Syracuse University

SURFACE

Electrical Engineering and Computer Science - Technical Reports

College of Engineering and Computer Science

5-1985

A Meta-Level Extension of Prolog

Kenneth A. Bowen
Syracuse University

Tobias Weinberg

Follow this and additional works at: https://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation
Bowen, Kenneth A. and Weinberg, Tobias, "A Meta-Level Extension of Prolog" (1985). *Electrical Engineering and Computer Science - Technical Reports*. 36.
https://surface.syr.edu/eecs_techreports/36

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

85-1

A Meta-Level Extension of Prolog*

Kenneth A. Bowen
Computer & Information Science
313 Link Hall
Syracuse University
Syracuse, NY 13210
kabowen%syr@csnet-relay

Tobias Weinberg
AI Technology Group
Digital Equipment Corp.
77 Reed Road
Hudson, MA 01749
weinberg%logic.DEC@DECWRL

SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

Technical Report CIS-85-1 May 1985
School of Computer and Information Science
Syracuse University

A Meta-Level Extension of Prolog*

Kenneth A. Bowen
Computer & Information Science
313 Link Hall
Syracuse University
Syracuse, NY 13210
kabowen%syr@csnet-relay

Tobias Weinberg
AI Technology Group
Digital Equipment Corp.
77 Reed Road
Hudson, MA 01749
weinberg%logic.DEC@DECWRL

* Work on this project was supported in part by AFOSR grant AFOSR-82-0292. The authors would like to thank Maarten van Emden and the students in Bowen's seminar at Syracuse University for helpful and stimulating discussions.

I. Introduction

Prolog has many attractive features as a programming tool for artificial intelligence. These include code that is easy to understand, programs that are easy to modify, and a clear relation between its logical and procedural semantics. Moreover, it has proved possible to create clear and efficient implementations. Nonetheless, we perceive several shortcomings. Chief among these is difficulty representing dynamic databases (databases which change in time) and an apparent restriction to backward chaining, backtracking, depth-first search. Our intent in this paper is to present an extension to Prolog, called metaProlog, which preserves the virtues of Prolog while introducing powerful constructions to attack these problems. This work is a direct continuation of the investigation into meta-level programming in logic begun by Bowen and Kowalski [1982].

Many applications of artificial intelligence demand facilities which amount to the ability to dynamically manipulate databases. Databases are naturally represented in Prolog as a set of assertions and clauses. This exploits all the advantages of Prolog's inherent deductive machinery. However, the logical core of Prolog provides no conceptual basis for segmenting or modifying the database. Most implementations of Prolog have provided ad hoc extensions to the basic logic programming paradigm which allow for dynamic modification of the program database by the program itself. But since the database is the program, the use of these facilities introduces difficulties similar to those introduced by global variables and self-modifying code in conventional programming languages. The effect of these features on the virtues listed above is catastrophic. Programs become difficult to understand, reliable modification of the code is almost impossible, and the logical semantics is utterly destroyed. We know of no mathematical or philosophical definition of first-order proof where the collection of axioms is not fixed. We would suspect any such notion to be incoherent. We believe these difficulties can be overcome by the introduction of theories as first-class objects which can be dynamically created and passed as parameters. In standard Prolog, goals are invoked with respect to a single background theory. In metaProlog, goals must be proved in an explicitly identified theory. We regard this system as simply a first-order logical theory of axiom sets and proofs.

The means of indicating that a metaProlog goal G should be solved in a particular theory T is an explicit call on the proof predicate demo. From a logical point of view, the proof predicate is really a relation between three objects: the theory T, the goal G, and the proof P which attests to the solvability of G in T. But logic programming is not only concerned with the static existence of proofs, but also the process of discovering them. That is, it is also concerned with the notion of search space and search strategies. Thus, for logic programming, the deep central relation is the one which holds between a theory T, a goal G, and the complex object consisting of a proof for G in T seen as a portion of a search space explored by a particular search strategy. Our investigations have led us to the conclusion that all of these entities must be treated as first-class objects (metaProlog terms) capable of being manipulated and passed as values of parameters.

