# Explaining Prolog Based Expert Systems Using a Layered Meta-Interpreter

**Leon Sterling** and **L. Ümit Yalçinalp***
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OHIO 44106

## Abstract

**Abstract**

This paper presents an improved method of explaining Prolog-based expert systems. The key idea is to make explicit failures during the computation. This allows the integrated explanation in a single interpreter of both successful and failed computations. It also allows rules containing cuts to be effectively explained. Furthermore, the explanation system is interactive, and allows full explanation of both successful and failed *partial* computations. We discuss in some detail a two-layer meta-interpreter which is at the heart of the system.

## 1 Constructs for Explanation

In recent years, several systems have been developed for explaining the execution of Prolog. Applications of these systems have ranged from teaching Prolog [Brayshaw and Eisenstadt, 1988], debugging Prolog [Pereira, 1986, Shapiro, 1983], to presenting the results of Prolog-based expert systems, [Clark and McCabe, 1982, Hammond, 1984, Niblett, 1984, Sterling and Lalee, 1986, Walker *et al.*, 1987, Yalçinalp and Sterling, 1988]. In each case, extra mechanisms were added to make explicit the relevant features of Prolog's backward chaining behavior.

For example, the following constructs are needed to give MYCIN-like explanations of expert systems written as collections of simple Prolog clauses. *Proof trees* represent successful branches of the search tree which is implicitly generated and traversed during a Prolog computation. After completion of the computation, a proof tree is presented in a suitable format to answer questions about *how* a solution is computed. *History lists* represent the current branch of the search tree being traversed and are implemented as a stack of successive goals investigated up to the current goal. Whenever the user is consulted for information, for example to find a missing fact, which might be used to prove or refute the particular goal in question, she is allowed to inquire *why* she is being consulted. More detailed information about these constructs can be found in [Sterling and Shapiro, 1986].

*Failure trees* represent failure branches of the search tree that have been traversed during the computation. A variety of techniques have been used to selectively collect such failure branches, all of which are based on separate interpreters for successes and failures [Bruffaerts and Henin, 1988, Hammond, 1984, Sterling and Lalee, 1986, Walker *et al.*, 1987]. Failure trees are used to answer *whynot* questions.

These constructs can be added directly to each clause in the Prolog program [Clark and McCabe, 1982]. A better approach is to write an enhanced meta-interpreter with the appropriate functionality. Most of the work cited above uses the standard four clause meta-interpreter at the clause reduction level [Sterling and Shapiro, 1986].

Two classes of explanations based on these constructs can be differentiated. *How* and *whynot* explanations are provided *after* the computation. *Why* explanations, on the other hand, are given *during* the computation.

This paper addresses two major limitations with current explanation systems based on the variants of the meta-interpreter at the clause reduction level. The first is their inability to explain adequately extra-logical Prolog predicates, such as negation and cut. We don't accept the solution adopted in [Bruffaerts and Henin, 1988, Sterling and Lalee, 1986] for explaining negation using a totally separate program. Limitations arise from the way the control flow is being modelled by the meta-interpreter. The abstraction of the computation in the systems above does not allow the failure mechanism of Prolog to be properly represented and controlled during this computation.

The second limitation arises from the difference between the view of the computation presented to the user during the computation and that after the computation. Specifically, when answering a *why* question, the user only sees the sequence of rules/clauses currently being investigated, but not completed or failed portions of the computation in different parts of the search tree. In other words, the user is not presented with a global picture. In contrast, *how* and *whynot* explanations present a complete computation.

In this paper, we describe a two-layer meta-interpreter

*This research was supported under NSF Grant No. 1R187-03911, and Equipment Grant No. DMC 8703210.

with a different abstraction of the control flow than the standard meta-interpreter. The new meta-interpreter allows proper explanations of extra-logical predicates such as the cut. Further, by introducing the concept of partial proof trees, it treats explanations given both during and after the computation uniformly, and faithfully to Prolog's computation model.

## 2 A Two-Layer Meta-Interpreter

A good explanation depends on a good representation of the computation at an appropriate level of abstraction. The major limitation of current explanations of Prolog-based systems stems from handling failure too implicitly. Failure in Prolog is composed of two different events. First, a failure occurs when a particular goal fails to unify with any of the existing clauses in the program. This failure causes *backtracking* to generate alternative solutions. Second, if all backtracking efforts result in failure for all clause definitions of a goal, Prolog answers "no".

