PROLOG/RDBMS Integration in the NED Intelligent Information System

F. Maier[1], D. Nute[1], W. D. Potter[1], J. Wang[1], M. Twery[2], H. M. Rauscher[3],
P. Knopp[2], S. Thomasma[2], M. Dass[1], and and H. Uchiyama[1]

[1] Artificial Intelligence Center, The University of Georgia, Athens, GA, USA
[2] Northeastern Research Station, USDA Forest Service, Burlington, VT, USA
[3] Bent Creek Experimental Forest, Southern Research Station, USDA Forest Service,
Asheville, NC, USA

Abstract. The following paper describes recent work on NED-2, an intelligent information system for ecosystem management currently in development by the USDA Forest Service. Using knowledge bases created by forestry experts and inference engines, NED-2 evaluates forest inventories according to a set of predefined goals. By integrating third-party simulation and visualization packages, NED-2 allows the user to plan, predict, and assess forest treatment scenarios. NED-2 is a blackboard system featuring agents implemented in PROLOG. Inventory, data calculation, and forest management plan creation modules are implemented in C++. The primary storage medium of NED-2 is a set of relational databases. The present paper focuses upon an issue of central importance to the project: the integration of PROLOG and relational databases to form the blackboard of NED-2.

Keywords. Intelligent Information System, Decision Support System, PROLOG, Relational Database System, Blackboard System

1 Overview

NED-2 is a software system currently in development by the USDA Forest Service (in conjunction with the University of Georgia Artificial Intelligence Center) to facilitate ecosystem management. It allows for the analysis of forest inventories to determine the degree to which they satisfy goals pertaining to timber production, water quality, aesthetics, wildlife habitat, and ecology. In addressing such diverse goals, NED distinguishes itself from the many decision support systems used in the forestry domain (which often deal only with maximizing timber). Through the integration of external simulation and visualization packages (such as FVS), NED-2 allows the user to plan treatment schedules, predict their outcome, and assess their worth. NED-2 is a second-generation product, building upon the capabilities of its immediate predecessor, NED-1.

NED-2 is a blackboard based system, agents being implemented in the PROLOG programming language. Calculation and user interface screens are implemented in C++. Most of the information used by NED, however, is stored in relational databases. Both relational databases and PROLOG clauses constitute

the blackboard of NED, and it is through the use of this blackboard that the PROLOG agents communicate.

As it integrates such diverse components, NED-2 is an Intelligent Information System (IIS), which may be viewed as a unified knowledge base, database, and model base [Potter02]. The main idea behind this notion is the transparent processing of user queries. The system is responsible for "deciding" which information sources to access in order to fulfill a query regardless of whether this involves a data retrieval, an inference, a computational method, a problem solving module, or some combination of these.

This paper is devoted to describing the methods used to couple PROLOG to relational database systems. As a set of relational databases constitutes NED's primary storage medium, and as NED's goal analysis modules are implemented in PROLOG, the interface between PROLOG and these databases is of central importance. It was found that the usual method of querying a database from PROLOG did not meet the needs of NED. Specifically, combining data from multiple tables using the usual query technique proved too slow to be of use. Furthermore, the usual method requires absolute knowledge of the database schema and requires that the schema remain constant. Alterations to the schema require changing a great deal of the code utilizing it.

In place of the usual methods of access, a query language was created especially for NED. This language provides the user an efficient and friendly way to access a database from PROLOG. Importantly, the user is not required to have full knowledge of the database schema, nor is the schema required to be fixed.

The following section provides a short history of the NED project and presents a few of NED-2's distinguishing characteristics. Later sections describe the kinship between PROLOG and relational database systems and give a brief account of the reasons for integrating the two. Barriers to their union are discussed. The solution used in the NED project is then presented.

2   A Brief History of NED

The NED project was conceived during meetings held in 1987 between members of the Northeastern Forest Experiment Station (now called the Northeastern Research Station) of the USDA Forest Service [Twery00, 172]. A desire was expressed for a piece of software incorporating all of the growth and yield models designed at that station. Particularly, a wish was voiced for "a single, easy-to-use-program that could provide summary information and expert prescriptions for any forest type in the northeastern United States." [Twery00, 172]. The name 'NED', an acronym for 'NorthEast Decision Model', was chosen [Twery00, 168].

The integration of independent and often relatively incompatible applications and data stores—so-called *heterogeneous data sources*—is currently a topic of some interest in computer science. Integration is not a trivial problem; nevertheless, a unification of that sort is a primary goal of NED [Twery00, 186-187, 189].
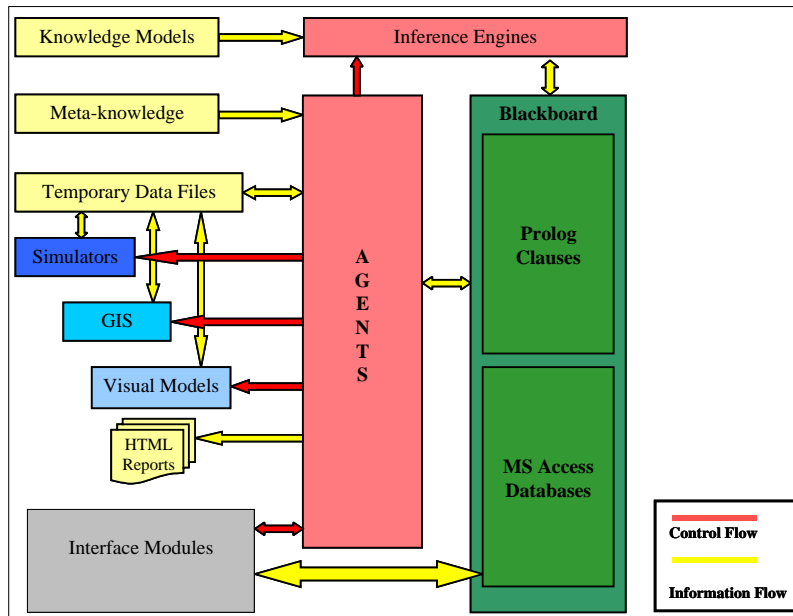
Fig. 1. NED-2 System Structure

The NED project has resulted in the development of several software products, all of which are described in [Twery00]. The current project centerpiece is NED-1, a decision support system designed to help manage forests down to the level of individual trees in stands. NED-2, which builds upon the capabilities of NED-1, is soon to be finished. When completed, it will serve as the glue binding several other software pieces together.

