18

Inheritance in Logic Programming Knowledge Bases

T. Finin and **J. McGuire**

Unisys Paoli Research Center, P.O. Box 517, Paoli, PA 19301, USA

Lockheed A.I. Center, 2710 Sand Hill Road, Menlo Park, CA 94025, USA

Abstract

This paper presents an extended model for a logic programming language's knowledge base. Instead of being restricted to one global knowledge base, as is the case with Prolog, we allow segmentation into units which are linked together into a lattice. Each unit defines a view on the knowledge base which includes those 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 knowledge base unit effectively blocks its inheritance for that unit and all of its descendants. Motivations for using this model are given. We also discuss the design-iterations of a Prolog-based implementation (*Phd*) of this model.

1. Introduction

Most logic programming languages have followed Prolog in having a single, global knowledge base or database in which both the program and data reside. The knowledge base 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 benefits. Conceptually, this is a very simple scheme which new users find easy to learn, use and reason 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 presents a simple model of the database as a 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 should not) – 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 access to it. This

Inheritance Hierarchies in Knowledge Representation and Programming Languages
Edited by M. Lenzerini, D. Nardi and M. Simi.

shows up in programming languages in a variety of ways, including the use of local variables, modules, packages, objects, environments, etc.

This paper describes a design for a hierarchical data model 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 consist of the union of the clauses in the local database unit and the inheritable clauses from the unit's parent units.

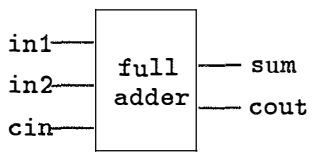
One important feature of our model is that it supports a notion of *relative retraction.* That is, one can retract a clause with respect to a particular database view. This can be done for a clause which is local to the view (i.e. is recorded in the unit which determines the view) as well as for a clause inherited from some ancestor. In either case, the retraction can only affect 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 affects 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; and (iii) thinking of a database as a set of clauses implies that they are unordered. When solving a goal with respect to a database unit (DBU), only the clauses in the view of that DBU will be used to prove the goal. The aim of this paper is to present our model and some of the implementation issues that we have addressed.

An Example

As an example, consider a circuit simulation scenario involving discrete time delays [17]. Figure 1 shows a typical device to be modeled, a *full adder*, and a portion of the Prolog code which models its behavior. A full adder takes three inputs, (*in1*, *in2*, and *carry in*) and produces two outputs (*sum* and *carry out*). The rule in the figure states that the value *Sum* of full-adder *FA*'s sum port can be inferred from the values of the carry-in, in1, and in2 ports after a time delay of 2. Other devices in this domain, such as multipliers, can be modeled in a similar fashion.

Figure 2 shows an organization of a simulator which uses the hierarchical database facility in three ways: (i) as a module system; (ii) to support temporal reasoning about the state of a circuit; and (iii) to reason about hypothetical cases. The rules defining the behavior of each class of device (e.g., full-adder, multiplier, etc.) are placed in a separate DBU. All port value information, which is true at the onset of the simulation is stored in DBU *time*0. Future port value information will be stored in descendant DBU 's of *time*0. Conceptually, each DBU descended from *time*0 represents a later discrete point in time. Old information inherits




```
fa_truth_table(1,1,1,1,1).
type(fa1,fa).
fa(FA,sum,Sum,Delay):-
  val(FA,cin,V1),
  val(FA,in1,V2),
  val(FA,in2,V3),
  fa_truth_table(V1,V2,V3,Sum,Cout),
  Delay = 2.
```

Figure 1: **This figure shows a typical device to be modeled, a full adder, and a portion of the Prolog code which defines its behavior.**

down, with new information being stored in spawned DBU's.