The inability to unify, and the exhaustion of alternatives for a goal are implicitly used in most reported work, but not explicitly represented. The standard meta-interpreter can be augmented with a result variable to make it fail-safe [Levi and Sardu, 1987, Sterling and Lakhotia, 1988]. This makes failure information available but changes the behaviour of Prolog.

Prolog's behaviour can be kept while maintaining explicit failures by building a meta-interpreter in two layers. We call the first layer of the interpreter the *goal layer*, while the second layer is called the *branch layer*. The goal layer uses the predicate *solve_branch* to generate each possible computation of the goal being solved. The result of the computation, either *yes* or *no*, is returned by the second argument of *solve_branch*. Note that the *solve_branch* layer is fail-safe. This result is then filtered by the goal layer to recover Prolog's behaviour. Its essence is captured in the predicate, *solve_goal*, defined as follows.

```
solve_goal(Goal,Result) ←
   solve_branch(Goal,Result),
   filter_failure(Result).
solve_goal(Goal,no).

filter_failure(yes).
filter_failure(no) ← fail.
```

The branch layer is similar to the standard meta-interpreter at the clause reduction level. The complete code for the two-layer interpreter is given in Figure 1. The scope of the interpreter is given by the clauses for *solve_branch*. In particular, it covers conjunctive goals, system goals, negated goals, single goals (namely rules), conditionals, set predicates, such as *findall* and askable goals.

Note that the two events involved in Prolog's failure mechanism have been made explicit. Failure to unify is indicated by a *solve_branch* clause, which returns a "no". This occurs when none of the clauses in the database are unifiable with the current goal. This is recognized by using the negation of the built-in *clause(Goal,Body)* in Prolog. Backtracking is initiated by the failure of *filter_failure*. This explicit representation of failure will be used in extending the interpreter later.

This two-layer interpreter follows the computation model of Prolog faithfully. Failure can be controlled and used to represent the result of a computation. Further, all computation is handled by a *single* interpreter. There is no reason to wait for the result of computation for a goal to detect failures, as in other systems [Bruffaerts and Henin, 1988, Sterling and Lalee, 1986]. The computation, as in Prolog, is performed only once, which allows the interpreter to correctly handle programs with side-effects.

The presence of two layers makes it straightforward to handle negation-as-failure correctly, as follows:

```
solve_branch(not Goal,Result) ←
   solve_goal(Goal,Result_Not), !, invert(Result_Not,Result).

invert(yes,no).
invert(no,yes).
```

We next demonstrate how the two layer approach facilitates the handling of cuts. A third result is used, namely *commit*, which represents a special type of failure due to commitment in the body of a clause. In the branch layer, Prolog's clause ordering is exploited to achieve the desired result.

```
solve_branch(!,yes).
solve_branch(!,commit) ← !.
```

Obtaining a *commit* result at a branch can affect the program in two ways, depending on the type of the goal.

- For a series of conjunctive goals, $\leftarrow P_1, P_2, ..., P_i, ..., P_n$, a cut can be any of these goals in the body. We retain the value of the computation, namely *commit*, as a failure occuring due to a commitment for the entire conjunction and inform the parent goal. If the cut is the final goal, $P_n$, then the result of the computation will be *commit* by the *solve_branch* clause above. If this goal is $P_i$ and the result of the computation of $P_{i+1}, ..., P_n$ is "no", then the result of the computation in the branch layer is *commit*. This is then handled in the goal layer of the interpreter, as seen in Figure 1.
- For a single goal, a commitment at the branches means an immediate failure as it is the parent goal. That means a *commit* result for a branch of the single goal actually indicates a *no* result in the goal layer, and the goal should no longer investigate other possibilities via different clause definitions in the branch layer.

