Frederick W. Maier
Notes on a Blackboard: Recent work on NED-2
(Under the direction of Donald Nute)

The following paper describes recent work on NED-2, an ecosystem management decision support system 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. Integrating third-party simulation and visualization packages, NED-2 allows the user to plan, predict, and assess treatments. It is a blackboard system, with agents implemented in PROLOG. Graphical interface, inventory, and plan creation modules are implemented in C++. A relational database is used as primary storage. NED-2 is a second generation product, building upon NED-1.

The paper addresses three issues. First, a blackboard integrating PROLOG with a relational database was created for NED-2; this is discussed. Second, the creation of domain control modules to accommodate the more sophisticated conceptual scheme of NED-2 is described. Third, techniques used in the generation of NED-2 reports are presented.

Index words: Ecosystem Management, Decision Support Systems, NED, Blackboards, PROLOG, Relational Databases, SQL

Notes on a Blackboard: Recent work on NED-2

by

Frederick W. Maier

B.A., Spring Hill College, 1996

M.A., Tulane University, 1999

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2002

© 2002
Frederick W. Maier
All Rights Reserved

Notes on a Blackboard:
Recent work on NED-2

by

Frederick W. Maier

Approved:

Major Professor: Donald Nute

Committee: Walter D. Potter
Charles Cross

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
May 2002

Acknowledgments

The pain and embarrassment of writing a thesis is only exceeded by that of proofreading and otherwise bearing witness to its production. I thank Dr. Nute, my major professor, and Drs. Potter and Cross, members of my committee, for subjecting themselves to it.

I thank my friends and classmates at the AI center—especially Vassi, Cartic, Anil, Nelson, Swaroop, Dev, Mayukh, Hajime, Jin, Sasa, Chris T., David B., Yunxiu, David C., B. C., and Shulei—for attempting to keep me relatively sane during my (extended) stay in the program. Thanks too to Angie Paul, for looking out for everyone.

I thank Drs. Twery and Rauscher, and Scott Thomasma and Pete Knopp—members of the Forest Service who allowed me to work with them on the NED project and who were also so very nice to work with.

Thanks to Scott Bitterman—for telling me about the AI program in the first place, and for being a great friend in general; and to my family members, who couldn't escape me even if they wanted to (and generally didn't want to).

Profound thanks to M. E. Baroco—What I owe to you exceeds any space allowed. Please know I would pay it back in kind.

Table of Contents

Page

Acknowledgments iv

List of Figures viii

List of Tables ix

Chapter

1 An Introduction to NED 1

1.1 Overview 1

1.2 A Brief History of NED 3

1.3 The NED Decision Process 5

1.4 Desired Future Conditions 6

1.5 NED Inventory 6

1.6 NED-1 Vs. NED-2 7

1.7 Other Ecosystem Systems 9

1.8 Conclusion 11

2 Blackboards, DSSTools, and NED 12

2.1 Introduction 12

2.2 Blackboards 12

2.3 DSSTools 13

2.4 The Current DCM Structure of NED-2 15

2.5 Report Generation in NED-2 16

3 PROLOG and Relational Databases: Background 24

3.1 Introduction 24

3.2 The Similarities Between PROLOG and Relational Databases 25

3.3 Reasons for linking PROLOG to a RDBMS 29

3.4 Stumbling Blocks 30

3.5 Techniques of Integration 31

3.6 Real World Systems 36

4 PROLOG and Relational Databases in NED 40

4.1 specifying data sources; loading metadata 40

4.2 Connecting to a database 42

4.3 Other Predicates 43

4.4 ODBC_PL: Augmenting ProData 43

4.5 Querying NED-2 Databases 45

4.6 Examples 46

4.7 Related Work 55

4.8 Conclusion 56

5 Conclusion and Future Directions 58

5.1 An Internal Representation 58

5.2 Integration of External Sources: The Constraint Problem 61

Appendix

A Database Query Benchmarks 64

A.1 Test Descriptions 66

A.2 Test Results 67

B Database Query Caching 69
B.1 The Caching Routines 69
B.2 Caching Benchmarks 73
C ODBC_PL: Reference Manual 75
C.1 The Structure of an ODBC Application 75
C.2 The ODBC_PL Library 77
Bibliography 88

List of Figures

2.1 NED-2 Graphical Interface 15

2.2 PROLOG/C++ Interaction in NED-2 16

2.3 NED-2 Report DCMS and Control Flow 17

3.1 Relational Level Access 34

3.2 View Level Access 35

4.1 The Query Process 49

5.1 A Simple Means of Data Integration 63

C.1 ODBC Application Architecture 76

List of Tables

A.1 Average Case Scenario: Internal Predicates 68

A.2 Average Case Scenario: External Predicates 68

A.3 Worst Case Scenario: Internal Predicates 68

A.4 Worst Case Scenario: External Predicates 68

B.1 Cache Tests 74

Chapter 1

An Introduction to NED

1.1 Overview

This paper describes recent work on NED-2, a decision support system for ecosystem management currently in development by the U.S. Forest Service (in conjunction with the University of Georgia Artificial Intelligence Center). NED-2 allows the analysis of forest inventories to determine the degree to which they satisfy a set of preselected goals. Goals pertain to the production of timber, but also to water quality, aesthetics, wildlife habitat, and ecology. In addressing goals of such diverse natures, NED distinguishes itself from 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, 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 are implemented in the PROLOG programming language. Chapter Two of this paper gives some background on blackboard systems and explains NED-2's current architecture. Some space is devoted to describing HTML report generation.

The majority of this paper, however, is devoted to the methods used in NED-2 of coupling PROLOG to relational databases. As a set of relational databases is used as NED's primary storage medium, and as NED's goal analysis and report

generation modules are implemented in PROLOG, the means of linking PROLOG to these databases is of central importance in NED-2. Chapter Three discusses the art of interfacing PROLOG to relational databases in general and explains the necessity of the database query language created especially for NED-2. This language is described in Chapter Four. Chapter Four also presents the mechanisms for 'registering' databases in PROLOG, which allows for their transparent use.

Three appendices to this paper are also devoted to PROLOG-RDBMS interaction. The first shows the results of tests determining access times to databases via various means; the tests indicate that some methods of access are better than others. The second describes the method used for caching the results of database queries in PROLOG's memory. In some cases, caching results considerably increases performance. The third describes the predicates of `ODBC_PL`, a library allowing PROLOG to utilize Microsoft's Open Database Connectivity API.

Since its earliest days, NED has been intended as a platform facilitating the integration of diverse third-party forestry products. In this way, the pre-existing tools used to solve pieces of a very hard problem can be joined into an organic whole. Chapter Five, the most exploratory of the chapters, touches upon ways of painlessly integrating these heterogeneous sources of knowledge.

The remainder of this first chapter is devoted to recounting the history of NED. Particularly, a few of the differences between NED-1 and NED-2 are discussed. It will be seen that NED-2 comes far closer than NED-1 in realizing the original vision of project members.

A sketch of a few other software systems relevant to the NED project is also presented. None of these systems fills quite the same niche as NED-2. However, as they all present methods of balancing competing forestry objectives, they may be considered similar to NED in spirit. The sketch is given primarily to inform the reader of a few other products in the field.

1.2 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[1]) 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',[2] was chosen [Twery00, 168].

The integration of independent and often incompatible programs and data stores—so-called *heterogeneous data sources*—is currently a very important topic in computer science; it is by no means confined to the forestry domain. Neither, it must be stressed, is the integration problem trivial. Nevertheless, such unification is a primary goal of NED [Twery00, 186-187, 189].

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 the 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

[1]The Northeastern Research Station, in Burlington, VT, is one of seven such stations of the Forest Service.

[2]As the scope of the project has expanded beyond the northeast, it is generally now said that 'NED' stands for nothing, just as 'SQL' once stood but no longer stands for 'Structured Query Language'.

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].

1.2.1 Ecosystem Management

The ecosystem management paradigm has become influential over the last ten to fifteen years. In 1993 and at the behest of Vice President Al Gore, an Interagency Ecosystem Management Task Force was formed. It was charged with analyzing the desirability and feasibility of what they called *the ecosystem approach*:[3]

> The ecosystem approach is a method for sustaining or restoring natural systems and their functions and values. It is goal driven, and it is based on a collaboratively developed vision of desired future conditions that integrates ecological, economic, and social factors....
>
> The goal of the ecosystem approach is to restore and sustain the health, productivity, and biological diversity of ecosystems and the overall quality of life through a natural resource management approach that is fully integrated with social and economic goals [Interagency95].

[3] The same document describes an ecosystem as "an interconnected community of living things, including humans, and the physical environment within which they interact." The needs of humans are thus an inherent part of the equation.

The approach has been adopted as the party line of the Forest Service [Rauscher00a, 196] and a great many other federal organizations (see "Memorandum of Understanding to Foster the Ecosystem Approach" [OEP95]). In a recent speech, the chief of the Forest Service said:

> We have come to realize that without healthy ecosystems, we cannot sustain the products and uses that our communities need for their health and stability. Our central mission in managing the national forests and grasslands has shifted from producing timber, range, and other outputs to restoring and maintaining healthy, resilient ecosystems. [Bosworth01]

Within the paradigm, human beings—with their social and economic needs—are considered an integral part of the system.[4] Humans are not the only part, however; their needs must be viewed in the context of the larger system.

It is within this context, too, that systems such as NED should be viewed. From a certain perspective, such systems are the embodiment of the ecosystem approach—goal driven and combining complex, competing objectives.

1.3 The NED Decision Process

Development of the NED software has been guided by the perception that forest management is fundamentally a goal-driven process [Nute00; Rauscher00a; Twery 2000]. This perception has led to what is called the *NED Decision Process*, the steps of which are outlined below [Rauscher 2002a]:

1. Select Goals
2. Assess Inventory
3. Design Alternative Courses of Action

[4] The report of the interagency task force is subtitled "Healthy Ecosystems *and* Sustainable Economies" The italics appear in the document itself.

4. Forecast the Future (through simulation)
5. Evaluate Goal Satisfaction
6. If not satisfactory, go back to step 1

The list itself is intuitively reasonable—seeming to apply to many decision processes that might come to mind. The first two steps are the most important (and it is debatable whether the order of the two can, or should, be maintained in every case). If one lacks goals, one can form no meaningful plan of action; the same can obviously be said about a lack of data. If one does not know where one currently is, one cannot know where one can go.

1.4 Desired Future Conditions

If humans use a decision process akin to the one sketched above, it is practiced in a slapdash fashion. For a computer, however, all steps must be made formal and rigorous. Crucially, the goals themselves must be formulated in such a way that the computer can easily determine whether and to what degree they have been met. In the case of NED, committees of experts were formed to determine a set of goals (such as enhancing biodiversity) appropriate for inclusion in a forest management support system [Twery00, 172]. These goals were then translated into logical complexes of *desired future conditions*. The latter are measurable quantities or states, such as 'canopy_closure = 80%'. They convert the original goals, which are abstract and lacking in clear meaning, into criteria that are quantifiable or at least observable. Without such a translation, analysis of goal satisfaction is impossible.

1.5 NED Inventory

NED-1 and NED-2 are project-level management systems, as opposed to systems operating at the regional and forest levels [Rauscher99, 186; Rauscher00a, 197]. The

differences between the three levels amount to differences in the size of land and time frames considered, and in the specificity of actions proposed. Forest plans might apply to 200,000 to 500,000 hectares of land, and set forest-wide requirements [Rauscher00a, 197]. They do not specify in any great detail how these requirements are to be met. Project level systems are applicable at much smaller scales (a few hundred to a few tens of thousands of hectares) and are responsible for devising specific activities to be applied to the land [Rauscher00a, 197; Nute00, 74].

For NED-1 and NED-2, users are required to provide summary information about the management unit as a whole,[5] about stands within the management unit, and about plots within stands. Tree observations are made for overstory and understory plots; observations of ground vegetation are also made [Twery00, 178-179]. The information provided by the user is used in NED-1 and NED-2 for goal analysis and in the generation of summary reports. Great effort has been made to keep the amount of information required from the user to a minimum. Derivative information is calculated by the system whenever possible [Twery00, 183].

1.6 NED-1 Vs. NED-2

NED-1 and NED-2 are alike in that they are both comprised of a mixture of C++ and PROLOG components. In both, PROLOG based inference engines are used to perform goal analyses. The two systems differ, however, in three significant respects. First, inventory information in NED-1 is stored on disk in a proprietary flat file format. This is controlled by a C++ component called the *data manager*. Internally, information is stored as C++ objects [Twery00, 183]. PROLOG components, in order to retrieve data, are required to communicate with the data manager via an

[5] The management unit can be considered to be the largest unit of land under consideration; it consists of multiple stands of trees. It might consist of a few hectares or several tens of thousands [Nute00, 76]

intermediary, the *Logic Server*. This component is based on work found in [Chen96]. As communication between PROLOG and C++ components must go through this intermediary, a potential bottleneck exists.

