Syracuse University
SURFACE

Electrical Engineering and Computer Science Technical Reports

College of Engineering and Computer Science

1988

MetaProlog User Manual

Hamid Bacha
Syracuse University

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

Part of the Computer Sciences Commons

Recommended Citation
Bacha, Hamid, "MetaProlog User Manual" (1988). *Electrical Engineering and Computer Science Technical Reports.* Paper 31.
http://surface.syr.edu/eecs_techreports/31

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.

MetaProlog User Manual

Hamid Bacha

Logic Programming Research Group
Technical Report LPRG-TR88-1

Logic Programming Research Group
School of Computer and Information Science
313 Link Hall - Syracuse University
Syracuse, New York 13210

hamid@logiclab.cis.syr.edu

Table of Contents

Introduction 1
Theories and Theory Names 2
Creating New Theories 2
Context Switching 4
Virtual Theories 4
Nested Theories 5
Proofs 5
Proofs as First Class Objects 6
MetaProlog Syntax 7
Listing Clauses 9
Meta-level Reasoning 9
Debugger 11
Miscellaneous 13
Summary of Commands 15

Acknowledgement: This version of MetaProlog was developed by the present author from the common system designed by the Logic Programming Research Group at Syracuse University. The author wishes to thank the many people who worked on the system over the years: Ken Bowen, Kevin Buettner, Ilyas Cicekli, Keith Hughes, Andy Turk, and Toby Weinberg.

MetaProlog User Manual[1]
(First Release V1.0)

H. Bacha

1. Introduction

MetaProlog is a logic-based programming language which subsumes the full Prolog language. This implementation is an incremental compiler (which also looks and feels like an interpreter) supporting meta-level constructs that are usually provided by the underlying architecture in other systems and are not directly available to the user. The most obvious feature of MetaProlog is the ability to handle multiple databases (referred to as theories) at the same time. In contrast to ordinary Prolog's single-theory database, a MetaProlog database is a collection of theories. A theory is a first-class object and can be passed around as the value of a variable. In fact, proofs are handled through the **demo** predicate which takes a theory as its first argument and a goal as its second argument: demo(Theory, Goal). The **demo** predicate is a meta-level construct which witnesses the derivability of the given goal from the given theory. Theories are collections of **viewpoints** (clauses and facts). The word **viewpoint** is used to reflect the fact that the same relation may be present in many theories with slight or substantial variations from one theory to the other. Each theory is then seen as having its own viewpoint on that given relation. The following example illustrates the idea:

Overall database entry for relation loves/2:

```
loves(john, jane).
loves(jane, jack).
loves(jane, john).
```

One theory T1 may view the procedure as:

```
loves(john, jane).
loves(jane, john).
```

[1]This work was supported by grant F30602-81-C-0169 from AFOSR and administered by RADC.

while another theory, say T2, may view it as:

loves(john, jane).
loves(jane, jack).

Both theories T1 and T2 coexist in the same MetaProlog database but have different viewpoints on the love relation.

2. Theories and Theory Names

Each theory is represented internally by a theory descriptor. When a new theory is created, this descriptor is returned as the value of the variable representing the new theory. In the example:

```
addto(OldTheory, Clause, NewTheory)
```

the variable **NewTheory** is unified with an internal representation of the theory descriptor for the new theory. The theory descriptor contains, among other things, the ID of the theory. The ID is just an integer starting with zero for **basetheory** (described in section 3), and incremented by one for every new theory created by the system. If the only way to get hold of a theory is through the variable unified with its descriptor, all theories would be temporary and cease to exist the moment we loose the variables representing them. Obviously, some theories need to be around permanently (at least those loaded into MetaProlog through **consult**), and we need some kind of global mechanism to refer to them. The predicate **nameof(Theory, TheoryName)** is provided to assign the ground term **TheoryName** as a global name to the theory whose descriptor is the value of the variable **Theory**. **TheoryName** can then be used throughout the system to refer to the corresponding theory. Assigning a name to a theory makes the theory permanent. A theory with no name is lost the moment the variable holding its descriptor becomes inaccessible and is therefore referred to as a temporary theory. For this reason, only default theories and some major ones should be assigned names. The rest should be considered temporary theories. The following example illustrates the idea.

```
test1(X) :-
    addto(basetheory, loves(paul, mary), NewTheory),
    demo(NewTheory, loves(paul, X)).
```

Assume our current context is basetheory. The goal test1(X) will first cause the creation of the theory NewTheory, then context is set to this new theory to prove the goal loves(paul, X), and finally it is reset back to basetheory. At this moment, the variable NewTheory ceases to exist. We have no way to access the newly created theory. That's why it is called a temporary theory. By contrast, a theory created from basetheory and clauses from an external file should be a permanent one. It is created as follows:

```
addto(basetheory, file(expert_shell), NewTh),
nameof(NewTh, expert)
```

or equivalently:

```
consult(expert_shell, expert).
```

The new theory could be an expert system shell, for example, and it is clear that it should be a permanent one.