NED-2 is intended to be more than a system for maximizing timber production. It is *multi-criterial* [Nute00]. As is usual for software in the forestry domain, it helps users manage for timber. It also, however, helps them manage their land from the standpoints of ecology, water quality, visual quality, and wildlife habitat [Twery00, 168]. In this way, NED-1 and NED-2 fall into the category of software systems designed for ecosystem management, where such management is responsible for obtaining "a sensible middle ground between ensuring long-term protection of the environment while allowing an increasing population to use its natural resources for maintaining and improving human life" [Rauscher99]. As society becomes more and more complex, and as natural resources become more and more precious, the need for such systems becomes increasingly urgent [Rauscher00b, 1ff; Rauscher99, 175ff].

NED-2's development has been guided by the perception that forest management is fundamentally a goal-driven process [Nute00; Rauscher00a; Twery00]. All action on the land is performed to achieve some goal. As this is so, the selection of goals (which are represented as logical complexes of measurable states

called desirable future conditions) plays a prominent role in NED-2 and influences the rest of program execution. The creation of treatment plans and the simulation of these plans, the generation of summary reports—all of this is performed with goal satisfaction in mind.

3 PROLOG and Relational Databases

Much ado has been made over the similarities between logic based languages such as PROLOG and relational databases, and there has been a sizable amount of effort exerted towards producing a practical marriage of the two [see Gray88; Kerschberg86; Kerschberg89]. Such a marriage would combine the inferencing capabilities of PROLOG with the ultra-efficient data handling capabilities of database systems. To date, however, no full integration has gained a significant degree of acceptance. The usual link between PROLOG and a database, when it exists, is often of a very tenuous nature—usually being a simple interface between two independent systems. Indeed, the usual method of connecting PROLOG to a database—as exemplified by the ProData [Lucas97] interface—has turned out to be quite unsuitable for use in the NED project. Ultimately, for want of an efficient means of querying NED-2 databases from PROLOG, project members were forced to develop a query language of their own.

3.1 The Similarities Between PROLOG and Relational Databases

It will be recalled that a PROLOG program consists of facts, such as

```
% a few facts stating stand adjacencies

adjacent(stand1, stand3).
adjacent(stand2, stand3).
adjacent(stand2, stand4).

% some facts listing stand size classes

size_class(stand1, 'small sawtimber').
size_class(stand2,sapling).
size_class(stand3, 'small sawtimber').
size_class(stand4,pole).
```

and of rules, such as

```
% two adjacent stands of the same size class
% are in the same patch

in_same_patch(X,Z):-
    adjacent(X,Z),
    size_class(X,Y),
    size_class(Z,Y).
```

The clauses above constitute a knowledge base which can be queried.

The structural similarity of the above facts to tuples of a relational database should be fairly obvious. Indeed, a predicate in PROLOG (as in the rest of logic) is normally interpreted to be nothing more than a relation over some domain. What is perhaps more interesting, however, is that a PROLOG rule, such as `in_same_patch/2`, may be viewed as a join of the relations `adjacent` and `size class` [Sciore86, 294; Zaniolo86, 221; Gray88,7]. The PROLOG predicate `in_same_patch/2`, which is the conclusion of the rule, would in the relational model be a derived relation, or *view*, over base relations [Lunn88, 42]. DATALOG, a language based upon PROLOG, has been proposed and has gained acceptance as a relatively graceful language for defining and querying databases [Ullman89, 100ff].

3.2 Motivations in Linking PROLOG to a RDBMS

There are good reasons for wishing to marry PROLOG to a relational database. One reason has already been mentioned—a PROLOG like language is a concise and intuitive way of specifying instructions to a database [Lunn88, 39; Zaniolo86, 219ff]. Compare

```
-? assert(( p(X,Y):- q(X,Z), r(Z,Y))).
```

```
-? p(a,Y).
```

to its SQL counterpart

```
CREATE VIEW P AS SELECT Q.X,R.Y FROM Q,R WHERE Q.Z = R.Z
```

```
SELECT Y FROM P WHERE X = a
```

Though the example is very simple, many would argue that the PROLOG rendition is easier to follow. The difference becomes more evident as the action to be performed grows more complex.

Another reason for desiring some sort of integration is that PROLOG clauses are generally held in primary memory. This places a severe restriction on the project size to which PROLOG can be reasonably applied [Sciore86, 295 Irving88, 83]. Many who appreciate the general programming abilities of PROLOG would prefer to have the large storage space of disk drives available to them. Furthermore, keeping data in primary memory limits access to data by multiple agents [Irving88,83; Venken88, 95]. Databases, in contrast, can store huge amounts of information on secondary storage such as disk drives, and almost universally allow concurrent access by multiple users.

Unlike most database systems, PROLOG makes little use of indexing schemes or other optimization techniques, making information retrieval, and therefore logical inference, relatively slow [Sciore86, 295]. Database systems, in contrast, are amazingly good at sifting through very large quantities of information. Much effort has been exerted in devising clever ways to speed retrieval.

3.3 Obstacles to Integration

There are differences between PROLOG and relational databases, however, which serve as obstacles to integration and which must be pointed out. The most important difference is that, while the average relational database manipulation language can be considered almost entirely declarative, PROLOG has a strong procedural bent. This is the obligatory result of PROLOG being developed as a general purpose programming language. Database languages, being special purpose, need not be so encumbered [Brodie88, 200; Zaniolo86, 221]. Furthermore, PROLOG has a fixed built-in search mechanism (depth-first) and is littered with elements for performing actions irrelevant to logical inference [Brodie88, 194-196].