This approach appears to provide a logically sound programming formalism sufficiently powerful to write clear reliable programs for experimental and applied artificial intelligence. We also believe it possible to construct efficient implementations of such a system, but will leave this question to a later paper. Although problem of efficient implementation has been of deep concern throughout our design process, our concern in this paper is with questions of conceptual and logical foundations. (Various portions of the system have been simulated by implementations in Edinburgh Prolog and parts of a prototype system have been written in C.)

Let us close this introductory section with an example illustrating the power of the approach. Suppose that one has two collections of goals, G1,...,Gn and H1,...,Hm and that one wishes to solve G1,...,Gn in theory T1 under one search strategy M1, and to solve H1,...,Hm under another strategy M2 in theory T2, where both M2 and T2 depend on the state of the computation resulting from the solution of G1,...,Gn, as well as on T1, G1,...,Gn, and H1,...,Hm. Let F be the problem to be solved by this work and let 'next_strategy' be some procedure acting on theories, goals, and computation states which will be used to compute T2 and M2. Then we could describe F as follows:

```
F is solvable if
        G1&... &Gn is solvable in T1 using strategy M1
        and S1 is the resulting computation state,
        and next_strategy acting on T1, G1&...&Gn,
                H1&...&Hm, and S1 yields T2 and M2,
        and H1&...Hm is solvable in T2 using strategy M2.
```

Let v1,...,vk be the variables of G1,...,Gn,H1,...,Hm. Then, in the metaProlog formalism we will introduce below, this could be expressed by:

```
all [v1,...,vk,T2,M2,S1]:
        F(v1,...,vk) ←
                demo(T1, G1&...&Gn, strategy(M1)+comp(S1))
                        & next_strategy(T1, G1&...&Gn,
                                        H1&...&Hm, S1, T2, M2)
                        & demo(T2, H1&...&Hm, strategy(M2))
```

II. Meta-Level Programming

It is important to make clear our notion of meta-level programming. Briefly, one distinguishes between the formal language being used to conduct some (unspecified) axiomatic investigation (the object language) and the language used to carry on any discussion about the object language (the metalanguage). For many purposes (including those of this paper), the metalanguage need only be powerful enough to discuss the combinatorial syntactic properties of the object language. The essential point is that the relations of the metalanguage are about the syntactic entities of the object language: the variables of the metalanguage range over various syntactic entities of the object language. In contrast, the variables of the object language either have no specified range (when it is viewed as a formally uninterpreted language) or (when the object language is treated as being interpreted) range over the members (possibly extremely mathematically complex) of some specified set.

Properly viewed, an ordinary Prolog interpreter is already a meta-level object. The object level consists of a fragment of ordinary first-order logic, a language and proof predicate. The latter describes which formulas of the language are consequences of sets of other formulas of the language. The meta-level of a theorem-prover is concerned with the manipulation of sets of object-level formulas in the search for a collection of formulas which witnesses the derivability of a given goal formula from a given set of axiom formulas. The prover proper is a meta-level object because its variables range over formulas (and other syntactic classes) of the object level language.

Thus a Prolog interpreter really defines a relationship between sets of formulas (the program database), goal formulas, and proofs, namely the relation that the proof witnesses the deducibility of the goal formula from the program database. (Note that the standard Prolog interpreters return a portion of the proof to the user, namely that part of the substitution applying to the variables occurring in the goal). As commonly implemented, pure Prolog interpreters incorporate the program database as a fixed part of the interpreter. Thus, from a meta-level point of view, a standard Prolog interpreter provided with a fixed program database defines a certain meta-level unary predicate applying to goal formulas. This meta-level unary predicate holds for just those goal formulas which are deducible from the program database by the interpreter. The fundamental operator of standard Prolog systems is thus a one-place operator (usually written call(...)) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the proposal set forth by Bowen and Kowalski was to utilize a system implementing the full deducibily relation described above. Such a system would have metavariables which not only range over formulas and terms, but would also allow the metavariables to range over sets of formulas (called theories). The fundamental operator of such a system is a three-place operator, usually written demo(Theory,Goal,Proof), which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