Consider the time context represented by DBU $t3$. New port values have been generated for full-adder fa1's cin and in2 input ports. To find out what can be derived from these new port values, the query *fa(fa1,Port,Value,Delay)* is posed with respect to DBU $t3$. The existing value for the in1 port has persisted from the past, and is inherited down. Using clauses in DBUs *fulladder*, *ti*, and $t3$, the new value of 1 can be inferred for fa1's sum port. This new value needs to be stored in a descendant DBU ($t4$) to represent the delay, but a value already exists for fa1's sum port. Therefore, we need to relatively retract the old sum value from DBU $t4$'s database view as well as assert the new value.[1] The relative retraction will prevent the inheritance of *val(fa1,sum,0)* into DBU $t4$ and its descendants.

This representation also supports the ability to ask 'what if' questions. For example, 'What if the value of fa1's cin port was a 0 at the time represented by DBU $t3$?'. To represent this, we could spawn a new DBU ($t5$), relatively retract *val(fa1,cin,1)*, assert *val(fa1,cin,0)*, and then store all inferred results beneath or in DBU $t5$.

As this example suggests, our model is similar to an *assumption-based truth maintenance system* or ATMS [11]. The analogy can be seen by considering treating each database unit as an *assumption* and each database view as an *environment* (i.e., as a set of assumptions). Under this analogy, the addition or removal of a clause from a database unit corresponds to the addition or a constraint or the modification of one or more existing constraints (see Section for details). Our model is of interest in that is a more specialized application of the general ATMS ideas. This specialization can be more efficiently implemented to support the needs of a maintaining a hierarchical database in a logic programming language.

The next section of this paper will present some motivations for developing a hierarchical model for the database of a logic programming language. The third

[1]Such bookkeeping can be done automatically with forward chaining rules using a system such as Pfc [15].

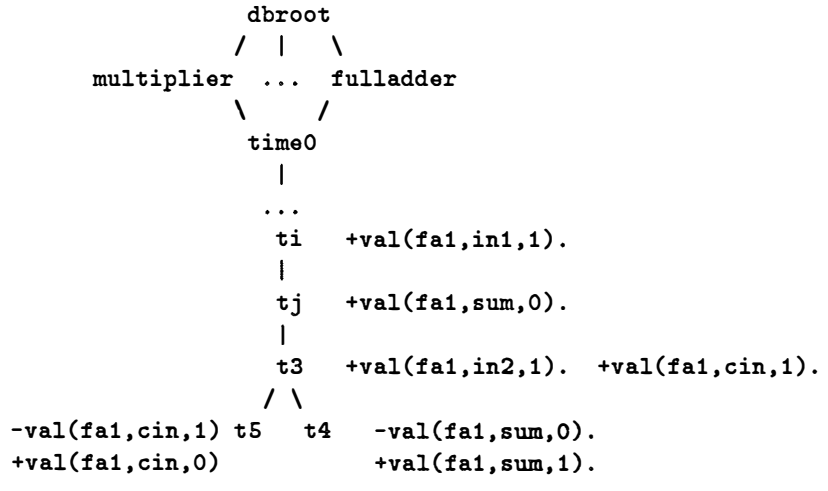


Figure 2: **The hierarchical database can be used as a module system, to support temporal reasoning and to support hypothetical reasoning**

section will define a slightly simplified version of the model through several Prolog predicates and present two different implementation techniques. The fourth section will discuss the relationship between our work and several related projects. We conclude with a final section which summarizes our research.

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 affect 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 better a better tool for knowledge representation (e.g., providing better support for 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. On the other hand, our model does not provide a way to define some information to be *private* within a module and thus unaccessible by external processes.

- *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 [9].

- *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 implementation and the applications to which it is put to use.

- *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 [14,

13] 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 [21, 20] to capture beliefs of users' along different dimensions.

3. The Model and its Implementations

This section presents our model for a hierarchical database for a logic programming language in more detail by way of presenting and discussing three different implementations. Each of these preserves what we take to be the essential characteristics of the model (i.e. a hierarchical database with multiple inheritance and relative retractions) but differ in a number ways (e.g., whether or not the *order* of clauses with a database view is defined).

