6-20-1988

dbProlog: a Prolog/relational database interface

Diane Oagley

Rochester Institute of Technology
School of Computer Science and Technology

dbProlog

A Prolog/Relational Database Interface

by

Diane M. Oagley

A thesis, submitted to
The Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

John A. Biles 7/7/88

Peter G. Anderson 6 July 1988

Walter Wolf 6/6/88

June 20, 1988

Title of Thesis: dbProlog: A Prolog/Relational Database Interface

I Diane M. Oagley prefer to be contacted each time a request for reprodiction is made. I can be reached at the following address:

Date: July 5, 1988

Acknowledgements

I would like to thank my thesis committee, Al Biles and Walter Wolf. They provided sound advice on determining the scope of this thesis, as well as encouragement along the way. Also, their availability and quick turnaround of revisions allowed me to stay on a rigorous schedule.

Steve cannot be thanked enough for his quiet understanding and patience. A more supportive husband would be hard to find. I also thank Frank, my "colleague at a midwestern university" and John, my "mentor" for their intellectual wit and support.

Mostly, I thank Mom and Dad; my first and best teachers.

Abstract

dbProlog is a prototype system that provides a C-Prolog user access to data in an external relational database via both loose and tight coupling. To the application programmer, dbProlog is a group of six built-in Prolog predicates that effect communication between a C-Prolog process and a database management system process. Prolog application program statements may be written using the six predicates to make the interface transparent to an end-user. The system is based on a "driver" process that must be customized to the interfaced DBMS and whose primary function is the translation of requests and replies between C-Prolog and the DBMS. dbProlog supports Prolog's depth-first search on database retrievals by producing the next record when the retrieval predicate is encountered upon backtracking. dbProlog also supports multiple active database retrievals, as may be required by a Prolog rule that references two or more database retrievals, or by a recursive rule.

Key Words

Prolog, DBMS, Relational database, Logic Programming, Query languages

Subject Categories

ACM Computing Review:

I.2.5 Computing Methodologies
Artificial Intelligence
Programming Languages and Software

H.2.3 Information Systems
Database Management
Languages

Inspec (IEEE/IEE) Computer and Control Abstracts:

6140D High-level Languages
6160D Relational DBMS
6180 User Interfaces

Table of Contents

Chapter 1

Introduction

The emergence of logic programming languages and relational database management systems (DBMS) have independently been very important to modern computer application development. Separately, these areas have much to offer application developers, while the coupling of the two technologies potentially offers much more in the way of application development tools. These tools can enhance the quality, efficiency and capabilities of many types of applications and also allow creation of new application types. Applications built using the current relational database or artificial intelligence technology have much to gain from such a coupling. These benefits provide the justification for such a system.

Researchers in the field of artificial intelligence are being challenged to find ways to support sophisticated logic programming applications, such as expert systems, that require extensive knowledge bases. Therefore, powerful data management systems are required to support the processing of complex application "knowledge." The use of a generalized DBMS to store an application's knowledge base can provide flexibility in the size and growth of the knowledge base, while at the same time take advantage of the access and storage mechanisms of the DBMS.

On the other hand, logic programming can provide benefits for DBMSs by providing capabilities for improving user interfaces, enhancing access methods, and easing DBMS control and operation. Databases which are integrated with a form of logic programming, are referred to as "Intelligent Database Systems" or "Expert Database Systems." The addition of logic programming techniques to DBMSs can enhance both database usage and database operation.

As a result of this thesis, a working prototype system that couples Prolog with a relational database was implemented. This was done by constructing an interface that can be used by the Prolog programmer to access a database through Prolog programs. The basis of this system is the addition of a set of standard predicates to Prolog, for access to any relational database system. The major goal was to provide a system that facilitates communication and interaction between Prolog and an external relational database. The system should provide the same results and functionality to an application end-user as if the external database tuples were a part of Prolog's internal database. The name given to the system is "dbProlog".

A detailed background of these areas is given in Chapter 2. Chapter 2 begins with a high level description of both relational database theory and Prolog, and a more detailed look at the benefits obtained by combining them. There is much literature describing the conceptual levels of coupling logic programming languages and databases. Chapter 2 provides an overview of this.