3. Creating New Theories

When the MetaProlog system is started, the **current theory** (or current context) is set to a top-level theory known as **basetheory**. This theory contains all the system built-ins. Any new theory created should have access to all these built-ins. Therefore, any new theory is created either directly out of basetheory (plus some other clauses), or out of some other theory that has access to all the system built-ins. In other words, the theories form a tree hierarchy with basetheory as the root (see Figure 1).

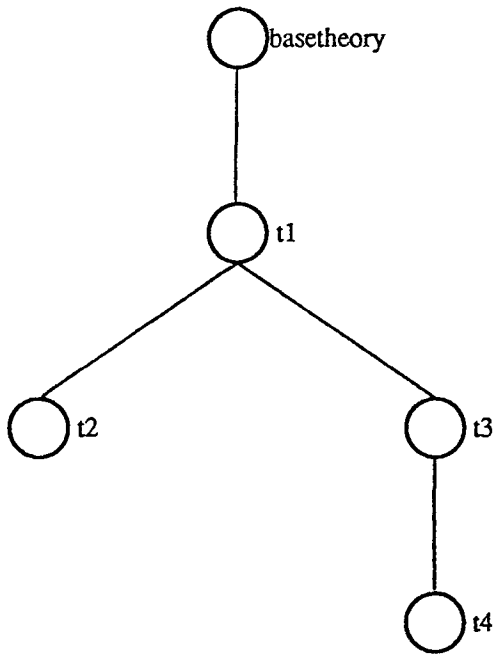

Figure 1: MetaProlog Theories as a Tree Hierarchy

New theories are created either by directly specifying the clauses they contain or by modifying some clauses of an existing theory. Top-level theories (also known as **default theories**) use basetheory as their starting theory. The predicates used to create new theories are:

consult(FileName, TheoryName)
addto(OldTheory, Clauses, NewTheory)
dropfrom(OldTheory, Clauses, NewTheory)
createtheory(NewTheory)

The first predicate is used to create a top-level theory from an external file called **FileName** containing MetaProlog clauses. The theory created is a permanent theory and the second argument of the predicate is the name to be associated with it. The second and third predicates use the clauses of an existing theory to create a new theory. Some of the clauses may be modified or deleted. OldTheory is basetheory (for top-level theories) or any user created theory. It may be either a theory name or a theory descriptor. NewTheory should be a variable which will be unified with an internal representation of the newly created theory. (When OldTheory is the name of a permanent theory, NewTheory may by the same as OldTheory instead of a variable. This has the effect of adding or 'asserting' to an existing theory. However, since this variation of addto has the same effect as assert in Prolog, its use should be discouraged) The predicate **nameof** may be used to explicitly name the new theory. The second

argument (Clauses) can be a clause, a name of an external file specified as **file(FileName)** and containing MetaProlog clauses, or a list of clauses and/or file name specifications. The predicate **consult(FileName, TheoryName)** may be redefined as:

addto(basetheory, file(FileName), NewTheory),
nameof(NewTheory, TheoryName).

Finally, the last predicate is used to create an empty theory. The argument can be either a variable or a ground term which will serve as the name of the new theory.

When a theory T2 is created from a theory T1, we say that T2 is a **descendant** of T1, or equivalently, that T1 is an **ancestor** of T2. The reason is that the clauses of T1 are not copied to T2 (that would be very expensive). Instead, through a clever representation of theories and viewpoints, T2 can access the clauses it shares with T1 in a very efficient manner. We say that T2 **inherits** those clauses from T1, or that T2 includes the viewpoints of T1 on those relations. Since every theory is a descendant of basetheory, every theory has access to the system built-ins. As an example, let's create theory t1 and its descendant t2:

addto(basetheory, [p(1), p(2)], T1), nameof(T1, t1).
addto(t1, p(3), T2), nameof(T2, t2).

If we submit the goal demo(t1, p(X)), we get the answers X = 1 and X = 2. If we submit the goal demo(t2, p(X)), we get X = 1, X = 2, and X = 3. Let's create another theory t3 as a descendant of t1:

dropfrom(t1, p(1), NT), nameof(NT, t3).

Now if we submit the demo(t3, p(X)), we get only one answer X = 2. Figure 1 shows the tree structure for the above theories.

4. Context Switching

Proofs can be carried out in any theory known to the MetaProlog system. Moving from one theory to another is referred to as context **switching**. Context switching is achieved either indirectly through the use of the **demo** predicate or directly through the **setcontext** predicate. The most common way of switching context is through the use of **demo(NewTheory, Goal)**. The context is temporarily switched from the current theory, say CT, to **NewTheory** to prove **Goal**, then reset back to CT. **NewTheory** may be either a regular theory or a virtual theory (as defined in the next section). When the system is started, the default context is basetheory. The predicate **setcontext(TheoryName)** may be used to specify a new default context, i.e. the theory the system should be in whenever it is done carrying out the proof of a top-level goal. **TheoryName** must be a permanent theory. Context switching is extremely fast since it only involves the setting of a register known as the Current Theory Register.

