer structure and be represented as:

Layer 1:

$$\{(0;\ q(2)),\ (0;\ w(1,\ 2),\ (0;\ w(5,\ 17)),\ (0;\ w(5,\ 20))\}$$

Layer 2:

$$\{(0;\ p(1)),\ (1;\ q(17)),\ (2;\ p(5)),\ (3;\ q(20))\}.$$

Each element of I is of the form (*stage-number*; *fact*). Furthermore, assume that each fact in layer 2 is edge-connected to all of the facts in stages below its own or in layer 1.

The response to the query $?p(X)$ is $p(1)$. The user expects to receive in addition the answer $p(5)$ and thus submits P and *missing fact* $p(5)$ to the debugger. Note that the stage number of the missing fact is 2. The debugger responds by a request to satisfy the rule: $p(5) \leftarrow w(5,\ Y),\ q(Y)$. The user response in form of: $p(5) \leftarrow w(5,\ 17),\ q(17)$ would be reliable, since only facts from stages < 2 are used. The user response: $p(5) \leftarrow w(5,\ 20),\ q(20)$. would be unreliable since $q(20)$ from stage 3 (> 2) is used in the justification. □

A *problematic fact* is either a missing fact $s \in I - M(P)$, or a wrong fact $s \in M(P) - I$, together with an indication as to which is the case.

Definition 2.5 (Debugger Input)
The input to a debugger consists of:

- A program P.
- A problematic fact.
- A reliable oracle. □

The debugger responds either by not generating output (it loops forever), or by an atom identified as a basic missing fact, or by identifying a clause as an incorrect clause.

Definition 2.6 (Sound Debugger)
A debugger is *sound* if, for all debugger inputs,

- A clause returned and identified as wrong is indeed wrong, or,
- An atom returned and identified as missing is indeed a basic missing atom. □

Definition 2.7 (Total Debugger)
A debugger is *total* if it is sound, and for all debugger inputs it returns an answer (i.e. does not loop forever). □

§3 Implementation of Diagnostic Systems

3.1 Diagnosis of Compiled Programs

The debuggers of Ref. 11) are of the *meta-interpreter* type. The computation of the debugger tracks the computation of the program to be diagnosed at the level of the procedure call. This solution is adequate if the intuitive understanding of the programmer w.r.t. his/her program is in terms of procedure calls; as is indeed the case in Prolog. When we come to $\mathcal{LDL}$ however, we cannot rely anymore on this simple model of execution. While the semantics of the language are described in a bottom-up, fixpoint model of computation,[8] the *actual* execution of a program is a complex sequence of low-level procedure calls. This sequence is the result of the compilation of the program and is optimized to take advantage of whatever information the user supplied. In particular, different query forms in the queries may result in different execution strategies. Consequently, a simple trace of these procedure calls is of little use to the programmer in understanding the behavior of his/her program. We considered several options and chose the following

> Indentify the origin of a fault in the real program by a top-down, query/subquery approach. The debugger, which itself is implemented in another language e.g., Prolog, is a shell program between the programmer and his/her $\mathcal{LDL}$ program: it poses queries and uses the results to drive the interaction with the user. In this process, the user is expected to act as a reliable oracle.

In the following section we present a basic version of this system and provide a proof for its totality. The basic scheme can be applied to $\mathcal{LDL}$ programs without set-terms, grouping terms, updates or the *choice* construct.

3.2 The Basic Debugger

The basic debugger presented in this section closely resembles that of Ref. 11). Let P be the program to be debugged. We make the following, fairly strong, important assumption:

> (A1) Only queries q such that (P, q) is in C are posed throughout the debugging process. This implies that all such queries will be correctly answered by the $\mathcal{LDL}$ run-time system within finite time.

The basic debugger and its variations are expressed in a Prolog* like language plus additional, ad-hoc constructs that serve to query the $\mathcal{LDL}$ program to be debugged. We are only interested in the first answer returned by the

* We could have used any other language for the purpose as well. The use of Prolog is merely a question of convenience.

debugger. The additional constructs that we use to obtain access to the $\mathcal{LDL}$ source program are the predicate *clause*(A, B) where A is instantiated to a clause-head and B to its body, the body is either a single literal (*true* is used for representing an empty body) or of the form ($B1$, $B2$) where $B2$ is a body. The debugger can execute an $\mathcal{LDL}$ query by *executeLDL*(q) where q is a single literal query. A crucial assumption is that each call of *executeLDL*(A) terminates correctly: either it fails, i.e. it correctly returns no answers, or it succeeds in which case the Prolog backtracking mechanism will eventually consider all, and only finitely many correct answers for A.

Unlike Prolog, $\mathcal{LDL}$ does not employ negation by failure. The answer to *executeLDL*(q) is always a set of general facts, which are ground instances of q, this is so regardless of whether q is a positive or a negative literal. That set may of course be empty, if this is the case then the Prolog call to *executeLDL*(q) fails.