Handling cuts correctly is achieved by modifying *filter_failure* and *solve_goal*. Now, *filter_failure* acts as another "port" for indicating overall failure upon commitment by returning a "no" result for a single goal. *Solve_goal* is altered to make the interpreter actually *commit* itself by placing a cut in the goal layer. This prevents backtracking of the interpreter back to the branches which generate alternative solutions. The modified goal layer is represented in Figure 1.

```
% GOAL LAYER

solve_goal(Goal,Result) ←
    copy(Goal,Copy),
    solve_branch(Copy,Branch_Result),
    (Branch_Result = commit → !;true),
    filter_failure(Branch_Result,Result,Goal,Copy).
solve_goal(Goal,no).

filter_failure(yes,yes,Goal,Goal).
filter_failure(commit,commit,Goal,Copy) ←
    non_singular(Goal), !.
filter_failure(commit,no,Goal,Copy). % for single goals
filter_failure(no,_,_,_) ← fail.

% BRANCH LAYER

solve_branch(true,yes)←!.
solve_branch((A,B),Result) ← !, % A and B
    solve_goal(A,RA),
    solve_branch_and(RA,A,Result,B).
solve_branch((A → B; C),Result) ←
    solve_goal(A,ResultA),!,
    solve_branch_if(ResultA, Result, B, C).
solve_branch((A → B),Result) ←
    solve_goal(A,ResultA), !,
    solve_branch_if_one(ResultA,Result, B).
solve_branch((A;B),Result) ← !, % A or B
    solve_goal(RA,A),
    solve_branch_or(RA,Result,B).
solve_branch(not Goal,Result) ←
    solve_goal(Goal,Result_Not), !,
    invert(Result_Not,Result).
solve_branch(findall(X,P,L),Result) ←
    findall(X-R,solve_goal(R,P),AList),
    eliminate(Result,AList,L).

solve_branch(!,yes).
solve_branch(!,commit) ← !.
solve_branch(A,Result) ←
    sys(A), A, Result = yes.
solve_branch(A,Result) ←
    sys(A), Result = no, !.
solve_branch(A,no) ←
    not clause(A,Body),!.
solve_branch(A,Result) ←
    clause(A,Body),
    solve_goal(Body,Result).
solve_branch(A,Result) ←
    askable(A),
    investigate_goal(A,Result).

invert(yes,no).
invert(no,yes).

% HANDLING CONJUNCTION

solve_branch_and(yes,!,Result,B) ← !,
    solve_goal(B,ResultB),
    ((ResultB = no ; ResultB = commit) →
                    Result = commit ; Result = yes).
solve_branch_and(yes,A,Result,B) ← !,
    % when A is not a cut
    solve_goal(B,Result).
solve_branch_and(no,A,no,B).

% HANDLING DISJUNCTION

solve_branch_or(yes,yes,B) ← !.
solve_branch_or(no,Result,B) ← !
      solve_goal(B,Result).
solve_branch_or(commit,commit,B) ← !.

% HANDLING IF BRANCHES

solve_branch_if_one(yes,Result, B) ← !,
    solve_goal(B,Result).
solve_branch_if_one(NoResult, NoResult, B) ←
    (NoResult = commit ; NoResult = no), !.

solve_branch_if(yes, Result, B, C) ← !,
    solve_goal(B,Result).
solve_branch_if(no, Result, B, C) ← !,
    solve_goal(C,Result).
solve_branch_if(commit, commit, B, C) ← !.
```

Figure 1: The *Layered* Interpreter

To handle the effect of cuts on instantiating variables correctly, a copy of the goal is needed. Since the commitment would instantiate the actual goal, the bindings will be propagated within the tree. In order to be faithful to the execution process, the actual goal should be uninstantiated upon failure. To have a uniform structure, the copy of the actual goal is used for creating the branches and explicit unification with this goal happens upon success.

Other control predicates can be handled in our framework. For example, the extension for the *if-then-else* construct, $P \rightarrow Q; R$, is immediate. The interpeter uses the *Result* of the goal defining the conditional part, *if*, to define and control the result of the branch defining the the construct. It is shown in the full interpreter in Figure 1.

## 3 Explanations Generated by the Layered Meta-Interpreter

An extra argument, *Proof*, is used for collecting proofs in the branch layer of the interpreter. This is a standard technique discussed in [Sterling and Shapiro, 1986] and used in [Bruffaerts and Henin, 1988, Sterling and Lalee, 1986, Yalçinalp and Sterling, 1988], etc. The proofs, both of successes and failures, are transmitted to the goal layer at this layer's exit points.