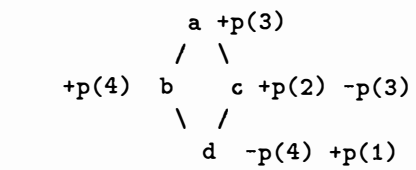
The section will first present an elegant (although inefficient) implementation of a depth-first inheritance using the notion of *relative retractions.* That will be followed by a presentation of the first-cut implementation of the *Phd* language. Next, we describe another scheme based on the labeling technique first used in the *Conniver* language [25]. Finally we discuss how the two schemes can be integrated.

3.1 A simple implementation

Constructing the database view from database unit u_1 can be conceptualized as 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 view of u_1. The *match* operation is like unification except that a variable can only unify with another variable and the shared variable relationships in the terms must be the same. Retractions are stored as negative assertions. As an example, consider the hierarchy in Figure 3 in which the depth-first inheritance from DBU d causes the DBU s to be searched in the following order: (d,b,a,c). The database view associated with d contains the clauses *p(1)*, *p(2)* and *p(3)*.

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




```
db(a,+,p(3))          parent(a,b)
db(b,+,p(4))          parent(a,c)
db(c,-,p(3))          parent(b,d)
db(c,+,p(2))          parent(c,d)
db(d,-,p(4))
db(d,+,p(1))
```

Figure 3: **The hierarchical database can be represented with the parent/2 relation, which encodes the lattice of database units, and the db/3 relation, which records the clauses and their status with respect to local database units.**

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 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.
  \+(localDb(Db,_,C,_)),
  assert(db(Db,+,C)).
```

The *localDb/4* predicate is used to find clauses which match the current one and 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(Db,Mode,Cin,Cout) :-
  db(Db,Mode,Cout),
  match(Cin,Cout).

match(X,Y) :-
  copy_term(X,X2),
  copy_term(Y,Y2),
  numbervars(X2,1,N),
  numbervars(Y2,1,N),
  X2==Y2.
```

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)).
```

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 by 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. The explicit set operations ensure that a clause will only appear in a view once, even though it might be inherited from several ancestors.

```
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,P^(parent(P,Db),clauses(P,C)),ListOfCs),
  mapSetUnion(ListOfCs,Cs).

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

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

Although this implementation is straightforward, it is quite inefficient and is presented for expository purposes only. There are two sources of gross inefficiencies. First, adding and removing a clause to a database unit takes time proportional to the number of clauses in the unit. This stems from the fact that the *localDb*/4 predicate does a sequential search through the local clauses looking for matches. Second, looking up a clause in a database unit takes (at best) time proportional to the number of clauses in the entire view of the unit. This stems from the need to explicitly compute and manipulate the sets of local clauses and clauses in the views of the parents.

3.2 The *Phd* implementation

Our first serious implementation of this model was in the *Phd* meta-interpreter [16]. This implementation provides additional functions for creating and mutating the hierarchy of databases as well as a mechanism for *context switching* – changing the interpreter's notion of the current database view.

The *Phd* implementation is similar to the simple one presented above. It differs in two details. First, it keeps pointers between local relative retractions and the ancestral clauses that they effectively retract. These pointers allow the implementation to be relatively more efficient. The second difference is the search strategy employed by the inheritance mechanism. The simple model presented above searches ancestor database units in a depth-first manner. In the *Phd* model, the order of inheritance reflects the partial ordering defined by the parent-child relationships in the hierarchy.

In the *Phd* model, each database unit is assigned an integer representing its 'level' in the hierarchy, with the requirement that a database unit's level be strictly less than the level of its immediate descendants. The inheritance order with respect to any DBU could be found by sorting all ancestor DBU s by their level.