It turns out that for the basic debugger of this section we can weaken the above assumption (A1). For the basic debugger the following assumption, concerning the *program* to be debugged, suffices:

> (A2) All of the *ground* single literal queries q, that can be posed to the program P are such that (P, q) is in C. Again, this implies that all such queries will be correctly answered by the $\mathcal{LDL}$ run-time system within finite time.

In general, it is unclear whether the fact that the program is safe w.r.t. the original query form and the assumption (A2) above imply that all the queries, invoked by some debugger's interactions, are safe too. However, in the basic debugger of this section, *executeLDL* is called only with (single) ground literals. In this case, due to (A2), all of the interactions are safe.

The program in Fig. 2 implements the debugging process. For typographical reasons we use *not*(q) instead of $\neg q$. The meaning of the statement:

Q:- A, *if* R *then* B *else* C, D.

is:

Q:- A, P, D.
P:- R, B.
P:- $\neg R$, C.

The debugger receives from the user (the oracle) a fact A in $M(P) - I$ or a fact A in $I - M(P)$, together with its proper classification. Initially, the search is directed to *wrong*(A, R) or to *miss*(A, R, A). In the above calls, R will contain the debugging conclusion. The third argument to *miss* serves as a *context*, i.e. the ground instance of the head of the clause for which the user is asked to supply ground instances for the body literals which are true in I, subject to causality constraints (i.e. justification edges from previously generated general facts). The context is needed because the n body literals are treated separately (see below).

In analyzing body non-satisfaction, n nested loops, for the n body literals, are handled by the two clauses of *miss*1. The first handles the case of a body composed of more than one literal and the second handles the case of a single literal body or the rightmost body literal (the innermost loop). Basically, we are doing a nested-loop-join of relations such that R_i contains all possible valid ground instances for body literal l_i in I, $1 \leq i \leq n$. While doing this join, each candidate added to form a result fact, is checked using $\mathcal{LDL}$. Another point worth mentioning is that the oracle's answers are guaranteed to form descending chains in the generated interpretation. This is essential for arguing termination. Example 3.1 shows a complete debugging session.

Example 3.1
Consider the program P:

```
q(X, Y) ← r(X, Y).
q(X, Z) ← q(X, Y), r(Y, Z).
r(1, 2).
r(3, 4).
```

The user poses the query $?q(1, 4)$ and receives the response *false*. Since the user expects to receive *true*, he submits $q(1, 4)$ *missing* to the debugger. The interpretation GI that the user has in mind has two layers. We assume that each fact is edge connected to all facts in stages or layer lower than its own. GI is represented as follows:
First layer:

$$\{(1;\ r(2, 3)),\ (2;\ r(3, 4)),\ (3;\ r(1, 2))\}.$$

Second layer:

$$\{(0;\ q(1, 2)),\ (1;\ q(3, 4)),\ (2;\ q(2, 3)),\ (3;\ q(1, 3)),\ (4;\ q(1, 4)),\ (5;\ q(2, 4))\}.$$

The chronology of events, between the user, the debugger, and the $\mathcal{LDL}$ system is described in Fig. 3. The debugger will return the conclusion *noClauseMatch*($r(2, 3)$) indicating that $r(2, 3)$ is a mbf. Note that the user responds to the debugger promptings in accordance with the interpretation GI. To prompts pertaining to facts e.g., "$r(1, 4)$? " the user response is "no" since $r(1, 4) \notin I$. To prompts pertaining to non-ground atoms e.g., "$q(1, Y)$? " the user response is with a fact from a lower stage in GI. The user response to $q(1, Y)$ in the context of the missing $q(1, 4)$ is $q(1, 3)$ which is at a lower stage in GI. □

We assume that there are no system, i.e. built-in, predicates. Adding such predicates presents little difficulty. Basically *validground* asks $\mathcal{LDL}$ instead of the

```
/* invariant for miss is that query is valid but fails in LDL */
miss(not(Q), R, Context):- wrong(Q, R).
miss(A, R, Context):-
                if clause(A, B) fails
                  then R = noClauseMatch(A) else fail.
miss(A, R, Context):- if (clause(A, B), miss1(B, R1, A)) succeeds
                      then R = R1
                      else R = atom(A). /* a mbf found */
miss1((A, B), R, Context):- validground(A, Context),
                        if executeLDL(A) succeeds
                        then (if miss1(B, R1, Context)) succeeds
                             then R = R1 else fail)
                        else miss(A, R, A).
miss1(A, R, Context):- validground(A, Context),
                /* the if check below is redundant */
                if executeLDL(A) fails then miss(A, R, A)
                                        else fail.

/* invariant for wrong is that query is invalid but succeeds */
wrong(A, R):- executeLDLplusTree(A, T),
              navigateDown(T, R).

executeLDLplusTree(A, T):- perform ExecuteLDL(A), returning one
                           instance A' at a time via backtracking.
                           Let T be the (partial) proof-tree for A'.

navigateDown(T, R):- T is a (partial) proof-tree.
                 case 1: T is a leaf
                 if T = (not(A1)) then miss(A1, R', A1);
                 if T = (A, A1) then R = wrongUnitClause(A1);
                 case 2: T = (clause, f, T1, ..., Tm)
                 if all children of T are valid in I
                 then R = wrongClause(clause)
                 else let Ti be T's first child which is
                        not valid in I, navigateDown(Ti, R)

validground(A, Context):- A reliable oracle: returns a valid
                          general fact for literal A with
                          generation prior to Context, and
                          which is edge-connected to Context.
```