The exit points of the goal layer are *filter_failure*, upon success or failure due to commitment, and the second clause of *solve_goal* upon ultimate failure of the goal. Since this interpreter does not know in advance whether a goal will ultimately fail or not, the proof is kept by the goal layer when a "no" result occurs in the branch layer. If there is no commitment, the alternative solutions are generated by imposing failure. Upon generating a success, a "yes" result by the branch layer, all the previously encountered failures are discarded since they are not relevant. However a failure, which occurs by commitment or ultimate failure of all branches, enables the interpreter to collect the proofs of all the failure branches that are previously stored. The method is shown below:

```
solve_goal(Goal,Result,Proof) ←
    copy(Goal,Copy),
    solve_branch(Copy,BResult,BProof),
    (BResult = commit → !;true),
    filter_failure(BResult,Result,Copy,Goal,BProof,Proof).
solve_goal(Goal,no,fail(Goal,Failures)) ←
    get_proof(Goal,Failures).

filter_failure(yes,yes,Copy,Goal,Proof,Proof) ←
    get_proof(Goal,Failures),
    Goal = Copy.
filter_failure(no,_,Copy,_,BProof,_) ←
    store(Copy,BProof), !, fail.
filter_failure(commit,commit,Copy,Goal,Proof,Proof) ←
    non_singular(Goal),!.
filter_failure(commit,no,Copy,Goal,Proof,
                          fail(Goal,Failures))←
    % for single goals
    get_proof(Goal, Prev_Failures),
    append(Prev_Failures,commit(Copy,Proof),Failures).
```

There are two reasons for using a copy of the goal. First, we are interested in keeping the actual uninstantiated goal and instantiate it as necessary. This is to handle the effect of cut on committing to bindings as explained above. Second, for failures, the structure *fail(Goal, Failures)* is used to collect the proofs of failures for the actual goal *Goal*, where *Failures* is a list of failures of this goal. Later, during explanation, the actual goal in this structure is used to resolve the variable differences in the tree by the scope information in the proofs of failure.

The *non_singular* predicate checks whether the given goal is a construct, such as a disjunction, conjunction, if-then-else or the cut, in fact anything other than a single goal defined as a set of clauses in the program. Since commitment is propagated in Prolog in these constructs, the interpreter must also adopt this behaviour. Upon checking the goal, *filter_failure* either propagates commitment or decides that a single goal ultimately fails and collects all the failures.

As mentioned earlier, generation of explanations actually depends on a good representation of the computation. With an appropriate representation such as a proof tree, stilted English statements can be formed using straightforward techniques, described for example in [Sterling and Shapiro, 1986]. Similar techniques are used to make the proofs of computation collected by the two-layer interpreter more understandable.

Instead of providing a full "trace" of the entire program, any of the subgoals within the body of the currently investigated clause can be chosen for further explanation. This method allows the user to navigate the tree as desired and the explanation for each clause is given upon demand. In addition, it is possible in our explanation system to reinvestigate the computation process by starting from the top goal at any point, or go back to the previous rule which has been investigated.

Let us illustrate the form of explanation generated with an example. Consider the simple program below that defines hypothetical rules for Ph.D. candidacy. A student is required either to take the qualifying test and pass, or to take advanced courses and achieve a sufficiently high grade point average.

```
candidacy(X) ←
    phd_student(X), qualified_in_math(X).

qualified_in_math(X) ←
    has_taken_test(X,math),!,
    passed_test(X,math).
qualified_in_math(X) ←
    has_advanced_courses(X,Y), !,
    satisfies_gpa(X,Y).
has_taken_test(Person,Subject) ←
    exam(Person,Subject,Date).
passed_test(Person,Subject) ←
    result(Person,Subject,Date,pass).

phd_student(jim).
exam(jim,math,jan1988).
result(jim,math,jan1988,fail).
```

The query *?-candidacy(jim)* gives a "no" answer. The explanation provided from the system is as below. User responses are italicized.

```
candidacy(jim) fails because in the clause definition
        1. phd_student(jim) succeeds.
        2. qualified_in_math(jim) fails.
> ? 2.
qualified_in_math(jim) fails because in the clause definition
        1. has_taken_test(jim,math) succeeds.
        2. passed_test(jim,math) fails.
and the goal qualified_in_math(jim) is committed
to this clause because there is a cut after has_taken_test.
> ? applicable_rules.
2 rules in the database match the current goal.
The computation is committed to the 1st (current clause).
Therefore, the clause below is not reachable:
qualified_in_math(jim) ←
        has_advanced_courses(jim,Y), !,
        satisfies_gpa(jim,Y).
> ? 2.
passed_test(jim,math) fails because in the clause definition
        1. result(jim,math,Date,pass) fails.
> ? 1.
This goal does not match with the clause(s) of result/4.
There is 1 fact in the database for result/4:
result(jim,math,jan1988,fail).
> ? quit.
```

As can be seen from the example, the user can further investigate the reasons of success or failure for each goal. It is possible to request further explanation regarding which clauses for the goal *qualified_in_math(jim)* are not considered as alternative choices due to the presence of the cut. Further meta-knowledge about facts, such as *result/4*, will enhance the explanation capabilities.

