Building Embedded Languages and Expert System Shells in Prolog

L. Ümit Yalçinalp and Leon Sterling

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OHIO 44106

Abstract

This paper concerns building embedded languages in Prolog, with special attention on expert system shells. First, the paradigm of meta-programming, of which building embedded languages is an example, is discussed. Second, we review interpreters for embedded languages, concentrating on meta-interpreters. Finally, two applications, explanation and uncertainty reasoning, are presented and the techniques that were used in their construction are discussed.

1 The Paradigm of Meta-Programming

Prolog is widely recognized as a good language for writing simple rule-based expert systems. There are two major factors. First, the primary language constructs, Horn clauses, are essentially rules. Second, Prolog's built-in backward chaining interpreter can be used as the inference engine. However, writing rule-based systems directly in Prolog has limitations due to the 'hardwiring' of both a particular type of representation and a particular inference strategy. Further, the language does not provide certain functionalities of an expert system shell, such as tools for interacting with the user, explanation, knowledge acquisition and debugging.

Three different approaches have been suggested to overcome Prolog's limitations for building expert systems [11]. The first method is to build the desired extensions within the underlying Prolog engine at the implementation level of a Prolog compiler or interpreter. The second method is to translate a knowledge-based system written in a special language to Prolog by using a special compiler. The third method is to exploit the meta-programming facilities of Prolog. Clearly the first method is the most efficient yet the most inflexible. Developing the desired extensions within Prolog is the easiest method and the most flexible one, since it is not necessary to change the underlying Prolog system. In this paper, we advocate this third method and illustrate techniques for building embedded languages and expert system shells in Prolog by exploiting its meta-programming facilities.

Let us start by addressing the question "what is meta-programming", a topic which has concerned many researchers. One common definition [2, 19] states

> Meta-programming is writing programs that treat other programs as data.

By this definition, familiar programs, such as assemblers, compilers and program transformers are meta-programs.

Another definition of meta-programming can be obtained by following Hayes-Roth's: **Meta-X is a macro definition for 'X about X'** [13]. Expanding this macro as in [3], yields another definition of meta-programming.

> Meta-programming is programming about programming.

Both of the above definitions are descriptive. However they are missing, in our opinion, the essence of meta-programming by ignoring both the purpose of writing meta-programs and the approach taken for developing them.

We regard meta-programming as a *paradigm*, which is according to Floyd [12], an approach to writing programs. A paradigm can also be described as a collection of methods and/or techniques to facilitate the construction of certain classes of programs. Other examples of paradigms are structured programming, dynamic programming, divide-and-conquer and object-oriented programming.

We offer a single sentence definition of the paradigm of meta-programming.

> Meta-programming is building an abstraction of an object language and developing an interpreter for the abstraction.

This definition does not contradict the previous definitions. However, it focuses on the purpose behind meta-programming activity. The reason for conceiving a programming task as meta-programming is to clarify and simplify the programming by giving a useful language to the user with which to build applications.

In the chapter on meta-linguistic abstractions in their well-known textbook, Abelson and Sussman [1] stress the importance of establishing new descriptive languages for programming. Tools and techniques are needed both to formulate new languages and to implement these languages by constructing evaluators. From an engineering point of view, it is much simpler to build within a high level language by delegating some of the work, i.e. matching, unification, or backward chaining to the underlying system.

CH2915-7/90/0000/0056$01.00 © 1990 IEEE

Meta-programming is an important paradigm for AI applications. The best way to develop a new application, in our experience, is to design a new language tailored to the application. This is only practical if it is easy to design and implement the language.

A particular language, such as Prolog, supports meta-programming by making it easy to write interpreters for a specific application. Interpreters are used for two related reasons for modelling computations. The first one is the execution of an embedded language within a programming language. The next section gives an example of developing an embedded language within Prolog, and gives a simple interpreter.