Lesser points of conflict are as follows: First, the domains for relations in the database model are explicitly specified in a relational system. If the elements are not enumerated, their respective data types (integer, character, etc.) are at least specified. [Date95, 81-86]. PROLOG, though it can be said to have a typing system of some sort, does not provide a means of specifying how predicate arguments are to be restricted [Brodie88, 197]. Furthermore, attributes in a relational system are generally referenced by name rather than by position. In PROLOG, no argument has a name. Also, values of attributes in a relational database are atomic, meaning that the tuples in a database table correspond only to atomic propositions containing no unbound variables. With very few exceptions, one cannot store a complex structure in a relation [Date95, 81]. In contrast, PROLOG predicate arguments can be as complex as one likes.

4 Techniques of Integration

According to [Brodie88] there are four general methods of combining elements of PROLOG and a relational database system. The classification is natural and is paralleled elsewhere (see [Singleton93], [Ioannidis89]). The four are:

1. Coupling of an existing PROLOG implementation to an existing relational database system;
2. Extending PROLOG to include some facilities of a DBMS;
3. Extending an existing DBMS to include some features of PROLOG;
4. Tightly integrating logic programming techniques with those of DBMS's [Brodie88, 203].

While the first three methods in some way add features to pre-existing systems, the last may be viewed as building a system from scratch. [Brodie88] recommends this fourth alternative, saying that it is no more work than the second or third, and that the end result will be a more capable system.

4.1 Loose Coupling Vs. Tight Coupling

Regarding coupled systems, the literature usually refers to systems of two types: tight and loose. The terms are used in various ways, however. Some authors

appear to use the terms in reference to the underlying architectural connections between PROLOG and the database system. Others appear to refer to the degree of integration from a programming point of view. [Venken88] defines a tight coupling to exist when PROLOG and a database system are compiled together forming a single program [88]. This matches the fourth architecture described by [Brodie88]. In this paper, such systems are called fully integrated. It is natural to suppose that in such a system, there would be only one language involved in its manipulation.

[Lucas97, 68] writes that a tight coupling exists "where external records are retrieved from the database and unified with PROLOG terms as and when required." With loose coupling, large chunks of information are fetched from a database into PROLOG memory prior to being used by the program. These are architectural considerations. A third parameter, *transparency*, is specified where each database relation appears to be just another PROLOG clause and can be invoked in normal PROLOG fashion. This appears to be a programmatic consideration.

[Date95] defines loose coupling as providing a call level interface between the logic language and the DBMS; users would program both in PROLOG and SQL, for instance—"The user is definitely aware of the fact that there are two distinct systems involved....This approach thus certainly does not provide the 'seamless integration' referred to above"[671]. With tight coupling, "The query language includes direct support for the logical inferencing operations. Thus the user deals with one language, not two."

Such uses of 'loose' and 'tight' coupling go against the usual meanings of the words in the computer industry, where systems are tightly coupled if they cannot function separately; they are loosely coupled if they can. Given this definition, all couplings of PROLOG to databases—since they connect independent systems via some software interface—are loose couplings. That, at any rate, is how the term will be used here.

4.2   Relational Level Access vs. View Level Access

Regarding programmatic considerations, the most natural way of representing (and accessing) data stored in an external database for use in PROLOG is simply to treat relations in a database as predicates and treat their tuples as one would treat PROLOG facts. This is by far the most common method encountered in the literature. Data in the database would be accessed in PROLOG's normal depth-first search fashion. Importantly, with the exception of the routines needed to implement the transparent use of these database predicates, this method requires no changes to either PROLOG or the database. PROLOG gains the use of secondary storage and concurrent access and otherwise escapes unscathed.

This is sometimes called *relational access* [Draxler93], sometimes *tuple-at-a-time* access [Napheys89]. It is relational because only a single relation is involved in the query. It is tuple-at-a-time because generally only a single tuple is returned as a solution. The two terms are not quite interchangeable; a query involving one relation might return an entire set of solutions, and a query involving multiple

relations could return solutions one-at-a-time. The terms are both appropriate here simply because it does not make sense to have a PROLOG variable simultaneously bound to multiple values. PROLOG prefers to backtrack for further solutions rather than having them presented all at once.

Because relational access requires few changes to PROLOG and the database, it is easy to implement. However, it is horribly inefficient. This fact cannot be over-stressed. Relation at a time access does not utilize any of the relational database's mechanisms for optimizing data retrieval. Relational databases are designed to take a complex query, determine an optimal plan for satisfying that query, and execute that plan. With relation at a time access, since queries are of the simplest possible variety, no optimization is possible.

The slowness of the relational approach has been offset to some degree in BERMUDA and Primo [Ioannidis89, 238; Ceri90]. Both of these systems implement a function which determines how much of a PROLOG rule can be grouped together and sent to the database without ruining the normal procedural execution of PROLOG. In the case of BERMUDA, the analysis of the predicates occurs at run time. In the case of Primo, a pre-compiler rewrites the original rules to designate which complexes should be passed to the database for processing. The technique preserves transparency. Particularly, the system acts exactly as a PROLOG program not involving an external database would. This is a significant virtue, as it makes the code more portable. However, the performance increase depends entirely on how clustered the so-called database predicates are.

The alternative to relation level access is called *view access*. Here, a complex query is passed to the database system, and it is the database system which does all of the work in satisfying the query (importantly, it is not PROLOG). Depending upon how it is implemented, solutions can be returned tuple-at-a-time or set-at-a-time.

The improvement in performance using this method can be staggering. Solving a given problem might take a single call to the database system and less than a second for view access; solving the same problem might take thousands of calls and many hours for relational access. This is not surprising, for relational access is merely a variation of a depth-first search, which is a blind search.

The drawback to view level access is that it generally ruins the transparent use of the database. Queries to the database, if they are to be efficient, are generally isolated from the rest of the PROLOG program upon being written. Upon execution, the queries are sent en masse to the database system.

