A HIERARCHICAL DATABASE MODEL FOR A LOGIC PROGRAMMING LANGUAGE

Tim Finin
Jim McGuire

MS-CIS-88-22
LINC LAB 108

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

March 1988

Acknowledgements: This research was supported in part by DARPA grant NOOO14-85-K-0018, NSF grants MCS-8219196-CER, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

A Hierarchical Database Model for a Logic Programming Language

Tim Finin
Paoli Research Center
Unisys Corporation
PO Box 517, Paoli PA 19301
(tim@prc.unisys.com)
215-648-7446

Jim McGuire *
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
(mcguire@linc.cis.upenn.edu)

March 1988

Abstract

This paper presents an extended Clausal Database Model for a logic programming language. Instead of being restricted to one global database, as is the case with Prolog, we allow segmentation of the database into *database units* which are linked together into a semi-lattice. Each database unit defines a database view which includes clauses which have been asserted into that unit as well as clauses inherited from its ancestors higher in the lattice structure. This model supports arbitrary retraction. Retracting a clause in a database unit effectively blocks its inheritance for that unit and all of its descendants. Motivations for using this model are given. We also discuss the implementation of a Prolog meta-interpreter that uses this model. (hereafter referred to as (*Phd*) or Prolog Hierarchical Database) This meta-interpreter is in the spirit of Prolog and therefore has a version of assert, retract and cut.

Submission Information. Submitted to AAAI-88. *Topic* = Machine Architecture and Computer Languages. *Subtopic* = logic programming. *Length* = about 5000 words. We believe that we can meet the length requirements by reducing the the motivation and related work sections and generally tightening things up. Please send all correspondence to the first author (Finin).

Contents

1 Introduction 1

2 Motivation 2

3 A Simple Implementation 2

4 The *Phd* model 4
- 4.1 Creating the Database Hierarchy . . . 4
- 4.2 Assert and Retract Mechanisms 5
- 4.3 Representation Details 6
- 4.4 Inheritance 7
- 4.5 *Phd* Meta-Predicates 7

5 Related Work 8

6 Conclusions 9

*This work was supported by a grant from ARO.

1 Introduction

Most logic programming languages have followed Prolog in having a single, global database in which both the program and data reside. The database is the only mutable data structure and also the only one which persists from one query to another. This has provided Prolog with a number of advantages. Conceptually, this is a very simple scheme which new users find easy to learn and think about. Pragmatically, it eases the prototyping of new programs, since one does not need to worry about issues of which programs need to access what other programs and data. Theoretically it corresponds to thinking of the database as consisting of the set of axioms on which the system rests. After all, a fact is either true (in which case it should be in the database) or it is not (in which case it is not in the database) – there is no middle ground.

Contrast this with the more typical ways of storing information in modern programming systems. Most higher-level programming languages provide various mechanisms to store and represent data in ways that control the access to it. This shows up in programming languages in a variety of ways, including the use of local variables, modules, packages, environments, etc.

This paper describes a design for a hierarchical database for a logic programming language. In particular, we have implemented this model for Prolog in the form of a meta-interpreter (*Phd*). In our model, the database is segmented into data base units each of which is a local collection of Prolog clauses. These database units are partially ordered by a *parent* relation along which clauses can be "inherited". Thus, each database unit defines a Prolog *database view*. The clauses in such a view consists of the union of the clauses in the local database unit and the inheritable clauses from the database unit's ancestors.

One important feature of our model is that it supports a notion of *relative retraction*. That is, one can retract a clause with respect to a particular database view. This can be done for a clause which is "local" to the view (i.e. is recorded in the unit which determines the view) as well as for a clause inherited from some ancestor. In either case, the retraction can only effect the database view in which the retraction was done and (potentially) its descendants.

A more precise definition for a database view as seen from database unit u is that it contains clauses equal to the set of local assertions in u unioned with all clauses in the views of u's immediate parents minus all of u's local retractions. There are several things to note: (i) retracting a clause from a database unit never effects any of the unit's ancestors or the database views they define; (ii) it is not possible for a clause to appear more than once in a database view. When solving a goal with respect to a DBU, only the clauses in the view of that DBU will be used to prove the goal. The aim of this paper is to discuss the assumptions made in our implementation of a Prolog meta-interpreter using this model.