5. Virtual Theories

A context can also be a virtual theory made up of multiple theories. This can be achieved through the demo predicate by specifying the virtual theory as T1+T2+... +Tn as in:

demo(T1+T2+T3, Goal).

The goal and all its subgoals are proved with respect to the virtual theory. It is a virtual theory since no new theory is created. The virtual theory is viewed as having all the clauses from all the theories in the order they appear (i.e. all clauses from the first theory, followed by all the clauses from the second theory, etc...). Recall that each theory is a first-class object in its own right regardless of how it was created. Therefore, if a procedure is shared by say two theories that make up the virtual theory (this

happens if one of the theories is an ancestor of the other), that procedure will appear twice in the virtual theory. This is a conscious design decision since we want to view, at the conceptual level, each theory as containing all the procedures that define it. The sharing of clauses associated with viewpoints inheritance is an implementation issue related to efficiency.

6. Nested Theories

MetaProlog theories form a tree hierarchy with basetheory as the root. Top-level theories are usually loaded directly from external MetaProlog files, while lower level theories are created using **addto** or **dropfrom**. This This would seem to mean that no external file can contain more than one theory since **consult(FileName, TheoryName)** is used to create one theory from one file. For a file to contain many theories, we need to specify the name of each theory and the boundaries between theories. The facilities provided for this purpose are **theoryname(ThName)** and **endtheory**. Many theories can now occupy the same file, and theories may be nested by specifying one theory inside another. However, the clauses of a theory have to be fully written down before the clauses of its descendant theories. That is, one cannot write some clauses from t1, then some clauses from its descendant t2, then some more clauses from t1. Also, files with explicit theories are loaded with the predicate **metaconsult(MyFile)** and not **consult(MyFile, TheoryName)**. Figure 2 shows how the previously defined theories t1, t2, and t3 can be specified in a single MetaProlog file called 'myfile'.

```
theoryname(t1).        % start theory t1

  p(1).
  p(2).

  theoryname(t2).      % start theory t2, within t1
    p(3).
  endtheory.           % end theory t2

  theoryname(t3).      % start theory t3, within t1
    -p(1).
  endtheory.           % end theory t3

endtheory.             % end theory t1
```

Fig. 2: Nested MetaProlog Theories

The '-' sign in front of p(1) in t3 is used to indicate that t3 is obtained from t1 by dropping the clause p(1). Note that all the clauses of t1 are written down before any clause from its descendant theories t2 and t3. To load this file, we type **metaconsult(myfile)**. The result is the same as if we created t1, then used addto and dropfrom to create t2 and t3 as in the previous example.

7. Proofs

The two or three place predicate **demo** is used to specify the goal to be proved as well as the theory in which the proof is to be carried out. The specified theory becomes temporarily the current theory or current context. The predicate **context(CT)** is used to check what the current context is. It binds the variable CT to an internal representation of the current theory. The current context can be either a regular theory or a virtual theory. Given the goal **demo(T2, Goal)** in theory T1, the context is set to T2 and the proof of **Goal** starts. If **Goal** succeeds, the context is reset back to T1. If **Goal** fails, the context is reset back to the theory which contains the goal we resume from after backtracking. All subgoals of **Goal** are evaluated in T2 except those that explicitly specify another context. If one of the subgoals is **demo(Ti, Gi)**, Gi is evaluated in the context of Ti, then the context switches back to T2.

8. Proofs as First Class Objects

The proof tree can be requested when MetaProlog is presented with a goal to solve. A three place **demo** predicate is used for this purpose:

```
demo(Theory, Goal, Proof).
```

A representation of the proof tree will be unified with the third argument. If this argument is partially or fully instantiated, it will serve as a control strategy guiding the inference system through the search space. The proof tree is represented as a list whose head is the root of the tree, and whose tail is a list of subtrees. The root represents a goal, and each subtree is the proof tree of a clause in the body of the goal. This representation should make it very easy to manipulate the proof tree. The following example illustrates the use of the three argument **demo**.

```
theoryname(t1).

p(X,Y) :- q(X), r(Y).

q(X) :- a(X).

r(X) :- q(X).

a(a).

endtheory.

?- demo(t1, p(X,Y), P).

X = a
Y = a
P = [p(a,a), [q(a), [a(a)]], [r(a), [q(a), [a(a)]]]]

yes.
?-
```