4.3   ProData

In practice, almost all real world systems linking PROLOG and a relational database system simply tack on a software interface between a pre-existing PROLOG implementation and a pre-existing relational database system. In other words, the PROLOG and database systems are loosely coupled. Often (at least in the case of MS Windows based machines) Microsoft's ODBC is used as an intermediary [Microsoft97]. An interface allows PROLOG to query the database

when needed, either by translating PROLOG goals to SQL or else by embedding SQL directly into the PROLOG code.

The database links allow PROLOG varying degrees of control over databases. Some are very low level, meaning that the user must keep track of things such as connection handles and cursors. The benefit of this is greater control over program execution and more subtle access to databases. The drawback is that an inexpert programmer can easily write dangerous code.

A higher level approach is exemplified by ProData [Lucas97], a library used in the LPA, SICStus, and Quintus implementations of PROLOG. [Lucas97] calls it a "transparent tight coupling", but according to the way the terms are used in this paper, it is better to call it a transparent loose coupling. Within ProData, all low-level database access functions are hidden from the user. It is a relation level system, users specifying which database relations to be used as PROLOG predicates. After such specification, database predicates are treated as PROLOG facts. Particularly, database predicates are re-entrant (meaning, basically, that separate calls to a database predicate do not interfere with each other—each is bound to answers in a top-down manner) and cuttable—meaning that they do not backtrack through a PROLOG cut [Singleton93].

ProData is a standard of sorts. Besides the already mentioned commercial systems using it, it appears that some effort was exerted to make the Ciao and XSB interfaces mimic it.

The benefit of this high level approach is that it distances the programmer from a great many difficult and time consuming details that he or she would otherwise be required to worry over. As a result of this, the finished code is usually less likely to cause a system wide crash.

5   PROLOG and Relational Databases in NED

The PROLOG components of NED-2 are implemented in LPA WIN-PROLOG, which uses ProData as the underlying mechanism to access databases. However, NED-2 does not make use of relation level ProData predicates. Instead, NED-2's primary data access predicates are special routines written on top of ProData. The important benefit of using these routines is that one need not know a schema in its entirety in order to query a database successfully, nor is this schema required to remain static. Rather, metadata about each database is gathered automatically, and this metadata is used to fill in any blanks left by the user. Another significant advantage of using these routines is that one is not limited to querying a database one relation at a time.

5.1   Specifying Data Sources; Gathering Metadata

The names of data sources to be used with NED-2 are stored in `data_source/1` PROLOG facts. Each such fact stores a single PROLOG string corresponding to a data source name registered with ODBC. When initialized, NED-2 uses these facts to map out the structure of each database.

ProData routines are used to determine the names of tables within a given database and the names of attributes within each table. This information is asserted to `database_table/5` clauses. The clauses have the form

```
database_table( +DataSource,
                +Table,
                +ColumnList,
                +PrimaryKeys,
                +ForeignKeys).
```

where `DataSource` is the name of the database; `Table` is a string representation of the name of a table in the database; `ColumnList` is a PROLOG list containing the names of all attributes in the table: `PrimaryKeys` and `ForeignKeys` store a list of attributes in the table serving as primary and foreign keys, respectively. The latter two arguments are used to specify join conditions during the construction of queries (this is discussed below). Relationships between tables are recorded in `table_relationship/7` clauses, which have the form

```
 table_relationship(+RelationName,
                    +DataSource1,
                    +Table1,
                    +Field1,
                    +DataSource2,
                    +Table2,
                    +Field2).
```

These clauses correspond to the referential integrity constraints specified in the database and are used by PROLOG when forming queries to that database.

It is important to note that ProData can provide only a portion of the metadata existing for a given database. While ProData routines exist which return the names of tables within a database and the names of attributes within a given table, there are no routines returning any information about primary and foreign keys, nor for returning the data types of the attributes (for instance, CHAR or INTEGER) within a table. This is a significant deficiency of the ProData package. Furthermore, though such information is normally stored in the data dictionary of a database itself, not every database calls this table by the same name or allows access to it by non-administrators.

This is indeed a troubling state of affairs. If PROLOG is to utilize a database in a reasonable fashion, it needs to know as much about the database as is possible. If it is to automatically access an arbitrary database, it must be able to retrieve this information. The inability of ProData to provide such information, and the lack of a standard data dictionary, makes this impossible in many cases.

NED-2 uses MS Access databases. Automatic generation of metadata can be had for these simply because NED-2 PROLOG routines have been tailored specifically for them. It is these routines that are used to determine the relationships between tables and to determine the primary and foreign keys within each table. If another type of database is used, then someone must encode the metadata by hand or else rewrite NED-2's source code to process this new format.

5.2   Database Queries in NED-2

One of the special features of NED-2 is the ability to retrieve information from multiple data sources without having to specify, within a query, where the data is to be found. In posing a query, one focuses only on the information itself and is not troubled by inessential details. This sort of transparent access is especially important when the location of data changes over time, or when the nature and availability of the data sources fluctuates.

Regarding the query language itself, it is not SQL; neither is it exactly correct PROLOG syntax. Rather, it grew out of the information storage structures used by the tool-kit used to create NED-2's PROLOG components. This tool-kit, called DSSTools, consists of PROLOG source code routines intended to facilitate the development of blackboard based systems [Zhu95]. Within DSSTools, information is stored on the blackboard in `fact/4` clauses, which have the following structure:

```
fact( +Attribute(+Object, +Value), +Confidence, +Source, +Time).
```

The first argument, sometimes called an *AOV* triple is the substance of the information itself. An example is `area([stand_x], 5)`, which may be interpreted fairly obviously as "The area of stand x is 5." The latter three arguments of a `fact/4` clause constitute metadata about the information—it's quality, where it came from, and the time at which it was recorded.

In order to allow the use of relational databases, the predicate `database_fact/4` has been added to DSSTools. Like `fact/4`, the standard $A(O,V)$ structure is used to represent information. Calls to `database_fact/4` translate a query in $A(O,V)$ format into SQL, which is then directed to a particular database. Metadata about the various data sources available is stored in the internal PROLOG knowledge base, and it is this metadata that is used in the translation process and to determine which database should be queried. ProData routines are used in the actual querying of the database.