Fig. 2 Basic Debugger.

user for valid ground instances. Because of the invariant for *miss* that query is valid but fails by *executeLDL*, there is never a possibility that *miss*1 will be called with $A = true$. An idiot-proof debugger can trap for this condition.

The notation for (partial) proof-trees T is as follows. If the tree is a leaf containing a negated fact $\neg f$ then $T = (not(f))$. If the tree is a leaf containing a fact f for unit clause A, then $T = (A, f)$. Otherwise T is of the form (*clause*, f, T_1, ..., T_m) where the first entry is the labeling clause for T's root, the second entry is the fact instance of the clause head contained in the root, and T_i, $1 \leq i \leq m$, are the subtrees whose roots are the children of the root of T. Also, *true* leaves are pruned from proof-trees.

Theorem 3.1 below is similar in structure to Theorem 2.1. The difference is that we are to prove correct a particular algorithm that implements the ideas of Theorem 2.1. This implies that we must take a careful look at the inner workings of the basic debugging algorithm and see how the assumptions we made guarantee totality. *miss* and *wrong* in Theorem 3.1 pertain to the miss and wrong clauses of the basic debugger in Fig. 2. The basic debugger supplies the following information. A wrong clause is classified as a *wrongUnitClause* or a *wrongClause*. A mbf is classified as either a *noClauseMatch* fact or an *atom*, the latter means there are matching clauses but the fact returned is a mbf.

Theorem 3.1
Given $f \in I - M(P)$ as input to *miss* or given $f \in M(P) - I$ as input to *wrong*, either a missing basic fact or an incorrect clause is returned.

Proof
See Appendix 1. □

3.3 Modifications to the Basic Debugger

As observed by Refs. 9) and 11), the basic debugging scheme can be improved by, (1) Utilizing the user-knowledge of those rules that were never meant to participate in the production of the missing fact and, (2) Reducing the number of questions to the oracle. The basic debugger does not take advantage of user's knowledge of which clauses are the relevant ones, i.e., they could be the potential source of the problem, and which clauses are not, i.e., they were never meant to participate in the production of the missing fact. Next, we try to take advantage of this knowledge. The clause:

```
miss(A, R, Context):- if (clause(A, B), missl(B, Rl, A)) succeeds
                      then R = Rl
                      else R = atom(A).
```

becomes now:

User(Oracle)	**Debugger**	**Context**	$\mathcal{CDL}$
$q(1, 4)$ missing !	—	$q(1, 4)$	—
—	$r(1, 4)$?	$q(1, 4)$	—
no !	—	—	—
—	$q(1, Y)$?	$q(1, 4)$	—
yes, $q(1, 3)$!	—	—	—
—	—	—	?$q(1, 3)$, no !
$q(1, 3)$ missing !	—	$q(1, 3)$	—
—	$r(1, 3)$?	$q(1, 3)$	—
no !	—	—	—
—	$q(1, Y)$?	$q(1, 3)$	—
yes, $q(1, 2)$!	—	—	—
—	—	—	?$q(1, 2)$, yes !
—	$r(2, 3)$?	$q(1, 3)$	—
yes !	—	—	—
—	—	—	?$r(2, 3)$, no !
$r(2, 3)$ missing !	—	$r(2, 3)$	—
—	noClauseMatch($r(2, 3)$)	—	—

Fig. 3 User-Debugger Interaction.

```
miss(A, R, Context):- if (askclause(A, B, Context), miss1(B, R1, A))
                      succeeds then R = R1
                                else R = atom(A).
```

Predicate *askclause*(A, B, *Context*) is true if A:- B is a clause instance which is satisfied in I so that the satisfying instance respects causality viz. Context and viz. the generation numbers of the head and the body literals. This is basically an oracle call to be answered by the user; giving an incorrect yes answer can not hurt, giving a correct no answer can limit the amount of questions asked by the debugger, giving an incorrect no answer may result in failure to find an answer (i.e. identifying a bug).