The following procedure can be used to display the proof tree using proper indentations to show the different subtrees.

```
show_head([], _).
show_head([H|B], N) :-
        tab(N),
        write(H), nl,
        N1 is N + 2,
        show_body(B, N1).

show_body([], _).
show_body([H|B], N) :-
        show_head(H, N),
        show_body(B, N).


?- demo(t1, p(X,Y), P), show_head(P, 0).
p(a,a)
```

```
  q(a)
    a(a)
  r(a)
    q(a)
      a(a)

X = a
Y = a
P = [p(a,a),[q(a),[a(a)]],[r(a),[q(a),[a(a)]]]]

yes.
?
```

Since the proof trees contain all the subgoals that participate in the evaluation of a goal, they can be used to justify the solutions reached. Explanation is a very important feature of expert systems. Since MetaProlog is suitable for expert systems, proof trees are a valuable feature to provide as an integral part of the system. Sure, one can always add the capability of building the proof tree on top of MetaProlog. But that makes the program wired to work in only one way or another. With proof trees as part of the underlying machinery, the same copy of the program can be run sometimes to get a result and a proof tree, and sometimes to get just the result. Take the example of a help system that takes questions from users and gives a brief answer. Once in a while, some user may ask why a certain conclusion has been reached. For this system, we would like all the queries to be run without the overhead of generating the proof tree (using the two place demo). Whenever there is a 'why' question, the query is rerun with the three place demo which generates the proof tree that can be used to justify the conclusion.

A pleasant surprise of the proof trees is their use as a control strategy to direct the search for a solution. If we know of certain nodes of the search space that are required or desirable intermediate goals, we can specify a proof tree that is partially instantiated with those nodes. This forces the system to choose these nodes and ignore their alternatives, resulting in an early pruning, in the search space, of some subtrees that would have lead either to failure or to correct but undesirable solutions. The example shown in Figure 3 illustrates how a partially instantiated proof tree is used to guide the search toward a more desirable outcome. In this case, we are interested in a flight from Syracuse to New Orleans such that the first stop is in Atlanta. By instantiating the root of the leftmost subtree to a direct flight from Syracuse to Atlanta (Figure 3b), we eliminate the flights where Miami or Orlando are first stops from consideration. Figure 3c shows the complete proof trees for the 2 possible solutions.

9. MetaProlog Syntax

In addition to supporting the full Prolog syntax, MetaProlog accepts its own special syntax. In particular, there is no special meaning attached to identifiers based on the fact that they start either with an upper case letter or a lower case letter. All MetaProlog clauses start with the word 'all' followed by a list of identifiers. These identifiers represent all the variables that may appear in the clause they precede. All variables must be explicitly quantified. The word 'all' together with the list and a following colon (':') constitute a logical universal quantifier. Any other identifier appearing in the clause is either a constant or a functor regardless whether it starts with an upper or lower case letter. Another particularity of the MetaProlog syntax is the replacement of the symbol ':-' by '<-', and the symbol ',' between goals by '&'. Some examples of MetaProlog clauses follow:

```
theory info:

flight(C1, C2) :- direct_flight(C1, C2).
flight(C1, C2) :- direct_flight(C1, Ci), flight(Ci, C2).

direct_flight(syracuse, miami).
direct_flight(syracuse, orlando).
direct_flight(syracuse, atlanta).
direct_flight(miami, atlanta).
direct_flight(miami, new_orleans).
direct_flight(orlando, new_orleans).
direct_flight(atlanta, new_orleans).
direct_flight(atlanta, orlando).
```

a) MetaProlog Program

```
demo(info, flight(syracuse,new_orleans), [S1, [direct_flight(syracuse, atlanta)] | S2]).
  Answer1:
  S1 = flight(syracuse,new_orleans)
  S2 = [flight(atlanta,new_orleans),[direct_flight(atlanta,new_orleans]];        Answer2:
  S1 = flight(syracuse,new_orleans)
  S2 = [flight(atlanta,new_orleans),
            [direct_flight(atlanta,orlando)],
            [flight(orlando,new_orleans),
                  [direct_flight(orlando,new_orleans)] ]];
```

b) Query and Answers

```
1) [flight(syracuse,new_orleans),
      [direct_flight(syracuse,atlanta)],
      [flight(atlanta,new_orleans),
            [direct_flight(atlanta,new_orleans)]]];

2) [flight(syracuse,new_orleans),
      [direct_flight(syracuse,atlanta)],
      [flight(atlanta,new_orleans),
            [direct_flight(atlanta,orlando)],
            [flight(orlando,new_orleans),
                  [direct_flight(orlando,new_orleans)]]]];
```

c) Complete Proof Trees

Figure 3: Proofs as a Control Strategy

```
all [var1, var2, VAR3] :
p(var1, Const1, Var3) <-
      Q(const2, var2) & R(CONST3, VAR3).

all [person1, person2] :
Ancestor(person1, John) <-
      Father(person2, John) & Ancestor(person1, person2).
```

In the first example, the only variables are var1, var2, VAR3. Notice that Q and R are both capital letters, yet they play the role of predicates. Also, Const1 and CONST3 both start with an upper case letter, yet they are constants just like const2.