The second reason for developing interpreters is to model a particular inference, such as backward chaining. The interpreter can then be used to develop expert system shells. In section 3, we discuss how to develop interpreters, classify the purposes of building them and present relevant techniques. In section 4, we discuss two expert system shells, for uncertainty reasoning and explanation, that exemplify the meta-programming paradigm.

2 Embedded Languages

This section gives an example of designing an embedded language for an expert system application. Before giving the details of the example, it is necessary to define some terminology.

The *domain language* is the language used to represent knowledge about the problem domain. An *object language* is a language describing a system or application program. A *meta-language* is a language which represents the constructs of an object language explicitly. In this paper, we identify the domain and object languages and refer to the two synonomously.

Consider writing an expert system to evaluate graduate student applications to a department in a university. Such a system must weigh attributes of a student such as GRE scores, grades, recommendation letters, and senior project topic, and classify the student application into categories such as accept, reject or consider further. We assume that the attributes can be described qualitatively, such as gre scores are excellent, recommendation letters are very good, and grades are poor. Further, the qualitative terms are assumed to lie on an ordinal scale, poor is worse than good is worse than very good, etc. A discussion on qualitative reasoning with attributes from ordinal scales is in [5].

Our system will evaluate students by trying to write down heuristic rules expressed in these qualitative terms, such as

"The student should be accepted if her gre scores are excellent, the grades are very good (or better) and the recommendation letters are good (or better)."

To build such a system we need a syntax for expressing rules. Borrowing from Prolog syntax, and taking advantage of the ability to define operators, one is lead to rules of the form `Conditions ⟹ Action`, where `Conditions` are a conjunction, denoted &, or disjunction, denoted V, of a set of goals of the form `(Attribute, RelOp, Value)`. Five sample rules for evaluating graduate students are given.

```
(gre,=,excellent) & (grades,>=,very_good) &
   (recommendations,>=,very_good)⟹accept.
(faculty_decision,>,good) ⟹accept.
(gre,=,excellent) & (grades,>=,good) &
   ((senior_project,>=,interesting) V
   (recommendation,>=,very_good))⟹consider.
(gre,>=,very_good) & (grades,=,excellent) &
   (senior_project,>=,interesting) &
   (recommendation,=,excellent) ⟹ consider.
(gre,<,excellent) &
   (grades,<,good) ⟹ refuse.
```

An interpreter is needed to evaluate whether a certain `Student` satisfies the requirements written in the rule language. Such an interpreter, is given below by the predicate `evaluate(Student, Decision)`. It tries the rules in turn until one is found which is applicable for the given student, essentially using a generate-and-test strategy. By having the representation built on Prolog, two of its features are exploited to evaluate the rules. Firstly, these rules are retrieved by using Prolog's built-in backward chaining interpreter. Secondly, the instantiation of variables is achieved by using Prolog's unification.

```
evaluate(Student, Decision) ←
   Qualifications ⟹ Decision,
   holds(Qualifications, Student).

holds((C1VC2), Student) ← holds(C1, Student).
holds((C1VC2), Student) ← holds(C2, Student).
holds((C1&C2), Student) ←
   holds(C1, Student), holds(C2, Student).
holds((Concept, RelOp, ExpVal), Student) ←
   lookup(Concept, Student, Value),
   values(Concept, Values),
   compare(RelOp, ExpVal, Value, Values).
```

The `holds` predicate tests that all the `Qualifications` on the left-hand side of a rule hold with respect to the `Student` under consideration. It depends on `compare/4`, which compares the `Student`'s `Value` with the expected value, `ExpVal`, of the `Concept` based on the operator `RelOp` defined in the rule. Code for `compare` can be found in [19].

Let us generalize the experience. An embedded language for an application is built by

- identifying the underlying abstraction necesssary for the task.
- defining its representation in Prolog by creating language constructs, such as a new representation scheme for rules.
- building an appropriate interpreter for the language.

The embedded language is more convenient for building expert systems. The knowledge engineer has the possibility of communicating more directly with the expert and reducing the gap between expert knowledge and expert system knowledge. The user does not need to know the full details of Prolog syntax and execution, but can focus on the embedded language.