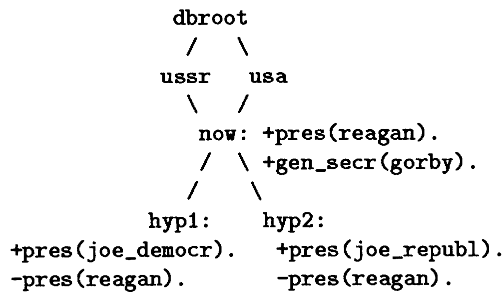

Figure 1: Simple Example

An Example

As an example, consider a system which must do hypothetical reasoning about American-Soviet relations (see figure 1). We would like to keep related data together. We have a DBU called *now* which describes the current state of affairs. We also have bodies of knowledge on Soviet and American political situations in DBUs *ussr* and *usa* respectively. The database view associated with DBU *now* includes the info stored in *ussr* and *usa*. Clauses inherit down. Now say we wish to consider the world after Reagan. We can spawn two new DBU's representing a hypothetical state of affairs where Reagan is not president. To prevent inheritance of a clause, we use a relative retraction. The relative retraction *-pres(reagan)* makes *+pres(reagan)* invisible to the database views of *hyp1* and *hyp2*.

The next section of this paper will present some motivations for developing a hierarchical model for a logic programming language database. The third section will define a slightly simplified version of the model through several Prolog predicates. The fourth section will then present the model in more detail and described its implementation in *Phd* . The fifth section will discuss the relationship between our work and several related projects. We conclude with a final section which summarizes our research and describes several open questions.

2 Motivation

There are a number of potential advantages to having a hierarchical database in a logic programming language as well as, of course, some disadvantages. An initial observation we can make is that the issues can be divided into those at the "programming language" level and those at the "representational" level. By "programming language" level, we mean issues that involve the features and limitations of a language like Prolog that effect its utility as a general higher-level programming language (e.g., the addition of *modules* to Prolog). There are also motivating issues which have more to do with making a logic programming language easier to use to represent certain kinds of information or knowledge (e.g., to support hypothetical reasoning). This section will briefly list some of the motivating factors that have caused us to begin this research.

Some programming language related motivations for a hierarchical database are:

- *Modules.* Our model for a hierarchical database is one way to provide a logic programming language with a *module* facility. It differs from most current module/package systems in that it is non-flat, predicates can be split across database units and it supports the notion of (relative) retraction. These differences offer additional flexibility at the expense of efficiency.
- *Theories as first-class objects.* Treating *theories* (e.g. collections of clauses) as first class objects is a powerful way to extend the richness and expressive power of a logic programming language [8].
- *Access efficiency.* Dividing the database into units arranged in a partial order can reduce the time searching the database for clauses. In our implementation this reduction comes through not having to search any database units not in the current view. Whether or not there is a realization of this potential depends on the particular application.
- *Debugging ease.* A hierarchical database can ease the process of debugging programs which alter the database. The general technique is to create a new database view which includes the current one and run the (buggy) program with respect to it, isolating any database updates to the new view.

The development of a hierarchical database model for a logic programming language has some advantages in representing knowledge for specific applications. Some examples are:

- *Generic vs. specific information.* Many AI applications such as diagnostic expert systems need to represent a knowledge base of general information (e.g. diagnostic rules) and are then applied to one or more specific *cases.* Processing each case might involve making additional assertions specific to just that case. The general knowledge needs to be included in the model for each individual case. The hierarchical database provides a language-level mechanism to keep such different information separate and have it share common knowledge.
- *Hypothetical reasoning.* A hierarchical database is a useful tool in applications involving *hypothetical reasoning* and *assumption-based reasoning.* Creating a new theory of a situation which contains a set of assumptions simply involves creating a new descendant database view of the current one and asserting the assumptions in the new view. The new view can then be used to attempt to prove any relevant queries.
- *Belief modeling.* Modeling the beliefs of other agents often involves a hierarchy of clauses representing the beliefs. For example, the GUMS system [10, 9] modeled users via a hierarchy of stereotypes, each of which was represented as a collection of clauses. A similar use of a hierarchy of user models is employed by Kass [15, 14] to capture beliefs of users' along different dimensions.

3 A Simple Implementation

The power of our model lies in the ability to supply different views of the data while incorporating *relative retractions.* The specific ordering of the clauses in the database view of some DBU is determined by some inheritance scheme. In this section, an elegant (although inefficient) implementation of a depth-first inheritance using the notion of *relative retractions* is presented. In the next section, we will describe our meta-interpreter which implements a more efficient inheritance scheme.