The interpreter has been extended to generate detailed explanations for arbitrary Prolog programs. Other features can be easily integrated to this shell, as in a previous version [Yalçinalp and Sterling, 1988], to provide history and depth information.

## 4 Explaining Partial Computations

The answer a user provides to a request for information may depend on what has transpired in the computation so far. Alternatively, as in an expert system for motor selection being developed at Case Western Reserve University [Discenzo *et al.*, 1988], the user may demand to see the history of the computation when answering a why question. In either case, a "global picture" of the computation must be provided.

Recall that a full description of the computation after it terminates is contained in a proof tree. A *fully* instantiated proof tree is not available for explanations during the computation. However, a *partially instantiated* tree which has full branches for completed portions of the computation and uninstantiated nodes for incomplete portions can be made available. We call this structure a *partial proof tree*. It is similar to the partially specified tree described in [Pereira and Shieber, 1987] for representing DCG parse trees during the parsing process.

A partial proof tree is easily generated by adding two extra arguments to the interpreter in Figure 1. The first argument represents the partial proof tree. It is different from the *Proof* argument described in the previous section and used in [Sterling and Lalee, 1986, Yalçinalp and Sterling, 1988] because it can be accessed at any time during the computation. The branches are generated as a structure by using the second argument which shows the current node of the partial proof tree. The tree is instantiated at its leaves, when computation finitely succeeds or fails. In the beginning of the computation, these two additional arguments represent the same entity, the uninstantiated tree, by the query :

*?- solve_goal(Goal, Tree, Tree, Result).*

The nodes are generated in the branch layer during the computation by the second extra argument and the full partial proof tree is passed to all layers of the computation. For example, the partial proof tree is generated as follows for a conjunctive goal in the branch layer:

```
solve_branch((A,B),Tree,(PA,PB),Result) ← !,
   solve_goal(A,Tree,PA,RA),
   solve_branch_and(RA,A,Result,B,Tree,PB).
```

When additional information is required by the interpreter, the user interacts with the system during the computation, by the last clause of *solve_branch* that represents askable goals. A *how* explanation of the partial proof tree can be presented during the computation. This is handled by passing the argument that represents the partial proof tree to *investigate_goal* which provides the *how* explanation. However, the explanation incorporates the partial instantiations for clauses and informs the user which goals which have not yet been solved.

## 5 Conclusions

The expressive power of the two-layer interpreter has been illustrated in the previous sections. The execution of the interpreter is faithful to Prolog's execution, allowing the collection of information about the reasons for failure, including branches having been pruned due to cuts. The execution does not require prior knowledge of success and failure as in [Bruffaerts and Henin, 1988, Sterling and Lalee, 1986] and all the computation is handled by a single interpreter. We have also shown the utility of generating partial proof trees. This allows explanations to the user about the full computation both during and at the termination of computation.

The two-layer interpreter can be modified, we believe, to handle debugging systems in the style of Shapiro's algorithmic debugging [Shapiro, 1983] as well as explanation systems. At the moment, the clauses that are not defined in the database are reported to the user as well as the clauses defined but do not match with the current goal. In addition, inaccesible clauses due to commitment are also noted. Hence, deguggers can benefit from this knowledge.

Our work can further be extended to explain Prolog programming cliches which have primarily a procedural reading. For example, a failure-driven loop, defined as follows, is a common construct.

$P \leftarrow P_1, P_2, ..., fail.$
$P.$

The success of the last clause is meaningful only in relation to the repeated failure of the previous clause. By recognizing this construct as a special case, the proof tree kept for explaining the final clause will also contain information about the previous failures. Other cliches in this category are *repeat* loops, and cut-fail combinations.

More generally, we plan to investigate the interaction between programming style and explanation generation. A good way of using Prolog to construct expert systems is to design and implement an embedded language. Explanations should be in terms of the constructs of the embedded language rather than Prolog structures. However Prolog constructs cannot be ignored entirely.

We conclude with a brief discussion of the significance of our work to meta-level programming. Certain aspects of the object level computation are explicitly represented in a meta-interpreter while others are kept implicit. Those explicit aspects are said to be *reified* while the implicit ones are *absorb*ed. Reification and absorption were originally discussed for the Lisp community by Smith [des Rivieres and Smith, 1984], and have been used informally in logic programming by Ken Kahn and Ehud Shapiro. Standard Prolog meta-interpreters reify clause reduction and absorb unification, except for the decomposition of constructs, such as conjunction and disjunction. Failure is also absorbed in the standard four-clause meta-interpreters hence we have no access to when it occurs or how backtracking affects the computation. In contrast, our layered abstraction reifies Prolog's control flow, where the *communication* between the goals and the state of the computation, namely success, failure or commitment, is explicit. Prolog's unification mechanism is absorbed in our model.