So that information stored both internally and externally to PROLOG may be viewed as a unified blackboard, the predicate `known/1` is defined; this calls, alternatively, both `fact/4` and `database_fact/4`.

5.3   An Example Query

As an example query, consider the call

```
 ?- known('STAND_AREA'(['STAND_ID' = 'patch-cut'], Value)).
```

This might be interpreted in English as "Show me the area of the stand called 'patch-cut'." PROLOG first looks to see if there is an appropriate `fact/4` clause on the internal portion of the blackboard satisfying the call. As information of this sort is stored in an external database, there will be no such clause, and so a call is made to `database_fact/4`. Execution of this call proceeds as follows:

1. `'STAND_AREA'` = `Value` is appended to the object list, forming
   `['STAND_AREA'` = `Value, 'STAND_id'` = `'patch-cut']`.
2. Each term in the list is then examined for references to database attributes. These are indicated by the use of LPA strings (These are a data type unique to LPA PROLOG and are denoted by backward quotation marks). If any are found, PROLOG then attempts to find the database and table associated with the given attribute, simply by looking at the metadata stored about each registered database.

   In the present example, there are two attributes mentioned. PROLOG determines that both `stand_area` and `stand_id` are recorded in the table called `stand_header` in the data source 'NED-2 working file'. If, as is not the case here, no suitable database could be found, the query would simply fail.

   If attributes are found in multiple tables, PROLOG will present multiple solutions to the query upon backtracking. (It is important to note, however, that PROLOG keeps track of the primary and foreign keys in each table. If an attribute appears as a primary key in one table, PROLOG will not backtrack to associate the attribute with another table. Thus, referential integrity is maintained.)
3. Attributes set equal to uninstantiated variables are set apart from the rest of the list; these will later be used in the `SELECT` part of the SQL statement.
4. A list of the tables associated with the attributes is kept and is used in the `FROM` part of the SQL statement.
5. The remaining elements of the list—which constitute constraints on the attributes to be selected—will be used in the formation of the `WHERE` part of the SQL statement.
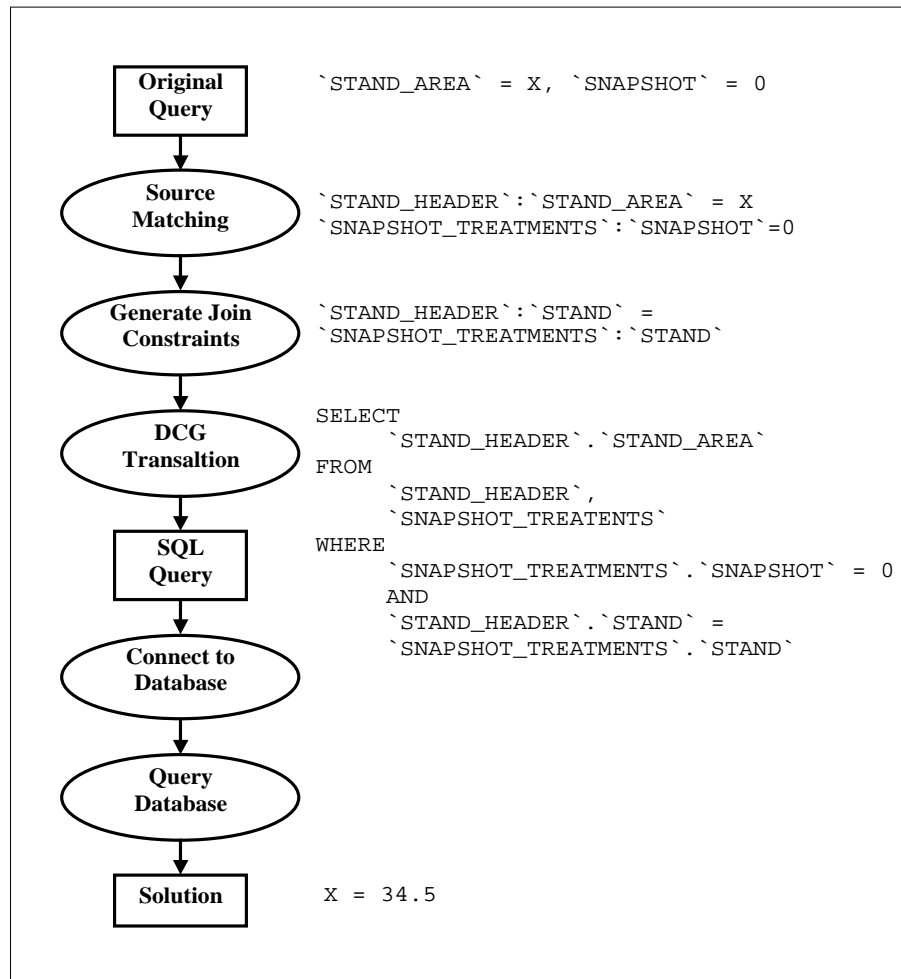
At this point, the attributes to be selected, the list of tables, and the list of constraints are fed to a definite clause grammar which translates them into SQL. In the present case, the resulting SQL query is:

```
SELECT
        'STAND_HEADER'.'STAND_AREA'
FROM
        'STAND_HEADER'
WHERE
        'STAND_HEADER'.'STAND_ID' = 'patch-cut'
```

This query is directed to the 'NED-2 working file' database. If it succeeds, a single value corresponding to the area of the patch-cut stand is returned.

6 Features of the Query Language

What follows is a brief description of prominent features of the the query language used in NED, as well as examples designed to illustrate its capabilities. The general technique is to expose the beneficial characteristics of SQL to PROLOG while not restricting the user to SQL's rigid syntax.

A query for the area of the stand associated with Snapshot 0 (A snapshot is a view of a stand at a give point in time). Note that a join of two tables on the attribute STAND is involved.