It is possible to build applications directly in Prolog. A sample rule from our running example follows.

```
evaluate(Student, accept) ←
   holds((gre, = , excellent), Student),
   holds((grades, >=, very_good), Student),
   holds((recommendation, >= , very_good),
          Student).
```

Comparing the system implemented directly in Prolog with building a new rule language illustrates the advantages of using an embedded language. This method provides a general framework for other applications which might use the same paradigm. For example, the generate-and-test paradigm for ordinal reasoning was used for two expert system applications, as described in [5]. In addition, flexibility and clarity is achieved by separation of the rules and how they are evaluated. Building on the power of Prolog, it is easier to include new rules or change the inference separately by using an embedded language.

3 Writing Interpreters

In the previous section, we developed a simple interpreter for an embedded language. In this section, we will contrast that example with writing a *meta-interpreter**.

Meta-interpreters can represent existing models of computation [24]. They can be used to abstract run-time properties of computations for building applications or tools. An explicit model of the inference written as a meta-interpreter can *reveal* or *alter* aspects of the computation of the domain language.

To state what may seem obvious, the most important principle behind building a meta-interpreter is to understand *why* the interpreter is needed. Thus, the needed abstraction of the computation model must be determined, and how that abstraction should be represented explicitly. Note that determining an abstraction will in general be an iterative process.

After constructing the meta-interpreter, enhancements are added to do a special task or combination of tasks, which we refer to as the *functionality* of the meta-interpreter. The functionality may involve revealing a particular run-time behavior, which may be hidden, e.g. generating a proof tree. The functionality may also be an altered computation model to obtain a specific behavior, i.e. invoking a depth-bound for computation termination or loop detection. To summarize, a general formula for constructing a meta-interpreter is

$$\textit{Construction} \equiv \textit{Abstraction} + \textit{Explicit Representation} + \textit{Adding Functionality}$$

Most meta-interpreters in Prolog follow a *single line of reasoning* via chaining, where the clause selection, or the rule selection, is revealed and unification is left implicit. This is essentially what happens in the well-known *vanilla* interpreter (see for example [19]) which describes Prolog's computation in Prolog. The meta-interpreter exploits Prolog's unification both for selecting appropriate clauses and for computing answers in the goals.