Constructing the database view from database unit u_1 can be conceptualized as a a set operation. It is a two step recursive process. First we must create the set of all local assertions in DBU u_1 unioned with all clauses in the *views* of u_1's immediate parents. From this set we must remove all clauses which *match* u_1's local retractions. The resulting set is the

```
         d1: +l(j,m)
           / \
+l(a,b) b2  c2: +l(b,c)
           \ /
         d3: -l(a,b)
             +l(t,j)

parent(d1,b2)
parent(d1,c2)
parent(b2,d3)
parent(c2,d3)

db(b2,+,l(a,b))
db(d1,+,l(j,m))
db(c2,+,l(b,c))
db(d3,-,l(a,b))
db(d3,+,l(t,j))
```

Figure 2: Sample Hierarchy with internal representation

view of u_1. The *match* operation is like unification except that a variable can only unify with another variable. Retractions are stored as negative assertions. As an example, consider the hierarchy in Figure 2 in which the depth-first inheritance from DBU d3 causes the DBU's to be searched in the following order: (d3,b2,d1,c2). The database view associated with d3 contains the clauses *l(t,j)*, *l(j,m)* and *l(b,c)*.

To present this model, we show the basic predicates for adding a clause to a database (add/2), retracting a clause from a database (rem/2) and querying a database for a clause (get/2). The hierarchical database is represented by two predicates: db/3 and parent/2. The db/3 predicate records the clauses which have been locally asserted and retracted in database units. In particular we have:

- *db(Dbu,+,C)* - true if the clause *C* has been locally asserted in the database unit *Dbu*
- *db(Dbu,-,C)* - true if the clause *C* has been locally retracted in the database unit *Dbu*

The parent/2 predicate records the lattice organization of the database units with *parent(A,B)* indicating that database unit B inherits from database unit A.

The *add/2* predicate takes two arguments, a database unit and clause to be asserted. It asserts the given clause into the database view determined by the given unit (and its descendants). Unlike standard Prolog, only one copy of a clause is recorded in a given unit.

```
add(Db,C) :-
  % already asserted locally.
  localDb(Db,+,C,C2).

add(Db,C) :-
  % already retracted locally.
  localDb(Db,-,C,C2),
  retract(db(Db,-,C2)),
  assert(db(Db,+,C)).

add(Db,C) :-
  % no local record.
  assert(db(Db,+,C)).
```

The *localDb/4* predicate is used to find clauses which match the current one which have been explicitly asserted or retracted in this database unit. Note the use of the *match/2* predicate which is like unification except that variables can only unify with other variables and no binding is done.

```
localDb(Dbu,Mode,Cin,Cout) :-
  db(Dbu,Mode,Cout),
  match(Cin,Cout).

match(X,Y) :-
  numbervars(X,X2,1,N),
  numbervars(Y,X2,1,N).
```

The *rem/2* predicate is used to retract clauses from a database associated with a database unit. As in standard Prolog, it retracts the first clause in the database which unifies with its argument. However, the retraction is only in the database determined by the given database unit and its descendants.

```
rem(Db,C) :-
  % recorded as a local clause.
  localDb(Db,+,C,C2),
  !,
  retract(db(Db,+,C2)),
  assert(db(Db,-,C)).

rem(Db,C) :-
  % unifies with an inherited clause.
  get(Db,C),
  assert(db(Db,-,C)).

rem(Db,C) :-
  % Record un-unified retraction
  assert(db(Db,-,C)).
```

Finally, the *get/2* predicate takes a database unit and clause and succeeds if there is a matching clause in the database determined by the database unit. This is determined be seeing if the clause sought is a member of the set formed from the union of all of the inherited and locally asserted clauses minus the locally retracted clauses. Note that the equality test

used in the *setUnion/3* and *setMinus/3* predicates must be the *match/2* predicate of above.

```
get(Db,C) :-  clauses(Db,Cs), member(C,Cs).

clauses(DB,Clauses) :-
  % Clauses is a list of clauses in view DB
  parentClauses(DB,Pc),
  localClauses(DB,Lc),
  localRetractions(DB,Lr),
  setUnion(Lc,Pc,C),
  setMinus(C,Lr,Clauses).

parentClauses(Db,Cs) :-
  % Cs is a list of clauses inherited by Db.
  bagof(C,
        (parent(P,Db),clauses(P,C)),
        ListOfCs),
  setsUnion(ListOfCs,Cs).

%% Cs is a list of locals clauses in Db
localClauses(Db,Cs) :-
  bagof(C,db(Db,+,C),Cs).

%% Rs is a list of locals retractions in Db
localRetractions(Db,Rs) :-
  bagof(C,db(Db,-,C),Rs).
```

4 The *Phd* model

This section describes a much more efficient model for the hierarchical database which we have implemented in the *Phd* meta-interpreter. This model differs from the simple one presented in the previous section in the order in which clauses appear in a database view.