In NED-2, this triangle has been abandoned; data in NED-2 are stored in a family of relational databases. This off-loads to an external system (in this case, MS Access) much data manipulation work that would otherwise be done by custom C++ routines. The move also allows PROLOG components (or any other component able to use Microsoft's ODBC library[6]) to have direct access to the data.

The second difference between the two systems is that, while many of the components of NED-2 are still written in C++, the PROLOG components play a more active role in program execution than did their NED-1 counterparts. Particularly, execution of the C++ modules is controlled by PROLOG via interaction with a C++ module called the *PnP* (this is short for *Plug and Play*). Messages indicating user activities are sent by the PnP to PROLOG. In response, PROLOG can inform the PnP which of the other modules to run.

The third and most significant difference between the two systems lies in their representational and functional capabilities. NED-1 performs goal analysis and provides reports only on initial inventory data. Though it can export data to simulators (and in many cases import such data), the process of forecasting future states of the management unit is very difficult. NED-2, in contrast, allows for the forecasting of future conditions based upon user created treatment plans—the forecasting is possible by running simulations of the plans using external programs. Goal analysis can be performed on these simulated states.

Thus, while NED-1 deals with stands at a single point in time, NED-2 deals with *snapshots* of stands over a period of time. The addition of the temporal dimension

[6]ODBC, which stands for Open Database Connectivity, allows applications access to any database system fitted with an ODBC driver. ODBC is discussed later in this paper.

and the ability to create plans greatly increases the power and utility of the program. It also, unfortunately, increases complexity. This is one reason why using a relational database system is preferable to storing data in a flat file. It is the capability of the program to incorporate data generated by simulators, thereby allowing the user to model land management over periods of time that brings NED-2 closer to project members' ideal of a single program bringing to bear the full spectrum of forestry knowledge.

1.7 Other Ecosystem Systems

[Rauscher99] provides an extensive list of decision support systems used in forestry. Below, however, are a few systems of some note (EMDS is listed in [Rauscher99]; the others are not). The systems do not bear much similarity to NED from an architectural standpoint, nor are they intended to fulfill quite the same role. However, they all in some sense are ecosystem management systems.

1.7.1 EMDS

Ecosystem Management Decision Support (EMDS) is a framework for the development of knowledge based systems utilizing GIS [Reynolds97]. It incorporates ArcView, a standard GIS, and NetWeaver, a shell for the development of logical dependency networks. EMDS is purported to be suitable for "ecological assessments at any geographic scale" [Reynolds97, 1]. It must be stressed that EMDS is a *framework* for creating decision support systems—it is the developer's responsibility to define data objects and specify logical relationships between them. In other words, the user is entirely responsible for representing knowledge in the system. This is not an easy task and likely serves as an obstacle to using EMDS. A particularly

interesting feature of EMDS is its ability to recognize missing pieces of information and evaluate their impact [Reynolds97, 2].

1.7.2 LEEMATH

Landscape Evaluation of Effects of Management Activities on Timber and Habitat (LEEMATH) is a system geared toward analyzing alternative management plans [Li00, 263]. The version described in [Li00] deals only with timber and wildlife habitat goals. However, economic, water quality, and social considerations are to be included in later versions [Li00, 266]. LEEMATH is written in FORTRAN, deals with the Southeast United States, and integrates GIS, growth and yield models (one for pine, one for hardwood) and expert systems (for habitat substantiality) [Li00,267].

1.7.3 SEIDAM

The System of Experts for Intelligent Data Management (SEIDAM) is a system for integrating remote sensing imagery, GIS data, growth and yield models, as well as field data [Goodenough97; Bhogal96]. A Notable feature is its ability to extract data from GIS information and images to update inventories (and vice versa). Agents are written at least partially in PROLOG. Planning modules and inference engines are utilized.

1.7.4 ECHO

The Echo Planning System (Echo) is a decision support system created in the period 1994-1997 [McGregor01, 16]. It is designed to generate and evaluate management strategies, with emphasis on balancing timber and non-timber objectives [McGregor01, 16]. The design philosophy appears to coincide with that of the

NED project: "Forests need to be managed to ensure that they will be productive, while still maintaining ecological balance and social and environmental values" [McGregor01, 20]. Users are allowed to set weights upon various objectives. ECHO is composed of three components, each geared toward a different operational scale (e.g. regional, forest level, project level [Rauscher99, 186]). Only the last of these levels appears comparable to the scale on which NED-2 operates.

1.8 Conclusion

Ecosystem management is becoming simultaneously more difficult and more important. Recently, there has been increased awareness at the governmental level of this fact. NED-2 is a decision support system intended to help users manage their land according to complex and competing objectives. It is goal driven and capable of integrating data sources of a diverse nature. The following chapters delve in greater detail into what enables NED-2 to fulfill its function. Special consideration is given to the techniques used in incorporating relational databases into NED.

Chapter 2

Blackboards, DSSTools, and NED

2.1 Introduction

NED-2 is a blackboard system, the blackboard design paradigm allowing for the high degree of modularity necessary to cast NED as the unifier of many separate pieces. The present chapter gives an overview of blackboard systems and describes the particular components currently constituting NED-2. DSSTools, a toolkit designed at the University of Georgia for the development of blackboard based systems and the kit used in the creation of NED-2, is presented.

2.2 Blackboards

A blackboard system is said to consist of three components [Ni89; [Englemore86]:

- A set of *Knowledge Sources*—in today's terminology, *knowledge agents*—which are software routines designed to perform a specific function or solve a particular problem.

- A *Blackboard*, which is a common store of information. Agents have access to all information on the blackboard; the results of their actions are posted to the blackboard.

- A *Control Mechanism.* Agents in the system monitor the current state of the blackboard and act when they see fit. However, it is common that some control

mechanism orchestrates the activity of groups of agents. Hopefully, the system is guided to a goal state.

Within the blackboard paradigm, agents do not communicate directly with each other. Rather, they interact solely by reading from and posting information to the blackboard. This design constraint ensures modularity. The activity of any one agent does not depend on the existence or nonexistence of any other agent.

The blackboard model is useful simply because many real world problems (forest management, for instance) are highly complex and difficult to solve, yet at the same time can be compartmentalized into subproblems. While it is nearly impossible to tackle the problem as a whole, each of the subproblems is individually tractable. Furthermore, such compartmentalization sidesteps the difficult task of planning, at every step, how the problem is to be solved [Englemore86, 14-15].

The blackboard paradigm fits nicely with the original intention of the NED project—the integration of pre-existing software products and techniques. In reviewing over two dozen support systems used in forestry, [Mowrer97] and [Rauscher99] point out that each of the systems dealt with some part of the forest management problem, but no one system was able to deal with the problem as a whole [Rauscher99, 185].

2.3 DSSTools

Decision Support System Tools (DSSTools) is a library of open source software routines intended to make designing a finished knowledge-based system easier [Zhu95]. Being a kit, the users of DSSTools will invariably be software developers; particularly, they must understand and be able to write PROLOG code. However, as the source code is available, DSSTools can be readily modified to meet the needs of a particular project.

The blackboard of a DSSTools project is the working memory of PROLOG.[1] Routines exist for:

- Implementing any number of agents, called *domain control modules* (DCM's);
- Reading from and writing to the blackboard;
- Invoking forward and backward chaining inference processes.

In the case of NED-1 and NED-2, the rules used by the inference engines specify the conditions for goal satisfaction.

The domain control modules of a DSSTools application operate on a sequential basis. Only one DCM can act at a given time; all others wait until the one has finished. In this sense, the agents of DSSTools can be considered *polite*. Each DCM is simply a PROLOG rule having `dcm(X)` as a head, and so the structure of a DCM is limited only by the limitations of the PROLOG language itself.

A DSSTools application works by repeatedly calling the rule `dcm/1`. As with any PROLOG program, some of the rules might succeed; some might not. The order of DCM execution is generally determined by a *request stack*, where the topmost request indicates a task that should be completed next. A DCM, seeing that it can process the request, takes control of the program. When the DCM has finished, it removes the request from the stack and relinquishes control).[2] The request forms a precondition for the success of a given DCM rule. When a rule is processed, if a request exists to which that DCM can respond, it is said to *fire* or *have been activated* [Ni89, 12].

[1]In implementing NED-2, however, the blackboard was expanded to include NED's relational databases.

[2]It should be said that the contents of the request stack can be modified by each DCM; it is not the case that DCM's can only push and pop requests from the stack. Furthermore, the stack can be ignored by DCM's. A given DCM needn't be programmed to fire only if it sees a particular request on the blackboard.

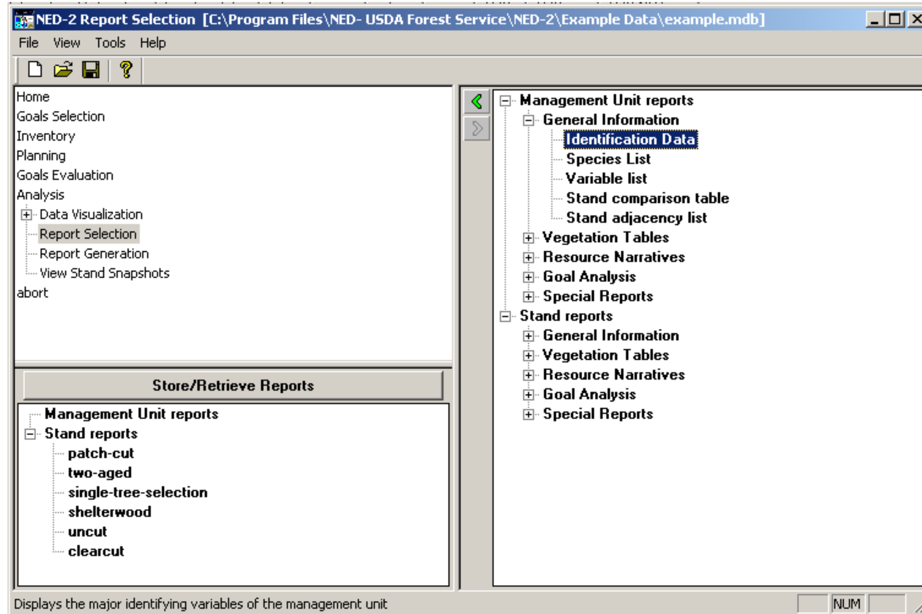


Figure 2.1: NED-2 Graphical Interface

Information is stored on the DSSTools blackboard in the form of *facts* (the structure of these is described in further detail in Chapter Four). Forward and backward chaining engines use these facts in conjunction with logical rules in order to derive further facts. DCM's may view the facts directly, and may invoke any of the inference engines.

2.4 The Current DCM Structure of NED-2

Program control in NED-2 lies mostly in the hands of PROLOG. When the user starts NED-2, it is a PROLOG executable file that is double-clicked, and it is a PROLOG program that is responsible for initializing the rest of the system. Particularly, the PROLOG program initializes the PnP module.[3] The PnP is responsible for presenting to the user the three-paned window shown in Figure 2.1.

[3] The PnP is a dynamic linked library. If PROLOG is the brains of NED-2, then the PnP is it's heart; as it is responsible for the graphical interface and for linking C++ modules to PROLOG, absolutely nothing could be done without it.

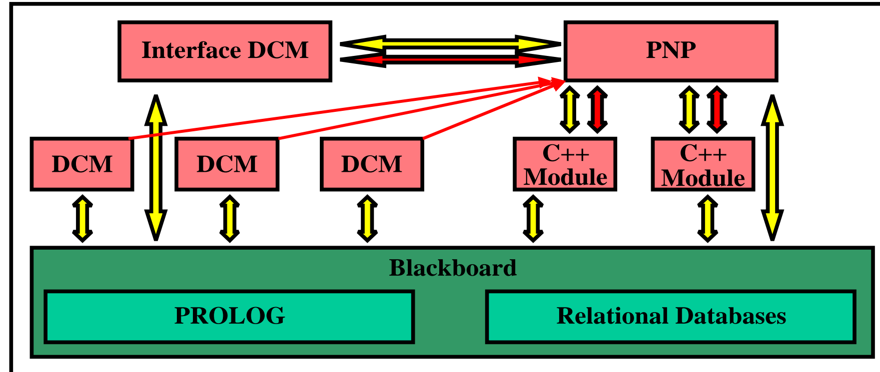


Figure 2.2: PROLOG/C++ Interaction in NED-2

The contents of the top left pane of the window, called the *A-pane*, are controlled by PROLOG. Through varying the contents, PROLOG can interact with and guide the user. When the user selects an item from the A-pane, a message is sent to PROLOG by the PnP. This constitutes the sole means by which the PnP communicates information to PROLOG.

2.4.1 The Interface DCM

A single DCM, called the Interface Module, is responsible for monitoring the A-pane. When NED-2 starts and shortly after PROLOG has initialized the PnP, a request to interact with the PnP is written onto the blackboard. When the interface module fires, it waits for a simple string message from the PnP. Using this string and a look up table, the interface module determines which tasks are to be performed next. It posts requests for them onto the blackboard and then relinquishes program control.