Further improvements can be introduced by changing the *miss*1 clause for conjunction so that *validground* is "pushed" inside. This change has the potential of reducing the number of oracle questions. The clause becomes:

```
miss1((A, B), R, Context):- if (executeLDL(A), miss1(B, R1, Context))
                            succeeds then R = R1
                                      else (validground(A, Context),
                                            if executeLDL(A) fails
                                              then miss(A, R, A)
                                              else fail).
```

Observe that it is now possible that *executeLDL*(A) be called with $A = \neg q$. Clearly, if A is not ground then there may be infinitely many answers to the invocation. To satisfy assumption A1, either we exclude programs in which the above scenario may happen or we use the new *miss*1 clause above only when A is positive, and otherwise use the older version of *miss*1. The same choices apply in the comprehensive debugger and its extensions (see Section 5).

It is shown that this debugger is total as well.

Theorem 3.2
For the debugger with the improvements of this section, given $f \in I - M(P)$ as input to *miss* or given $f \in M(P) - I$ as input to *wrong*, either a mbf or an incorrect clause is returned.

Proof
See Appendix 1. □

§4 A Debugging Scheme for $\mathcal{LDL}$ Updates

The basic debugging scheme, discussed in the previous section, can be applied to $\mathcal{LDL}$ programs that do not include any set, or grouping constructs. Nor can they include any update rules or procedural extensions. In this paper we only discuss in some detail the extensions required for updates; the extensions required for handling sets, for the procedural extensions i.e., *if — then — else*, *forever* and the *choice* construct will not be detailed.

For a detailed description of updates in $\mathcal{LDL}$ the reader may consult Ref.

8). Briefly, updates apply only to base relations. There are two kinds of updates: for adding facts, an *EDB* atom prefixed with '+', and for deleting facts, an *EDB* atom prefixed with '−'. An updating literal is called an *updater*. Due to $\mathcal{LDL}$'s restrictions, any legal $\mathcal{LDL}$ program containing updates can be transformed into the following canonical form:

$$\begin{array}{l} h \leftarrow L_0,\ U_0,\ ...,\ L_{n-1},\ U_{n-1},\ L_n. \\ p1 \leftarrow b1. \\ \vdots \\ pm \leftarrow bm. \end{array}$$

Above, each L_i, $i = 1, ..., n$, is a sequence of literals whose predicates are defined by rules within $pi \leftarrow bi$, $i = 1, ... m$, U_j, $j = 1, ..., n - 1$, is an updater, and no rule references in its body the predicate in h.

So, without loss of generality, we consider a program containing a single updating rule where the head predicate of this rule does not appear in any clause body. The semantics of the constructs in this section is based on the notion of a *state*. The state of the program contains all of the facts of the program at some point in time. As a result of the execution of an update the state may change to a new state.

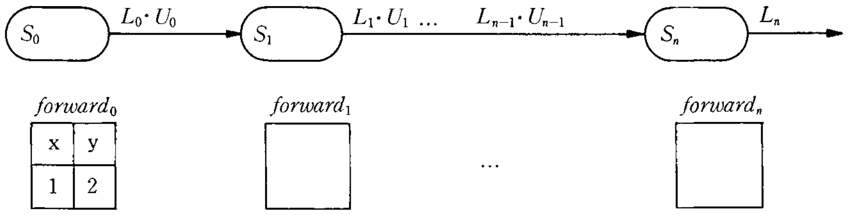

Fig. 4 State Transitions in the Presence of Updates.

Suppose we have a terminating program. Figure 4 depicts the execution — a sequence of states S_0, S_1, ..., S_n. Each state-transition (S_i, S_{i+1}), $i = 0, ..., n - 1$, is associated with a particular *updater* U_i, i.e. $+p_i$ or $-p_i$ where p_i is an atom, $i = 0, ..., n - 1$. For debugging purposes this sequence of states must be kept as well as the particular updater applied to each state. Each literal in the execution is interpreted relative to a current state — the one generated as a result of the execution of the most recent updater in the execution path. With each S_i there is a relation *forward*$_i$ of allowed combinations of bindings that were applicable in all of the states S_j, $j < i$.

Intuitively, *forward*$_i$ can be thought of as the relation for the join of the relations corresponding to the body literals upto the i-th updater, projected onto columns each of which corresponds to a variable appearing in these literals or the head. *forward*$_0$ contains the bindings supplied from the rule head, if any. The sequence of literals, between two consecutive updaters U_i and U_{i+1}, is denoted as L_i. L_0 precedes the first updater (may be empty), and L_i which follows the i-th updater, for $i > 0$, may also be empty.