4.1 Creating the Database Hierarchy

There is initially a distinguished database unit, *dbroot* which is the ancestor of every other database unit. This is where all the system clauses and special utilities reside. Each DBU is assigned an integer representing its "level" in the hierarchy with the requirement that a DBU's level must be strictly less than the level of it's immediate descendants. By convention, the level assigned to *dbroot* is 0. When a DBU is created, it is assigned as a level one plus the maximum level of its parents. All parent, child, and level information is stored in the underlying Prolog database. Initially, the newly created DBU is assigned level 1, since it's only parent is *dbroot*.

Phd provides four predicates for constructing and modifying the hierarchy:

- create(DBU)
- adopt(Parent,Child)

```
phd> create(a),create(c),create(b),
     create(d),create(e),
     adopt(a,c),adopt(c,e),
     adopt(a,b),adopt(b,d),
     adopt(d,e).
```

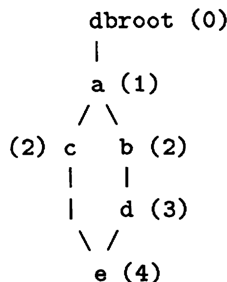

Figure 3: Initial hierarchy with levels indicated

- disown(Parent,Child)
- kill(DBU)

The create/1 predicate creates a new database unit which is implicitly a child of *dbroot*. The adopt/2 predicate adds a new inheritance link between its two arguments. There are restrictions on the use of adopt — a descendant DBU cannot adopt an existing ancestor. Thus cycles are disallowed. To remove a link, call disown(Parent,Child). A DBU is not allowed to disown itself from *dbroot*. A database unit can be purged from the database with a call to kill/1. This will wipe out all clauses local to DBU and perform the necessary disowns from the DBU's parents. A DBU cannot be killed if it has offspring DBU's. When an adopt or disown is performed, the child DBU may require a change in its level. Remember the level of a DBU is strictly one plus the max level of the DBU's parents. This in turn may require the need for recomputation of the levels for the child DBU's descendants.

Figures 3 and 4 show the effects of adopts and disowns on the level assigned to a database unit. The level is listed in parentheses next to the DBU. Remember, dbroot is the only database unit that exists initially. Everybody has dbroot as a direct parent, although all the links are not portrayed here.

In logic programming languages such as Prolog the order of clauses in the database is significant. This is one way the programmer has of controlling the search for a solution. Thus we felt that it was important to have a well defined model which determines the ordering of the local and inherited clauses in a database view. Our choice was that you inherit from your closest ancestors first. This is why we associate a level

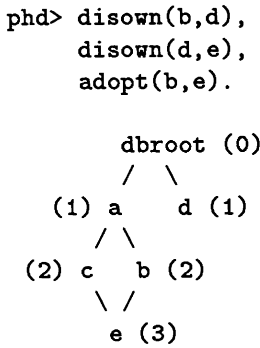

Figure 4: Altered hierarchy and levels

with each DBU. You inherit from an ancestor DBU with a larger level before another ancestor with a smaller level. With multiple parents, there may be more than one lineage to the same ancestor DBU. This approach prevents you from searching ancestor u_1 before ancestor u_2 if u_1 is itself an ancestor of u_2. What about ancestor DBU's located at the same level? We arbitrarily chose the ordering to be right to left. (i.e. most recently defined ancestor at that level). So for example, when proving a goal with respect to DBU *e* in Figure 3 , the DBU's would be searched in the following order: (e,d,b,c,a,dbroot).

An inheritance order is precomputed for each DBU and must be recomputed for affected DBU's on an adopt or disown. Adopts and disowns create additional subtleties with the assert/retract mechanism which will be discussed later.

4.2 Assert and Retract Mechanisms

The clauses visible in any database view depend upon the assertions and relative retractions found in the view. For convenience we supply an equivalent for Prolog's *listing* predicate:

- *visible(u).* Perform a listing of all clauses in the database view of *u*.

The addition and removal of clauses from DBU's is performed by

- *assert(Clause,Dbu).* The semantics of assert(C,U) is, "Clause C is known to be present in database unit U"
- *retract(Clause,Dbu).* The semantics of retract(C,U) is, "Clause C is known to be absent from the view of database unit U."

Assuming no retracts have been performed, assert(C,U) immediately makes C visible in U and in the views of DBU's which are descendants of U. Clauses inherits down. If there exists a retraction in U which *matches* the new assertion, physically remove the retraction from U. Remember, matching is different than unifying.

In Prolog, retracting a clause means "physically removing" the first unifying clause found in the database. Similarly, in our model, retraction involves first finding a unifying assertion in the view and second updating the database hierarchy to make this assertion invisible in the view. If the unifying clause is local to (stored in) U, physically remove that clause from U. Note, we do not physically remove the unifying clause from the database if it is not local to U but explicitly store the unified retraction in U.