Chapter 2 also discusses basic issues that must be addressed before building such a system, such as the user interface and the various types of systems that can be built as a result of the extent of physical coupling. Chapter 3 contains the results of a literature search of previous work in this area, including systems that are available in the software marketplace. Chapter 4 presents design details of the proposed system and includes structure charts, data flow diagrams, and a data dictionary. Chapter 5 provides implementation details such as system structure and interprocess communication mechanisms while Chapter 6 discusses the resulting implementation including operation and problems encountered. Conclusions and ideas for future enhancements are given in Chapter 7.

Chapter 2

Background

2.1 Prolog and relational databases

Both Prolog and relational databases are powerful and popular tools used in application development. They are typically used independently; however, there are many application possibilities when they are combined. The following sections will provide brief overviews of the relational database model and Prolog, and will also compare and contrast the two.

2.1.1 The relational database model

The relational database model was first defined by E.F. Codd [Codd70]. Many commercial DBMSs have been implemented on the basis of this model, such as INGRES and DB2. "The relational model can be characterized as a way of looking at the data, that is, a prescription for a way of representing the data and for a way of manipulating that representation" [Date86].

Relational databases have proven to be powerful yet simple systems for managing large volumes of data in many varied application areas. However, not all DBMSs termed "relational" adhere to the full extent of the relational model. There are three levels of the relational database model (minimally

relational, relationally complete, and fully relational) used to describe the "relationality" of a DBMS [Codd70].

Relational databases are databases that can be perceived by the user as a collection of rectangular tables. The table columns correspond to attributes, and the rows correspond to tuples. Each tuple of a relation (table) must contain the same number of values and be defined over the same set of attributes. Also, each relation must have a set of attributes which uniquely identifies each tuple in the relation, called the primary key. Information is neither contained in the ordering of the tuples in a relation, nor in the ordering of attributes in a tuple. Therefore, order of tuples and/or attributes is not important [Date86], [Ullm82].

A relational DBMS must support the operations of *SELECT*, *PROJECT* and *JOIN* (defined in Appendix A) without requiring the predefinition of physical access paths to the data. Other functions are also commonly supported, such as the capability to create and destroy tables. In addition, relations must be normalized in order to avoid maintenance anomalies. The are several levels of normalization [Date86].

A more extensive review of the relational model is provided for the interested reader in Appendix A.

2.1.2 Prolog

Prolog is a language for programming in logic. It had its beginning around 1970, the year that the relational theory for databases had its start, and since then has been found to be of

use in many application areas [ClMe84]. Prolog is used in the field of artificial intelligence in applications such as expert systems and natural language parsing and understanding. Prolog also seems to be well suited for applications such as rule-based decision systems, compilers, database extensions, and specification and symbolic manipulation of complex physical systems like digital circuits [Pere87].

Prolog is a very high-level language based on the Horn clause subset of the first-order predicate logic. It is a language for symbolic computation and thus is an appropriate language for solving problems that involve objects and relations among objects. Prolog has a built-in theorem prover, or inference engine, which operates in a top-down, depth-first manner. Prolog also maintains a built-in database that stores the components of a Prolog program [Brat86], [ClMe84], [StSh86].

A Prolog program consists of clauses of which there are three types: facts, rules, and questions. Facts declare things that are always unconditionally true, rules declare things that are true depending on a given condition, and questions provide the means for a user to ask the program what things are true. Prolog clauses consist of a head and a body. The body is a list of goals (or questions) separated by a character indicating conjunction (a comma in C-Prolog) [Brat86].

The execution of a Prolog program involves depth-first search with backtracking, to attempt to satisfy the goals in the body of a clause. These goals are satisified by "unification", which is the process of matching Prolog structures. Unification may result

in a variable being substituted with an object, which is called "instantiation" [Brat86], [ClMe84].

The interested reader is directed to Appendix B for a more extensive review of Prolog.

2.1.3 Prolog and relational database systems - similarities and differences

Prolog has seldom been recognized for its use as a relational DBMS. However, Prolog has some of the characteristics that are commonly found in a relational DBMS [BrJa84]. A fact in Prolog is analogous to a tuple in a relational database. Therefore, a group of facts sharing the same predicate and arity is similar to a table of a relational database. A rule in Prolog can be used to specify a "view" into the Prolog database of facts. This is similar to the view concept of relational databases. Prolog has mechanisms for manipulating the database (inserting and deleting) via built-in predicates such as *assert* and *retract*. It also has powerful built-in mechanisms for specifying and answering queries to the database, such as depth-first search with backtracking.