The exact actual implementation of recording the *forward*$_i$ relations and the S_i states is left unspecified; one way is to record changes from the previous S_i or *forward*$_i$. The user's mental picture is assumed to be a sequence of generated interpretations. Given some recording mechanism, the debugger presents the states, one at a time in order, to the user. With each such presentation, the user is asked to certify the state as correct, i.e. agreeing with the corresponding generated interpretation. The certification may be done by the user by examining the facts of *forward*$_i$ and S_i directly, by asking queries against S_i or *forward*$_i$, or by simply assuming they are correct. Let S_w be the first state that is not certified as correct. The reasons for non-certification which we analyze are:

- There is wrong fact $p(t_1, ..., t_n)$ in S_w; there are three cases to consider:
  (1) S_w was neither created by an updater $+p(T_1, ..., T_n)$ nor by an updater $-p(T_1, ..., T_n)$. The p-content of S_w is the same as that of S_{w-1}. Then, the certification is incorrect, i.e., S_{w-1} should not have been certified as correct.
  (2) S_w was created by an updater $+p(T_1, ..., T_n)$. Intuitively, $p(t_1, ..., t_n)$ should not have been added, however, it was added.
  Let P' be P together with a new clause

  $$a \leftarrow forward_{w-1},\ L_{w-1},\ T_1 = t_1, ..., T_n = t_n$$

  where a is a new predicate symbol. Perform *wrong*(a) on P' with the relations for *EDB* predicate symbols being S_{w-1}. This may identify the reason why $p(t_1, ..., t_n)$ was added. If the reason is not found then, again, S_{w-1} should not have been certified as correct.
  (3) S_w was created by an updater $-p(T_1, ..., T_n)$. Intuitively, $p(t_1, ..., t_n)$ should have been deleted, however, it was not deleted.
  Let P' be P together with a new clause

  $$a \leftarrow forward_{w-1},\ L_{w-1},\ T_1 = t_1, ..., T_n = t_n$$

  where a is a new predicate symbol. Perform *missing*(a) on P' with the relations for *EDB* predicate symbols being S_{w-1}. This may identify the reason why $p(t_1, ..., t_n)$ was not deleted. If the reason is not found then, again, S_{w-1} should not have been certified as correct.

- There is a missing ground atom $p(t_1, ..., t_n)$ in S_w. Again, there are three cases similar to the ones analyzed above.

We expect the system to inform the user about the column names (i.e.

variables) of table *forward*$_{w-1}$.

§5 A Comprehensive Debugger

The debuggers, that we have discussed so far, suffer from some deficiencies. First, if *executeLDL*(A) is used, instead of *validground*, to guide the search then a call may generate an incorrect result, i.e. in $M(P) - I$. Based on this result *miss*(B, $R1$, *Context*) is used and it may discover a bug. This bug may look confusing as it comes from an unexpected source. The sequence of questions leading to this bug may also appear strange. It seems that wrong should be invoked, if possible, on this incorrect *ExecuteLDL*(A) answer. This idea is used in the N.3 debugger by Naish.[9] However, it seems bothersome to ask a question for each fact. The alternative is to ask a question for each fact-set containing all *ExecuteLDL*(A) answers.

Second, the search is not deterministic. Depending on (accidental) scanning order, the bug discovered may be different. If all answers to *ExecuteLDL*(A) are generated then it is possible to ask the user (a) are all answers correct, and (b) are all correct answers present. Then, we can decide which direction takes precedence when multiple bugs are present, e.g. if the answer to (a) is no and to (b) is no, we can decide that we always activate wrong on an incorrect answer.

```
/* invariant for miss is that query is valid but fails in LDL */
miss(not(Q), R, Context):- wrong(Q, R).
miss(A, R, Context):-
                if clause(A, B) fails
                  then R = noClauseMatch(A) else fail.
miss(A, R, Context):- if (clause(A, B), miss1(B, R1, A)) succeeds
                then R = R1
                else R = atom(A).
miss1((A, B), R, Context):- allAnswersExecuteLDL(A, S),
            if examineSet(A, S, R1, Context) succeeds
            then R = R1
            else (member(A, S), miss1(B, R, Context)).
            /* (member(A, S) is used to obtain bindings as in executeLDL(A) */
miss1(A, R, Context):- allAnswersExecuteLDL(A, S),
            if examineSet(A, S, R1, Context) succeeds
            then R = R1 else fail.
examineSet(A, S, R, Context):-
        if there is Q in S such that Q is not valid in I
        and (Q = not(Q1) or Q's layer is lower than that of Context)
        then wrong(Q, R)
        else
          (if notExhaustive(A, S, Missingone, Context)
          then miss(Missingone, R, A) else fail).
notExhaustive(A, S, Missingone, Context):-
          Missingone is a valid instance of A whose
          generation is prior to Context which is not in S.
allAnswersExecuteLDL(A, S):-
          S is the set of all answers to executeLDL(A).
/* invariant for wrong is that query is invalid but succeeds */
/* wrong, navigateDown: as before. */
```

Fig. 5 Comprehensive Debugger.