2.5 Report Generation in NED-2

The next DCM to run depends upon the user's selection and the current state of the program. DCM's exist for performing analyses on goals selected by the user, for

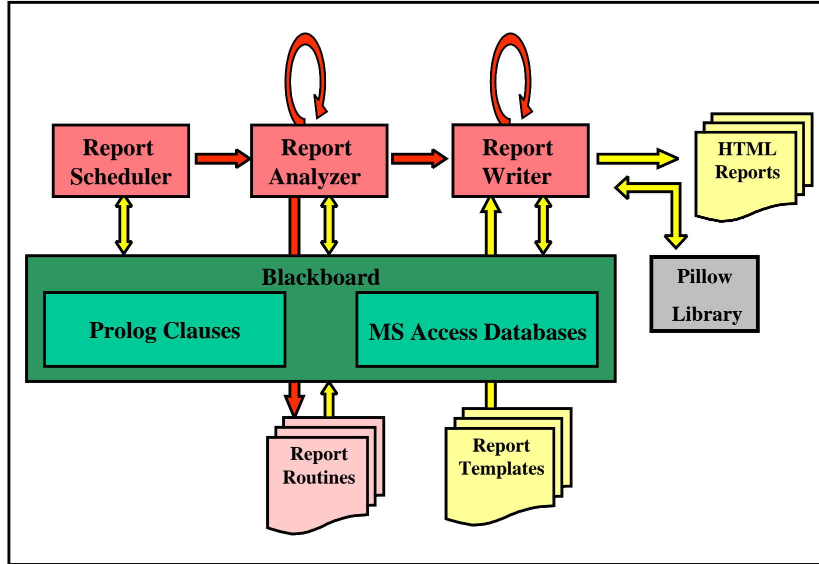


Figure 2.3: NED-2 Report DCMS and Control Flow

generating a list of patches present on the management unit,[4] and for controlling simulators such as FVS. The present section discusses the domain control modules involved in report generation.

Reports in NED-1 were generated by C++ routines using custom written templates. The finished reports were displayed in a window of the graphical user interface and could be exported to an HTML file specified by the user. As the common store of information lay on the C++ side of the program, producing reports in this manner made sense.

In NED-2, the above process has been abandoned. Instead, three PROLOG domain control modules are responsible for gathering the information needed for the reports and directly writing the reports in HTML format. Figure 2.3 indicates the general flow of program control.

[4]Patches are groups of contiguous stands sharing similar properties. in NED-2, patches can be based on forest type, size class, or canopy closure.

Using the PnP, the user selects the names of reports to generate. The selections are then stored in a table of the working database. PROLOG has nothing to do with these actions. However, selecting 'Report Generation' from the A-pane causes a message to be sent to the Interface DCM. The interface module, using its lookup table, places onto the blackboard requests that reports be scheduled and written. It then exits.

2.5.1 The Report Scheduler DCM

A DCM exists for scheduling reports. Since a request for such a task is at the top of the request stack, it is this DCM that fires next. It's purpose is to retrieve from the working database a list of reports to generate.[5] Specifically, it retrieves the following information from the database: The identification number of the stand to which the report pertains; the name of the report to be generated; a list of user selected options for each report (such as whether species should be displayed by common name or by Latin name). The DCM also sorts the reports to be generated by a key stored in a separate database; this key indicates the order in which reports are to be displayed to the user. A unique integer is then assigned to each report—a necessary step, since it is possible for the user to select two reports of the same name and for the same stand but with different options. By assigning a unique identifier to each report, it is possible to distinguish facts generated for one report from facts generated for another.

Once the above information has been collected, the report scheduler requests that the reports be generated. A separate request exists for each report, which allows the program to recover from an error encountered while generating a single report. Each request is of the form:

[5]This information is stored in the 'Reports' relation in the database. Each tuple in the relation corresponds to a single report to be written.

```
selected_report([   ReportNumber,
                    StandID,
                    ReportName,
                    ReportOptions])
```

Once these requests have been pushed onto the stack, the report scheduler gives up control of PROLOG execution.

2.5.2 The Report Analyzer DCM

Seeing that `selected_report/1` clauses are now on the blackboard, the report analyzer DCM will execute. It is responsible for retrieving and formatting the data to be presented in each report. Note that the report analyzer takes control of the program for each `selected_report/1` clause on the blackboard and relinquishes control after it has finished with the report specified in that clause. In this way, it is possible for any number of other DCM's to fire between reports. For instance, perhaps it is necessary that some inference engine be run. In such a case, the report analyzer would push a request for the inference onto the stack; since the `selected_report/1` clause still exists behind this new request, the report analyzer will eventually run again. After the report analyzer has completed its job for a given report, it removes a `selected_report/1` clause from the blackboard and then shuts down.

Currently, the processing of report data is of a fairly procedural nature. For each report in NED-2, a separate PROLOG file exists containing routines for processing the data needed by the report. As there are roughly 50 such reports in NED-2, there are roughly 50 PROLOG files. Each of these files contains a single rule called by the report analyzer; it is this rule that calls all of the other routines in the file. This rule's head is:

```
ned_report(+ReportName(+StandNumber))
```

`ReportName` and `StandNumber` are the same as found in the `selected_report/1` structure.

When a piece of information is stored by these routines, the DSSTools predicate `update_fact/4` is used. It is vitally important that these facts are indexed by the current report number and the source of the fact is listed as `report_generator`. For example, the clause

```
update_fact(
            mu_ident_table([current_report_number(2)],String),
            1,
            report_generator
            ).
```

saves a fact to be used in the second report to be written. These facts are used when the final HTML report file is created.

When the `ned_report/2` clause exits (successfully or not), the report analyzer asserts a clause to the blackboard indicating that all analysis for that report has been completed. This clause has the form

```
generated_report(   +ReportNumber,
                    +StandID,
                    +ReportName,
                    +Options,
                    +OutputFile)
```

In general, the arguments are the same as those found in `selected_report/1`. The exception, `OutputFile`, is the name selected by the report analyzer for the file to which the report will eventually be written. This name is simply the concatenation of the report name, the stand number, and the report number. As the report number is unique, the file name will be unique. If the analysis does not exit successfully, the output file name is set to `error`.

After the assertion of such a clause, the report analyzer exits. It may, however, be the very next DCM to run.

2.5.3 The Report Writer DCM

When all `selected_report/1` requests have been removed from the blackboard, a request that the reports be written to HTML and displayed to the user will (likely) be at the top of the stack—specifically, the atom `write_html` will be present. The Report Writer DCM is responsible for taking the data generated by the report analyzer and actually writing it to an HTML file.

The Report Writer works in three phases. (1) It collects the `generated_report` clauses asserted to the blackboard and, for each, pushes a `report_to_write/1` clause onto the request stack. It also pushes the request `table_of_contents` onto the stack (this request is buried under the `report_to_write` requests). It then gives up program control, perhaps permitting another DCM to fire. (2) When the DCM fires again, it produces an HTML file for each `report_to_write/1` clause, retracting each clause as it proceeds. (3) When all such requests have been removed, a table of contents is written and displayed to the user. This is an HTML file containing links to all generated reports.

2.5.4 Writing the HTML: Pillow and PROLOG

The actual writing of the HTML file makes use of templates stored in PROLOG files. Each template is simply an HTML file containing the special tag `<PROLOG>`[6] indicating that pieces of data produced by the report analyzer are to be inserted. Arbitrary PROLOG routines can be delimited using these tags. The routines are treated as small programs and executed; any output produced by them is placed in the finished HTML file in lieu of the `<PROLOG>` element.

If, for instance, information about biodiversity has been previously saved for a given report, then the HTML fragment

[6]The inclusion of the `<PROLOG>` tag means that the templates are not quite HTML. Most web browsers, however, are capable of overlooking this deviation.

```
The current stand exhibits
<PROLOG>
    biodiversity([current_report_number])
</PROLOG>
biodiversity.
```

would be converted into

```
The current stand exhibits a high level of biodiversity.
```

Similarly, the HTML fragment and PROLOG code

```
<PROLOG>
    len('Hello world', Length),
    writeq('Hello world'),
    write(' contains '),
    write(Length),
    write(' characters.')
</PROLOG>
```

would generate the output

```
'Hello world' contains 11 characters.
```

At the moment, no further specialized tags have been included (for instance, there is no `<IF_THEN>` element). All such functions could be performed using PROLOG code directly inserted into the HTML, and so the use of such tags is superfluous.

The report templates exist as a list of ASCII codes; these are converted to PROLOG structures using Pillow, a library of routines for interfacing PROLOG and HTML. Pillow was developed by the Computational Logic, Implementation, and Parallelism Lab (CLIP) of the Technical University of Madrid.[7] It was developed for use with Ciao PROLOG and is included with the standard installation of SICStus PROLOG. For the purposes of NED, it has been modified to work with LPA Win-PROLOG.

[7] http://www.clip.dia.fi.upm.es/

Within Pillow, each HTML element is converted into a PROLOG structure. The tag of the element becomes a functor; text delimited by the tag becomes a predicate argument. In the case of the `<PROLOG>` element, the argument is either treated as a call to insert information from the NED blackboard, or else as a PROLOG program to be executed. If, for some reason, both of these fail, the element is simply passed back unchanged. The presence of unevaluated PROLOG code, of course, makes for a confusing and aesthetically displeasing HTML page.

Chapter 3

PROLOG and Relational Databases: Background

3.1 Introduction

Much ado has been made over the similarities between logic based languages such as PROLOG and relational databases,[1] 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. Work on creating such a system began in the 1970's—the decade that saw the birth of both PROLOG and relational systems. It reached a peak of sorts in the 1980's, seeing the creation of systems of some sophistication, notably Bermuda and Primo [Ioannidis89, 229ff; Ceri90]. The contemporary field of deductive databases is, in part, the descendent of such work [Ullman93, 2ff; Date 1995, 792ff].

To date, however, no full integration has gained a significant degree of acceptance, perhaps because the resulting system, while debatably a relational database system of some sort, certainly would not be an implementation of PROLOG. 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.

The purpose of this chapter is to introduce relevant connections between PROLOG and relational databases, and to describe the impetus behind integration

[1]Consider [Sciore86]: "The point we wish to make is not that relational databases are easy to program in the PROLOG language, but that the PROLOG language itself is a relational database language" [294].

as well as the techniques used for integration. It will be explained why the usual method of connecting PROLOG to a database—as exemplified by the ProData [Lucas97] interface—is unsuitable for use in the NED project. The real point of the chapter, however, is to justify the existence of the query sub-language described in Chapter Four. Ultimately, for want of an efficient means of querying NED-2 databases from PROLOG, project members were forced to develop a language of their own.

3.2 The Similarities Between PROLOG and Relational Databases

3.2.1 Prolog

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. Derivation of theorems from the knowledge base proceeds via a backward chaining mechanism (from conclusions to premises); variables are unified where necessary. The query

```
-? in_same_patch(A,B).
```

causes the PROLOG theorem prover to return with variables bound as follows:

```
A = stand1
B = stand3
```

If more answers are available, the inference mechanism can be made to backtrack to produce them.

3.2.2 Relational Databases

Relational database systems got their start with the publication of E. F. Codd's "A Relational Model of Data for Large Shared Data Banks" [Codd70]. Within that model, a database consists of a set of *relations* (less formally called *tables*), where each relation is a set of *tuples* (less formally called *rows*) over some specified domain or domains [Codd70, 379]. The positions in a tuple are generally called *attributes*; they correspond to columns in a table.

The most important characteristic of a relational database system—or, more properly, the language used to control it—is that it is completely *declarative* in nature. The user does not specify how information is to be stored, or the means of retrieving it. He or she simply specifies *that* the information be stored or retrieved. This is a virtue dwelled upon in some detail in Codd's article, and it is this virtue that made relational databases superior to previous database systems.

Within the relational model, *integrity constraints* are forced upon databases [Date95, 110ff]. All tuples in a relation must be unique, and there must be some

attribute or set of attributes of each relation that uniquely name the tuples of that relation. These are generally called *primary keys.* An attribute of a relation that is the primary key of some other relation is called a *foreign key* and is said to reference the primary key. Thus, values of foreign keys are required to correspond to primary key values.

All of the information describing the structure of a database is kept in what is commonly called a *catalog* or *data dictionary.* Generally, the catalog is nothing more than another set of relations, and so may be accessed as any other relation would be [Date95, 60]. In this way, the querying user or process is able to know everything about the database that is needed to know.

3.2.3 SQL

There are a number of languages used in the manipulation of relational databases. All are at least as powerful the relational algebra first described by Codd [Date95, 140-141] and proposed as a standard for comparison. Any database manipulation language capable of expressing the same relations as the algebra is said to be *relationally complete* [Date95, 160].