There was one major drawback to the *Phd* model, which caused us to search for better implementation strategies. By searching through each DBU in the inheritance order, all the clauses that are potentially in the current view are known. However, we must also examine relative-retractions encountered along the way to confirm whether a candidate clause is in fact in the current view. This turns

out to be a difficult problem, since an ancestor clause may be retracted along one path from the current DBU but not along another path from the current DBU . We concluded that this process was much to expensive to be done at query time. It would be preferable if this information could be 'compiled' at update time. Techniques for doing just this are described in the next section.

3.3 Clause labeling implementations

In this section, we describe extensions of the elegant labeling technique first used in the Conniver language [25] and described by McDermott [22]. This general technique is also used to implement *assumption based truth maintenance systems* (ATMSs) [11]. Our clause-labeling implementation can be thought of as a special purpose ATMS with specialized labeling and propagation algorithms [24, 12].

McDermott's basic idea works for a system in which the database units form an extensible tree (i.e., one which can grow from leaves), where clausal assertions and retractions can be made only to existing leaf database units. The internal representation for this technique prevents the enforcement of any inheritance order. First we present the basic model. Then we discuss extensions to the model that allow retractions on non-leaf DBU's, allow multiple inheritance, and support the notion of an inheritance order (i.e. provide a clause ordering).

McDermott's technique

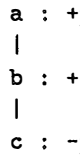
The basic idea in this scheme is to associate with each clause in the database a label which encodes those views from which the clause is visible. In order to retrieve all clauses visible in view V which unify with a goal G, we define the *current context* to be V, retrieve all clauses in the entire database which unify with G, and then filter these candidate clauses by discarding any whose labels do not evaluate to true in the current context.

To realize this scheme, we assign each DBU a unique identifier (DBID) which also acts as a boolean variable. The view defined by a DBU can be established by only assigning the value *true* to the DBID's of DBU itself and all of its ancestors and assigning all other DBID's the value *false*).

All clauses are stored in a single global database. Each clause contains a label, which is boolean combination of DBIDs. By evaluating a clause's label, it can be determined whether the clause is visible (true) in the current view. Since the clauses are internally stored in the same global database, clause ordering within views cannot be enforced. The views and DBUs define conceptual segmentations of the database, but there is no corresponding internal physical separation.

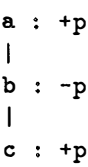
Adding a clause p to a database unit U with identifier u simply requires changing p's label from l to $l \vee u$. In the absence of relative retractions, p's new label will also evaluate to *true* in the views of newly added DBU's descended from U.

To erase a clause p from U, we need only change p's label from l to $l\wedge \sim u$. P's new label will evaluate to *false* in the views of newly added DBU's descended



time	event	label
t1	add(b,p)	b
t2	rem(c,p)	$b \wedge \sim c$
t3	add(a,p)	$(b \wedge \sim c) \vee a$

Figure 4: **The standard clause labeling technique will assign an incorrect label to the clause p when it is added to the non-leaf unit a. This label shows p to be visible from unit c from which it had previously been removed.**



time	event	label
t1	add(a,p)	a
t2	add(c,p)	$a \vee c$
t3	rem(b,p)	$(a \vee c) \wedge \sim b$

Figure 5: **The standard clause labeling technique will assign an incorrect label to the clause p when it is removed to the non-leaf unit b. This label shows p to be invisible from unit c to which it had previously been added.**

---

from U.

Unfortunately, this technique does not work if we allow arbitrary assertions and retractions to any database unit or if we allow database units to have more than one parent. The example in Figure 4 shows a situation in which the clause p is mistakenly marked as being visible in database view of C, due to the order in which the assertions and retractions were made. The example in Figure 5 shows another problematic situation where the label assigned to p indicates that it is not visible from view C.

Updates in a tree

It is relatively easy to extend McDermott's technique to allow arbitrary updates (i.e., including those not restricted to leaf DBU's) if we assume that the database units are organized into a tree structure. We sketch the solution. Associated with each clause p we have three variables:

- *in* – list of all DBUs in which the clause has been locally asserted.
- *out* – a list of all DBUs from which the clause has been locally retracted.
- *label* – the membership formula in disjunctive normal form.