These ideas led to the *comprehensive debugger* shown in Fig. 5.

Theorem 5.1
For the comprehensive debugger, given $f \in I - M(P)$ as input to miss or given $f \in M(P) - I$ as input to wrong, either a mbf or an incorrect clause is returned.

Proof
See Appendix 1. □

§6 Conclusion

We have addressed the problem of debugging declarative programs. A major contribution of this paper is in modeling the user's "mental picture" viz. acceptable structure and generated interpretation. Some formalization is needed in order to guarantee bug discovery by an algorithm which interacts with the user. It is an open question whether all the conditions specified for an acceptable structure are needed. The same question applies to the definition of a generated interpretation. Another issue is whether these concepts indeed capture the essence of "mental pictures".

We have presented a basic debugger and proved that under certain assumptions it is total. Extensions for handling updates and improving user-debugger interactions, as well as a more sophisticated debugger, were also presented.

"It is argued that declarative error diagnosers will be indispensable components of advanced logic programming systems."[6] Our formalization differs from that of Ref. 6) in that a different language is addressed with different semantics (especially w.r.t negation) and hence, we have a different notion of correctness. Other than that, we fully agree with the above statement.

Acknowledgements

We'd like to thank the referees for their useful comments and N. Francez for his many suggestions.

References

1) Beeri, C., Naqvi, S., Shmueli, O. and Tsur, S., "Set Constructors in a Logic Database Language," *Journal of Logic Programming, 10, 3-4*, pp. 181-232, April/May 1991.
2) Chimenti, D. and Gamboa, R., "The SALAD Cookbook; A User/Programmers' Guide," *MCC Technical Report, ACT-ST-346-89.*
3) Dershowitz, N. and Lee, Y., "Deductive Debugging," in *Proceedings Fourth IEEE Symposium on Logic Programming*, San Francisco, California, pp. 298-306, Aug. 1987.
4) Drabent, W., Nadjim-Tehrani, S. and Maluszynski, J., "Algorithmic Debugging with Assertions," in *Proceedings Workshop on Meta-Programming in Logic Programming*, University of Bristol, June 1988.

5) Ferrand, G., "Error Diagnosis in Logic Programming: An Adaptation of E. Y. Shapiro's Method," *Reporte de Recherche*, *375*, INRIA, France, 1985.
6) Lloyd, J. W., "Declarative Error Diagnosis," *New Generation Computing*, *5*, pp. 133-154, 1987.
7) Lloyd, J. W., *Foundations of Logic Programming* (2nd Edition), Springer-Verlag, 1987.
8) Naqvi, S. and Tsur, S., *A Logical Language for Data and Knowledge Bases*, W. H. Freeman, 1989.
9) Naish, L., "Declarative Error Diagnosis of Missing Facts," *Technical Report*, *88/9*, Department of Computer Science, University of Melbourne.
10) Pereira, L. M., "Rational Debugging in Logic Programming," in *Proceedings Third International Conference on Logic Programming*, London, England, Springer-Verlag, LNCS 225, pp. 203-210, July 1986.
11) Shapiro, E. Y., *Algorithmic Program Debugging*, MIT Press, Cambridge, Massachusettes, 1983.
12) Shmueli, O. and Tsur, S., "Logical Diagnosis of LDL Programs," in *Proceedings Seventh International Conference on Logic Programming*, Jerusalem, Israel, pp. 112-129, June 1990.
13) Sterling, L. and Shapiro, E. Y., *The Art of Prolog*, MIT Press, Cambridge, Massachusettes, 1986.

Appendix 1

We first define generation numbers. Consider a program P and a general fact f. Let L be the layer in P in which the predicate of f is defined. Let m be the stage number of f within L if $f \in I$ and otherwise $m = 0$. The *generation number* of f is the tuple (L, m); define $(L, m) < (L1, m1)$ if either $L < L1$ or $L = L1$ and $m < m1$.

Proof of Theorem 3.1

The proof is divided into two parts: (a) showing that the debugger terminates, and (b) showing that it returns an answer and that this answer is sound.

We show that each invocation of the debugger terminates. First, note that each call to *ExecuteLDL* terminates, by assumption. There are two basic loops which may invoke each other recursively. One is the loop involving *miss* and *miss*1 and the other is the *NavigateDown* loop. Calling from one loop to another loop decreases the layer number and hence can only be done a finite number of times. Thus, it suffices to show that each loop by itself terminates, i.e. with calls to the other loop considered as a single statement.

First, we argue that the *NavigateDown* loop terminates. As proof-trees are finite, each call to *navigateDown* either terminates or reaches a leaf in finitely many steps. Thus, each invocation of *navigateDown* terminates.