*Some confusion exists between the terms meta-level interpreters and meta-interpreters. There is insufficient space here to discuss a technical basis for distinction. Here, meta-interpreter will refer to an interpreter for Prolog written in Prolog.

```
solve_top(Goal,Result) ←
   solve_bottom(Goal,Result),
   filter(Result).
solve_top(Goal,no).

solve_bottom(true,yes) ← !.
solve_bottom((A,B),Result) ← !,
   solve_top(A,RA),
   solve_bottom_and(RA,B,Result).
solve_bottom(not(A),Result) ← !,
   solve_top(A,R), invert(R,Result).
solve_bottom(A,Result) ←
   sys(A), !,
   (A → Result=yes ; Result=no).
solve_bottom(A,no) ←
   not clause(A,B), !.
solve_bottom(A,Result) ←
   clause(A,B), solve_top(B,Result).

solve_bottom_and(no,B,no).
solve_bottom_and(yes,B,Result) ←
   solve_top(B,Result).

filter(yes).
filter(no) ← fail.

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

Figure 1: Basic Layered Interpreter

In this section, we present a meta-interpreter which makes explicit *multiple lines of reasoning.* Multiple lines of reasoning are necessary to handle failure and backtracking explicitly. Further constraints we place on our meta-interpreter are that the lines of reasoning are considered in the same order as Prolog.

The meta-interpreter is based on a generate-and-test approach to traverse the AND-OR tree implicitly searched by Prolog. Backtracking is used to search alternative paths upon failure. The meta-interpreter has *two layers.* The bottom layer performs the computation explicitly, similarly to the well-known vanilla interpreter, with the addition that branches of the search tree are labelled with an auxiliary parameter `Result`, which takes the values `yes` for success nodes and `no` for failure nodes. The computation is monitored by the top layer which generates the computation results and tests them. This abstraction gives the power to represent both success and failure by using a single interpreter. Note that we could not model the computation by using a `Result` parameter and a bottom layer only, because the goal is to be able to represent multiple lines of reasoning within Prolog. The code for the layered interpreter is given in Figure 1 and is used as a basis for the uncertainty reasoning and explanation shells presented in the next section.

Other techniques are required for meta-interpreters which make different aspects of the computation explicit and available. For example, in order to construct systems for reasoning about belief, or modules, we need to make theories, i.e. sets of rules, explicit. This is the approach taken in MetaProlog[6]. In order to reason about unification, all aspects of unification such as renaming, matching and propagation of matching in the program clauses have to be explicitly represented. The demo meta-interpreter represents both unification and theories explicitly and is further discussed in [7].

The work presented above is complementary to previous work [17] which advocated using meta-interpreters for expert system construction. There, the emphasis was on systematic ways of adding multiple functionalities to a single meta-interpreter and composing them to get a final interpreter which incorporates the composite functionalities. Composing meta-interpreters was discussed in detail in [18]. Here we emphasize writing different meta-interpreters which are to be used as a basis for describing computations, where the functionalities are added later. How to add and compose functionalities is discussed in [17, 18].

4 Expert system Shells

This section describes two applications based on the paradigm of meta-programming in Prolog, an expert system shell for reasoning under uncertainty and another shell for explanation. Both of these systems exploit the multiple lines of reasoning allowed by the layered meta-interpreter presented in Section 3.

4.1 Uncertainty Reasoning in Prolog Based Expert Systems

Many researchers have addressed methods for incorporating uncertainty reasoning within the paradigm of logic programming [23]. Describing the uncertainty reasoning scheme by using an appropriate Prolog meta-interpreter is the usual approach [15, 19, 16]. However, several issues have not been adequately addressed by previous research. In this section, we show how handling multiple lines of reasoning as allowed by the layered meta-interpreter addresses some of the limitations of previous research. We begin by reviewing the two major concerns for handling uncertainty with Prolog meta-interpreters, namely the representation of uncertainty and the uncertainty reasoning calculus.

Uncertainty is usually represented by augmenting the rule representation with an attached certainty factor. A Prolog clause with augmented certainty factor** indicates a conditional certainty of the head of the clause when the body of the clause is true. For example, in $A \leftarrow cf(CF), B_1, ..., B_n$, the certainty factor CF represents the certainty of A when the body of the clause is true. Clauses without certainty factors are assumed to have the maximum certainty factor 1. The exact form of the certainty factor is not specified, and we discuss both single valued and two valued uncertainties in this section.

The uncertainty calculus describes the methods to evaluate the rules during inference. For example, the best known uncertainty reasoning scheme is used in MYCIN [9], and reasons with single value uncertainties. Dempster-Shafer theory is an example of two valued uncertainty reasoning [4].

The major difficulties with previous work are:

1. Combining Lines of Reasoning: The result of a successful Prolog deduction simply involves one branch in the search tree. There might be, however, other branches in the search tree with the same solution, for a particular answer substitution. It seems sensible to include multiple lines of reasoning when we calculate an uncertainty for a predicate. This is not possible by computing uncertainties considering a single line of reasoning. In [15, 19], uncertainties are calculated only by pruning the current branch of the proof tree if necessary. Therefore, different values of uncertainty are given to the same answer substitution corresponding to different branches of the search tree. Given several bodies of evidence as different paths in the search tree, Baldwin [4] addresses computing a composite number in finding a solution for a predicate. There, the separate lines of reasoning are combined by using the computation of the set union of the answers†.

Thus multiple lines of reasoning for a predicate are needed to compute a composite uncertainty. The naive approach to get all the lines of reasoning uses a set predicate such as *setof* as in [16]. However, this approach does not combine the lines of reasoning at different levels of the computation. It is necessary to consider different lines of reasoning for each rule to compute a composite uncertainty and this should be applied to all the rules in the computation.

2. Handling Failure and Negation: Operationally, dealing with negation requires the computation of all lines of reasoning for a goal. The earlier approaches [15, 19], do not deal with negation correctly, since there is no mechanism to incorporate all lines of reasoning. Note that for single valued probabilities, a probability of **not p** can be obtained after a composite certainty of **p** is acquired, with the formula $prob(not\ p) = 1 - prob(p)$.

3. Explicit Representation of True, False and Unknowns: Single valued probabilities do not allow a distinction between false and unknown values. In contrast, a value pair $[\alpha, \beta]$ allows us to represent predicates that are definitely known to be false, by the pair $[0, 0]$, or known to be true by the pair $[1, 1]$. Other methods to represent and calculate with certainties by using two values are discussed in [14, 10, 4]. For example, in [4], the interval $[\alpha, \beta]$, also represents the interval of the certainty of the predicate p, where $1 - \beta$ is defined to be the support for *not p* to be true.

One way to handle unknown predicates is to assign them the value $[0, 1]$, indicating the probability can have any value and there is no supporting or refuting evidence. The COOP shell [16] uses a similar convention to [4]. However, unknowns are represented by adding an auxiliary clause in the knowledge base. This method requires updating the knowledge base, in contrast to the meta-interpreter we present which performs all the necessary calculations. The meta-interpreter appropriately enhanced both to compute

**Note that rules augmented with certainty factors are an example of an embedded language.

†For single valued uncertainties, if there are two deductions for a predicate p with the same answer substitution θ with the certainties p_1 and p_2 respectively, then $prob(p) = p_1 + p_2 - p_1 * p_2$.