The *label* of a clause is a disjunction of conjunctions. Each conjunction contains exactly one non-negated variable, called its *head*. For each member of *in*, there will exist a conjunction in the *label* with that member's DBID acting as the *head*.

When a clause p is **asserted** into a database unit d, the following algorithm is used to update $label(p)$:

1. If p has already been locally asserted into database unit d (i.e. $d \in in(p)$) then stop.
2. Add d to $in(p)$.
3. If p had previously been locally retracted from d (i.e., $d \in out(p)$), then delete d from $out(p)$ and remove negated d's from each conjunction in $label(p)$.
4. Create a new conjunction for d and add it to $label(p)$. The negated identifiers in the conjunction will be those members of $out(p)$ that are also descendants of d.

Note that if the clause did not exist before, then $in(p)$ would be set to $[d]$, $out(p)$ would be set to $[\,]$ and $label(p)$ would be set to (d).

When a clause p is **retracted** from database unit d, the following algorithm is followed.

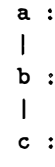
1. Add d to $out(p)$.
2. If $d \in in(p)$, then delete d from $in(p)$ and delete the conjunction corresponding to d from $label(p)$.
3. For each conjunction in $label(p)$, if d is a descendant of the conjunction's head, then add $\sim d$ to the conjunction.

Applying this method for our two earlier problem examples results in the situations shown in Figures 6 and 7. Note that the labels are not the simplest ones possible. The final label for p in Figure 6 could be simplified to $a \wedge \sim c$.

Updates in a lattice

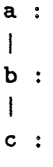
This revised method does not, unfortunately, handle some cases in which the database units form a lattice, as is demonstrated by the example in Figure 8. In this example the final label for p is $b \wedge \sim d$, indicating that p would not be visible from the view from DBU e. The problem arises because there may be several paths between a database unit and one of its descendants. If a clause p is added to the unit, there may or may not exist a *valid inheritance path* for p from the unit to a given descendant, depending on the existence of relative retractions along the way. The revised clause labeling method will block inheritance if any one path contains a relative retraction. What is needed is a modification to the technique which reasons about all potential inheritance paths.

We have developed a further extension to the basic technique which will assign a correct label to each clause when the database is structured as a lattice. For the example in Figure 8 this algorithm will assign p the label $b \wedge \sim d \vee e$, showing it to be visible in units b, c, and e and not visible in d. We present some preliminary definitions and sketch the solution.



time	event	in	out	label
t1	add(b,p)	(b)	()	b
t2	rem(c,p)	(b)	(c)	$b \wedge \sim c$
t3	add(a,p)	(b,a)	(c)	$(b \wedge \sim c) \vee (a \wedge \sim c)$

Figure 6: **The revised clause labeling technique will assign a correct label to the clause p when it is added to the non-leaf unit a. This label shows p to be in a and b but not c.**



time	event	in	out	label
t1	add(a,p)	(a)	()	a
t2	add(c,p)	(a,c)	()	$a \vee c$
t3	rem(b,p)	(a,c)	(b)	$(a \wedge \sim b) \vee c$

Figure 7: **The revised clause labeling technique will assign a correct label to the clause p even when it is removed to the non-leaf unit b. This label shows p to be in databases a and c but not b.**

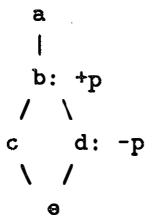
An *inheritance path* for a clause p holds between two database units $d1$ and $d2$ if p has been locally asserted into $d1$ and $d1$ is an ancestor of $d2$. Such a path is a *valid inheritance path* if p has not been locally retracted at any intervening database unit along the path. A *merge point* is a DBU which has multiple parents. Let mp_d represent the set of all merge points descended from d. An *uppermost merge point* beneath a DBU d (ump_d) is a member of mp_d to which a path exists from d that does not traverse another member of mp_d. The following simple Prolog predicate defines this relationship:

```
uppermostMergePoint(D,Ump) :-
  parent(D,Ump),  % must be a member of mpd.
  parent(D2,Ump),
  \+ D == D2,
  !. % Don't search beneath identified umpd.

uppermostMergePoint(D,Ump) :-
  parent(D,Child),
  uppermostMergePoint(Child,Ump).
```