All metaProlog program databases are the values of metaProlog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the built-in predicate demo/3, the system predicates include

```
add_to(Theory, Axiom, NewTheory)

drop_from(Theory, Axiom, NewTheory)
```

which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause containing calls of the form

..., add_to(T1, A, T2), demo(T2, D, P),... (*)

where the theory which is the value of T1 has been constructed by the earlier calls. The effect of (*) would then be to construct a new theory T2 resulting from T1 by the addition of the formula A as a new axiom, and then the invokation of a search for a proof of the formula D from the theory T2. Since demo implements the proof relation, such programs as (*) preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of demo are expressed by what were called reflection rules by Bowen and Kowalski:

If demo(T, A, P), then A is derivable from T via proof P.

If A is derivable from T via proof P, then demo(T, A, P).

These rules provide the justification for the implementation of calls on demo in the abstract metaProlog machine as context switches. In essence, at most times the machine behaves as a standard Prolog machine with the current theory (the analogue of the usual fixed program database) indicated by a register. When a call demo(T, A, P) is encountered, the database (theory) register is changed to point to T and a new search for a deduction of A is begun. Thus the efficiency of standard Prolog computations is preserved and the overhead of meta-level computation is localized in the construction of new theories from old. This approach provides a meta-level programming methodolgy suitable for constructing other methods of exploring the search space of derivations of A from T besides the top-down depth-first approach of standard Prolog. Exploitation of this approach will ultimately provide the meta-level programmer with a library of search strategies which can be (programmatically) invoked depending on the particular problem and context.

In order for any language M to serve as a metalanguage for another language L, M must contain names for all the appropriate syntactic entities of L. Thus, since metaProlog is to serve as its own metalanguage, it must contain names for all of its own syntactic entities, just as any natural language does. To this end, constants act as names of themselves. For non-constant items, metaProlog provides structural or non-structural names (and sometimes both), where the former are compound terms whose structure reflects the syntactic structure of the syntactic item they name. Facilities for manipulating names are provided, such as methods of obtaining the name of a compound expression from names of its components. And methods for moving between a name and the thing it names are included, analogous to univ (= ..) of ordinary Prolog.

One further subtle point regarding variables must be treated at this point. The logical interpretation of Prolog's theorem prover stipulates that variables actually occurring in the program's clauses are in fact implicitly universally quantified object level variables, even though they are syntactically indicated by metavariables. In using a clause, the interpreter replaces these universally quantified object level variables by existentially quantified meta-level variables. The syntactic conflation of object- and meta-level variables is acceptable for pure Prolog deductions, but causes difficulties as soon as assert and retract are added to the system. If an expression (say p(X)) contains a metavariable X which is uninstantiated at the time when assert(p(X)) is executed, there is a natural sense in which the call assert(p(X)) is incoherent: the formula to be added to the database is not fully specified. The Prolog approach to this problem is to once again conflate the existentially quantified metavariable X with a corresponding universally quantified object-level variable, actually asserting (all X)[p(X)]. This approach destroys the logical semantics of clauses in which such calls occur. Assuming that there are no clauses for the predicate p already in the database, the goal statements of the following two clauses should be logically equivalent:

$$\mathrm{h} :- X = a, \mathrm{assert(p(X)),\ p(b)}. \qquad (A1)$$

$$\mathrm{h} :- \mathrm{assert(p(X)),\ X\ = a,\ p(b)}. \qquad (A2)$$

But the first fails, since it only adds p(a) to the database, while the second succeeds, since it adds (all X)[p(X)]to the database. To avoid such difficulties, the metaProlog system requires that programmers be explicit about their intentions, clearly indicating universally quantified object variables. Thus, to add (all X)[p(X)]to a theory T, one would write