```
solve_top(Goal, U, Proof) ←
   solve_bottom(Goal, Uncertainty, Proof),
   filter_uncertainty(Uncertainty, Goal, U,
                     Proof).
solve_top(Goal, U, cl(Goal, U, ProofSet)) ←
   combine_prob(Goal, U, ProofSet).

solve_bottom(true, T, fact)← truthval(T),!.
solve_bottom((A,B), and(CA,CB), (PA,PB)) ←
   !, % A and B
   solve_top(A, CA, PA),
   solve_bottom_and(CA, CB, B, PB).
solve_bottom(not A,invert(CA),not(A, PA))←
   !, % negation
   solve_top(A, CA, PA).
solve_bottom(A, Uncertainty, sys(A)) ←
   sys(A), truthval(T), falseval(F), !,
   (A → Uncertainty = T ;
        Uncertainty = F).
solve_bottom(A, F, notclause(A)) ←
   % does not exist in the knowledge base...
   not clause_new(A, Body, Ci),falseval(F),!.
solve_bottom(A, body(Ci,CBody),
                 clause(A,Body,ProofB)) ←
   clause_new(A, Body, Ci),
   solve_top(Body, CBody, ProofB).

solve_bottom_and(C, C, B, PB) ←
   not continue_conj(C), !.
   % stop calculation !...
solve_bottom_and(C, CB, B, PB) ←
   continue_conj(C), !,
   % if the first conjunct has not failed, continue...
   solve_top(B, CB, PB).

filter_uncertainty(1, true, 1, fact) ← !.
filter_uncertainty(U, Goal, U, sys(_G)) ← !.
filter_uncertainty(0, Goal, _U,
                 Proof) ←
   store_proof(Goal,Proof,0), !, fail.
filter_uncertainty(invert(Cf), Goal, U,
                 Proof) ← !,
   U is 1 - Cf.
filter_uncertainty(and(U1,U2), Goal, U,
                 Proof) ← !,
   Uis U1 * U2.
filter_uncertainty(body(CBody,Ci), Goal, _U,
                 Proof) ←
   U is CBody*Ci,
   store_proof(Goal,Proof,U), !, fail.