SQL is by far the most popular such language. Developed in the early 1970's by IBM for their System R, [Date95, 65; Ullman89, 210], SQL became an ANSI standard in 1986 and an ISO standard in 1987. It has undergone three major changes since then, corresponding to the published standards SQL89, SQL92, and SQL99.

The general means of retrieving information from a relational database with SQL is via a SELECT statement. Such a statement has the form *SELECT* - *FROM* - *WHERE* [Date95, 71]. The SQL statement for retrieving from the NED-2 database the names of stands and their associated snapshots would be:

```
SELECT
        `STAND_HEADER`.`STAND_ID`,
        `STAND_SNAPSHOT_TREATMENTS`.`SNAPSHOT`
FROM
        `STAND_HEADER`, `STAND_SNAPSHOT_TREATMENTS`
WHERE
        `STAND_HEADER`.`STAND` =
        `STAND_SNAPSHOT_TREATMENTS`.`STAND`
```

Here, values from the two relations `STAND_HEADER` and `STAND_SNAPSHOTS_TREATMENT` are combined using an attribute the relations have in common—`STAND`.

3.2.4 The Resemblance

By this brief sketch, at least a few of the similarities between a PROLOG knowledge base and a relational database should be obvious. A predicate in logic is commonly interpreted to be a relation over some domain called the universe of discourse. Thus a collection of PROLOG facts such as of `adjacent/2` clauses, can be seen as a database relation named `adjacent` existing over the domain of stand names. Furthermore, a PROLOG rule such as `in_same_patch` may be viewed as a *join* of the relations `adjacent` and `size_class` [Sciore86, 294; Zaniolo86, 221; Gray88,7]. A join $R \bowtie S$ of relations R and S may be defined as

$$R \bowtie S = \{(a, b) : (a, c) \in R, (c, b) \in S\}$$

and each tuple (a,b) satisfies some further constraint [Codd70, 383].

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]. Unlike PROLOG, DATALOG restricts predicate arguments to ground terms and variables and contains none of the procedural aspects of PROLOG.

3.3 Reasons for linking PROLOG to a RDBMS

There are good reasons for wishing to marry PROLOG to a relational database. Each system has its special virtues, but each also has significant deficiencies. It is thought that in joining them, a more powerful and elegant creature would be created.

One reason has already been mentioned—a PROLOG like language is a concise and intuitive way of specifying queries 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 available to them the large storage space of disk drives.

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.

With the exception of first argument indexing, PROLOG makes little use of indexing schemes or other optimization techniques, making information retrieval, and therefore logical inference, relatively slow [Sciore86, 295]. In contrast, databases are very good at sifting through very large quantities of information. Much effort has been exerted in devising clever ways to speed retrieval. Also, in comparison to languages such as C, C++, and Java, there has traditionally not been much support for PROLOG. As a result, it is often difficult to integrate PROLOG into larger projects [Brodie88, 197], (many PROLOG implementations—such as SICStus, Quintus, XSB, Visual PROLOG—allow the mixed use of C and PROLOG, thereby opening a whole new world to PROLOG programmers).

3.4 Stumbling Blocks

There are more than a few 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.[2]

3.5 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

[2] Techniques for storing complex PROLOG clauses in a relational database are discussed in [Singleton93].

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. One may disagree with this, of course.

The second method has been attempted. The latest version of LPA Win-PROLOG (v4.2) allows for static predicates to be compiled using a hashing optimizer [Shalfield01, 121]. The SWI and SICStus PROLOG implementations include *Berkeley DB module*s which allow storing complex terms to secondary storage (this may solve the storage problem, but not the concurrent access problem); indexing on arguments is provided for [Wielemaker00; SICS02]. Ciao PROLOG has what is called a *persistent predicate database*, which can store predicates in external files or, via an SQL translator, in a relational database [Bueno00, 401ff].

3.5.1 Loose Coupling Vs. Tight Coupling

Regarding *coupled systems*, the literature usually refers to systems of two types: *tight* and *loose*. There is some ambiguity, however, as to what these terms mean. 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 the present paper, such systems have been 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][3] 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

[3] Lucas is the originator of the ProData interface used to some extent in NED-2.

considerations. A third parameter is specified—*transparency*—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.

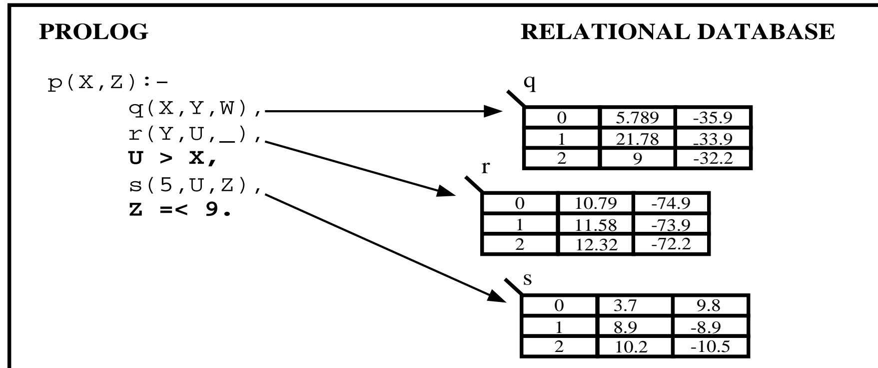
[Date95] defines loose coupling as providing a call level interface between the logic language and the DBMS; users would program both in PROLOG and SQL, for instance—"The user is definitely aware of the fact that there are two distinct systems involved....This approach thus certainly does not provide 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.

3.5.2 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 the way that has been presented at the outset of the discussion and 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*,[4] this method requires no changes to

[4] The term is used in [Ioannidis89, 231]. Lucas uses *external predicate*.



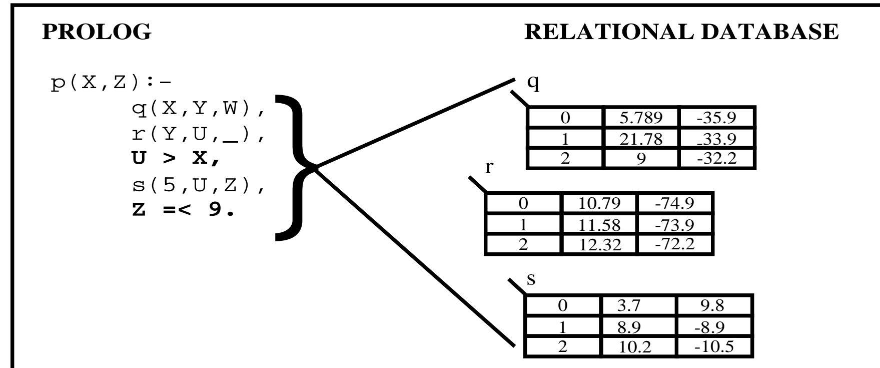
With relation level access, a database is queried one relation at a time. PROLOG performs constraint tests.

Figure 3.1: Relational Level Access

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, however; 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. It 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



With view level access, multiple relations are involved in the database query. The DBMS performs constraint tests.

Figure 3.2: View Level Access

query, and execute that plan. With tuple-at-a-time access, since queries are of the simplest possible variety, no optimization is possible.[5]

The alternative 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. Appendix A of this paper is intended to illustrate the difference between relational and view level access. 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.

3.6 Real World Systems

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 two systems are loosely coupled. An interface allows PROLOG to query the database when needed, either by translating PROLOG goals to SQL, the de facto standard of relational database systems, or else by embedding SQL directly in the PROLOG code.

Most PROLOG systems[6] providing a database link use Microsoft's ODBC library as an intermediary [Microsoft97]. More than a few implementations,[7] in lieu of or in addition to an ODBC interface, link directly to more popular databases (such as Oracle). Others use a Java interface similar to ODBC.

ODBC was developed in the early 1990's to facilitate database interoperability. It provides a uniform interface between applications (including PROLOG) and any database fitted with an ODBC driver. The ODBC library consists of a set of functions for opening and controlling a connection to a database and a set of functions for preparing and executing SQL statements against the database. Importantly, the peculiarities of a particular database system are masked by ODBC. In this way, the application need worry only about communicating with ODBC and not about the underlying architecture of the database it is accessing.

---

[5] As this access method is inefficient because only a single relation is involved, the term 'relational' is perhaps a more apt term for it.

[6] ALS, LPA, SICStus, Amzi, XSB, Visual, Trinc

[7] XSB, SICStus, Strawberry, MasterPROLOG

3.6.1 Low Level Access vs. High Level; ProData

The database links allow PROLOG varying degrees control over databases. Some[8] are very low level, merely providing PROLOG wrappers to ODBC functions. This means that the user must keep track of connection handles and cursors.[9] 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.

The high 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. ProData is a standard of sorts; besides the commercial systems already mentioned using it, some effort was exerted to make Ciao and XSB interfaces mimic it.

Within ProData, all low-level ODBC functions are hidden from the user. It is a relation based 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[10][Singleton93].

With ProData, the user can also fetch information from a relation using `db_tuple/2`, which accepts a relation name and returns a tuple. Alternatively, complex SQL statements can be sent to ODBC and the results collected one at a time upon backtracking (hence, ProData is not restricted to transparent access).

[8] Trinc, PDC, ALS

[9] A cursor in this context can be thought of as a marker of which row in a table would be returned next when retrieval routines are called.

[10] The term 'cuttable' means that once a cut is encountered, no further solutions are retrieved from the database upon backtracking

The benefit of the 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. A significant drawback is that the programmer has less control over what is going on behind the scenes. In the particular case of ProData, a great many useful ODBC functions are unavailable to the programmer. This has frustrated NED programmers on more than a few occasions.

Another significant drawback to the high level approach is that it generally goes hand and hand with treating relations as PROLOG predicates, and although this works well in many cases, there are also cases where it does not. Specifically, database relations often have arities of over 100, and writing PROLOG rules involving predicates with over a hundred arguments is tedious in the best of times, especially when one is only interested in the 42nd argument. It would be preferable in such circumstances to refer to the argument by name, just as one would do in SQL.

BERMUDA and PRIMO

A technique which allows for a significant degree of transparency while at the same time avoiding the speed drawbacks inherent in relational level access is implemented in the systems BURMUDA and PRIMO [Ioannidis89, 238; Ceri90]. Both implement a *look ahead* 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 precompiler 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 together database predicates are.

3.6.2 Elegance at the Price of Efficiency

The PROLOG components of NED-2 are written in LPA Win-PROLOG and consequently use ProData to access databases. It must be stressed that the phrase 'transparent tight coupling'—as ProData supposedly is—should not be construed to mean 'efficient' or 'desirable'. It merely means that access to the database proceeds in much the same fashion as access to the internal PROLOG knowledge base—via depth first search and unification. The point has already been made that this method of access is quite inefficient. Indeed, this method was used by NED-2 in its early development phase, but its inefficiency became immediately apparent. Furthermore, as the structure of the NED-2 databases changed so often in their early days, treating relations as normal predicates (which occurs in a transparent coupling) was unworkable; it would simply take too much effort to ensure that the predicate argument structure matched the database table structure. The relational means of access was quickly abandoned, and development of the language described in Chapter Four, was begun.

Chapter 4

PROLOG and Relational Databases in NED

For NED-2, various PROLOG predicates have been written in order to allow easier communication with relational databases. These are built on top of the normal ProData routines. This chapter describes the process one must go through in order to use a database with NED-2.

4.1 specifying data sources; loading metadata

The names of data sources to be used with PROLOG are stored in `data_source/1` facts. Each such fact stores a single PROLOG string, consisting of all uppercase letters, as an argument. This string must correspond to a data source name registered with ODBC. The current NED-2 data sources are:

```
data_source(‘NED-2 WORKING FILE‘).
data_source(‘NED GOALS DATABASE‘).
data_source(‘NED PLANTS MASTER DATABASE‘).
data_source(‘NED REPORTS DATABASE‘).
data_source(‘NED TREATMENTS DATABASE‘).
data_source(‘NED VARIABLES DATABASE‘).
```

Any database registered with ODBC can be accessed, but only those that have `data_source/1` clauses associated with them.

When NED-2 is initialized, two PROLOG predicates are called. The first, `load_dblink/0` simply loads the ProData library and sets various attributes for use with NED-2.[1] The second, `load_data_sources(+Dir)` looks in a specified directory

[1] For instance, error messages and SQL statements generated by ProData, which would otherwise be shown to the user, are hidden.

for infomation stored about the data sources named in `data_source/1` clauses. Such information, which duplicates in many regards the data dictionary of each database, is stored in a '*name*.dic' file, where *name* is (again) the registered name of the data source. There is one such file for each data source. For each relation/table in the database, a `database_table/5` clause will be recorded. It has 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 in forming joins in 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 relational database and are used by PROLOG when forming queries to that database.

If no metadata about a database is stored in a '*.dic' file, `load_data_sources/1` attempts to generate this data itself using ProData routines, and, once generated, saves the metadata to disk. It is important to note that ProData can provide only a

portion of the metadata which exists 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 the 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.