Even though Prolog possesses most of the data manipulating capabilities of a relational DBMS, there are many areas where Prolog falls short of being a relational DBMS [ChWa86], [Rett87a]. One of the most obvious shortcomings of Prolog is the limitation of database size. Most implementations of Prolog are "in-memory" systems; therefore, Prolog's database cannot exceed the available

system memory. This prevents using Prolog's database for an application requiring a substantial capacity.

Prolog seldom has an efficient way of accessing facts such as direct hashing or B-trees. Prolog usually stores facts that share the same predicate together; however, to find a specific fact, it usually must read sequentially through the group of facts.

The relational model states that the order of tuples in a relation is immaterial, as is the order of attributes in a tuple. However, in Prolog, the order of arguments to a predicate is important, especially for unification. Prolog's depth-first search characteristics and top-down executing inference engine make the order of clauses in a Prolog database important.

Prolog does not support a multi-user environment for applications where many users require concurrent accesss to the same knowledge base. It also does not provide any scheme of access security to its database. Transaction and recovery support are also important areas in DBMSs which are not supported by Prolog.

Prolog does not provide any type of query optimization functionality, which is a feature offered by some relational DBMSs. It also has no clear way of expressing database structure. More specifically, the Prolog database cannot easily be structured into heirarchies and networks. There is no support for relationships such as primary key/foreign key. Prolog also does not provide any built-in mechanism for enforcing integrity constraints on the database.

Even though Prolog has several shortcomings when compared to a full-scale DBMS, it does have some features that are not typically found in DBMSs [Rett87a]. Prolog can be very flexible and powerful as a query language. Its built-in theorem prover can be very effective, especially when used in combination with other language features. Database objects can be defined dynamically, and new relations or tuples of any type can be added to the database at any time. This allows for flexibility of data modeling and expression. Prolog also permits structures as attributes of a tuple, whereas relational databases require all attributes to be atomic data elements. Since Prolog supports recursion, both data structures and queries may be defined recursively. Prolog has built-in facilities, such as backtracking and execution trace, which could be very useful in database application implementation.

2.2 Reasons for coupling logic programming languages with relational databases

A system constructed by the coupling of a logic programming language and a relational DBMS can provide benefits to both logic programming applications and relational database applications. This section will describe the ways that such application areas can benefit from such a tool. Outside of these types of applications, there also exist hybrid applications such as decision support systems that could benefit from such a tool. These systems usually require large amounts of data (probably

stored in a database), an "expert" component for making inferences, and possibly a mathematical component for complex mathematical calculations. Clearly, a tool that directly supports all three of these components would be desirable.

2.2.1 Benefits for artificial intelligence applications

Some artificial intelligence applications, such as sophisticated expert systems, may require a very large knowledge base of facts and rules. Most current implementations of Prolog are "in-memory" systems, i.e. the program's knowledge base is limited by the size of the computer's memory. One solution to this problem is a virtual memory paging system for the knowledge base. However, this may be costly in algorithm development time, efficiency, and swapping overhead. An alternate solution is to store the fact portion of the knowledge base in a generalized DBMS.

Certain applications may require data that are already available in some database. "Snapshots" could be taken from the database and loaded into the expert system; however, this would be of little value if the data in the database were dynamically changing. Also, memory restraints may cause problems if the database is extensive. Allowing the expert system direct access to the actual database would provide it with timely information with which to work.

The use of a general purpose DBMS to store application knowledge provides alternate methods for processing the data in a

knowledge base, such as a powerful query language. A DBMS may also provide efficient methods for loading the knowledge base and maintaining it.

2.2.2 Benefits for database applications

Database applications may be extended and enhanced by logic programming techniques. The benefits to database applications fall into two distinct areas: database usage and database operation.

2.2.2.1 Enhanced database usage

The enhancements to database usage that a logic programming language such as Prolog can provide generally fall into three categories: extraction of more meaning from the database, wider variety of queries supported, and support for sophisticated data modeling techniques [Berg85], [Chan78].