The problem in Figure 8 is that we have an uppermost merge point, e , which can still inherit from DBU b. Our approach to solving the problem displayed in Figure 8 will associate with p the membership formula $(b \wedge \sim d) \vee e$.

In the tree-scheme, the positive literal in each conjunction was chosen from the set *in*, representing each DBU containing an instance of the clause in question



time	event	in	out	label
t1	add(b,p)	(b)	()	b
t2	rem(d,p)	(b)	(d)	$b \wedge \sim d$

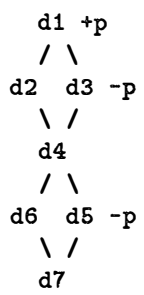
Figure 8: **For this simple lattice example, the standard labeling technique assigns p the label $b \wedge \sim d$, incorrectly showing it to be invisible to database unit e.**

(p). To handle lattices, we also associate with each clause a set of *Hypotheses* whose members are a subset of all merge-points in the lattice. Each *Hypothesis* gains support from some member of *in*. The support indicates that a *valid inheritance path* exists from the *in* member to the *Hypothesis* member. The idea is to introduce the minimal necessary number of *Hypotheses* as positive literals into our membership formulas.

Upon a relative retraction of p from DBU U, identify for clause p all uppermost merge points below U to which a valid inheritance path exists from one of the U's ancestors. The DBID's of the members of this set represent the new *heads* of conjunctions which need to be added to *label*(p). By restricting ourselves to uppermost merge points, we introduce the minimal necessary number of extra conjunctions.

Upon an **assertion** p to a DBU d, the following algorithm is performed:

1. Execute the algorithm defined for assertions within a Tree hierarchy.
2. Identify all relative retractions of p that are descendant to d.
3. For each such relative retraction r, do:
   For each uppermost merge point below r, ump_r, such that there is a valid inheritance path for p from d to ump_r, do:
   (a) Create Hypothesis ump_r with support from d.
   (b) Set ump_{out} to the set of all DBU's descended from ump_r that contain a relative retraction of p.
   (c) Add $(ump_r \wedge \sim ump_{out})$ to *label*(p) (i.e. make it one of the conjunctions) with support (i.e justification) from d.



time	event	in	out	label
t1	add(d1,p)	($d1$)	()	$d1$
t2	rem(d3,p)	($d1$)	($d3$)	$d1 \wedge \sim d3 \vee d4$
t3	rem(d5,p)	($d1$)	($d3$, $d5$)	$d1 \wedge \sim d3 \vee d4 \wedge \sim d5 \vee d7$

Figure 9: **A correct label of $d1 \wedge \sim d3 \vee d4 \wedge \sim d5 \vee d7$ is assigned to p.**

Upon a **retraction** of clause p from DBU d, the following algorithm is performed:

1. Execute the algorithm defined for retractions within a Tree hierarchy.
2. If this is a local retraction (i.e. $d \in in(p)$) then remove the support d from any Hypothesis relying on d.
3. For each database unit $d_p \in in(p)$, such that d_p is an ancestor of d: For each uppermost merge point ump below d such that there is a valid inheritance path for p from d_p to ump:
   (a) Create Hypothesis ump with support from d_p
   (b) Set ump_{out} to the set of all relative retractions of p beneath ump
   (c) Add $(ump \wedge \sim ump_{out})$ to $label(p)$

Figure 9 shows the effects of retractions on a more complicated lattice of database units. At time t2, a retraction of p is made from DBU $d3$. The tree algorithm will produce the label $(d1 \wedge \sim d3)$. This we know to be incorrect as p is visible in DBU $d4$ and its descendants. DBU $d1$ is the only ancestor of $d5$ containing clause p. DBU $d4$ is the only uppermost merge point beneath $d3$ containing a valid inheritance path from $d1$. Thus according to the new algorithm we should introduce a new conjunction with $d4$ as the positive literal (head) into p's label.