truthval(1).
falseval(0).
```

Figure 2: Extended Layered Interpreter for Handling Single Valued Uncertainties

with single valued probabilities and to obtain proof of deduction is given in Figure 2.

Due to the layered structure of the interpreter, the *status* of the computation is available to the top layer. Recall that the traversal of the search tree in the basic layered meta-interpreter is monitored by a `Result` variable at the top layer. In a computation that involves certainties, the meta-interpreter computes an uncertainty rather than a `Result`, where the uncertainty can be either a single value or a pair of values. The certainty for a particular goal is computed from the certainities of all the branches that led to the solution at the top layer. The bottom layer provides the certainty calculated from a single line of reasoning. The modified `filter` predicate at the top layer, `filter_uncertainty`, continues to compute all different paths that lead to a solution upon success as well as failure in contrast to `filter` in Figure 1. The predicate `store_proof` is used to store the branches of the proof temporarily.

When the computation terminates after calculating all lines of reasoning, by the ultimate failure of the first clause of `solve_top`, the certainty is calculated by the `combine_prob` predicate. This predicate first groups solutions with respect to different answer substitutions and then combines the certainties for each answer substitution from different lines of reasoning while eliminating the branches that do not support an answer, i.e. branches with 0 probability. It also returns each different solution, in the order it is obtained, consistent with Prolog's behaviour.

The layered interpreter provides several advantages for uncertainty reasoning.

- **Flexibility:** We assign certainty functions at the bottom level and compute them at the top level. This gives us the flexibility to use single valued or interval valued certainties for representing certainties. By modifying `filter_uncertainty` and `compute_prob` *different strategies for calculating uncertainty* can be obtained by using *the same architecture*. For example, an interval valued uncertainty calculus as outlined in [4, 16] can be used in the same architecture only by changing `filter_uncertainty`.
- **Calculation with Unknowns and Negation:** Since the layered interpreter represents failure and hence negation adequately, the calculation of certainties for negated rules poses no problem in our approach. Actually, this is the first meta-interpreter that we know that can handle rules with negation in reasoning with uncertainties. Furthermore, the fail-safe nature of the layered interpreter allows unknowns to be represented by using the meta-interpreter. Unlike [16], no assigment or alteration of the knowledge base is required. This is desirable because the domain independent inference is separated from the representation of the knowledge base.
- **Dependency of Solutions:** Although this is not implemented, the layered approach can be easily extended to check the dependency of the computations for a specific solution. This issue is discussed in [4]. The layered structure can be extended for resolving conflicts at the top layer by examining the lines of reasoning at the bottom layer. Just like the naive representation of multiple lines of reasoning, a non-layered

architecture is not capable of doing this, because the independence of conflicts with all possible solutions must be checked for each predicate.

4.2 Explanation

An expert system user may require explanation of the system's behaviour, actions, or decisions. Explanation can be *introspective*, where the user is presented with *what* is happening in the system and *why*. An explanation system can also provide a *justification* of the system's behaviour.

Different users have varying degrees of knowledge about the system and in general need different explanations varying in content and detail. Explanations can be classified according to what type of information they provide in response to users' queries. For example, *how* and *whynot* explanations highlight the reasons behind the system's decisions and computations. When the expert system queries a user to aid in the solution, by supplying additional information, the user might want to know the reasons for the query. A *why* type of explanation provides the line of reasoning which lead to the system's queries. Other explanations relate to assertions that are made by the user or the system, such as *when* and *by whom* certain facts are provided, *which* of the known parameters have changed. A discussion of classifying user queries and explanations can be found in [21]. Among types of explanations, *what-if* explanation requires the simulation of another computation by changing certain variables or facts in the system. It does reflect the current or completed behaviour of the system.