The Prolog if-then-else construct, as in 'a -> b; c', may be replaced by the more explicit 'if a then b else c'. Since MetaProlog supports the full Prolog syntax, all MetaProlog clauses have to start with the word 'all' followed by a list of variables. If the clause is ground, an empty list should be used (all [] : p(A,B,C)). The prefix 'all [' indicates to the parser that a clause with a strict MetaProlog syntax follows. This allows a mixture of Prolog and MetaProlog syntax to be used in the same program. (In the future, MetaProlog may support its own syntax as the default one, and Prolog syntax as an option. At that point, no mixture of the two syntaxes will be allowed.)

10. Listing Clauses

Many forms of the **listing** predicate are supported to provide a wide range of selection of output. The output of listing is always relative to the current theory, i.e. what you see depends on the context you are in. A summary of the different forms of **listing** follows:

listing:
show all visible clauses in the current theory.

listing(Pred):
show all clauses in the current theory with predicate Pred and any arity.

listing(Pred/Arity):
show all clauses in the current theory with predicate Pred and arity Arity.

listing(Theory, Pred):
perform listing(Pred) in theory Theory. Equivalent to demo(Theory, listing(Pred)

listing(Theory, Pred/Arity):
perform listing(Pred/Arity) in theory Theory.

list(Theory):
listing(Theory, _):
same as **listing**, using Theory as the current context.

In all of the above, Theory is either a theory descriptor or a theory name.

11. Meta-level Reasoning

Some limited form of meta-level reasoning is supported by the MetaProlog system. Even though the inference mechanism follows a backward chaining regime, the system can be set up to pursue several lines of reasoning at the same time. Recall that the theories are organized in a tree hierarchy. The idea is to consider each branch of the tree as pursuing a certain line of reasoning. Starting from a theory T, we can add some assumptions creating a theory T_{i1}. After doing some work in T_{i1}, we can add some more assumptions creating T_{i2}. We can repeat the same process until we reach a certain theory T_{in}. The branch of the tree consisting of the theories from basetheory to T_{in} (basetheory,... T, T_{i1},... T_{in}) is referred to as our current line of reasoning. At any time, we can suspend work along this branch and start a different line of reasoning along another branch, say T, T_{j1},... T_{jn}, using a different set of assumptions. Of course, we can also decide to freeze everything along this new branch and either start a new line of reasoning, or go back to a previous one. Sometimes, we may want to abandon a line of reasoning altogether or just retract to a previous position. Four built-ins are provided to support these features: **assume(Belief)**, **reject(Belief)**, **suspend(State)**, and **resume(State)**. The first predicate states that we want to add a certain assumption to the current context, creating a new context. The new