2.2.2.1.1 Extracting more meaning from the database.

Prolog can provide the ability to create virtual relations by use of Prolog rules. These virtual relation are more general than the concept of a "view" in relational database theory. A view in relational database theory is defined statically, while a virtual relation is a moving window, dependent upon the parameters that it is given by way of variable instantiation. Another way a virtual

relation is more general than a view is that a virtual relation may be defined recursively, where a view cannot. An example of a Prolog virtual relation follows [Chan78]:

father(X,Y) - base relation

ancestor(X,Y) - virtual relation

The virtual relation *ancestor(X,Y)* can then be defined by the following Prolog statements:

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Z) :- father(X,Y),ancestor(Y,Z).
```

Therefore, by the use of the concept of a virtual relation, meaning is added to the database, that did not exist explicitly.

Prolog can also provide the ability to add pieces of information to the database, similar to the addition of attributes to a relation, without the implications of changing the physical database structure. Also, if the rules of relational database normal forms are strictly adhered to, additions of relation attributes may sometimes violate these rules, causing the need for additional relations. Using Prolog to add attributes to a database is illustrated by the following example [Berg85]:

employee(Name,Number,Gender) - base relation

If it is also desired to store the room number of the locker room

that the employee uses (one for men, one for women), this attribute, "lockroom", could be added to the employee relation, which would require restructuring the database. This restructured database would violate the second normal form rule of relational theory because of the functional dependency of the attribute "lockroom" on the attribute "gender." The correct way to add the attribute "lockroom" to the database would be to add a new relation, such as *lockroom(Gender,Number)*. The addition of this rather small relation would cause modifications to the database structure and schema, and would require using the *JOIN* operation with the employee relation in order to use the data in the lockroom relation. However, by creating two simple Prolog rules, the "lockroom" data can be easily accessible without the aforementioned complications:

```
lockroom(Empname,a120) :- employee(Empname,_,male).
lockroom(Empname,a122) :- employee(Empname,_,female).
```

Where *a120* and *a122* are the room numbers for the men's and women's locker rooms, respectively.

2.2.2.1.2 Supporting a wider variety of queries and user interfaces

Relational database query languages are typically derived from the relational algebra. The commands of these query languages are oriented to row and column operations, such as

SELECT, *PROJECT* and *JOIN*. A logic programming language, such as Prolog, makes possible other types of queries, which are typically not available in relational DBMSs. One such type of query, is that based on set theory. Prolog has built-in predicates to accomodate such a query as illustrated by the following example [Berg85]:

> One may wish to query the database to see if the word "MALTIC" exists as the value of any attribute of any tuple. If typical relational DBMS queries are used, many queries may have to be issued (testing values of each attribute) until an answer is found. This query could be answered more simply, using Prolog, by converting each tuple into a set, and then using the predicate *member* to test for the occurrence of "MALTIC."

Prolog's lack of data-typing may also make it attractive for queries where one wishes to ignore data type declarations. Social Security numbers are an example of this. Traditional uses of Social Security number require it to be entered with embedded hyphens. Therefore in a typical relational DBMS environment, its type would be defined as character, and therefore retrievals on the basis of numerical order would be cumbersome. However, since Prolog can compare alphabetic order as well as numeric, requests could be given such as [Berg85]:

"Find the tuples whose Social Security numbers are between 123-00-0000 and 125-99-9999."

Logic programming has already been noted for its usefulness in the development of natural language systems [Walk84], [Dahl82]. The existence of a coupling between a DBMS and Prolog would clearly support the rapid development of natural language user interfaces to databases. This concept also can be taken one step further into the area of speech understanding, where complex expert systems may be required to accept verbal commands and construct database queries from them.

2.2.2.1.3 Support of sophisticated data modeling techniques

Data modeling is "a representation of the data which is organized to accomodate particular applications while remaining faithful to the structure of the phenomena being modeled" [Berg85]. Relationships such as a hierarchy are not supported explicitly by a traditional relational database. Through the use of a logic programming language, the concept of structured attributes can be created by adding rules to the data.

2.2.2.2 Enhanced database operation

Enhancing a DBMS with expert system features may not only increase the capabilities of the DBMS but may make it safer and

more efficient to use. Two areas where logic programming techniques may be employed are transaction management and query optimization.

One type of transaction management that could be enhanced by logic programming techniques is integrity checking. Integrity checking refers to defining a set of rules that the data in the database must adhere to, and then enforcing them on transactions that are attempted on the database. Integrity rules can be expressed in Prolog, and an expert system may be used to determine which rule should be applied to an incoming transaction. An expert system can also be useful in determining what action is to be taken when an integrity rule is violated. Similar techniques may also be used to implement or enhance security or locking schemes.

Another form of integrity checking in databases is the concept of "domains." A domain is a pool of values from which one or more attributes draw their actual values [Date86]. For example, when specifying months of the year as integers, the domain is "the integers between 1 and 12 inclusive." Logic programming techniques, such as Prolog rules, can be used to implement the concept of domains when coupled to a DBMS.

Logic programming techniques can also be used in identification of a fast and efficient way to execute a query. According to [JaVa84], there are three ways that logic programming techniques can be used to perform query optimization.

1. Use of rules to guide the transformation of a query into a faster and more efficient query from many applicable query transformations.
2. Application of the integrity constraints of the database to the query to simplify query selection - In the extreme case, where a query's conditions are disjoint from the allowable conditions as specified by an integrity rule, the query can be resolved, (returning a null set) without ever actually accessing the database.
3. Optimization of multiple queries - Selected query results can be "remembered" to be used at a later time (when a query is issued that is based on the result of a previous query). Also, an expert system can be used to recognize commonality in a batch of queries, increasing efficiency by answering them collectively.

2.3 Conceptual levels of coupling Prolog and databases

The coupling of Prolog to a database system can be done at one of several levels. The levels range from pure, unenhanced Prolog, with its built-in database, to Prolog communicating with a general purpose DBMS, appearing to the user as a transparent extension to Prolog's own database.

2.3.1 Elementary data management in Prolog

This is the simplest case of Prolog-database coupling. All data are kept in Prolog's built-in database. Prolog's built-in predicates such as *assert* and *retract* are used to manage the data. Application-specific routines must be written for other database operations and retrievals.

At this level, it is difficult to distinguish between Prolog programs and data, since both are stored in the database. However, this type of implementation allows for direct interaction of Prolog's inference engine with the data. Prolog supports the concept of virtual relations by using rules, and also acts as a query language.

There are two major limitations to using pure Prolog as a DBMS [VaC183]:

1. Size of database - Most Prolog implementations only support databases that fit into main memory. Therefore, the applications that use such a system are limited.
2. General DBMS facilities - Prolog typically does not support data management functions that are usually supported by a general purpose DBMS. These include data dictionary capabilities, multiple-record retrievals, and type restrictions.