The characteristics of the loop involving *miss* and *miss*1 are as follows. One possible behavior is that once procedure *miss* calls *miss*1, *miss*1 recurses on itself a number of times without calling *miss*, including backtracking. We now show that this possible behavior terminates. *validground* supplies finitely many answers whose generation numbers are less than the context's generation number. Each *executeLDL* call, which is issued by the debugger, terminates (by assumption A2). Thus, each invocation of *miss*1, under this behavior, terminates.

Procedure *miss*1 can also call *miss*, recursively. Suppose *miss*1 is called from *miss* operating on f. The answers to *validground* in *miss*1 are of generation numbers smaller than that of f, since a reliable oracle is assumed. This is true also for the recursive invocations of *miss*1, if any. Thus, the generation number in the recursive call from *miss*1 to *miss* is smaller than that of f. As there are no infinite decreasing chains in a generated interpretation, this global behavior of *miss* calling *miss*1 which calls *miss* etc., can be repeated only finitely many times. Thus the loop involving *miss* and *miss*1 terminates.

We conclude that the debugger terminates. We now prove that the debugger always returns an answer and that the answer is sound. The proof is by induction on generation numbers. The induction hypothesis is the following, where f is a *general fact*. Given f valid in I but not valid in $M(P)$ as input to *miss* or given f valid in $M(P)$ but not valid in I as input to *wrong*, either a missing basic fact or an incorrect clause is returned. (The hypothesis is stronger than what is strictly needed to prove the theorem.)

Basis: $(L, m) = (1, 0)$
If *miss*(f, R, f) is called with f positive, then since f is a fact for an *EDB* predicate, no clause head can unify with f and *noClauseMatch*(f) is correctly returned identifying f as a mbf. If *miss*(f, R, f) is called with $f = \neg q$, then $q \in M(P)$ although $q \notin I$; *wrong*(q, R) is invoked with q a positive *EDB* fact. Procedure *wrong* invokes *navigateDown* with T consisting of a single leaf with a positive literal, and *wrongUnitClause*(q) correctly identifies an incorrect unit clause. The basis case of *wrong* called with a positive fact is thus treated as well. Now suppose *wrong* is called with $\neg q$; i.e. $q \notin M(P)$ and $q \in I$; *wrong* calls *navigateDown* which invokes *miss*(q, R, q). As argued above this call correctly identifies a mbf.

Induction: $(L, m) > (1, 0)$
Case 1: f is valid in $M(P)$ but f is not valid in I. Let T be a partial proof-tree for f, recall that such a tree has leaves which are either positive facts, which are instances of unit clauses, or of the form $\neg q$. Since the tree is of finite height, *navigateDown* eventually stops, following some number of recursive calls, at an internal node or a leaf containing q. If it stops at an internal node, an incorrect clause has been identified. So, suppose it stops at a leaf.

If q is positive an incorrect unit clause is identified. If $q = \neg p$ then $p \notin M(P)$ and $p \in I$; *miss*(p, R, p) is called with a smaller (L, m) as p is defined in a layer below that of q, by induction hypothesis a mbf or an incorrect clause is returned.
Case 2: f is valid in I but f is not valid in $M(P)$. Suppose $f = \neg q$. Then, *wrong*(q, R) is called with $q \in M(P) - I$. This was handled in Case 1 above. So, suppose f is positive. If *noClauseMatch*(f) is returned, clearly f is a mbf. We now treat the case where the third clause of *miss* was invoked at the top level call. Either an answer is returned by the then-branch or one is returned via the else-branch, at any case an answer must be returned because the invocation of the third clause of *miss* terminates. It remains to show that if an answer is returned by the third clause of *miss* then indeed a mbf or an incorrect clause instance is returned.

So, the answer came either from (i) the then-branch or (ii) the else-branch.

(i) An answer from the then-branch was supplied recursively by *miss*1. However, *miss*1 manufactures only answers that are obtained from *miss*. Furthermore, a call to *miss* must have used a fact, say q, generated prior to f such that g is valid in I but not in $M(P)$; this is ensured explicitly by the code. By induction, this answer is either a mbf or an incorrect clause.

(ii) An answer was supplied from the else-branch; the answer is *atom*(f). For the sake of deriving a contradiction assume an incorrect answer was returned, i.e. f is not a mbf. So, there is a clause instance A': $-B_1'$, ..., B_n' for some clause C: A: $-B_1$, ..., B_n in P such that $A' = f$ and for $i = 1, ..., n$, B_i' is valid in I and in which B_i', $1 \leq i \leq n$, are of generations prior to that of f. Clearly, C cannot be a unit clause, this would imply $f \in M(P)$.

Consider now the if condition in the third clause of *miss*. Clearly, *miss*1 will be invoked as C's head unifies with f. If $n = 1$ the second clause of *miss*1 is invoked with a literal g. Let g' be a result to *validground*(g). If *executeLDL*(g') succeeeds then $f \in M(P)$, a contradiciton. So, *executeLDL*(g') fails. This results in *miss* being called on g' which is generated prior to f and such that g' is valid in I but is not valid in $M(P)$. By induction, an answer is returned; this would imply an answer which is supplied from the then-branch of the third clause of *miss*, a contradiction.