Fig. 2. The Query Process

6.1 Arithmetic

The comparators $=$, $<=$, $>=$ can be used in queries, as well as normal arithmetical operators $(+, -, *, /)$. The question, "Show me the species of trees with diameters between 0 and 5", might be expressed in PROLOG as

```
?- known('TREE_SPP'(['TREE_DBH' >= 0,'TREE_DBH' < 4+1],Value)).
```

This becomes the SQL statement

```
SELECT
      'OVERSTORY_OBS'.'TREE_SPP'
FROM
      'OVERSTORY_OBS'
WHERE
      'OVERSTORY_OBS'.'TREE_DBH' >= 0
         AND
      'OVERSTORY_OBS'.'TREE_DBH' < 4+1
```

Note that either $<=$ or $=<$ can be used to denote 'less than or equal to', and that either $>=$ or $=>$ can be used to denote 'greater than or equal to'. These are automatically translated into the proper relation symbols (thus the PROLOG prohibition on $=>$ and $<=$ as relations does not apply).

6.2 Logical Operations

Logical operations can be used: ',' denotes conjunction, ';' denotes disjunction, and '\+' denotes negation, as is normally the case in PROLOG. Scope is indicated in the usual fashion—via the use of parentheses. No operator precedence conventions have been implemented. A sequence such as `a,b,c,d,e` or `a;b;c` is acceptable, but not `a ; b , c ; d`. For example, the query

```
?- known('STAND_ID'([\+('STAND' = 0 ; 'STAND' = 1)],Value)).
```

which might be read as "Show me the names of stands other than stand 0 and 1", becomes the SQL

```
SELECT
      'STAND_HEADER'.'STAND_ID'
FROM
      'STAND_HEADER'
WHERE
      NOT ((
            'STAND_HEADER'.'STAND' = 0
            OR
            'STAND_HEADER'.'STAND' = 1
            ))
```

6.3 Colon Notation

In the original PROLOG query list (as opposed to the SQL translation) either the structure `TableName:Attribute` or `Attribute` can be used to indicate a column identifier. When processed, all attributes will get expanded to the `TableName:Attribute` form. This eliminates possible ambiguities, such as when a given (non-key) attribute appears in more than one table. Multiple occurrences of an unaccompanied attribute string are taken to refer to the same attribute. For instance, the below expression

```
'B' = X, 'T':'A' = Y, 'A' > 5
```

is expanded to

```
'T2':'B' = X, 'T':'A' = Y, 'T':'A' > 5
```

In ambiguous cases, such as

```
'T1':'A' = X, 'T2':'A' = Y, 'A' > 5
```

the most recently encountered table is used:

```
'T1':'A' = X, 'T2':'A' = Y, 'T2':'A' > 5
```

6.4 Aggregates

The Aggregates `MIN`, `MAX`, `COUNT`, `AVG`, `STDEV`, `VAR` can be used. However only a single aggregate can be used in the select portion of the query, and even then it must be the only column selected. This is due to an apparent limitation of the MS Access driver used with ODBC. The query

```
?- known('MIN'('STAND_AREA')([], X)).
```

becomes the SQL

```
SELECT
      MIN('STAND_HEADER'.'STAND_AREA')
FROM
      'STAND_HEADER'
```

When aggregates are used in constraints, they are converted into subqueries. Here they must be phrased so as to return a single value.

```
?- known('STAND'(['STAND' < 'MIN'('STAND_AREA')+5], X)).
```

becomes

```
SELECT
      'STAND_HEADER'.'STAND'
FROM
      'STAND_HEADER'
WHERE
      'STAND_HEADER'.'STAND' <
      (SELECT  MIN('STAND_AREA') FROM 'STAND_HEADER') + 5
```

6.5 Subqueries

Derived tables can be specified using the `subquery(Alias, Query)` expression, where Alias is a string to be used to denote the table in the SELECT statement. In order for the derived table to be of use in the query, some column identifier must make use of it. The subquery is a query list of the usual form; this means that `SELECT` variables are required in the subquery. They cannot be displayed, however, for they would not be returned from the primary SQL query; it is recommended that anonymous variables be used.

```
?- known('MyTable':'STAND'([subquery('MyTable', ['STAND' =_])],X)).
```

becomes

```
SELECT
      'MYTABLE'.'STAND'
FROM
       (SELECT
            'STAND_HEADER'.'STAND'
        FROM
            'STAND_HEADER') 'MYTABLE'
```

6.6 Automatic Joins

As was stated above, included in the metadata stored by PROLOG is a knowledge of the relationships between tables in each database. This is vital if PROLOG is to retrieve accurate results. Specifically, it is necessary if joins are to be created between multiple relations. Were it not for these, any query to multiple relations would return attributes from the Cartesian product of these relations—too much data!

```
?- known('STAND_ID'(['SNAPSHOT' =0],ID)).
```

```
SELECT
      'STAND_HEADER'.'STAND_ID'
FROM
      'STAND_HEADER','STAND_SNAPSHOTS_TREATMENT'
WHERE
      'STAND_SNAPSHOTS_TREATMENT'.'SNAPSHOT' = 0
      AND
      'STAND_HEADER'.'STAND' = 'STAND_SNAPSHOTS_TREATMENT'.'STAND'
```

6.7 IN and BETWEEN

The expressions `in(Attribute, Set)` and `between(Attribute, Min, Max)` can be used in queries, and can either use subqueries or explictly specified value lists as arguments:

```
?- known('TREE_SPP'([in('SNAPSHOT',subquery(['STAND' = X]) )],Spp)).
```

becomes

```
SELECT
      'OVERSTORY_OBS'.'TREE_SPP'
FROM
      'OVERSTORY_OBS',
      'STAND_SNAPSHOTS_TREATMENT'
WHERE
      'STAND_SNAPSHOTS_TREATMENT'.'SNAPSHOT'
      IN ( (SELECT 'STAND_HEADER'.'STAND' FROM 'STAND_HEADER') )
        AND
      'OVERSTORY_OBS'.'SNAPSHOT' =
        'STAND_SNAPSHOTS_TREATMENT'.'SNAPSHOT'
```