$$\mathrm{add\_to(Theory, allX : p(X), NewTheory).}$$

Note that in the above expression, the symbols X, Theory, and NewTheory are metaProlog constants. There is no way a metaProlog programmer can write the name of a metaProlog variable. He or she can only indicate the position of such variables to the metaProlog interpreter by using the explicit universal quantifier which is represented by the symbol all/1. From the syntactic point of view, all/1 is just a function symbol used to form terms. The symbol all/1 functions as a quantifier only when a term formed with it occurs as a clause in a theory or as an argument to certain meta-level predicates. As an example, consider the following two metaProlog clauses which achieve the same effects as the clauses (A1) and (A2) above:

```
all [X,T1,T2]:
        h(T1, T2) ←
                X = a & add_ to(T1, p(X), T2)
                & demo(T2, p(b), _).(B1)
all [T1,T2]:
        h(T1, T2) ←
                add_ to(T1, all X : p(X), T2)
                & demo(T2, p(b), _).(B2)
```

(N.B. There is no sense in which the symbol X as it occurs in (B2) is a variable – it is a metaProlog constant.)

III. The metaProlog System.

The metaProlog system is syntactically similar to the Edinburgh system. We use ← for the implication symbol (instead of :–) and use & as the conjunction operator, rather than comma. The major difference lies in our treatment of variables and constants. For the reasons we discussed above, we require that the implicit universal quantifiers on clauses be made explicit. Quantification is indicated by applying the function symbol all/1 (which is parsed as a prefix operator) to a term whose principal functor is the binary infix operator :/2 and whose first argument is either a metaProlog constant or a list of metaProlog constants and whose second argument is an arbitrary metaProlog term (we call such things "indicated terms"). Thus, for example, the Edinburgh clause

```
    append([Head | Tail], Rt, [Head | R_Tail ]) :-
            append(Tail, Rt, R_Tail).
```

could be written

```
    all [Head, tail, Rt, r_Tail]:
            append([Head | tail], Rt, [Head | r_Tail]) ←
                    append(tail, Rt, r_Tail).
```

If the clause contains only one variable, the list brackets in the quantifier can be dropped. (Dropping the convention of regarding symbols beginning with upper case as variables reduces the need for single quotes and the awkwardness that entails.)

The set of built-in predicates of pure Prolog exists as a subset of the metaProlog built-ins. (Indeed, pure Prolog is a subset of metaProlog, modulo the conventions regarding variable naming and quantification.) As is clear from the preceding sections, the three predicates demo, add_to, and drop_ from constitute the core built-ins for manipulating theories and proofs (replacing call, assert, and retract from Prolog). We have already discussed add_to and drop_ from. We need to discuss demo in somewhat greater detail.

Calls on demo support a convenient idiom for describing implicit unions of theories. Specifically, a call of the form

demo(Theory1&Theory2, Goal, Search_Info)

is logically equivalent to the call

demo(Theory3, Goal, Search_Info)

where Theory3 is the ordered union of Theory1 and Theory2 in the following sense: If theories are regarded as the ordered list of their axioms, then Theory3 satisfies

append(Theory1, Theory2, Theory3).

However, the system does not physically create Theory3, but regards the expression Theory1 & Theory2 as a description of a virtual theory. In effect, when searching for a rule or fact to apply to a selected subproblem of the current goal, it first searches Theory1 for a candidate, and only on failing to find such a candidate in Theory1, it then searches Theory2. Another usage supported is the explicit indication of the axioms of the theory. Namely, if it is desired to search for a deduction of G from A1,...,An, this is achieved by the call

demo([A1,...,An], G, Search_Info).

The two usages can be combined, as in the calls:

demo([A1,...,An]& Theory2, Goal, Search_Info).

demo(Theory1 & [A1,...,An], Goal, Search_Info).

We have stated earlier that the proof predicate demo is a three-place relation holding between a Theory, a Goal, and a Search Space/Proof. We need to explain further the nature and use of the third argument. It may be used for a variety of purposes. These include extracting pieces of the proof or search space, controlling the search strategy, and introducing or extracting annotations to the proof, such as confidence factors. We intend this facility to be user-extensible. As a first step in this direction, search information expressions can be combined using the infix operator +/2, as in

demo(Theory, Goal, Search_Info1 + Search_Info2).