2.3.2 Generalized data management in Prolog

If the application database is too large to fit in main memory, the data can be stored in external files, and file-handling operations can be implemented in Prolog to access the data. This is basically the same as implementing a general purpose DBMS with Prolog.

Preferably, the database implementation in Prolog would provide data independence. Therefore, the preferred architecture would be similar to the traditional leveled architecture found in a DBMS. This Prolog implementation of a DBMS requires "the definition of an internal representation of a relational database" [JaVa84]. Prolog's DBMS must provide a mapping between the storage structure of the external files to the internal record representation, as well as a mapping from Prolog operations to external file-handling operations.

An advantage of such a system may be efficiency. By writing the DBMS in Prolog, storage may be tailored more efficiently to match Prolog's search and access strategy.

A Prolog data management system is a disadvantage when an existing large database is required by a Prolog application. Therefore, this Prolog-written DBMS solution does not work, unless the database is converted into the file format required by the Prolog DBMS. Writing a DBMS in Prolog is also a disadvantage, as it seems to be wasteful when there are many general purpose DBMSs available.

2.3.3 Loose coupling with existing DBMS

In cases where a Prolog application needs to access an existing database, the simplest solution is to provide a loose coupling. Loose coupling refers to the extraction of "snapshots" from the external database and storage of these "snapshots" in the Prolog database. After the data are stored internally in the Prolog system, they may then be accessed by the standard Prolog operations. Such a system requires the existence of a communications channel between Prolog and the DBMS.

Three components are required to implement loose coupling [JaVa84]:

1. Link to DBMS with unload facilities
2. Automatic generation of Prolog database from downloaded data
3. Knowledge of which portion of database to extract

Some of the disadvantages of this system are related to determining and extracting only a specific portion of the database. In a loose coupling system, the portion of the database to be extracted must be known and specified in advance. In most cases, a superset of the required data will be extracted, thus reducing efficiency. A problem occurs if the portion of the database that is to be extracted is too large to store in the Prolog database. In addition, the process of selecting a portion

of a database, extracting it and loading it into Prolog may be time consuming. The capacity and speed of the communication channel will effect the amount of time required to download data. Another disadvantage to be considered is the fact that snapshots of a rapidly changing database quickly become obsolete.

2.3.4 Tight coupling with existing DBMS

In cases where Prolog needs direct, real-time access to a database, tight coupling is the best alternative. However, the implementation of a tightly coupled system requires solutions to some non-trivial problems. In a tightly coupled system, Prolog and the DBMS communicate via a communications channel. However, the communication is performed in such a way that it is transparent to the user of the Prolog application. In tight coupling the database appears as if it were an extension of Prolog's own database. In a tight coupling system, Prolog variables may be instantiated via the unification of a Prolog goal with a tuple of an external database relation.

The efficient operation of a tightly coupled system depends on the intelligent management of the communication channel. Naive use of the channel simply redirects all Prolog queries of external relations directly to the DBMS. Intelligent use of the communication channel presents some implementation difficulties such as the number of DBMS calls. Potentially, each Prolog goal may invoke a DBMS call. Also, Prolog works in a tuple-at-a-time fashion, while DBMSs are usually capable of returning sets of

tuples. This incompatibility may require many DBMS calls, one tuple per call, rather than one DBMS call to retrieve all tuples. The complexity of Prolog goals may also present an implementation challenge. Some Prolog goals, expecially those involving recursion may be difficult, if not impossible to translate into DBMS calls.

The above difficulties may be overcome by jointly executing DBMS calls. A way of implementing this is described by [VaC183], [JaVa84] as follows:

> "An amalgamation of the Expert System language with its own meta-language is used. This allows for deferred evaluation of predicates requiring Database calls, while at the same time the inference engine of the Expert System is working."

The "meta-language" optimizes and translates a set of complex queries into a feasible set of DBMS queries. The technique is described in [VaC183].

In a successful tight coupling system, the Prolog system must know how and when to generate and direct queries to the DBMS. In addition to this, the system must also be able to understand the replies from the DBMS and be able to translate them into a form suitable to the Prolog system.

2.4 Design and implementation issues

When planning to connect a logic programming language such as Prolog to a relational DBMS, decisions must be made as to how the system will appear to the user, along with which operations the system will support. The desired efficiency of the interface must also be considered during the early stages of design and implementation. These factors will govern the choice of method for connecting the systems. The implementation chosen may also affect the degree of coupling between the systems.

2.4.1 User interface

One decision concerning the design of a user interface, is determining the placing of the user interface. The user interface may be positioned in one of several places:

1. Prolog - In this case, the user would access the system by accessing Prolog. This approach requires Prolog programming expertise of the user, but also affords the user all of the powerful constructs of Prolog directly.
2. DBMS - In this case, the user only has direct access to the DBMS functionality. Any logical inference or Prolog functionality is done in the background.
3. Prolog or DBMS - This type of interface is a combination of 1 and 2. It provides the user with the capabilities