An exception exists in the case of MS Access databases, which is the format used by NED-2. Automatic generation of metadata can be had for these databases only because NED-2 PROLOG routines have been tailored specifically for them. If another type of database is used, then someone must encode the metadata by hand and save it to a file, or else rewrite NED-2's source code to process this new format.

4.2 Connecting to a database

Once the metadata about a given database has been loaded, agents can connect to a database and query it. To connect to all registered databases the command `connect_to_databases` is given. To connect to one or more specified databases, the command `connect_to_databases(X)` is given, where `X` is either a string indicating an individual datasource, or else a list of such strings. Note that if a database

connection is already open, a new connection will not be created. The predicates `disconnect_from_databases` and `disconnect_from_databases(X)`, as their names imply, close all or some connections.

4.3 Other Predicates

The above constitute the most commonly used database handling predicates. However, there are a few others, which will now be quickly mentioned.

`create_data_dictionary/1` accepts a string or list of strings and generates metadata about specified data sources. Similarly, `save_data_dictionary/1` saves metadata about one or more databases to '*.dic' files.

The predicate `get_data_sources/1` returns a list of all data sources for which `data_source/1` clauses exist. The predicate `set_data_sources/1` asserts `data_source/1` clauses for all data sources specified in a list.

4.4 ODBC_PL: Augmenting ProData

ODBC provides functions that can describe in great detail the capabilities of a database system, as well as describe the current state of the system. ProData, as has been mentioned, generally does not provide a means for using such functions. In order to make up for this lack, a small library of C routines has been written which accesses ODBC directly. It's functionality is briefly described here (a list of all the predicates constituting the library is found in Appendix C). The library is called `ODBC_PL`.

With ODBC_PL, PROLOG connects to a database using `odbc_connect/2`, which accepts a data source name and returns a connection handle. One closes the connection with `odbc_disconnect`. Both of these predicates mimic ProData routines.

The names of tables in a database as well as the columns in each table can be retrieved using, respectively, `odbc_tables(+DSN,+Flag,-Tables)`, and `odbc_columns(+DSN, +Table, +Flag, -Columns)`. These, too, are like ProData routines, with a few important differences. By specifying a value for a flag, `odbc_tables` can be made to return only the data dictionary tables. Similarly, `odbc_columns` can be made to return not only attribute names, but also their data types and the amount of data that each can store.

`odbc_primary_keys(+DSN,+Table,-Keys)` accepts a datasource handle and a table name, and returns a list of the primary keys in that table.
`odbc_foreign_keys(+DSN,+Table,-Keys)` returns the foreign keys. (Please note that these functions are only supported by some databases. MS Access, for instance, does not support them.)

`odbc_get_info(+DSN, +Characteristic, -Info)` returns information about dozens of characteristics of the data source, such as: the number of columns and rows that are allowed in a table; the sort of aggregate functions it supports in SQL statements; the name of the file associated with the the data source (note that ProData cannot do this); the maximum size of an SQL statement that can be passed to it; the driver that is connected to the source; and the maximum number of simultaneous connections that the driver can support. Such information is vital if truly automatic and transparent access to the database is to be had.

A data source can be registered with ODBC using `odbc_create_dsn(+Type, +Driver, +DSN, +File)`, where `Type` indicates whether the data source is to be registered as a user or system source; `File` is the name of the file to be registered. ProData, in contrast, offers no means of creating and registering a database with ODBC.

The library is not finished, but a critical mass has been reached which will allow relatively easy expansion. It is already very useful. An effort has been made to

duplicate and, where possible, exceed the functionality of ProData. At least in the limited tests performed, `ODBC_PL` was able to outperform ProData. For instance, it can retrieve tuples from a fairly large database in roughly one half of the time taken by ProData. Whether it is a more stable library than ProData, however, is another question entirely; this can only be determined by further testing.

4.5 Querying NED-2 Databases

Information in NED-2 is stored in relational databases and in the working memory of PROLOG. 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 structures used in DSSTools to store information.

In working memory, information is stored in DSSTools `fact/4` clauses, which have the following structure:

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

The first argument, sometimes called an *AOV* triple (commonly called an *OAV* triple in other works) is the substance of the information itself. An example is `area([stand_x], 5)`. This may be interpreted, fairly obviously, as “The area of stand x is 5.” The latter three arguments constitute metadata about the information—it’s quality, where it came from, and the time at which the fact was recorded.

In order to allow the use of relational databases, the predicate `database_fact/4` has been added to DSSTools.[2] Like `fact/4`, the standard `A(O,V)` DSSTools 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. LPA's 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 predicates `known/1` and `known/3` are defined; these call, alternatively, both `fact/4` and `database_fact/4`. `known/3` is shown below:

```
known(Attr(Object,Value),CF,Source) :-
     (
     fact(Attr(Object,Value),CF,Source,_)
     ;
     database_fact(Attr(Object,Value),CF,Source,_)
     ).
```

4.6 Examples

Following are a series of examples designed to illustrate the nature and capabilities of the query language. The first is perhaps the simplest sort of query possible. Later examples are of a more complex nature.

A Simple Question

Suppose, for instance, that the call

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

[2] This predicate is defined in the file ‘sql.dst’ in the DSSTools directory.

is made, which 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.[3] 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 fail.

   If attributes are found in multiple tables, PROLOG will present multiple solutions to the query upon backtracking. It is important to note 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.

[3]LPA strings are indicated using backward quotation marks.

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’’
```

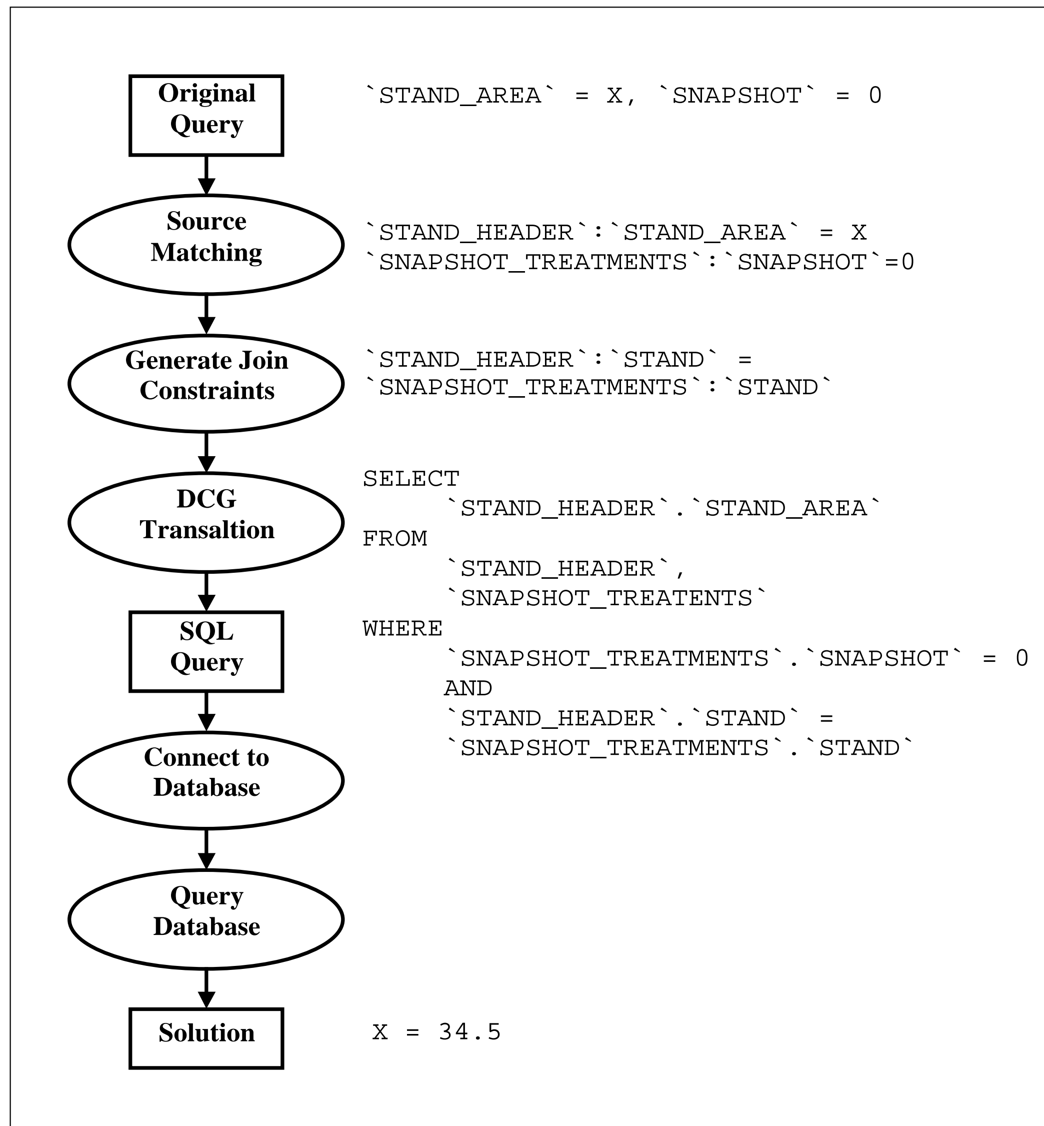
Note that column names and tables appear as LPA strings. Column values are usually either numbers, which are not converted in the query, or else atoms, which must be doubly-quoted in order to be evaluated (there is a built-in routine which automatically adds extra quotation marks to atoms).

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. The steps involved in a usual query are shown in Figure 4.1.

Arithmetic

The comparators =, <=, >= can be used in queries, as well as normal arithmetical operators (+, −, ∗, /).

```
database_fact(‘TREE_SPP‘([‘TREE_DBH‘ >= 0,‘TREE_DBH‘ < 4+1], Value),_,_,_)
```



A query for the area of the stand associated with Snapshot 0. Note that a join of two tables on the attribute `STAND` is involved.

Figure 4.1: The Query Process

becomes

```
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).

Logical Operations

Logical operations can be used: `','` denotes conjunction, `';'` denotes disjunction, and `'\+'` denotes negation, as is 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,

```
database_fact(`STAND_ID`([\+(`STAND` = 0 ; `STAND` = 1)], Value),_,_,_).
```

becomes

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

Colon Notation

In the original 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
```

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 appears to be a limitation of the MS Access driver used with ODBC. The query

```
database_fact('MIN'(‘STAND‘)([], X), _,_,_).
```

becomes the SQL

```
SELECT
      MIN(‘STAND_HEADER‘.‘STAND‘)
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.

```
database_fact(‘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’
```

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.

```
database_fact(
        ‘MyTable‘:‘STAND‘([subquery(‘MyTable‘, [‘STAND‘ = _])],X),
        _,_,_).
```

becomes

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

Automatic Joins

Included in the metadata stored by PROLOG about each database is 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!

```
database_fact( `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`'
```

IN and BETWEEN

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

```
database_fact(
        `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`'
```

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 indicates 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.

```
database_fact(`STAND_ID`([distinct],X),_,_,_).
```

becomes

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

4.7 Related Work

It is noted that translation of PROLOG expressions into SQL is absolutely vital if information is to be retrieved in a timely fashion. This was the point of Chapter Three. While it might take the RDBMS a second to evaluate a fairly complex query, it might take PROLOG several minutes 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 actually quite similar to a language called TREQL (Thorton Research Easy Query Language) developed some time ago. The intention behind the development of that language is similar in at least one respect to the one used in NED-2—namely, it permits meaningful queries to be posed to databases despite ignorance of the underlying database schema [Lunn88, 48]. As in the present language, 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.[4] 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 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.

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

4.8 Conclusion

The query language described in the above sections is an essential component of NED-2. If it did not exist, then something very much like it would need to be created to fulfill its function. 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 painlessly posed from within PROLOG. Though it certainly could

[4] It is likely correct to view the first argument of `translate/3` as the head of a rule, and `Goal` as the body.

be expanded and improved upon, the language described here accomplishes this. Though it does not allow transparency, this is not considered a horrible loss. Since the databases of NED-2 involve many attributes and underwent frequent changes in their developmental phases, transparency would have offered few advantages.

Chapter 5

Conclusion and Future Directions

The preceding chapters have described NED-2 as an ecosystem management system. It is in many ways the embodiment of the ecosystem approach, which takes a holistic view of forest management and which has gained prominence in recent years. NED-2 is goal driven and attempts to combine many competing objectives. It is modular, this modularity made possible both by the use of a blackboard design and by the use of relational databases as primary storage. Some success has been met in devising a convenient way of accessing these databases from PROLOG.

However, it is intended that NED-2 or its offspring will be capable of incorporate many more sources of knowledge. Though much progress has been made towards this goal, it cannot be argued that NED has achieved this.

Below are a few comments, offered in a very tentative way, about what is needed in the near future in order to increase NED's abilities as a platform of unification. These suggestions can realistically be viewed only as being about which small steps to take next in the ongoing development process. Hopefully, however, these steps would be in the right direction.