For example, consider the initial database hierarchy in Figure 5 (note - the inheritance order associated with *e* is (e,b,c,a,dbroot). After the following exchange in which the current database is *e*,

```
phd> retract(l(X,Y),e).
X=j
Y=m;

X=a
Y=b

yes
phd> visible(e).
+ l(h,X) :- g(X).
+ l(r,t).
+ g(p).
```

the altered hierarchy in Figure 6 results. We find the first candidate for retraction within DBU e. Since it is local, we physically remove +l(j,m) and record the unified retraction. In the second retract, the assertion unifying with l(X,Y) is inherited, so we do not physically remove l(a,b) from DBU b. With respect to DBU e, + l(a,b) is known to be absent. This is demonstrated by the call to visible.

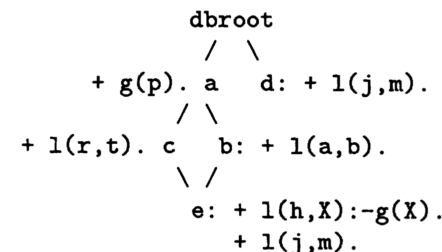

Figure 5: Initial hierarchy

After retract(Cl,U) has been called and a unifying retraction found, our meta-interpreter tags ev-

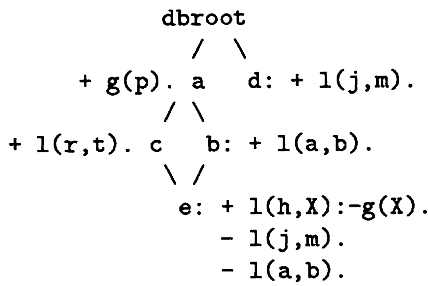

Figure 6: After Retractions

ery clause in the ancestor DBU's of U which *matches* the unified retraction. The tagging takes the form of creating pointers from the retraction to all ancestor assertions which match the retraction. Also, all the affected assertions contain a pointer back to the retraction(s) which makes them invisible to a portion(s) of the database.

This means that a newly asserted clause may be already logically retracted by a retraction in some descendant DBU. Thus in our implementation, every assertion is followed by a search for descendant retractions which *match* the new assertion. This way the retract pointers can be kept up to date. If an assertion results in the physical removal of a matching retraction in the same DBU, we can use the retract pointers to unretract matching ancestor assertions. We must also consider the effect of calling adopt(u_1,u_2). Assertions located in u_1 and ancestor DBU's of u_1 may now be logically retracted by retractions in u_2 and descendants of u_2. Figure 7 shows the changes which occur to the hierarchy in Figure 6 after the following exchange:

```
phd> adopt(d,a).
yes.

phd> visible(e).
+ l(h,X) :- g(X).
+ l(r,t).
+ g(p).
```

There is no change to the view associated with DBU e because the relative retraction -l(j,m) immediately makes + l(j,m) in DBU d invisible.

Our version of retract has more of a declarative reading than in Prolog. The time in which a retract is done does not matter. Retracts will immediately hide new assertions in ancestor DBU's which *match* them. Likewise, asserts will always undo relative retractions local to the DBU where the assertion is taking place.

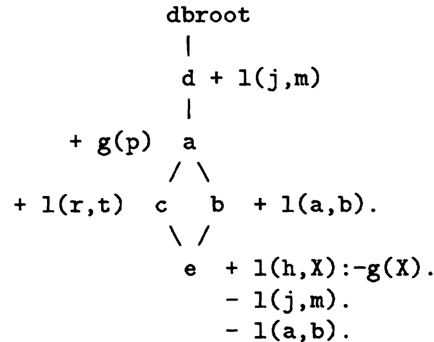

Figure 7: Adoption and existing retractions

The design decision was to make assertions, retractions, adopts, and disowns expensive to optimize goal satisfaction with respect to a specific DBU. Later we will see how these retract pointers are used to weed out invisible clauses.

4.3 Representation Details

Since our meta-interpreter is implemented in Prolog, we must represent all the separate DBU's within Prolog's global database. This section is a description of clauses used only by the system to simulate our model. These clauses are kept hidden from the user of the extended model. Information on each DBU is stored within the database as:

```
db_struc(Level,DBU,Parents,Children)
```

All clauses within a DBU are stored in the underlying Prolog database in a term with a functor corresponding to the DBU's name. For example, the arbitrary clause (Head:-Body) local to dbu1 would be represented as follows:

```
dbu1(Id,Head,Body,'+',Prev,Next,Retract)
```