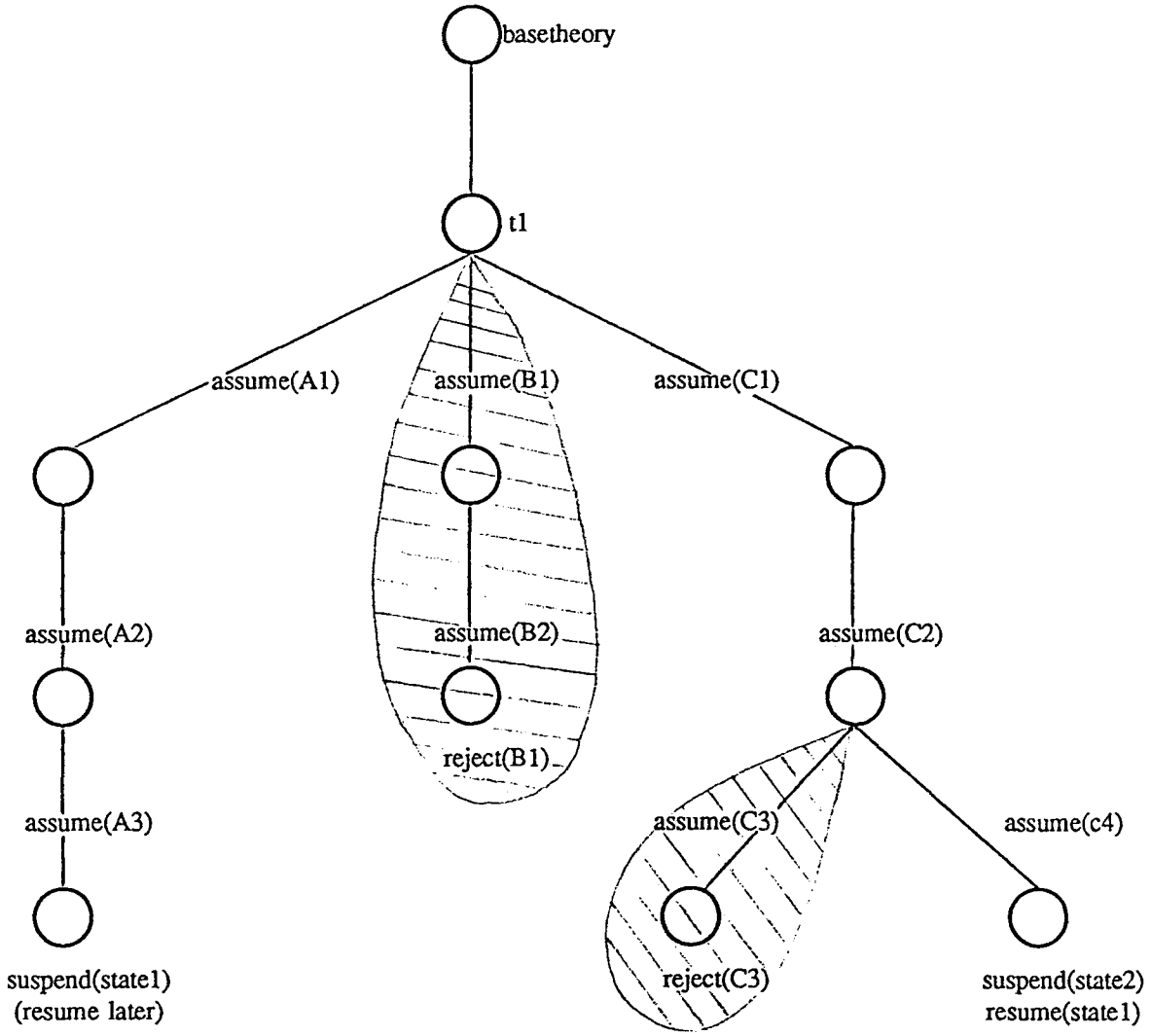

Figure 4: Pursuing Multiple Hypotheses at the Same Time in MetaProlog

context will be our current context. The second predicate states that we want to retract to a previous context, the one just before the specified assumption was added. Any work done between the moment the specified assumption was added and the current context will be forgotten. The third predicate assigns the name State (a ground term) to the current line of reasoning and instructs the system to leave everything as is until resumed later. The fourth and final predicate directs the system to go back to a previous line of reasoning suspended under the name State. Note that we can only reject assumptions made along the current line of reasoning. To reject any other assumption, we must first go back to the branch along which it was added using the built-in **resume**. The following example gives an idea on how these features may be used.

starting in theory t1
assume(A1),
do some work,
assume(A2),
do some more work,

```
assume(A3),
do some more work,
difficulties encountered,   % nowhere to go from here for the moment
suspend(state1),            % save what was done so far
setcontext(t1),             % back to where we started from (t1)...
assume(B1),                 % ...and try another line of reasoning
do some work,
assume(B2),
do some more work,
major difficulties,         % we want to reconsider
reject(B1),                 % abandon current line of reasoning (back to t1)
assume(C1),                 % start over
do some work,
assume(C2),
do some more work,
assume(C3),
do some more work,
some problems,
reject(C3),                 % back to the context just before last assumption
assume(C4),
do some more work,
nothing promising,
suspend(state2),
resume(state1),             % back to where we left in state1
  ...
```

The tree structure corresponding to this example is show in Figure 4. This example is not really a program, but rather a high level description of the different steps a program may go through. A top-level program could recursively add some assumptions, evaluate the results, then decide whether to continue, suspend, retract to a previous position, completely abandon a certain line of reasoning, or resume a previously suspended line of reasoning.

The features just outlined, combined with MetaProlog's support of multiple contexts and fast context switching, should provide excellent tools for writing expert system shells and other kinds of intelligent systems. Though the method of inference is a goal-driven backward chaining mechanism, breadth-first reasoning can still be simulated through the use of multiple lines of reasoning. Many hypotheses can be pursued at the same time, alternating between them through the use of the **suspend** and **resume** mechanisms[2]. Each hypothesis, in turn, is explored using backward chaining. A limited from of belief revision is provided through the use of **assume** and **reject**[3]. After an assumption has been made, we may find out that it is erroneous and decide to remove it and all inferences based on it. Unfortunately, all we can do is go back to a state just prior to when that assumption was made. All inferences made along the line of reasoning containing that assumption between the moment it was added and the context in which it was rejected become inaccessible. The other lines of reasoning are totally unaffected. A true belief system would remove only the inferences associated with the erroneous assumption.

[2]Alternatively, a top-level program could create and carry around a list of theories representing a breadth-first development. However, such an approach would not necessarily take advantage of the underlying MetaProlog machinery.

[3]This machinery is neutral with regard to the reasons for revision and the goals for this revision. It can be used to implement specific philosophies of belief revision.