5.1 An Internal Representation

Within the average DSSTools application, information is stored as *facts*, the structure of a fact being an `Attribute-Object-Value` triple. These facts can be retrieved using the predicate `known/1`. As originally intended, the facts were to be as simple

as possible—atomic propositions—and calls to `known` were to return a single AOV triple at a time.

In NED-2, however, the incorporation of relational databases forced movement away from this simple method. When retrieving facts, `Object` arguments were allowed to contain the logical complexes described in the last chapter. It was too costly in terms of performance to continue to retrieve information a single attribute at a time (for exactly the same reason that relational level database access is unworkable). So, in calling `known/1`, one was not getting one simple fact, but several.

Though generally much needed and beneficial, the move to relational databases as the primary representational medium has had at least one detrimental effect. Specifically, the routines developed to access data held in the databases are perhaps too closely tailored to the relational model. There is no well thought and well organized system within PROLOG itself representing the objects that constitute the forestry domain as envisioned by NED.

While DSSTools facts were originally intended to represent simple AOV triples, there are no objects per se in the relational model to occupy the middle position—there are only attributes, and relations between attributes, and relations between relations. This is not a deficiency of the relational model. However, from the standpoint of PROLOG and DSSTools, it perhaps makes it difficult to speak about objects in a clear manner. Since what constitutes objects and their attributes may be spread across several tables of a relational database, making assertions about objects certainly becomes problematic.

What is needed at this point is the delineation of an ontology to be used by the internal PROLOG processes of NED-2. It must parallel the world represented in the relational databases but must be separated from it. Such an internal representation is needed for two reasons: (1) to allow smoother interaction with the NED-2 databases themselves; and (2) to allow the easy incorporation of further sources of knowledge

(which almost certainly will not have the same representational schema as NED-2's databases).

Better interaction with the NED-2 databases would be facilitated by the level of abstraction that the internal representation would provide. For instance, if one wishes to create a new snapshot, it would be better if there was a single routine `create_new(snapshot, Arguments, ID)`, where `Arguments` is a list of attribute values needed to sufficiently describe the snapshot in question, and `ID` is a unique identifier assigned to the snapshot. Built into this routine (or accessible to it) would be declarations of the interrelationships between the tables of the NED-2 databases. In this way, the routine itself would be responsible for ensuring that the snapshot created is well-defined in the database. Furthermore, if the database schema changes (e.g., if another database entirely was used), then all that would need to be changed is the metadata accessed by the routine.

This process can be contrasted with how updates to the databases are done now. Here, the developer, using ProData routines (perhaps even SQL statements), must update each table, always keeping in mind the dependencies which exist between them.

If other sources of knowledge are to be used by NED-2, it is essential that the language spoken by PROLOG components be well thought out. Once in place, it becomes much easier to specify what sort of information various data sources contain. There might be lookup tables indicating that one source, for instance, contains information only on plots, and only on certain attributes of plots. The tables would also need to contain the name of the information in the source that is needed to represent the plot, as well as rules for translating this information into the NED/PROLOG language. The table might also specify how long it would take for the source to provide the information (e.g. the source might be a simulator), or

whether the source can be written to as well as read from. This sort of information will be needed to utilize the source in a meaningful way.

5.2 Integration of External Sources: The Constraint Problem

Given the existence of this ontology, queries for information could be posed in a single language, as if one were speaking to a single agent, even though the answers come from many places. The query could be translated to the languages spoken by whatever sources are capable of responding.

5.2.1 A very naive integration technique

A profoundly simple method of carrying out this querying process is shown in Figure 5.1. One starts with a query phrased in the internal language. The query is analyzed to determine the objects and attributes referenced in the query. A list of these is kept. Then, using a simple look up table linking the objects and attributes both to the names of external sources storing information about them and to the internal schemas of the sources, new queries could be posed to each source. These would be phrased in a format understandable by each source and referencing the objects in that source.

The results of these queries could then be translated back into PROLOG's own schema using translation rules and inference engines. The newly translated results would be substituted into the original query and the constraints of the query tested for truth.

This method, while simple, has the flaw of being greatly inefficient, for the testing of constraints is performed only at the end of the query process. As a result, there will be a great amount of backtracking involved before a valid solution is found.

It turns out that queries involving both constraints and multiple data sources make for a difficult problem (at least one author has given it a name: the *constraint mapping problem*) [Chang99]. Developing a means of translating queries that yield solutions in practical times is not an easy task.

5.2.2 A Slightly Better Technique

One possible solution is to convert the original query into disjunctive normal form.[1] Each disjunct would then consist of a conjunction of constraints. The conjuncts can then be grouped by source. The conjuncts involving only a single common source can be translated and 'pushed' to that source. In this way, the work of satisfying the constraints is offloaded to the foreign source. Conjuncts involving multiple sources must be handled in a manner somewhat similar to the naive method described above.

[1] an elegant method of performing this in PROLOG is found in the code for the compiler described in [Draxler93].

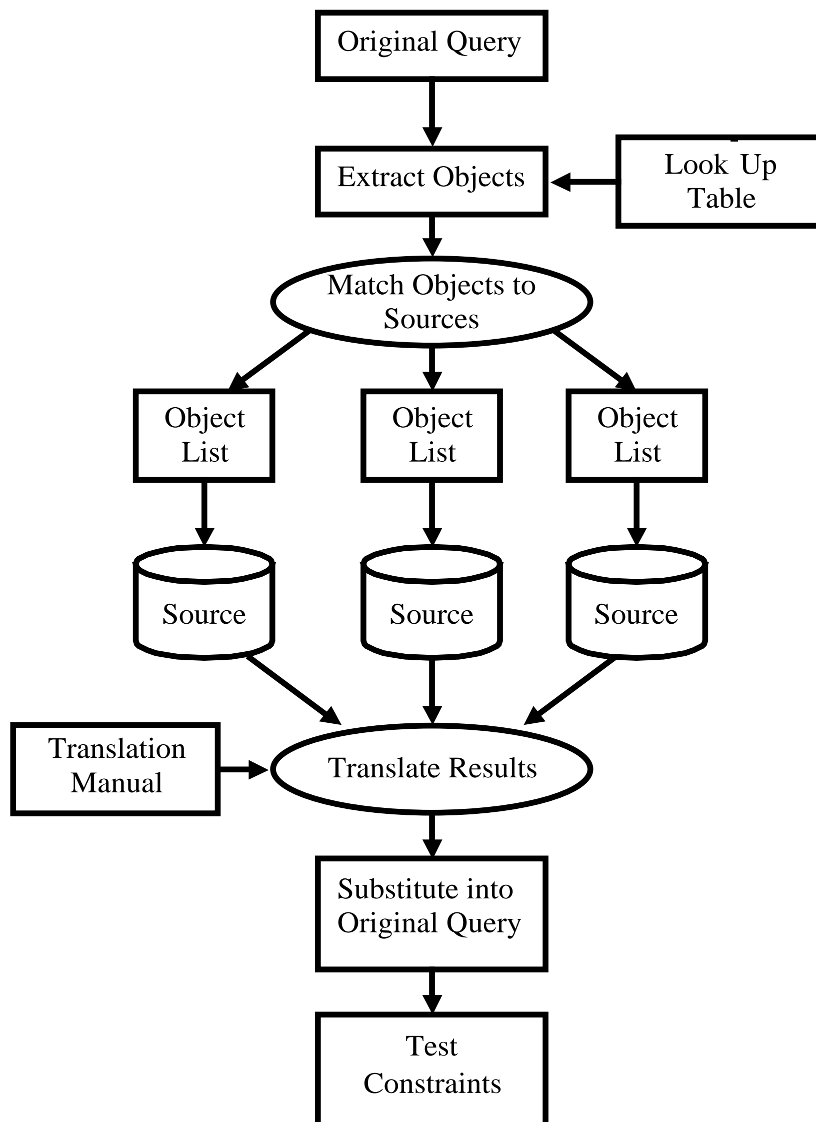


Figure 5.1: A Simple Means of Data Integration

Appendix A

Database Query Benchmarks

The following is a summary of tests performed to determine the time required to solve various PROLOG goals involving database predicates. The bulk of the tests utilize the top-down access mechanism used by LPA ProData (and PROLOG in general). These are compared to translating the original PROLOG goals to SQL SELECT statements. The tests are intended to stress the unacceptable performance of querying an external database in the same manner as internal PROLOG Knowledge bases are queried.

The goals were also solved for clauses stored internal to PROLOG (i.e., with the data in the external database retrieved and asserted as PROLOG facts). This was done to illustrate the potential performance advantage of storing data in primary memory.[1]

For the tests, two MS Access 2000 tables, `A` and `B`, were created. Each table consisted of 100 columns and 1000 rows. Each column of A constituted a list of integers randomly chosen without replacement from [1,1000]. Table B was similarly created, save that numbers were chosen from [1000, 1999]. A given column in A matches a given column in B in **exactly** one place.

In order to test access times for data stored in RAM, the tuples of the two database tables were asserted as clauses of PROLOG predicates a/100 and b/100.

[1]However, [Ioannidis94] indicates that the performance gain decreases as the size of the database increases. Particularly, tests in that paper show that PROLOG performs quite poorly in comparison to an RDBMS when performing joins of 10,000 tuple relations.

The tests were performed several times, in two batches. The first batch of tests was intended to simulate queries on an average looking database (here, each test was performed 100 times). The second batch was intended to simulate the worst case scenario, where solutions to goals can be found only at the bottom of the tables (Tables A and B matched only on the 1000th row; given that these tests took much longer to execute, they were only performed 25 times each).

In viewing the results, it can be clearly seen that attempting to solve for a goal involving database predicates using PROLOG's normal top-down search strategy takes an impressively long amount of time. Simply retrieving a tuple from the database and then checking to see if it satisfies a given constraint takes, on average, over four seconds (in the worst case, it took roughly eleven seconds). In contrast, the same query performed against an internal PROLOG knowledge base takes about four milliseconds; converting the goal to SQL and retrieving the results takes roughly 1/10th of a second. Similarly impressive, performing a join using the top-down technique takes between 40 seconds and 2 minutes (and this for a single goal!). The equivalent SQL query takes, on average, 80ms.

It must be stressed that the tests below involved at most two database tables of relatively small size (in comparison, two NED-2 tables storing information about plant species have over 2,000 and 80,000 rows, respectively). Furthermore, the sort of queries actually posed to the NED databases during run-time are often of a more complex nature than those presented below (for instance, some involve joins of three or four tables).

Descriptions of the tests performed and the results of the tests are listed below. Tests were performed on a 1.2GHz PC with 512MB of double data rate RAM. All times are measured in milliseconds.

A.1 Test Descriptions

Fetch Test 1

`Description: unifies with a clause of a/100 where Nth arg = 1000.`

This test calls a/100 with some randomly selected argument instantiated to 1000. All other arguments are unbound.

```
Example Goal:
| ?- a( A1,A2, A3,1000, A5, A6,...,A100).
```

Fetch Test 2

`Description: unifies with a clause of a/100,`

`THEN checks to see if Nth arg = 1000.`

This test is like the first, save that a randomly selected argument is checked for equality with 1000 `after` the clause has been fetched. In comparison to the first test, searching in this manner should take considerably longer.

```
Example Goal:

| ?- a( A1,A2, A3,A4, A5, A6,...,A100), A5 == 1000.
```

First Argument Indexing

`Description: unifies with a clause of a/100 where 1st argument is`

`instantiated to some random integer in [1,1000].`

PROLOG implementations almost invariably use some form of first argument indexing. This test attempts to see if such indexing causes a gain in performance.

```
Example Goal:
| ?- Rand is rand(1000)//1+1,
     a(Rand,A2, A3,A4, A5, A6,...,A100).
```

Join Test 1

`Description: where Mth arg of a/100 = Nth arg of b/100.`

This test determines the amount of time needed to perform an *equijoin*—joining tuples of A and B only where some specified argument of A matches a specified argument of B.

`Example Goal:`

```
| ?- a( A1,A2, A3,JOIN, A5, A6,...,A100),
     b( B1,B2, B3,B4, B5, B6,...,JOIN, B100).
```

Join Test 2

`Description: Where Nth arg of a/100 = Nth arg of b/100.`

This test is exactly like the previous test, save that A and B must match on the same argument.