Each clause has a unique Id within its DBU. All clauses of a certain functor and arity within a dbu are stored as a doubly-linked list. The fourth argument is a "-" for a relative retraction and a "+" for a (local) assertion. The `Retract` argument is a list of [DBU,Id] pointers to relative retractions in other database units which retract this clause. The `Prev` and `Next` arguments are respectively pointers to the previous and next clause in the list. We need to keep clauses in doubly linked lists to maintain their correct ordering. Whenever a clause is made invisible by a retraction, the Retract list must be updated. Asserta and assertz are not adequate to maintain the proper ordering of clauses. A doubly-linked list does

```
db_struc(0,dbroot,[],[e,d,c,b,a])
db_struc(1,d,[dbroot],[a])
db_struc(2,a,[d,dbroot],[b,c])
db_struc(3,b,[a,dbroot],[e])
db_struc(3,c,[a,dbroot],[e])
db_struc(4,e,[b,c,dbroot],[])

a(info,g,1,1,1).
b(info,l,2,4,4).
c(info,l,2,3,3).
d(info,l,2,2,2).
e(info,l,2,5,5).

a(1,g(p),true,+,nil,nil,'[]').

b(4,l(a,b),true,+,nil,nil,[[e,8]]).

c(3,l(r,t),true,+,nil,nil,'[]').

d(2,l(j,m),true,+,nil,nil,[[e,7]]).

e(5,l(h,_16),g(_16),+,nil,nil,'[]').
e(8,l(a,b),true,-,_7,_8,[[b,4]]).
e(7,l(j,m),true,-,_7,_8,[[d,2]]).
```

Figure 8: Internal representation

not have to be maintained for retractions, because the order in which they are done does not matter. When proving a subgoal, we want to search the doubly linked list corresponding to functor and arity of the subgoal. For a DBU named dbul, we have system clauses of the following form to isolate on the appropriate linked list:

```
dbu1(info,Functor,Arity,First,Last)
```

The arguments **First** and **Last** are the id's of the first and last clauses in DBU u_1 of a specific **Functor** and **Arity**. Figure 8 shows the internal Prolog representation in *Phd* of the hierarchy in figure 7.

4.4 Inheritance

Assume we wish to prove a goal against DBU u_1. First, exhaust all local clauses in u_1. Then try to inherit clauses. At all times you carry around a list of all DBU's which you *have already* searched. Retrieve the inheritance order (discussed in the hierarchy section) associated with u_1. Examine each DBU in the search queue in turn. When you have exhausted a DBU, search the next DBU in the search queue. Continue this process until you find a unifying visible clause or fail to do so. Eventually you will reach *dbroot*. This is the equivalent of falling through to Prolog.

Invisible clauses will not be searched. Every assertion has a pointer to the retraction which retracted it. The pointer indicates which DBU the retraction is local to. Therefore, if we encounter a candidate clause whose head unifies with the current SubGoal, we will just check if it was retracted by a retraction in one of the DBU's we've searched. Remember that we carry around the list of already searched DBU's.

Incidentally, our meta-interpreter does implement the cut operation. Since there is an ordering to the clauses in a view, the cut operation has the same semantics as in Prolog. It would be interesting to explore whether two types of cut were useful. One would be a local cut which prevented backtracking to clauses within the same DBU, but allowed inheritance from other DBU's. The other would be a global cut similar to Prolog's cut.

4.5 *Phd* Meta-Predicates

We have implemented the basics of our hierarchical database model in the form of a meta-interpreter *Phd*This system maintains the notion of one database view as the *current database view* against which goals are attempted to be proven. Most of the usual Prolog predicates which interact with the database have been extended to take an additional, optional argument which specifies the database unit (and therefore the view) that the predicates is to be run against. If a database view is not specified explicitly, then the "current database" is assumed. For completeness, here are the additional system predicates:

- *localclause(P,Q,u_1)*. (P:-Q) is present in u_1.
- *clause(P,Q,u_1)*. (P:-Q) is visible in the database view of u_1.
- *consult(File,u_1)*. Consult from a file to a specific database unit.

Some predicates used for switching database views are:

- *demo(Goal,u_1)*. Attempt to prove Goal using the view of u_1.
- *cdb(X)*. True if X is the current database unit.
- *makecdb(X)*. Make X the current database unit.