12. Debugger

A standard four port debugger similar to that of regular Prolog is provided. For every goal displayed, it shows the context in which it is being evaluated. The context is shown as a theory descriptor and the name associated with the theory if it is a permanent one. Typing **trace** or **spy** causes the debugger to be loaded if not already in the system. The debugger is invoked with the command **trace(Goal)**. Spy points are set by typing **spy(Pred/Arity)** for predicates in the current theory, or **spy(TheoryName, Pred/Arity)** for predicates in the theory named TheoryName. The rest of the commands are exactly as in the standard Prolog debugger. Note that we are always back in the original context whether the goal succeeds or fails. In other words, if the **trace(goal)** command is given from theory t1, the context may change several times during the evaluation of Goal, but will always be set back to t1 when the goal terminates whether it succeeds or fails. However, if the trace is aborted the context may be set to any theory when the decision to abort is made, including a temporary theory. To avoid the problem of finding ourselves in a non-existent theory (one that was temporary), the context is reset to basetheory. As an example, let's go back to the theories in Figure 2 and add a new theory t4 as shown below, then trace the goal q(X). Watch carefully how the contexts change as we move back and forth between different theories.

```
?- addto(t3, [(q(X) :- demo(t2, p(X)), (X = 2;X = 3))], N), nameof(N, t4).

X = _6312
N = <theory 4>

yes.
?- trace(demo(t4, q(X))).
Reconsulting '/usr/accts/hamid/meta/dbg.pro'...

    (1) 1:    theory: <theory 0>  name: BASETHEORY
Call: demo(t4,q(_6449))
    (2) 2:    theory: <theory 4>  name: t4
Call: q(_6449)
    (3) 3:    theory: <theory 4>  name: t4
Call: demo(t2,p(_6449))
    (4) 4:    theory: <theory 2>  name: t2
Call: p(_6449)
    (4) 4:    theory: <theory 2>  name: t2
Exit: p(1)
    (3) 3:    theory: <theory 4>  name: t4
Exit: demo(t2,p(1))
    (5) 3:    theory: <theory 4>  name: t4
Call: 1=2
    (5) 3:    theory: <theory 4>  name: t4
Fail: 1=2
    (6) 3:    theory: <theory 4>  name: t4
Call: 1=3
    (6) 3:    theory: <theory 4>  name: t4
Fail: 1=3
    (4) 4:    theory: <theory 2>  name: t2
Redo: p(_6449)
    (4) 4:    theory: <theory 2>  name: t2
Exit: p(2)
    (3) 3:    theory: <theory 4>  name: t4
Exit: demo(t2,p(2))
    (7) 3:    theory: <theory 4>  name: t4
Call: 2=2
```

```
    (7) 3:    theory: <theory 4>  name: t4
Exit: 2=2
    (2) 2:    theory: <theory 4>  name: t4
Exit: q(2)
    (1) 1:    theory: <theory 0>  name: BASETHEORY
Exit: demo(t4,q(2))

X = 2;
    (7) 3:    theory: <theory 4>  name: t4
Fail: 2=2
    (8) 3:    theory: <theory 4>  name: t4
Call: 2=3
    (8) 3:    theory: <theory 4>  name: t4
Fail: 2=3
    (4) 4:    theory: <theory 2>  name: t2
Redo: p(_6449)
    (4) 4:    theory: <theory 2>  name: t2
Exit: p(3)
    (3) 3:    theory: <theory 4>  name: t4
Exit: demo(t2,p(3))
    (9) 3:    theory: <theory 4>  name: t4
Call: 3=2
    (9) 3:    theory: <theory 4>  name: t4
Fail: 3=2
    (10) 3:   theory: <theory 4>  name: t4
Call: 3=3
    (10) 3:   theory: <theory 4>  name: t4
Exit: 3=3
    (2) 2:    theory: <theory 4>  name: t4
Exit: q(3)
    (1) 1:    theory: <theory 0>  name: BASETHEORY
Exit: demo(t4,q(3))

X = 3;
    (10) 3:   theory: <theory 4>  name: t4
Fail: 3=3
    (4) 4:    theory: <theory 2>  name: t2
Fail: p(_6449)
    (3) 3:    theory: <theory 4>  name: t4
Fail: demo(t2,p(_6449))
    (2) 2:    theory: <theory 4>  name: t4
Fail: q(_6449)
    (1) 1:    theory: <theory 0>  name: BASETHEORY
Fail: demo(t4,q(_6449))

no.
?-
```

13. Miscellaneous

The source code to all the user defined predicates is available through some system predicates such as **listing**, and during debugging. To hide the source code of a clause, the head of the clause should be declared a system predicate using **setsyspred(Predicate/Arity)**. To reverse this effect, use **resetsyspred(Predicate/Arity)**.

Another useful feature is the **list_wamcode** predicate. It shows the actual WAM code generated by the compiler for the Prolog clauses. It is used as follows:

list_wamcode(Pred/Arity)

where Pred and Arity are the predicate name and its arity.