`Example Goal:`

```
| ?- a( A1,A2, A3,JOIN, A5, A6,...,A100),
     b( B1,B2, B3,JOIN, B5, B6,..., B100).
```

SQL Query Test

`Description: Performs a Join of A and B on some argument.`

For this test, an SQL SELECT statement is created to return the join of A and B on some randomly selected argument (in this case, the argument is called `x`).

```
Example Goal :'SELECT ‘A‘.*, ‘B‘.* FROM ‘A‘,‘B‘ WHERE ‘A‘.x =
‘B‘.x’.
```

A.2 Test Results

Test	MIN	MAX	AVG	VAR	STDEV
Fetch test 1	0	1	0.05	4.80E-002	0.22
Fetch Test 2	2	136	4.45	186.47	13.66
1st Arg Indexing	0	1	0.08	7.43E-002	0.27
Join Test 1[2]	10	1053	24.58	11896.23	109.07
Join Test 2[3]	10	4244	67.67	192840.43	439.14

Table A.1: Average Case Scenario: Internal Predicates

Test	MIN	MAX	AVG	VAR	STDEV
Fetch test 1	94	916	135.41	10903.07	104.42
Fetch Test 2	4017	12948	4478.55	845758.86	919.65
Join Test 1	43870	52627	44504.7	1536164.56	1239.42
Join Test 2	44311	85593	45221.71	21366590.17	4622.40
SQL Query	70	224	77.53	320.71	17.91

Table A.2: Average Case Scenario: External Predicates

Test	MIN	MAX	AVG	VAR	STDEV
Fetch test 1	0	0	0	0	0
Fetch Test 2	9	9	9	0	0
1st Arg Indexing	0	0	0	0	0
Join Test 1	36	37	36.12	0.11	0.33
Join Test 2	36	39	36.68	0.73	0.85

Table A.3: Worst Case Scenario: Internal Predicates

Test	MIN	MAX	AVG	VAR	STDEV
Fetch test 1	97	987	162	31805.08	178.34
Fetch Test 2	11586	14017	11758.64	226820.41	476.27
Join Test 1	125848	143871	131246.92	48914565.33	6993.90
Join Test 2	127315	182601	137619.32	113408594.06	10649.35
SQL Query	73	145	80.36	239.91	15.49

Table A.4: Worst Case Scenario: External Predicates

Appendix B

Database Query Caching

Retrieving data stored on a spinning disk is invariably slower than retrieving it from primary memory. It consequently makes sense to keep in primary memory data fetched from the NED-2 databases, so that it can be accessed more quickly the next time it is needed. The present appendix discusses certain PROLOG routines created for this purpose and presents the results of a few benchmark tests performed to illustrate the time savings allowed by caching.

B.1 The Caching Routines

B.1.1 cached_query/5

In the current NED-2 design, queries such as described in Chapter Four of this paper, as well as their results, are recorded in `cached_query/5` clauses. This predicate has the form:

```
cached_query(+QueryTemplate, +Query, +DB, +SQL, +Flag).
```

Each answer from a database (i.e., a projection of an individual tuple) is stored in a separate `cached_query` clause. The argument structure is a bit complex and requires some explanation. The query and its results are actually stored in the second argument of the predicate. It is in the format described in Chapter Four; the variables in the query constituting the solution have been bound. The first argument is a template made from the original query by replacing all solution variables with

structures of the form `'$VAR'(N)`, where `N` is the Nth unique variable encountered in the query. This template is created to prevent the querying process from returning too few answers. (This can occur if a query subsumes another query for which solutions have been cached. In such a case, if is possible that some solutions for the more general query would not be returned.) The template ensures that answers will be provided only to queries exactly matching the form of the original.

The third argument stores the name of the database orignally providing the answer. The fourth argument stores the SQL translation of the query. The last argument is either `true` or `false`, and indicates whether there might be more answers to the query in one of the databases. It is used to prevent the query managing routines from re-querying a database, thereby producing redundant answers.

B.1.2 CLEAR_CACHE/0, SET_CACHE_FLAG/1, GET_CACHE_FLAG/1

The cache can be cleared by invoking `clear_cache`. It is very important that the cache be cleared after any changes are made to the NED databases. Caching can be turned on or off by invoking `set_cache_flag(true)` (cache queries) or `set_cache_flag(false)` (off). Note that during the process of turning off the cache, all cached queries will be discarded. To determine whether caching is enabled, one may invoke the predicate `get_cache_flag/1`. By default, caching of queries is turned off.

B.1.3 CACHE_MAX_CLAUSES/1, KEEP_CURRENT_GOAL/1

To prevent the cache from exhausting all of PROLOG's allotted memory, the number of `cached_query` clauses asserted is limited to an integer value stored in `cache_max_clauses/1`. This value may be set via the predicate `set_cache_max /1`. If the cache limit is exceeded, then the entire set of results for a single query will be removed from memory. (It is important that the entire set of solutions be removed;

if only some are removed, subsequent calls to the query would return only a partial set of solutions.) The query chosen for removal is the one that was called least recently. The process of removing solution sets for individual queries loops until the maximum is no longer exceeded.

It is possible that the number of solutions to a single query exceed the maximum number allowed by the cache. If this occurs, then the technique described above—of removing the entire set of results—will itself cause the problem it was designed to avoid. To prevent this, the user may assert the clause `keep_current_goal(true)`. This allows the cache limit to be exceeded until all answers to the current query are fetched from the databases. Doing this is itself dangerous, for the set might also exceed the memory allotted to PROLOG as a whole. (The only solution to this is to allot more system resources to PROLOG.)

B.1.4 DB_QUERY/2

The three clauses of `db_query/2` are listed below. `db_query/2` is actually the primary predicate used to query NED databases. It has been altered, however, to allow for caching.

In the first of the below clauses, an attempt is made to match a given goal query to one that has already been stored. If a match can be made where the `Flag` argument is `true`, then the clause succeeds (note that backtracking for further solutions is possible). If a match is made where the `Flag` argument is `false`, then a cut is executed (thus preventing going into the lower `db_query/2` clauses) and the clause is forced to fail.

If no further matches can be made with cached queries, execution moves into the second `db_query/2` clause; the goal query is translated to SQL, and this statement is directed toward some database. It is important to note, that PROLOG remembers (using `cached_count/2`) how many answers to the query have already been cached

and counts the answers from the database, discarding those already stored (e.g., if eighteen answers have been cached, then PROLOG will query the database nineteen times, throwing away the first eighteen answers). New answers from the database are asserted to working memory.

When there are no more answers to be found even in the databases themselves, a final `db_query/2` clause—this time containing a `false` flag value, is asserted to memory. This, again, states that there are no more answers to that particular query.

```
% FIRST LOOK IN CACHE
db_query(Goal, DB):-
    format_term_vars(Goal,GoalTemplate),
    cached_query(GoalTemplate, Goal, DB, SQL, Flag),
    (
    Flag == true
    ;
    Flag == false,
    !,
    fail
    ).

% THEN LOOK IN DB AND UPDATE CACHE.
db_query(Goal, DB):-
    cached_count(Goal,Count),
    translate_to_sql(Goal,DB, SQL, Variables,Maps),
    connect_to_database(DB,HDBC),
    format_term_vars(Goal,GoalTemplate),
    start_flag(Flag),
    db_sql_select(HDBC, SQL, Variables),
    % ignore answers until the number
    % exceeds number of cached answers
    increment_flag(Flag, Count,N),
    N > Count,
    close_flag(Flag),
    update_cache( GoalTemplate,Goal, DB, SQL,true).


% UPDATE CACHE, INDICATE THAT QUERY FAILED.
db_query(Goal, DB):-
```

```
format_term_vars(Goal,GoalTemplate),
update_cache( GoalTemplate,Goal, DB, SQL,false),
!,
fail.
```

B.2 Caching Benchmarks

Below are listed the results of a few tests performed to determine the advantages of caching queries. Specifically, the times required to generate three NED HTML reports—the Overstory Summary Report, the Understory Summary Report, and the Management Unit Identification report—were recorded, as was the time required to perform habitat analysis for all wildlife species in the NED species database. The latter requires running a backward chaining inference engine on each species. The reports and the inference process require retrieving data from external databases.

From the results, one can see that the habitat analysis benefited the most from caching, the analysis with caching taking roughly a third of the time as analysis without caching (the actual number of seconds or minutes required for any program is contingent upon the size of the data set). The report generation times benefited not at all from caching.

It must be stressed that the utility of caching depends greatly upon the sort of query encountered during program execution. If the same query occurs many times within a program (as it does in the habitat rules), it makes good sense to cache results. If a program makes many queries, but no query is repeated, then it makes no sense at all to cache the results.

B.2.1 Test Results

Test	Cached	MIN	MAX	AVG	VAR	STDEV
Wildlife Anal.	No	130277	184195	164157.16	77517311.39	8804.39
MU Ident.	No	2884	4686	3837.5	227273.833	476.73
MU Over.	No	3215	5337	4920.1	373270.1	610.96
MU Under.	No	4336	5508	4982.2	97739.29	312.63
Wildlife Anal.	Yes	17976	50543	44911.6	91371941.6	9558.87
MU Ident	Yes	2954	35671	6350.1	106203826.99	10305.52
MU Over.	Yes	3495	9523	4382.1	3285300.32	1812.54
MU Under.	Yes	4437	15483	5770.1	11776271.88	3431.66

Table B.1: Cache Tests

Appendix C

ODBC_PL: Reference Manual

The following is a brief description of ODBC followed by a list of the predicates constituting the odbc_pl library.

C.1 The Structure of an ODBC Application

An ODBC application is composed of four components [Microsoft97, 33ff]:(1) the calling application; (2) the ODBC driver manager; (3) ODBC drivers; (4) data sources. The interaction of these components is indicated in Figure C.1. An application accesses each data source via the routines provided in the ODBC API. Some of the routines are implemented by the ODBC driver manager; others are implemented by the data source driver itself.

C.1.1 Handles

The ODBC API makes extensive use of handles—32-bit values identifying different objects [Microsoft97, 51ff]. The basic objects to which handles are assigned: (1) environments; (2) connections; (3) statements.

Environment handles may be viewed as the “Global context” of the application [Microsoft97, 53]. Generally, only one environment handle is used in an application.

Connection handles identify a particular connection to a data source. Multiple connections to a single source can be made. Connections are tied to a given environment (meaning that a connection cannot exist outside of some environment and

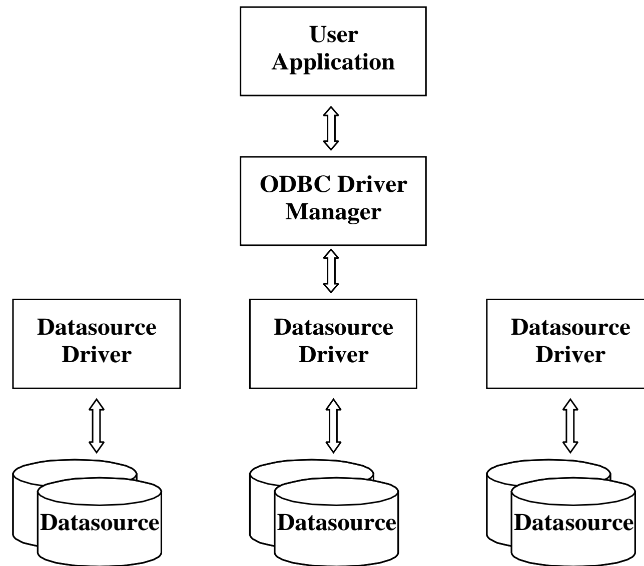


Figure C.1: ODBC Application Architecture

that the environment must be specified when the connection handle is created) and must be freed before the environment handle can be freed.

Statement handles are used when querying the data source (e.g. when making an SQL SELECT statement or inquiring how many tables are present in a data source). Statement handles are tied to a given connection handle. Multiple statement handles can be assigned to the same connection.

C.1.2 Application Steps

An application wishing to access a data source must go through the following steps [Microsoft97, 83ff]:

1. Allocate an environment handle.

2. Allocate connection handles.

3. Connect to data sources using connection handles.

4. Allocate statement handles.

5. **Perform queries**.

6. Deallocate statement handles.

7. Disconnect from data sources.

8. Deallocate connection handles.

9. Deallocate environment handles.

Note that all statement handles associated with a given connection must be freed before the connection's handle can be freed, and all connection handles must be freed before the environment handle can be freed. Failure to do so results in (possibly dangerous) errors.

C.1.3 Result Sets

Queries to data sources—such as asking for a list of tables in a database, asking for a list of columns in a table, or posing some SQL statement—generally utilize statement handles. The results of the queries may be retrieved using the statement handle and are often in the form of tables called *result sets* [Microsoft97, 175ff]; values in the result set are accessed one cell at a time.

C.2 The ODBC_PL Library

The library consists of the PROLOG files 'odbc_functions.pl' and 'odbc_data.pl'; the PROLOG routines utilize the ODBC API via the DLL 'odbc_pl.dll'. The predicates of the ODBC_PL library mask much of the complexity associated with accessing a datasource. In many cases, the routines mimic the look and functionality of ProData routines (this is intentional). Sometimes, the functionality of ProData has been exceeded.

Please note the following:

- Connection handles follow the design of ProData. Handles are integers; they start at 1 and continue indefinitely.

- Statement handles are allocated and deallocated dynamically. PROLOG users need never know of their existence.

- Closing a connection automatically closes all statement handles associated with that connection and deallocates the connection handle. The same can be said for environment handles.

- In the case of SQL select statements (made using `odbc_sql_select`), values in a result set are retrieved one row at a time and placed into a list; further rows can be returned upon backtracking. Alternatively, the data can be returned as a single list; each member is also a list (corresponding to a row of the result set).