Much of the power of our model comes from very procedural operations (e.g. adopt, kill, assert, retract) and through the explicit use of assert and retract to modify the database. It is our intention to provide a language with such side-effecting operations to allow programmers to easily and efficiently build useful meta-level predicates to regain a declarative

reading to programs. As an example, consider the ability to to ask hypothetical questions. Assuming clause P in the presense of the current database view, is Q true?

```
assuming(P,Q) :-
   newContext(C),
   assert(P),
   Q.

newContext(C) :-
   create(NewDb),
   cdb(CurrentDb),
   adopt(CurrentDb,NewDb),
   makecdb(NewDb).
```

5 Related Work

The idea of a hierarchical database of clause-like objects can be found in some of the earliest programming languages for AI applications. Both Conniver [18] and QA4 [7], for example, had hierarchical databases. More recent work on enriching the model of the clausal database can be divided into three groups: the encorporation of (flat) modules and packages, combining the object-oriented and logic programming paradigms, and the extension of the usual clausal database model to (non-flat) hierarchies. Our work falls into the last category. We will briefly mention the first two approaches and then describe in more detail some of the other work in extending the database model to hierarchies.

In the last few years, the encorporation of modules into logic programming languages has been studied at both the theoretical[17, 12]) and practical [3, 4, 1] level. Most of the module systems we have studied differ from our approach in that (1) the modules are "flat" (2) predicates must usually be defined entirely in one module and (3) there is no support for relative retraction.

The past few years have seen a number of proposals for languages which combine the logic programming and object-oriented programming paradigms [20]. Some examples which include some kind of hierarchical database-like facility include KEE [2], BiggerTalk [13] and LOGIN [5].

Our work falls into a category in which a logic programming language has a hierarchical database in which units inherit clauses, in some fashion, from their ancestors. Examples in this category include Bowen's meta-prolog system, an efficient implementation of it designed by Bacha [6], the Horne theorem proving system [11] and work by McDermott [16].

In [8], Bowen adopts a meta-level approach to provide a logical semantics to assert and retract. His equivalent operations are add-to and drop-from. Related sets of clauses, grouped together in lists, are called theories. By executing **add-to**(t_1,C,t_2), the clause C is effectively cons'ed to theory t_1 to create the new theory t_2. The *demo/3* predicate is used to solve a goal with respect to a given theory yielding a particular proof tree. A constant name is associated with a theory which can be used to access the newest version of the theory by using the **name-of(Name**,t_1**)** predicate. Remember, the original theory (implemented as a list of clauses) will dynamically change through successive use of add-to and drop-from.

Bowen shows that an inheritance scheme between the theories can be implemented in a frame-like fashion. Inorder for Theory2 to inherit the clauses in Theory1, the clause
is-a(Theory2-name,Theory1-name) must exist in Theory2. Say we call demo(Theory2,Goal,P). We have the capability to access Theory1. Simply call name-of(Theory1-name,Theory1). Now we can call demo(Theory1,Goal,P), and the inheritance is achieved. The point of his paper was to explore the power inherent in the meta-level approach, rather than describe a specific implementation.

Whenever an add-to(t_1,C,t_2) or drop-from(t_1,C,t_2) is called, a new theory is created. This new theory is conceptually a slightly modified copy of the original theory. Bacha [6] presents an efficient representation for theories in Bowen's model. A copy of the theory is not actually made. Instead, theory t_2 inherits all clauses in t_1 except those of the same functor and arity as clause C. This way, t_2 need only contain an effective copy of the modified collection of clauses sharing the same functor and arity of C. Most importantly, he discusses how to implement his work in the context of the Warren Abstract Machine [19]. It appears that the add-to and drop-from operations prevent a theory from having multiple parents, since a new theory is alway a modification of one existing theory.

Another interesting method is described by [16] and which is a descendant of the scheme used in Conniver [18]. Our notion of a data base unit is equivalent to McDermott's notion of data pools. There is also a view associated with each data pool, since clauses in ancestor data pools inherit down. Associated with each clause in the database is a formula indicating which data pool views the clause is in and which it is known to be absent (by virtue of a retraction). Solving a goal with respect to a data pool involves retrieving clauses in the database and checking their membership formulae to see if they are really in the pool. One problem is that the membership formulas can get quite large. In addition, you must sort

through all the clauses unifying with the goal to determine if they are in the current data pool's view.

6 Conclusions

This paper has described a design for a hierarchical database for a logic programming language and its implementation in a meta-interpreter *Phd* . In this model, the clausal database is segmented into database units each of which is a local collection of clauses. Each database unit defined a database view which consists of the union of the clauses in the local database unit and those clauses in the parent database units which have not been explicitly retracted in this view. An important feature of our model is that it supports a notion of *relative retraction.* That is, one can retract a clause with respect to a particular database view. This can be done for a clause which is "local" to the view (i.e. is recorded in the unit which determines the view) as well as for a clause inherited from some ancestor. In either case, the retraction can only effect the database view in which the retraction was done and (potentially) descendant views (by preventing inheritance of the retracted clause).