Two examples of search information annotations are proof(P) and branch(B). The proof(P) annotation causes the system to accumulate a representation of the proof branch in the (uninstantiated) variable P, allowing the programmer to extract a successful proof for furthur processing, such as providing explanations, etc. We see no reason to prevent the programmer from passing a partially constructed proof cum search_space to demo through the use of proof(P). The branch(B) expression causes the call

demo(Theory, Goal, branch(B))

to succeed in all cases, binding the uninstantiated variable B to the left-most branch of the search tree. Note that in the case that the left-most branch is theoretically infinite, the call will still succeed due to depth bound limitations of the system. Backtracking into this call will cause B to be bound to successive branches of the search tree. As discussed in detail below, the call

setOf(B, demo(T, G, branch(B)), Branches) would cause Branches to be bound to the (lazy) list of all branches of the search tree for G relative to T in the order that they are explored by the system.

As part of our program of providing powerful tools for AI programming, we seek to offer the programmer control of stream-based communication between concurrent processes, while still holding to our program of preserving the essential elements of Prolog semantics. In the logic programming context, this amounts to implementing some form of and-parallelism. The most straight-forward sort of and-parallelism to attack is simple producer - consumer computations. However, since the implementation of producer - consumer relations in which the producer is allowed to non-determinately reconsider the stream it has produced is difficult to say the least, we restrict ourselves to determinate and-parallel situations. Other approaches to parallelism in Prolog (e.g., Parlog (Clark and Gregory [198?]) or Concurrent Prolog (Shapiro [1983])) achieve this restriction by introducing committed choice. However, while preserving the correctness of the computations, this approach loses Prolog's deductive completeness. In contrast, we preserve both the correctness and completeness by restricting ourselves to running in parallel only producer - consumer computations in which the production of the stream is determinate. (Note that the computation of the stream may involve non-determinate aspects; it is simply at the point of adding a new element to the stream that the producer must act determinately. Also, consumption of the stream may be entirely non-determinate.) The essential point appears to us that it is not really the processes which must be forced to be determinate, but rather the communication between them. Thus our approach is to force the producing process to determinately fill the communication buffer; all else can be non-determinate.

We have identified two useful classes of producer - consumer computations which meet our requirement (and the possibility of others certainly exists). The first is the (lazy) production of sets via complete exploration of a search tree (i.e., the lazy form of Prolog's setof construct) and the production of streams by determinate tail-recursive procedures. These are indicated in metaProlog programs by the constructs

all_solutions(Template, Goal, Stream) and
streamOf(Goal, Stream).

We see these as entirely encapsulated independent computations: their only method of communication with parent or sibling processes is via the stream variable. Every element of the stream must be ground. If the producing process would have otherwise produced a partially instantiated term as a stream element, that term must be converted to a ground term by use of the 'naming' or 'indicating' operator discussed above in conjunction with quantification. The same restrictions clearly must apply to the Goal argument of both stream_of and all_solutions. One method of implementation is that of producer variables. The first invokation of Goal binds the variable Stream to a buffer together with a description of Goal and its environment. Subsequent attempts to access the variable stream by the consumer causes Goal to be run through one cycle of its computation, binding Stream to a cons cell whose first element is the item produced and whose second element is a description of the rest of the buffer together with the current state of the computation of Goal. It is important to recognize that the producer variable does not act like normal Prolog variable. Indeed, since any attempt to match a non-variable term against an element of the stream causes the stream element to be instantiated to a ground term by the producer, and since the producer is determinately committed to the binding it produces, producer variables behave for all intents and purposes as ground objects. Thus it is perfectly permissable for producer variables to appear in the Goal arguments of other producer processes. This allows for two-way communication between producers. Process synchronization is achieved by requests for bindings passed from process to process. It is clear that the two communicating processes must created simultaneously. The construct

simultaneous(Process1, Process2)

achieves this effect. It can be invoked with any number of arguments.