6.8   DISTINCT and ALL

The atoms distinct and all can appear at the head of the query list. The
former indicates that only unique solutions are returned. Use of the latter indi-
cates that all solutions should be returned. (The difference between distinct
and all thus parallels somewhat that between setof/3 and findall/3—note,
however, that answers to database queries are still provided on a tuple-at-a-time
basis). The benefit of specifying the restriction in the query itself (as opposed to
using findall/3 or setof/3) is that it is the database management system and
not PROLOG which takes on the computational task of eliminating duplicate
answers.

```
?- known('STAND_ID'([distinct],X)).
```

becomes

```
SELECT  DISTINCT
      'STAND_HEADER'.'STAND_ID'
FROM
      'STAND_HEADER'
```

7   Related Work

The translation of PROLOG expressions into SQL is absolutely vital if informa-
tion is to be retrieved in a timely fashion. While it might take the RDBMS a
second to evaluate a fairly complex query, it might take PROLOG several min-
utes or even longer to produce the same results via its normal fetch, check, and
backtrack search mechanism. Optimization of this sort really is unavoidable if
one intends to build a useful system.

The querying technique just described is quite similar to a language called
TREQL (Thorton Research Easy Query Language) developed some time ago

[Lunn88, 48]. The intention behind the development of that language parallels the development of NED in at least one respect—namely, TREQL permits meaningful queries to be posed to databases despite some ignorance of the underlying database schema. As in NED, the poser of the query need not specify join constraints; TREQL provides these automatically. TREQL, however, is translated directly into PROLOG predicates attached in ProData fashion to database relations. This, as has been said several times already, is an unacceptably inefficient means of querying a database. The developers of TREQL note that the TREQL queries could be translated to SQL rather than PROLOG; however they say that to do so would be "much more difficult." [51].

[Draxler93] describes a PROLOG to SQL translator. Queries can be any complex PROLOG query involving: predicates linked to database relations; the PROLOG equivalents of AND, OR, and NOT; the existential quantifier '^'; arithmetical comparators; and aggregate functions. The top level predicate of the translator is

```
translate(+Projection, +Goal, -SQL).
```

where `Projection` and `Goal` are structures abiding by the above rules. The argument `SQL` is returned bound with the SQL equivalent of the original goal. Like the language used by NED-2, the translator does not allow transparent access to a database (the database goals are isolated from the rest of PROLOG). This is in contrast to BERMUDA and PRIMO; however, a translator such as Draxler's could be used to make a system such as BERMUDA or PRIMO—the translation process would simply be hidden from the user.

In many ways the query language described in [Draxler93] is more expressive than the language described here. However, since database relations there are specified explicitly by PROLOG predicates, a knowledge of the database schema is necessary. Furthermore, for databases containing tables with large numbers of attributes, writing them as PROLOG predicates is tedious and makes uneconomical use of space. Referring to attributes by name is far easier. It is for these two reasons that a technique more closely resembling that proposed in [Draxler93] was not used in NED-2.

The routines described in [Draxler93] are used in both Ciao and XSB implementations of PROLOG [Bueno00, 421ff; Sagnonas00, 82, 85ff, 101ff]. The endorsement of Draxler's technique is telling. Relation based access is not a viable solution.

8   Conclusion

The query language described in the above sections is an essential component of NED-2. What was needed was a means of retrieving information from a database both quickly and without requiring the programmer to possess complete knowledge of the database's schema. Furthermore, what was required was that these queries be succinctly posed from within PROLOG. Though it certainly could be expanded and improved upon, the language described here accomplishes this.

Though it does not allow transparency (in that database relations are not treated as PROLOG facts), this is not considered a horrible loss. Since the databases of NED-2 involve many attributes and underwent frequent changes in their developmental phases, maintaining transparency would have caused severe disadvantages. Particularly, every change to a database schema would require a change in the PROLOG routines designed to access that database.

References

[Brodie88] Brodie, Michael, and M. Jarke. 1988. "On integrating logic programming and databases." In *PROLOG and Databases: Implementations and New Directions*, edited by P.M.D. Gray and R. Lucas. Chichister, West Sussex: Ellis Horwood Limited.

[Bueno00] Bueno, F., D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. 2000. *The Ciao PROLOG System: Reference Manual.* http://www.clip.dia.fi.upm.es/Software/Ciao/ciao.ps (accessed April 2, 2002).

[Ceri89] Ceri, Stefano, G. Gottlob, and Gio Wiederhold. 1989. "Efficient database access from PROLOG." *IEEE Transactions on Software Engineering.* 15(2):153-164.

[Ceri90] Ceri, Stefano, F. Gozzi, and M. Lugli. 1990. "An overview of Primo: A portable interface between PROLOG and relational databases." *Information Systems.* 15(5):543-553.

[Chang 1999] Chang, C., and H. Garcia-Molina. 1999. "Mind your vocabulary: Query mapping across heterogeneous information sources." In *Proceedings of the 1999 ACM SIGMOD Conference.* May 31 - June 3, 1999, Philadephia, Pennsylvania:335-346.

[Date95] Date, C. J. 1995. *An Introduction to Database Systems.* New York: Addison-Wesley Publishing Company, Inc.

[Draxler93] Draxler, Christophe. 1993. "A powerful PROLOG to SQL compiler." Technical report, CIS Centre for Information and Speech Processing, Ludwig-Maximilians-University, Munich. http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/io/pl2sql/pl2sql.tgz (accessed April 2, 2002).

[Gray88] Gray, P. M. D., and R. J. Lucas, eds. 1988. *PROLOG and Databases: Implementations and New Directions.* Chichister, West Sussex: Ellis Horwood Limited.

[Ioannidis89] Ioannidis, Yannis E., Joanna Chen, Mark A. Friedman, and Manolis M. Tsangaris. 1989. "BERMUDA—An architecutal perspecitve on interfacing PROLOG to a database machine." In *Expert Database Systems, Proceedings From the Second International Conference*, April 25-27, 1988, Vienna, VA. Edited by Larry Kerschberg. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc.