Other research in meta-level programming has pondered the issue of reflection. We believe our work can be discussed within this terminology, but to do so is beyond the scope of this paper. For an overview of current meta-level programming and applications, see [Maes and Nardi, 1988, Lloyd, 1988].

## References


[Brayshaw and Eisenstadt, 1988] M. Brayshaw and M. Eisenstadt. *Adding data and Procedure Abstraction to the Transparent Prolog Machine(TPM).* Proceedings of the 5th International Conference and Symposium in Logic Programming, pp. 532-547, 1988.

[Bruffaerts and Henin, 1988] A. Bruffaerts and E. Henin. *Proof Trees for Negation as failure: Yet Another Prolog Meta Interpreter.* Proceedings of the 5th International Conference and Symposium in Logic Programming, pp. 343-358, The MIT Press, 1988.

[Clark and McCabe, 1982] K.L. Clark and F.G. McCabe. *PROLOG: a language for implementing expert systems.* Machine Intelligence 10 (eds. Hayes, Michie and Pao), pp. 455-470, Ellis-Horwood, 1982.

[Discenzo *et al.*, 1988] F. Discenzo , G. W. Ernst, Z. M. Ozsoyoglu, L. Sterling. *Integration of Expert Systems and Database Technologies.* Proceedings of AAAI Workshop in Databases in Large AI Systems, pp. 71-77, St. Paul, MN, August 1988.

[Hammond, 1984] P. Hammond. *micro-PROLOG for expert systems.* in micro-PROLOG: Programming in Logic, pp. 294-319, Prentice Hall International, 1984.

[Levi and Sardu, 1987] G. Levi and G. Sardu. *Partial Evaluation of metaprograms in a "multiple worlds" language,* Proceedings of the Workshop on Partial and Mixed Computation, Denmark, 1987.

[Lloyd, 1988] J. Lloyd *ed. Proceedings of the Workshop on Meta-Programming in Logic programming,* Bristol, June 1988. *(to be published by MIT Press in 1989).*

[Maes and Nardi, 1988] P. Maes and D. Nardi *eds. Meta-Level Architectures and Reflection.* North-Holland, 1988.

[Niblett, 1984] T. Niblett. *YAPES - Yet Another Prolog Expert System.* Tech Report, TIRM-84-008, The Turing Institute, Glasgow, UK, 1984.

[Pereira and Shieber, 1987] F. Pereira and S. M. Shieber. *Prolog and Natural-Language Analysis.* CSLI Lecture Notes, CLSI, 1987.

[Pereira, 1986] L. M. Pereira. *Rational Debugging in Logic Programming.* Proceedings of 3rd International Logic Programming Conference, Lecture Notes in Computer Science 225, pp. 203-210, Springer-Verlag, 1986.

[des Rivieres and Smith, 1984] J. des Rivieres, B. C. Smith. *The Implementation of Procedurally Reflective Languages,* Proceedings of ACM Symposium on LISP and Functional Programming, pp. 331-347, 1984.

[Shapiro, 1983] E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Sterling, 1986] L. Sterling. *Meta-Interpreters: The Flavors of Logic Programming?.* Proceedings of Workshop on Foundations of Logic Programming and Deductive Databases, Washington, 1986.

[Sterling and Lakhotia, 1988] L. Sterling and A. Lakhotia. *Composing Prolog Meta Interpreters.* Proceedings of the 5th International Conference and Symposium in Logic Programming, pp. 386-403, The MIT Press, 1988.

[Sterling and Lalee, 1986] L. Sterling and M. Lalee. *An Explanation Shell for Expert Systems.* Computational Intelligence, pp. 136-141, 1986.

[Sterling and Shapiro, 1986] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Walker *et al.*, 1987] A. Walker, M. McCord, J. Sowa and W. Wilson. *Knowledge Systems and Prolog.* Addison-Wesley, 1987.

[Yalçinalp and Sterling, 1988] L. Ü. Yalçinalp and L. Sterling. *An Integrated Interpreter for Explaining Prolog's Successes and Failures.* Proceedings of the Workshop in Meta Programming in Logic Programming, pp. 147-159, June, 1988. *(to be published by MIT Press in 1989)*