So, suppose $n > 1$, thus the first clause of *miss*1 is invoked. Suppose it is called with ((A, B), R, *Context*) and let A' be returned by *validground*(A). The else-branch in the first clause of *miss*1 is not taken, if it is taken then, by induction, an answer would be returned as A' is valid in I but not in $M(P)$ and A' is of a generation prior to that of f. This would imply an answer which is supplied from the then-branch of the third clause of *miss*, a contradiction. Thus, the *miss*1 internal loop (first *miss*1 clause) basically performs, with backtracking, the following query:

$$validground(B_1),\ executeLDL(B_1),\ ...,\ validground(B_{n-1}),\ executeLDL(B_{n-1}).$$

However, *validground*(B_1) will eventually supply the answer B_1'. *ExecuteLDL* will succeed on B_1', then *validground*(B_2') will eventually be supplied and again *executeLDL* will succeed on B_2'. This will be repeated for B_3 through B_{n-1} since if at any point j *executeLDL*(B_j') fails then the else-branch will be taken in the first clause of *miss*1. Finally, for B_n, the second clause of *miss*1 is invoked with B_n. This case was handled above (the case $n = 1$ and $q = B_n$) and resulted in a contradiction. □

Proof of Theorem 3.2

The proof is similar to that of Theorem 3.1 with the following differences.

Termination of the loop involving *miss* and *miss*1 hinges on the observation that, by assumption A1, *executeLDL*(A) can return only finitely many answers whether it is preceded with *validground*(A) or not.

Induction, Case 2, part (ii)

The first two paragraphs here are the same as in the proof of Theorem 3.1. The rest of the proof is as follows.

So, suppose $n > 1$, thus the first clause of *miss*1 is invoked. Suppose it is called with ((A, B), R, *Context*). The then-branch in the first clause of *miss*1 is not taken, if it is taken then, this would imply an answer which is supplied from the then-branch of the third clause of *miss*, a contradiction. Thus, the *miss*1 internal loop (first *miss*1 clause) basically performs, with backtracking, the following query:

For $1 \leq i < n$:

$c_i(B_i, R, Context)$: $-executeLDL(B_i)$, $c_{i+1}(B_{i+1}, R, Context)$
$c_i(B_i, R, Context)$: $-\neg(executeLDL(B_i), c_{i+1}(B_{i+1}, R, Context))$,
$validground(B_i, Context)$, $\neg executeLDL(B_i)$,
$miss(B_i, R, B_i)$

For $i = n$ the second clause of *miss*1 does:

$c_n(B_n, R, Context)$: $-validground(B_n, Context)$, $executeLDL(B_i)$, *fail*
$c_n(B_n, R, Context)$: $-validground(B_n, Context)$, $\neg executeLDL(B_i)$,
$miss(B_n, R, B_n)$

Since no answer was returned, the second clause of c_1 was tried. During the execution $validground(B_1', Context)$ was returned in the second clause of c_1. Since *miss* was not called, $executeLDL(B_1')$ succeeded. Thus the value B_1' was also returned by $executeLDL(B_1, Context)$ in the first clause of c_1. By the same reasoning, $validground(B_i', Context)$ was returned in executing the first clause of c_i, $1 < i < n$. Now, it cannot be that this holds for $1 \leq i < n$, because then either the second clause for c_n does not call *miss*, which means that $executeLDL(B_n')$ succeeds and hence f is in $M(P)$, or the second clause for c_n would execute *miss* and an answer will be returned, by induction, to the then-part of the third clause of *miss*. In both alternatives we have a contradiction. □

Proof of Theorem 5.1

The proof is similar to that of Theorems 3.1 and 3.2. Termination follows from previous arguments and the following observations:

- The call to *wrong* ensures either decreasing context or decreasing layer number.
- The call to *notExhaustive* returns a lower context for Missingone.
- The line $(member(A, S), miss1(B, R, A))$ is doing essentially what $(executeLDL(A), miss1(B, R1, A))$ did before. One difference is that $member(A, S)$ may return a fact whose generation is *not* prior to Context. However, we assumed that S is finite and hence termination is not affected.

The correctness argument is as in Theorem 3.2. The possibility that $member(A, S)$ may return a fact whose generation is *not* prior to Context presents no major problem as the proof of Theorems 3.1 and 3.2 depended on (i) that finitely many solutions to A be considered, and (ii) that all valid facts are scanned through. Fact (ii) still holds as executing $(member(A, S), miss1(B, R, Context))$, implies that S contains all valid ground atoms whose generation is prior to Context (with the edge connections constraints). □