[Ioannidis94] Ioannidis, Yannis E., and Manolis M. Tsangaris. 1994. "The design, implementation, and performance evaluation of Bermuda." *IEEE Transactions on Knowledge and Data Engineering.* 6(1):38-56.

[Irving88] Irving, T. 1988. "A generalized interface between PROLOG and relational databases." In *PROLOG and Databases: Implementations and New Directions*, edited by P. M. D. Gray and R. Lucas. Chichister, West Sussex: Ellis Horwood Limited.

20

[Kerschberg86] Kerschberg, Larry, ed. 1986. *Expert Database Systems, Proceedings From the First international Workshop*, October 24-27, 1984, Kiawah Island, SC. Menlo Park, CA: Benjamin/Cummings Publishing Company, Inc.

[Kerschberg89] Kerschberg, Larry, ed. 1989. *Expert Database Systems, Proceedings From the Second International Conference*, April 25-27, 1988, Vienna, VA. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc.

[Lucas97] Lucas, Robert, and Keylink Computers, Ltd. 1997 *ProData Interface Manual.* Kenilworth, UK: Keylink Computers Ltd.

[Lunn88] Lunn, K; and I. G. 1988. "TREQL (Thorton Research Easy Query Language: An intelligent front-end to a relational database." In *PROLOG and Databases: Implementations and New Directions*, edited by P.M.D. Gray and R. Lucas. Chichister, West Sussex: Ellis Horwood Limited.

[Microsoft97] Microsoft Corporation 1997. *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference.* Vol. 1. Redmond, WA: Microsoft Press.

[Napheys89] Napheys, Ben, and Don Herkimer. 1989. "A look at loosedly-coupled PROLOG/database systems." In *Expert Database Systems, Proceedings From the Second International Conference*, April 25-27, 1988, Vienna, VA. Edited by Larry Kerschberg. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc.

[Nute99] Nute, Donald, Geneho Kim, Walter D. Potter, Mark J. Twery, H. Michael Rauscher, Scott Thomasma, Deborah Bennett, and Peter Kollasch. 1999. "A multi-criterial decision support system for forest management." In *Environmental Decision Support Systems and Artificial Intelligence*, AAAI-99, Technical Report WS-99-07, Menlo Park CA: AAAI Press. 74-81.

[Potter02] W. Potter, D. Nute, F. Maier, M. Twery, M. Rauscher, P. Knopp, S. Thomasma, M. Dass, and H. Uchiyama. 2002. "The NED IIS Project—Forest Ecosystem Management," to appear in *Proceedings of the IFIP World Computer Congress WCC2002— Intelligent Information Processing (IIP-2002)*, August 25-30, 2002, Montreal, Canada.

[Rauscher99] Rauscher, H. M. 1999. "Ecosystem management decision support for federal forests in the United States: A review." *Forest Ecology and Management.* 114:173-197.

[Rauscher00a] Rauscher, H. M., F. Thomas Lloyd, David Loftis, and Mark J. Twery. 2000. "A practical decision-anaysis process for forest ecosystem management." *Computers and Electronics in Agriculture.* 27:195-226.

[Rauscher00b] Rauscher, H. M., Richard E. Plant, Alan J. Thomson, and Mark J. Twery. 2000. Forward to *Computers and Electronics in Agriculture.* 27:1-6.

[Sagnoas00] Sagonas, Konstantinos, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Steve Dawson, and Michael Kifer. 2000. *The XSB System V2.2.* Vol 2. http://www.cs.sunysb.edu/ sbprolog/manual2/index.html (accessed April 2, 2002).

[Sciore86] Sciore, Edward, and David S. Warren. 1986. "Toward an integrated database-PROLOG system." In *Expert Database Systems, Proceedings From the First International Workshop*, October 24-27, 1984, Kiawah Island, SC. Edited by Larry Kerschberg. Menlo Park, CA: Benjamin/Cummings Publishing Company, Inc.

[Shalfield01] Shalfield, Rebecca. 2001. *Win-PROLOG Programming Guide.* Lonodon: Logic Programming Associates Ltd.

[SICS02] Swedish Institute of Computer Science. 2002 *SICStus Prolog User's Manual.* Kista, Sweden: Swedish Institute of Computer Science.

http://www.sics.se/isl/sicstuswww/site/documentation.html (accessed April 2, 2002).

[Singleton93] Singleton, Paul, and Pearl Brereton. 1993. "Storage and retrieval of first-order terms using a relational database." In *Advances in Databases, Proceedings of the 11th British National Conference on Databases*, July 7-9, 1993, Keele, U.K. Edited by Michael F. Worboys and A. F. Grundy.

[Twery00] Twery, Mark J., H. M. Rauscher, D. J. Bennett, S. Thomasma, S. Stout, J. Palmer, R. Hoffman, D. DeCalesta, E. Gustafson, H. Cleveland, J. M. Grove, D. Nute, G. Kim, and R. P. Kollasch. 2000. "NED-1: Integrated analysis for forest stewardship decisions." *Computers and Electronics in Agriculture*. 27:167-193.

[Ullman89] Ullman, Jeffrey D. 1989. *Principles Of Database and Knowledge-Base Systems*. Vol. 1. Rockville, MA: Computer Science Press, Inc.

[Venken88] Venken, R., and A. Mulkers. 1988. "The interaction from PROLOG to a binary realtional database." In *PROLOG and Databases: Implementations and New Directions*, edited by P.M.D. Gray and R. Lucas. Chichister, West Sussex: Ellis Horwood Limited.

[Zhu95] Zhu, Guojun. 1995. "DSSTOOLS: A toolkit for development of decision support systems in PROLOG." M. S. thesis, University of Georgia.

[Zaniolo86] Zaniolo, Carlo. 1986. "PROLOG: A database query language for all seasons." In *Expert Database Systems, Proceedings From the First International Workshop*, October 24-27, 1984, Kiawah Island, SC. Edited by Larry Kerschberg. Menlo Park, CA: Benjamin/Cummings Publishing Company, Inc.