At time t3, a retraction of p is made from DBU $d5$. Once again, DBU $d1$ is the only ancestor of $d3$ containing clause p. DBU $d7$ is the only uppermost merge point beneath $d5$ containing a valid inheritance path from $d1$. Thus, we now introduce a new conjunction into p's label with $d7$ as the positive literal. Careful inspection of the final label in Figure 9 will show that p's label will evaluate to *true* in the views of DBU's $d1$, $d2$, $d4$, $d6$, and $d7$.

It turns out that there is a very natural (and potentially very efficient) way to co-routine backtracking and clause evaluation in Prolog, without the need for meta-interpretation. The basic idea is to store a fact p with label l as `p :- l -> true` and a rule with head p, body q and label l as `p :- l -> q` where the `->`

operator is the standard Prolog *if - then* operation. For example, the clause p in Figure 9 would be represented as:

```
p :- (in(d7) ; in(d4),\+in(d5) ; in(d1),\+in(d3)) -> true.
```

Similarly, to add the rule that 'All full adders are digital devices' to DBU $d1$ we can assert:

```
digital_device(X) :-  in(d1) -> full_adder(X).
```

The use of the -> operator prevents Prolog from backtracking into additional and entirely redundant attempts to 'prove' the membership label.

In order to prove a goal with respect to some DBU, you must first switch context to the new view and then simply attempt to prove the goal. Switching to a new database view is accomplished by asserting $in(D)$ for each database unit D in the view and removing any other in/1 assertions. In practice, the view could be represented via an efficient tree-based data-structure which was passed along as an extra argument between the sub-goals. This would make context switching less expensive, since Prolog asserts are a relatively more expensive operation.

Clause labeling with an inheritance order

The clause-labeling algorithm, as described, stores all clauses in a single global-database. The problem with this is that there may be many clauses stored that unify, but few which are in the current view. Thus at query time, each label associated with a unifying clause might be evaluated. For many applications, this will be an unsatisfactory state of affairs. Our solution is to merge the clause-labeling scheme with the original inheritance-order scheme. In this approach, copies of a clause will be stored in each DBU containing the clause. This can be done efficiently in the module systems found in many Prolog systems (e.g. Quintus Prolog). The membership labels for each clause will be stored once in the global database. At query evaluation time, we simply retrieve the inheritance order for the current view and then search each DBU in turn. This gives us back general access efficiency. Also, we have in our clause labels a 'compiled' representation of the relative-retractions.

4. 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 [25] and QA4 [8], for example, had hierarchical databases. More recent work on enriching the model of the clausal database can be divided into three groups: the incorporation 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

cover 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 [23, 18] 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 also seen a number of proposals for languages which combine the logic programming and object-oriented programming paradigms [27]. Some examples which include some kind of hierarchical database-like facility include KEE [2], BiggerTalk [19] and LOGIN [5]. There are also parallels to be drawn with discussions of inheritance in *frame inference systems* [26, 10].

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 [9], an efficient implementation of it designed by Bacha [7], the Rhet representation system [6] and work by McDermott [22].

Finally, the type of hierarchical knowledge base that we have defined can be modeled using an ATMS [11].

Although the basic idea of a hierarchical database for a prolog-like language seems relatively simple, there are several factors which make an efficient implementation somewhat difficult to achieve. Several of these (1, 2 and 3 below) are decisions we have made in the design and several are a consequence of using a logic programming language like Prolog.

1. *relative retraction.* This makes the model much more complex than the related models for languages such as LOGIN [5] and BIGGERTALK [19] and much of the recent work on object oriented programming systems.