There are several ways in which our model differs from previous hierarchical models for a logic programming language. First, our model allows the database to have a lattice structure rather than just a tree structure (as in Conniver, Horne and meta-Prolog). This greatly increases the expressivenes of the language. Secondly, our model does not involve the copying of any portions of the database, unlike Meta-Prolog. Third, it supports the notion of "relative retraction" which is essential for a number of applications such as hypothetical reasoning and default reasoning. This notion is not supported in Biggertalk and Login. Finally, our model has an "efficiency profile" which makes it much more attractive for some applications (the ones we are interested in, in particular!).

We leave for future work a number of important and interesting questions having to do with compilation, the structure of a hierarchical database and efficiency.

Compilation. We have looked briefly at the issue of compiling a logic programming language using our extended model of the clausal database into instructions for a modified version of the Warren abstract machine. In particular, we would like to discover what changes would be required to the indexing instructions (i.e. try-me-else, switch-on-term, etc). We believe that this modification may not be straightforward. One approach is to produce a static WAM encoding for each unique database view. Unfortunately, much of the power of our model comes from the ability to make dynamic changes to the structure of the hierarchy and to the contents of the DBU's which comprise the hierarchy. It is not reasonable to assume that we can recompile affected database views upon an adopt or a disown.

Our initial approach is to require that all predicate names that can be asserted or retracted must be indicated beforehand. We will then maintain two representations of the database. One would be an interpretive representation in the spirit of *Phd* into which dynamic predicates would be stored. In the second representation, the static predicates of each DBU would be compiled into a WAM encoding. Dynamically created DBU's could only contain dynamic predicates, so they would not have to be compiled. The dynamic (interpretive) and static (WAM) representations would cooperate to solve goals together.

Database structure. Do we really need the lattice capability? Trees certainly would be much more efficient. We would not have to maintain a dynamically changing inheritance order based on the notion of levels. Also, we might be able to apply some of the results of Bacha in [6]. We would like to do a careful analysis of the efficiency gains to be made by restricting the database hierarchy to be a tree.

Efficiency. There is a strong need for some empirical tests to study the efficiency of the three general approaches to implementing a hierarchical database in a language like Prolog.

References

[1] *Arity Prolog Reference Manual.* 1986.

[2] *KEEworlds Reference Manual.* Intellicorp, level 3.0 edition, 1986.

[3] *Quintus Prolog Reference Manual.* Quintus Computer Systems, Inc, Mountain View, CA, version 2.0 edition, 1987.

[4] *ZYX Prolog Reference Manual.* 1987.

[5] Hassan Ait-Kaci and Roger Nasr. LOGIN: a logic programming language with built-in inheritance. *Journal of Logic Programming,* 3:185–215, 1986.

[6] Hamid Bacha. Meta-level programming: a compiled approach. *Proc. of the 4th Int. Conf. on Logic Programming,* 1987.

[7] Danny Bobrow and Bertram Raphael. New programming languages for artificial intelligence research. *Computing Surveys,* 6(3), 1974.

[8] Kenneth A Bowen. Meta-leval programming and knowledge representation. *New Generation Computing*, 359–383, 1985.

[9] Tim Finin. GUMS—a general user modelling shell. In Alfred Kobsa and Wolfgang Wahlster, editors, *User Models in Dialog Systems*, Springer Verlag, Berlin—New York, 1988.

[10] Tim Finin and David Drager. $GUMS_1$: a general user modelling system. In *Proceedings of the 1986 Conference of the Canadian Society for Computational Studies of Intelligence*, pages 24–30, 1986.

[11] Allen Frisch and James Allen? *The Horne Reference manual?* Technical Report, 1987?

[12] Joseph Goguen and Jose Meseguer. *Functional and Logic Programming*, chapter Equality, Types and Generic Modules for Logic Programming, pages 295–363. Prentice-Hall, 1986.

[13] Eric Gullichsen. *BiggerTalk: Object-Oriented Prolog.* Technical Report MCC Technical Report Number STP-125-85, MCC, December 1985.

[14] Robert Kass and Tim Finin. A general user modelling facility. In *Proceedings of CHI'88*, 1988.

[15] Robert Kass and Tim Finin. Modelling the user in natural language systems. *Computational Linguistics*, Special Issue on User Modelling, 1988.

[16] Drew McDermott. Contexts and data dependancies: a synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):237–246, May 1983.

[17] Dale Miller. A theory of modules for logic programming. In *Proceedings of the IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986.

[18] Gerald Sussman and Drew McDermott. From planning to conniving: a gennetic approach. In *Proc. FJCC*, 1972.

[19] David H. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, Artificial Intelligence Center,SRI International, 1983.

[20] Carlos Zaniolo. Object-oriented programming in prolog. In *International Logic Programming Symposium*, 1984.