An introspective account of a system can only be provided if the system can *represent* and *present* its own computations adequately. Meta-interpreters are a flexible and easy means for providing this type of explanation. The required functionality for explanation is to construct a proof structure representing the computations of the domain language. This is achieved by extending an appropriate interpreter by standard techniques. In [19], the vanilla interpreter is extended to give a simple explanation shell.

We suggest the following features from a general purpose explanation shell based on Prolog:

a. It should be interactive, posing queries to the user when the information is not in the knowledge base and recording the responses. This demands a dynamic knowledge base.

b. It should be able to display reasons for successful, failed and partially completed computations - thereby providing how, why and whynot explanations.

c. It should provide alternative solutions when requested.

d. User supplied information should be easily obtainable for providing which, when, and what explanations.

Building an explanation shell from single layer interpreters, such as vanilla, suffer from the limitation that success and failure do not mesh well together. Previous research has written separate meta-interpreters for successful and failed computations [8]. This method requires the system to first determine whether a query has succeeded or failed, in order to construct the proof by the appropriate interpreter. However, expert systems are interactive. Users can add knowledge to the knowledge base during the solution of a goal. Using different meta-interpreters to provide explanation for successful and failed queries causes changes to the knowledge base to be lost, or the computation be misrepresented, because a recomputation of the query does not guarantee the currently constructed proof to represent the previous computation correctly. Therefore, we require an integrated interpreter which can compute queries in one pass.

The layered interpreter provides an appropriate basis for an explanation shell. We omit the interpreter for explanation, as its capabilities are discussed in [20] in detail. However, we present specific points which makes this tool desirable.

1. An integrated single interpreter provides the explicit representation of most aspects of Prolog computations by representing the computation in two layers. This feature has two consequences for explanation.

• The layered interpreter can represent both successful and failed computations in one pass, since it can handle multiple lines of reasoning. This is a very important feature since the proof for both successful and failed queries can be obtained by using the same meta-interpreter. The dynamic changes to the knowledge base does not impose a problem for this framework, and the proof, hence the explanation we obtain from the proof, is faithful to the current computation.

• Representing computations with *negation* and the pruning operator *cut* are no longer a problem by using the correct techniques [22].

2. We expect the expert system shell to provide alternative solutions when requested by the user. The layered interpreter satisfies this constraint.

3. Using Prolog for knowledge representation directly does not provide any information about the assertions to the database. However, using an embedded language provides information about the structure of the knowledge itself. As in our example of Section 2, the special language represents `Conditions` leading to an `Action`. If we have the representation available, then the explanations can be extended to incorporate the features of the language. Regarding our example, it is possible to answer queries of the form, *What* `Conditions` *hold to lead to* `Action` *?*, *Which* `Conditions` *do not hold to lead to* `Action` *?*.

4.3 Conclusions

In this paper, we have characterized meta-programming as a paradigm, and given a definition. Our specific interest is the use of the paradigm for building expert systems in Prolog. We advocate building an embedded language for specific applications, and presented a simple example for ordinal reasoning. An embedded language for a specific task can be designed very easily in Prolog. The advantage is that a user building an application can work with a restricted language instead of having to deal with the full complexities of Prolog. By exploiting the meta-programming capabilities of Prolog, specifying the language constructs, such as special purpose rules or objects, poses no problems. For knowledge engineers, more functionality can be included in the interpreter of the embedded language by separating knowledge representation from inference. Further, a knowledge engineer can modify the system by using both the embedded language and Prolog directly, since they are both directly available.