14. Summary of Commands

In addition to the standard Prolog built-ins, MetaProlog supports the following commands:

addto(Theory, Clauses, NewTheory):

NewTheory contains all the clauses of Theory plus those specified by Clauses. Theory is either a theory name or a theory descriptor. NewTheory must be a variable which will be unified with a theory descriptor. The second argument, Clauses, may be either a single clause, a file name specified as **file(filename)**, or a list of clauses and/or file names. Examples:

```
addto(T1, p(a), T2).
addto(T1, file(f1), T2).
addto(T1, [p(1), p(2), file(f1), q(a), file(f2)], T2).
```

assume(Belief):

Add the clause Belief to the current context creating a new context. The new context becomes the current context. If the assumption Belief is ever rejected, the old context becomes the current context again.

basetheory(B):

The variable B is set to the theory descriptor corresponding to **basetheory**.

createtheory(Th):

Create an empty theory as a descendant of basetheory. If Th is a variable, bind it to an internal representation of the new theory. If Th is a ground term, assign it as the name of the new theory.

consult(file1, TheoryName):

consult([file1, file2,... filen], TheoryName):

Create a theory from the clauses in file1 (file2, ... filen) and current theory and name it TheoryName. Equivalent to:

```
context(CT), addto(CT, file(file1), T), nameof(T, TheoryName).
    and
context(CT), addto(CT, [file(file1), file(file2), ... file(filen)], T), nameof(T, TheoryName).
```

context(T):

The variable T is set to the theory descriptor corresponding to the current theory (current context).

createtheory(T):

Create an empty theory and either bind T to its internal representation if T is a variable or assign the name T to it if T is a ground atom.

demo(Theory, Goal):

demo(T_1 + T_2 +... +T_n, Goal):

Prove Goal in the given theory or virtual theory. T_1, T_2,... , T_n may be either theory names or theory descriptors. Multiple theories T_1+T_2+... +T_n are considered as one virtual theory containing all the clauses from T_1 followed by all the clauses from T_2, etc.

demo(Theory, Goal, Proof):

Same as the two argument demo, except that a representation of the proof tree is unified with the third argument. The third argument may be partially or fully instantiated. In this case, it serve as a control strategy guiding the inference system through the proof tree.

dropfrom(Theory, Clauses, NewTheory):

NewTheory has all the clauses of Theory except for Clauses. Theory is either a theory name or a theory descriptor. NewTheory must be a variable. The second argument is either a single clause or a list of clauses. Built-ins provided in basetheory cannot be altered with dropfrom.

list(TheoryName):

List all clauses in the theory named TheoryName. Same as listing(TheoryName, _).

listing:

The predicate listing comes in a variety of flavors. In its 'vanilla' form, it lists all the clauses in the current theory. The listing can be restricted to clauses whose heads have a specific predicate (listing(Pred)), or a specific predicate and arity (listing(Pred/Arity)). Clauses in theories other than the current one can be listed by specifying the theory name as the first argument and the option as the second argument as in listing(TheoryName, Option). Option is either Pred or Pred/Arity as above. listing(TheoryName, _) lists all clauses in theory TheoryName and is equivalent to list(TheoryName).

metaconsult(file1):

Consult the MetaProlog file 'file1'. 'file1' is expected to have theories enclosed between the theory delimiters **theoryname(ThName)** and **endtheory**. This provides a very flexible way of specifying many theories in the same file, as well as specifying their hierarchy. example:

```
theoryname(t1).        % start theory t1
p(1).
p(2).

theoryname(t2).        % start theory t2, within t1
q(1).
q(2).
endtheory.             % end theory t2

endtheory.             % end theory t1
```

Theory t1 is a descendant of basetheory, while theory t2 is a descendant of t1.

nameof(Theory, TheoryName):

Assign to Theory the name TheoryName, or get the theory descriptor corresponding to TheoryName; e.g. if T is an uninstantiated variable and t1 has previously been assigned as the name of some theory, the goal **nameof(T, t1)** will cause T to be unified with the corresponding internal representation of t1.

reject(Belief):

Used in conjunction with a previously executed **assume(Belief)**. If the program discovers that the assumption Belief is erroneous, this goal resets the current context to the context just before the assumption Belief was made, and discards what was done since then along the current line of reasoning.

resetsyspred(Pred/Arity):

Reverse the effect of setsyspred(Pred/Arity).

resume(State):

Used to resume work along a previous line of reasoning suspended under the name State by

setting the current context to state.

setsyspred(Pred/Arity):
Turn the predicate defined by Pred/Arity into a system predicate, i.e. prevent the source code from being displayed.

setcontext(T):
Make T the current theory, i.e. switch the context to the new context represented by theory T. T may be a theory name or a theory descriptor.

suspend(State):
Used to freeze the current line of reasoning as is for possible resumption later on. The ground term State is applied as a name of the current context. Later use of resume(State) will get back to the current context.