- Solutions to SQL select statements are fetched in *blocks* which are held in PROLOG's memory (i.e., only the first 50 answers are fetched from a data-source). When the solutions in a block have been exhausted, the query is posed again. Answers which were returned in the first block are discarded (i.e. the first 50 answers are discarded, and the second 50 are returned). While expensive (the query is posed several times and answers are discarded), this technique ensures re-entrancy and cuttability.

C.2.1 ODBC_PL Predicates

---

odbc_start/0

Loads the 'odbc_pl.dll' and allocates an environment handle. This predicate must be called if any of the odbc_pl predicates are to be used.

---

odbc_stop/0

Frees the environment handle and unloads 'odbc_pl.dll'. If any connections are open, then these are closed.

---

close_open_connections/0

Closes all open connections. In addition, all statement handles are closed.

---

odbc_connect/2

```
odbc_connect(+DSN, -Handle)}

DSN:        A string indicating a data source to which to connect.
Handle:     An integer; the handle of the connection.
```

Opens a connection to the specified datasource (compare ProData's db_connect). A given datasource may be connected to multiple times.

---

odbc_connect/4

```
odbc_connect(+DSN,+USER,+PASSWORD, -Handle)}

DSN:        A string indicating a data source to which to connect.
USER:       A string; the user ID used when the connection is made.
PASSWORD:   A string; the password used when the connection is made.
Handle:     An integer; the handle of the connection.
```

Opens a connection to the specified datasource with the specified user ID and password. A given datasource may be connected to multiple times.

---

odbc_disconnect/1

```
odbc_disconnect(+Handle)}

Handle:     An integer; the handle of the connection to be closed.
```

Closes a connection to a datasource previously opened with `odbc_connect`. All open statements associated with the Handle are also closed.

---

odbc_tables/2

```
odbc_tables(+DSNHandle, -Tables)

DSNHandle:  An integer; the handle to a datasource.
Tables:     A list of the tables in the datasource.
```

Returns a list of the tables in the datasource. See `odbc_tables_complete`.

---

odbc_tables_complete/2

```
odbc_tables_complete(+DSNHandle, -Tables)

DSNHandle:  An integer; the handle to a datasource.
Tables:     A list of the tables in the datasource.
```

Returns a list of the tables in the datasource. Each element of the list will be a row from the result set created by the ODBC function `SQLTables`. The table's name and type are included.

---

odbc_columns/3

```
odbc_columns(+DSNHandle, +Table, -Columns)

DSNHandle:      An integer; the handle to a datasource.
Table:          A string; the table to which the columns belong.
Columns:        A list containing the names of columns in the table.
```

Returns a list of the columns in the specified table in the datasource.

---

odbc_columns_complete/3

```
odbc_columns_complete(+DSNHandle, +Table, -Columns)

DSNHandle:  An integer; the handle to a datasource.
Table:      A string; the table to which the columns belong
Columns:    A list containing the names of columns in the table.
```

Returns a list of the columns in the specified table in the datasource. Each element of the list will be a row from the result set created by a call to the ODBC function `SQLColumns`.

---

odbc_attach/3

```
odbc_attach(+DSNHandle, +Predicate, +Table)

DSNHandle:      An integer; the handle to a datasource.
Predicate:      An atom; the predicate to create using
                the specified table.
Table:          An atom; the table to be attached to
                using the specified predicate.
```

Attaches a predicate name to a table so that the table can be used as a normal PROLOG predicate.The predicate will have the same arity as the table. (Compare to ProData's `db_attach`).

---

odbc_sql_select/3

```
odbc_select(+DSNHandle, +SQL, -Result)

DSNHandle:      An integer; the handle to a datasource.
SQL:            A string; an SQL select statement.
Result:         A list; the result of the SQL select statement.
```

Returns a result set to the specified SQL select statement posed to the datasource indicated by the handle (compare ProData's `db_sql_select/3`). Results are returned one row at a time (tuple-at-a-time); more solutions can be found upon backtracking. Alternatively, results can be returned as a single set (a list of lists, each element being a single row).

Whether results are to be returned tuple-at-a-time or set-at-a-time is specified in the dynamic fact `odbc_sql_select_flag(X)`, where `X` is either `tuple` or `relation`.

---

odbc_tuple/3

```
odbc_tuple(+DSNHandle, +Table, -Result)

DSNHandle:  An integer; the handle to a datasource.
Table:      A string; the table from which tuples are fetched.
Result:     A list; a tuple from the table.
```

Returns a tuple (row) from the specified table (compare ProData's `db_tuple/3`). Note: no unification is possible in the `Result` argument.

---

odbc_sql/2

```
odbc_sql(+DSNHandle, +SQL)

DSNHandle:      An integer; the handle to a datasource.
SQL:            A string; the SQL statement to be executed.
```

Executes the specified SQL statement. Returns nothing.

---

odbc_getinfo/3

```
odbc_getinfo(+DSNHandle, +Attribute, -Return)

DSNHandle:  An integer; the handle to a datasource.
Attribute:  An integer or atom; the attribute to be checked.
Return:     Integer or atom: the value associated with the attribute.
```

Each connection has associated with it a myriad of attributes (see ODBC function SQLGetInfo); values associated with each attribute may be obtained using this predicate. Valid attributes are contained in textttodbc_getinfo_data(Atom,Integer) clauses, which store the atomic and integer representation of each attribute.

---

odbc_getfunctions/3

```
odbc_getfunctions(+DSNHandle, +Function, -Return)

DSNHandle:  An integer; the handle to a datasource.
Function:   An integer or atom; the function to be checked.
Return:     0 or 1: indicates whether a given function is supported
```

Determines whether the data source supports a specified ODBC function; Valid functions are contained in texttttodbc_getfunction_data(Atom,Integer) clauses, which store the atomic and integer representation of each function.

---

odbc_primarykeys/3

```
odbc_primarykeys(+DSNHandle, +Table, -Keys)

DSNHandle:  An integer; the handle to a datasource.
Table:      A String; a database table.
Keys:       The attributes serving as Primary Keys in the table.
```

Returns the primary keys of a table. Note that not every datasource supports this function. If it does not, then an error message will be generated.

---

odbc_foreignkeys/3

```
\begin{center} \rule{5.5in}{.02in}   \end{center}
DSNHandle:  An integer; the handle to a datasource.
Table:      A String; a database table.
Keys:       The attributes serving as Foreign Keys in the table.
```

Returns the foreign keys of a table. Note that not every datasource supports this function. If it does not, then an error message will be generated.

---

odbc_datasources/2

```
odbc_datasources(+Type, -Sources)

Type:       an atom from the list [all, user, system].
Sources:    a list of the Sources of the given type.
```

Returns a list of sources of specifed type that have been registered with ODBC. Elements of the list will be a two-element list containing the datasource name and an English description of it.

---

odbc_datasources/1

```
odbc_datasources(-Sources)

Sources: a list of the Sources of the given type.
```

Returns a list of sources registered with ODBC. Both user and system sources are returned. Elements of the list will be a two-element list containing the datasource name and an English description of it.

---

odbc_drivers/1

```
odbc_drivers(-Drivers)

Drivers:    A list of drivers registered with ODBC.
```

Returns a list of drivers that have been registered with ODBC.

---

odbc_datatype/2

```
odbc_datatypes(+DSNHandle, -Types)

DSNHandle:      An integer; an open connection
Types:          A list of SQL data types supported by the datasource.
```

Given a connection handle, returns a list of SQL data types supported by the datasource associated with the handle. Elements of `Types` will be of the form `[Type, Length]`, where `Type` is the name of the data type and `Size` is the data type's maximum allowed size (in characters).

---

odbc_typeinfo/2

```
odbc_typeinfo(+DSNHandle, -Types)

DSNHandle:      An integer; an open connection
Types:          A list of SQL data types and associated information
```

Given a connection handle, returns a list of SQL data types supported by the datasource associated with the handle. Answers correspond to the result set produced by ODBC function `SQLGetTypeInfo`.

---

odbc_getconnect_att/3

```
odbc_getconnect_att(+DSNHandle,+Attribute, -Return)

DSNHandle:      An integer; an open datasource connection.
Attribute:      An atom or integer; the attribute for which
                information is desired
Return:         The current state of the attribute.
```

Given a specified connection handle and attribute name, returns the current state of the attribute. Valid attribute names are stored in PROLOG of the form

```
odbc_connect_att_data(Name,Integer,Type).
```

`Type`, a `0` or `1`, indicates whether the attribute returns a number (0) or a string (1).

---

odbc_create_dsn/4

```
odbc_create_dsn(+Type, +Driver, +DSN, +File)

Type:   0 or 1; indicating whether the data source
        is to be registered as user (0) or system (1).

Driver: The driver to be used with the source.
DSN:    The name to be used for the source.
File:   The physical file to be used as the datasource.
```

Registers a file with ODBC using the specified name, driver, and type. User ID and password are set to null strings.

---

odbc_remove_dsn/3

```
odbc_remove_dsn(+Type, +Driver, +DSN)

Type:   0 or 1; indicating a user (0) or system (1) datasource.

Driver: The driver associated with the datasource.
DSN:    The datasource to be removed.
```

Un-registers the specified datasource from ODBC.

Bibliography

[Bosworth01] Bosworth, Dale. 2001. "Working together for the health of the Land." Speech at Natural Resources Summit, September 28, 2001, Sacramento CA. http://www.fs.fed.us/intro/speech/2001/2001sep28-Sierrafinl.htm (accessed April 2, 2002).

[Bhogal96] Bhogal, A. S., D. G. Goodenough, D. Charlebois, S. Matwin, S. Portigal, H. Barclay, A. Thomson, and O. Niemann. 1996. "SEIDAM for forestry: Intelligent fusion and analysis of multi-temporal imaging spectrometer data." *Proceedings of the 26th International Symposium on Remote Sensing of Environment*. Vancouver, BC, Canada: 59-62.

[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.

[Chen96] Chen, N. 1996. "Logic Server: A client/server model for decision support systems." M. S. thesis, University of Georgia.

[Codd70] Codd, E. F. 1970. "A relational model of data for large shared data banks." *Communications of the ACM*. 13(6):377-387.

[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).

[Englemore86] Engemore, R., and T. Morgan, eds. 1986. *Blackboard Systems*. Reading, MA: Addison-Wesley Publishing Company, Inc.

[Goodenough97] Goodenough, D. G., D. Charlebois, A. S. Bhogal, and S. Matwin. 1997. "Automated forest inventory update with SEIDAM." In *Proceedings of the 1997 IEEE International Geoscience and Remote Sensing Symposium*. August 4-8, 1997, Singapore:670-673.

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

[Interagency95] Interagency Ecosystem Management Task Force. 1995. "The Ecosystem Approach: healthy ecosystems *and* sustainable economies. Volume I—Overview." http://www.denix.osd.mil/denix/Public/ES-Programs/Conservation/Ecosystem/ecosystem1.html (accessed April 2, 2002).

[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.

[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.

[Li00] Li, Harbin, David I. Gartner, Pu Mou, and Carl C. Trettin. 2000. "A landscape model (LEEMATH) to evaluate effects of management impacts on timber and wildlife habitat." *Computers and Electronics in Agriculture*. 27:263-292.

[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.

[McGregor01] McGregor Model Forest Association and the University of Northern British Columbia. 2001. *Echo Planning System: Overview Manual*. Prince George, BC, Canada. http://www.mcgregor.bc.ca/publications/Echo/OverviewManual.PDF (accessed April 2, 2002).

[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.

[Ni89] Ni, H. Penny. 1989. "Blackboard systems." In *Handbook of Artificial Intelligence*, Vol IV, edited by Avron Barr, Paul R. Cohen, and Edward A. Feigenbaum. Reading, MA: Addison-Wesley 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.

[OEP95] Office of Environmental Policy. 1995. "Memorandum of understanding to foster the Ecosystem Approach." December 15, 1995, Washington DC. http://www.fhwa.dot.gov/legsregs/directives/policy/memoofun.htm http://www.usace.army.mil/inet/usace-docs/eng-pamphlets/ep1165-2-502/a-a.pdf (accessed April 2, 2002).

[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.

[Reynolds97] Reynolds, K., S. Murray, M. Saunders, J. Slade, and B. Miller. 1997. "Knowledge-based decision support for environmental assessment." In *Proceedings of the 7th symposium on systems analysis in forest resources*, Traverse City, MI. Edited by Jeremy Fried, Larry Leefers, and Mike Vasievich.

[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.

[Shalfield2001] 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.

[Ullman93] Ullman, Jeffrey D and Raghu Ramakrishnan. 1993. "A survey of research on deductive database systems." *The Journal of Logic Programming.* 23(2):125-149.

[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.

[Wielemaker00] Wielemaker, Jan. *SWI-Prolog External Database.* University of Amsterdam, The Netherlands. June 26, 2000. http://www.swi-prolog.org (accessed April 2, 2002).

[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.