Because we see these processes as entirely sealed computations with their own environments, it is possible, in appropriate hardware settings, to run them truly in parallel, allowing the producing process to fill the buffer up to some pre-set limit or even run to completion when the stream is finite. On sequential hardware, the implementation is simple co-routining of the producer and consumer, with the additional overhead entirely localized in the communication – there is no slow down of the basic Prolog computation. In particular, the computational children of the Goal of one of these processes do not inherit the parallel mode: they run as normal Prolog processes. It should be possible to mix parallel and co-routined execution with no change to the program or its behavior. Finally, while we have not attempted to do so, it seems evident that or-parallelism could be introduced with a stream operator whose top level was expanded in an or-parallel manner. One might even introduce committed-choice versions of such an operator without disturbing the semantics of the rest of the system.

IV. A Programming Example.

In this section we describe approaches to fault-detection in digital circuits based on the ideas of Esghi [1982]. For the purposes of fault-finding, the devices must be described in some sort of predicate calculus formalism, for example andGate(G, In1, In2, Out), which expresses that G is an and-gate with input lines In1 and In2, and output line Out. Similarly for orGate. The topological description of the circuit is contained in the theory c1, which besides expressions such as those above, indicates the lists of input and output wires for the circuit. The behaviors of the circuit components are described in the theory tt (for truth tables) which contains such rules as:

```
all [Gate, In1, In2, Out]:
        andTable(Gate, In1, In2, Out) ←
                not(exceptional(Gate))
                & standardAnd(In1, In2, Out).
standardAnd(high, high, high).
all In2 :
        standardAnd(low, In2, low).
all In1 :
        standardAnd(In1, low, low).
```

The significance of the predicate "exceptional" will be described later. The topology and component behaviors can be used to predict the circuit outputs given the inputs as described in the theory laws which defines a predicate predict(InputValueList, OutputValueList) which calculates the output wire values by backchaining through the circuit from the output wires back to the input wires. Normal simulation of circuit function would be carried out by the call:

```
demo(c1&tt&laws, predict(InList, OutList), _).
```

The fault detection problem consists of attempting to locate the source of the fault based on faulty input-output behavior. We will make the common simplifying assumption that the fault is caused by a single wire of a single gate being stuck at high or low. The basic method we will apply (due to Esghi) attempts, given a faulty input-output pair (If, Of), to determine a theory Tf obtained by minimal perturbation of the theory tt such that Tf correctly describes the behavior of the faulty circuit. Examination of Tf will then reveal the location of the fault. The basic algorithm runs as follows:

```
1.   From tt and (If, Of), construct a set HYP of theories Hi such that:
     i)    For all i, demo(c1 & Hi & laws, predict(If, Of), _ ) succeeds;
     ii)   For some i, Hi correctly describes the faulty circuit.
2.   If HYP contains only one element, halt and output HYP.
3.   Otherwise, proceed as follows:
     i)    Choose distinct Hi and Hj from HYP;
     ii)   Construct a discriminating input Id which distinguishes Hi
           and Hj; if no such input exists for any choice of
           Hi and Hj, halt and output HYP.
4.   Apply Id to the faulty circuit, obtaining output Od.
5.   Delete from HYP all Hi for which the following call fails:
                demo(c1 & Hi & laws, predict(Id, Od), _).
6.   Goto 2.
```

Once the set HYP is constructed in step 1, the remainder of the algorithm is basically straight-forward (though we will return to it below). The set HYP is constructed by first making the call:

```
        setOf(B, demo(c1&tt&laws, predict(If,Of),
                               user_choice+branch(B)), BadBranches)
```

First note that the goal of this setOf would fail without the control information since tt describes the correct circuit, while (If, Of) is faulty for this circuit. But the control branch(B) causes the setOf to produce the list of all branches of the search tree. The user_choice forces these branches to be constructed according to the user choice theory uc which describes, in the style of Pereira[], a next-goal choice procedure which delays as long as possible selecting goals of the form

"andTable(_, _, _)" or "orTable(_, _, _)".