2. *lattice structure.* We assume that the DBUs comprising the database can, in general, have multiple parent DBUs and thus form a lattice. This extends the expressiveness of the model at the expense of increasing the complexity of inheritance.

3. *dynamics.* We would like to be able to dynamically assert and (relatively) retract clauses to any DBU at any time, not just to a 'leaf' DBU.

4. *non-ground queries and multiple answers.* A database query will, in general, contain uninstantiated variables which enable it to be satisfied by more than one clause in the database view. However, a particular ground instance of a unit clause (i.e. an assertion of a fact) should only appear once as a solution even if it can be inherited from several different ancestors.

On the other hand, there is one important factor which make this model much more manageable than most of the related models used in object-oriented representation languages designed for AI work (e.g. KEE [2]), — the lack of negation.

The problem of determining what clauses are visible in a given database view is very similar to determining the attribute values[2] for a given object. Most AI languages, however, allow negation of one form or another. This can be expressed either in the form of an explicit negation operator or in the ability to explicitly *cancel* or mark as an *exception* certain attribute values. This introduces the possibility of *contradictory evidence* and *ambiguity* into the database. This problem is avoided in our model since logic-programming languages do not directly support negation.

5. Conclusions

This paper has described a model for a hierarchical database for a logic programming language such as Prolog. In this model, the clausal database is segmented into database units, each of which is a local collection of clauses. Each database unit defines 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 affect the database view in which the retraction was done and (potentially) its descendant's views.

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, Rhet and meta-Prolog). This greatly increases the expressiveness 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.

In addition to presenting the model, we have described several different implementations which differ in their complexity and efficiency profiles. 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.

[2]More accurately, the problem is usually to determine what an object's attributes are and for each, to determine the relevant values and/or other facets (e.g. types)

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

[6] J. Allen and B. Miller. *The Rhetorical Knowledge Representation System: A User's manual.* Technical Report 238, Department of Computer Science, University of Rochester, September 1988.

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

[8] D. Bobrow and B. Raphael. New programming languages for artificial intelligence research. *Computing Surveys*, **6**(3), 1974.

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

[10] G. Brewska. The logic of inheritance in frame systems. In 10^{th} *International Conference on Artificial Intelligence*, August 1987.

[11] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, **28**:127–162, 1986.

[12] J. de Kleer. A general labeling algorithm for assumption-based truth maintenance. In *Proceedings of the* 7^{th} *National Conference on Artificial Intelligence*, pages 188–192, 1988.

[13] T. 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.

[14] T. Finin and D. 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, CSCSI/SCEIO, May 1986.

[15] T. Finin, R. Fritzson, and D. Matuzsek. Adding forward chaining and truth maintenance to prolog. In *Proceedings of the Fifth IEEE Conference on Artificial Intelligence Applications*, March 1989.

[16] T. Finin and J. McGuire. *A Hierarchical Database Model for a Logic Programming Language.* Technical Report MS-CIS-88-108, CIS Dept, University of Pennsylvania, 1988. Also available as LBS Technical note 87, Paoli Research Center, Unisys, Paoli, PA.

[17] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, **24**:411–436, 1984.

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

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

[20] R. Kass and T. Finin. A general user modelling facility. In *Proceedings of the Human Factors in Computer Systems Conference (CHI'88)*, 1988.

[21] R. Kass and T. Finin. Modelling the user in natural language systems. *Computational Linguistics*, Special Issue on User Modelling, 1988.

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

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

[24] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *Proceedings of the* 6^{th} *National Conference on Artificial Intelligence*, pages 183–188, 1987.

[25] G. Sussman and D. McDermott. From planning to conniving: a gennetic approach. In *Proc. FJCC*, 1972.

[26] D. Touretzky, J. F. Horty, and R. Thomason. A clash of intuitions: the current state of nonmonotonic inheritance systems. In 10^{th} *International Conference on Artificial Intelligence*, August 1987.

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