The other issue addressed is the use of meta-interpreters. Meta-interpreters provide an explicit representation of a particular model of computation. We focussed on a layered interpreter which allows multiple lines of reasoning. The expressive power of the layered interpreter was demonstrated in two applications - a shell for uncertainty reasoning and for explanation.

Acknowledgements

We thank Arvind Bansal for pointing out the relationship between the layered architecture and the generate-and-test paradigm. We also thank the reviewers for their valuable comments and the COMPOSERS group for providing a fruitful environment to work. This research was supported under NSF Grant No. 1R187-03911, and Equipment Grant No. DMC 8703210.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] H. Abramson and M.H. Rogers, editors. *Meta Programming in Logic Programming.* MIT Press, 1989.

[3] L. Aiello and G. Levi. The Uses of Meta-Knowledge in AI Systems. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 243–254. North-Holland, 1988.

[4] J. F. Baldwin and M. R. Monk. Evidence Theory, Fuzzy Logic and Logic Programming. Technical report, Inf. Tech. Research Centre, U. of Bristol, 1987.

[5] A. Ben-David, L. Sterling, and Y. H. Pao. Learning and Classification by Monotonic Ordinal Concepts. *Computational Intelligence*, 5(1):45–49, 1989.

[6] K. Bowen. Meta-Level Programming and Knowledge Representation. *New Generation Computing*, 3(4):359–383, 1985.

[7] K. Bowen and R. Kowalski. Amalgamating Langauge and MetaLanguage in Logic Programming. In K.L. Clark and S-A. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[8] A. Bruffaerts and E. Henin. Proof Trees for Negation as Failure: Yet Another Prolog Meta-Interpreter. In R. Kowalski and K. Bowen, editors, *Logic Programming, Proceedings of the 5th International Conference and Symposium*, pages 343–358. MIT Press, 1988.

[9] B.G. Buchanan and E. Shortliffe. *Rule Based Expert Systems The MYCIN Experiments of the Stanford Programming Project.* Addison-Wesley, 1984.

[10] A. Bundy. Incidence calculus: A mechanism for probabilistic reasoning. In *Proceedings of the International Conference on Fifth Generation Computer Systems,* pages 166–174. ICOT, 1984.

[11] P. Coscia, P. Franceschi, G. Levi, G. Sardu, and L. Torre. Object Level Reflection of Inference Rules by Partial Evaluation. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 313–327. North-Holland, 1988.

[12] R. W. Floyd. The Paradigms of Programming. In *ACM Turing Award Lectures- The First Twenty Years*, pages 131–142. ACM Press, 1987.

[13] F. Hayes-Roth, D. Waterman, and D. Lenat, editors. *Building Expert Systems.* Addison-Wesley, 1983.

[14] R. Ng and V. S. Subrahmanian. Probabilistic Logic Programming. Technical Report CS-TR-2399, Dept. of Computer Science, U. of Maryland, 1990.

[15] E. Shapiro. Logic Programs with Uncertainties: A Tool for Implementing Rule-Based Systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 529–532. William Kaufmann Inc., 1983.

[16] S. Shekhar and C.V. Ramamoorthy. COOP: A Shell for Cooperating Expert Systems. In *Conference on Tools for Artifical Intelligence 89*, pages 2–11. IEEE-Computer Society Press, 1989.

[17] L. Sterling and R. Beer. Meta-Interpreters for Expert System Construction. *Journal of Logic Programming*, 6(1-2):163–178, 1989.

[18] L. Sterling and A. Lakhotia. Composing Prolog Meta-Interpreters. In R. Kowalski and K. Bowen, editors, *Logic Programming, Proceedings of the 5th International Conference and Symposium*, pages 386–403. MIT Press, 1988.

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

[20] L. Sterling and L. Ü. Yalçinalp. Explaining Prolog-Based Expert Systems Using a Layered Meta-Interpreter. In *Proceedings of 11th International Joint Conference in Artificial Intelligence*, pages 66–71. Morgan–Kaufmann, 1989.

[21] M. Wick and J. Slagle. An Explanation Facility for Today's Expert Systems. *IEEE Expert*, 4(1):26–36, 1989.

[22] L. Ü. Yalçinalp and L. Sterling. Layered Meta-Interpreters for Describing Prolog's Control Flow. Technical Report CES–89–08, Dept. of Computer Eng. and Sci., Case Western Reserve U., 1989.

[23] L. Ü. Yalçinalp and L. Sterling. Uncertainty Reasoning in Prolog with Layered Meta-Interpreters. Technical Report 90-110, Center for Automation and Intelligent Sys. Res., Case Western Reserve U., 1990.

[24] L. Ü. Yalçinalp, L. Sterling, A. Lakhotia, and A. Bansal. The COMPOSERS Guide to Meta-Programming. Technical Report 89-166, Center for Automation and Intelligent Sys. Res., Case Western Reserve U., 1989.