Next each of the failed proof branches on BadBranches is used to guide the generation of candidate theories Hi for HYP. Essentially, tt is modified so that failing calls of the form

"andTable(G, _, _)" or "orTable(G, _, _)"

become successful: essentially the failing call is added to tt together with the assertion "exceptional(G)" to produce Hi. HYP is then filtered by steps 2-6.

For any realistic circuits, the lists BadBranches and HYP will be unmanagably large if produced in their entirety. However, the lazy nature of setOf causes the production of BadBranches to be co-routined with the action of gen(BadBranches, HYP) which generates HYP from the elements of BadBranches. The procedure gen is defined as a tail-recursive streamOf construct, so that it can in turn be co-routined with the filter process implementing 2-6. The nature of the streamOf and simultaneous constructs allows the dynamic generation of processes. This permits filter to be organized in a manner analogous to the classic parallel implementations of the seive of Eratosthenes. First, each Hi making it through the current filter is recorded on a working list. Next, as each pair Hi, Hj makes it through the filter, the discriminating pair (Id, Od) is generated, and used to produce a small check process check(Id, Od, H) which tests H to determine whether H correctly predicts the i-o pair (Id, Od). This check process is attached at the current end of the filter, much as the divisor test for the most recently generated prime is attached at the end of the seive. Also, as each pair (Id, Od) is generated, check(Id, Od, H) is applied to each element of the current temporary scratch list, and any H on that list which fail the test are removed. The entire process gradually terminates as each of the processes, from the initial setOf call through last check process gradually close down (by seeing the streams they are consuming being closed). When the last of these processes closes down, the elements remaining on HYP all correctly predict the faulty i-o pair and pass all the tests for behavior of the real faulty circuit which have been generated. If HYP contains more than one element, these hypothetical cannot be distinguished by i-o behavior. They are all candidate Hi descriptions of the possible source of the fault. Finally, let us note that these methods can be adapted to a setting of hierarchical diagnosis in the style of Genesereth [1982].

V. Conclusion.

We have elaborated a system called metaProlog which, narrowly conceived, is an extension of Prolog. The real power (meta_power) of this system lies not in the specific system facilities we have described, but in the programming methodology they introduce. The example in the preceeding section only beings to explore the possibilities of this system. Using this approach, we have begun to logically characterize frames and default hierarchies, generalized networks of theories and semantic nets, and more general control strategies such as bottom-up or breadth-first search. There is no logical requirement that the only notion of proof in metaProlog be the Horn clause-oriented demo predicate we have introduced. We see no reason why other methods of proof cannot co-exist with demo. We envisage the situation in which another method of proof would be rapidly prototyped using explicit recursive calls on the present demo, and later integrated into the system at a low level using the same bootstrapping methods we are adopting for the implementation of the basic metaProlog system.

By stepping up to the full meta-level point of view wherein all components of the system have become first-class objects, we have entered the realm of a logical construal of Theories, Goals, and SearchSpaces in which it is possible to axiomatically and programmatically characterize elements of the system previously regarded as parts of the implementation. This allows us to introduce powerful logical approaches to the construction of artificial intelligence systems.

References

Bowen, K.A. and Kowalski, R.A., *Amalgamating language and metalanguage in logic programming*, in **Logic Programming**, eds Clark and Tarnlund, Academic Press, 1982, pp. 153-172.

Clark, K., and Gregory, S., *Parlog: A parallel logic programming languag e*, Research Report DOC 83/5, Imperial College, March 1983.

Eshghi, K., *Application of meta-language programming to fault finding in logic circuits*, in **First International Logic Programming Conference**, 1982, pp 240-246.

Genesereth, M., *Diagnosis using hierarchical design models*, **Proc. Nat'l Conf. on Artificial Intelligence**, 1982, pp. 278-283.

Pereira, F., and Warren, D.H.D., *Definite clause grammars for language analysis – a* *survey of the formalism and a comparison with augmented transition grammars*, **Artificial Intelligence** 13 (1980), pp. 231-278.