of both types of systems; however, both types of expertise are expected of the user.

4. Custom Interface - In this case, the user is shielded from the details of directly accessing either type of system, by a "User Interface" program that accepts user commands and routes them to the appropriate system. This may be a less powerful interface for the user experienced in Prolog and/or relational DBMSs; however, it may yield many advantages for the naive user.

The second issue affecting the design of a user interface, is the structure of the commands that must be entered by a user to access the interface.

If the user accesses the system via a Prolog interface, two of the ways that a retrieval command may appear are [Bocc86]:

1. *retrieve([employee.name=Name,*
   employee.salary=Salary],
   employee.salary=100).
2. *Salary=100*
   employee(Name,Salary,...).

Both statements are used to retrieve a record in the relation *employee* where *employee.salary=100* and the variable *Name* gets instantiated to the value of *employee.name*. Example 1 is referred to as a "loose" user interface (not to be confused with loose coupling) between the two systems, while 2 is referred to as a

"close" user interface.

The loose user interface appears as nothing more than the data manipulation language (DML) of the DBMS embedded in Prolog. In this type of interface, the positions of relation attributes is immaterial, and it is not necessary to list all attributes when constructing a query. This is an advantage when users must query relations that contain many attributes. Another advantage of the loose interface is its ease of implementation. Prolog requests for DBMS service could easily be mapped into equivalent DML expressions. One disadvantage of the loose interface is that there is no obvious solution for handling multiple queries and recursion, just as these things are not supported by most relational DBMSs.

The close user interface makes external database access transparent to the user, since queries to the external database (EDB) appear the same as queries to the internal database (IDB) of Prolog. The transparency allows for easy prototyping and growth of applications. When designing an application, Prolog's IDB may be used for data storage. However, when the application is implemented, or when an application outgrows Prolog's database, a DBMS may be linked to for data storage without any change to the application programs. The obvious disadvantage to this type of interface is the amount of mapping and translation of DBMS requests, as compared to the loose user interface.

When constructing an interface between two systems, the designer must determine the extent of the functionality of one

system that will be available to the other system. For example, in the case of a Prolog system interfacing to a relational DBMS, the designer may wish to restrict the interface to support only database retrievals from the Prolog system. In such a system, database content modification via adds, deletes and updates or data definition language (DDL) operations which modify the structure of the database would not be supported.

2.4.2 Enhanced, integrated and coupled systems

The words "enhanced", "coupled" and "integrated" are relative terms used to describe the level of physical connection of the two systems.

2.4.2.1 Enhanced systems

One way to provide a system that connects logic programming and DBMS technologies, is to incorporate the functions of one type of system into the other. This can be done in one of two ways:

1. Enhancing a relational DBMS with logic programming capabilities.
2. Enhancing a logic programming language, such as Prolog, with relational DBMS functionality.

These types of enhanced systems usually require extensive modifications to some existing system to incorporate the added

capabilities.

An enhanced DBMS is most likely implemented by the addition of logic programming inference facilities to a conventional DBMS. A deductive component is embedded into the DBMS, resulting in a DBMS that supports rules and inference. However, unless a full-scale logic programming language, such as Prolog, is incorporated into the DBMS, it is doubtful that such a system could support all functions of Prolog, such as lists and structures. Three ways that the embedded deductive component could be used by the DBMS are [ZoGr87]:

1. Integrated method - Deductive routines are embedded as an integral part of the DBMS.
2. Deductive filtering - Queries are filtered through the deductive component before being processed by the DBMS. This filtering is transparent to the user.
3. Interactive method - The DBMS interacts with the deductive component when required to.

An enhanced logic programming language is one that contains data management facilities to permit support for database users. The logic programming language could be enhanced to support operations on a general purpose database; however, this would require rewriting a large portion of a DBMS in the logic programming language. Another approach is to incorporate a file system and file manager into the language. However, this approach may not afford the degree of storage and access efficiency

provided by a general purpose DBMS [ZoGr87].

2.4.2.2 Integrated systems

In most cases, when two or more systems are said to be integrated, the program code of at least one of the systems has been modified. The two separate systems actually appear as one with a blurred interface. Access to one of the components by the other is accomplished by some "back door" route, rather than the standard access methods of the program.

In a system where Prolog is integrated with a DBMS and the storage structure of the database is known, one can take advantage of this knowledge and access the stored data directly, thus bypassing the access mechanisms of the DBMS. However, by doing so general DBMS facilities, such as concurrency control, are not supported. Therefore, very few total integrations of Prolog with a DBMS exist. The majority of the work that has been done deals with single-user, query-only mode. An example of such a system is EDUCE, which was developed by Bocca [Bocc86]. In this type of system, recursive queries are simpler to implement, because Prolog controls the one-tuple-at-a-time access to the database.

2.4.2.3 Coupled systems

The term "coupling" is used to represent a spectrum of techniques to connect Prolog with DBMSs. In general, when two systems are coupled, they remain independent processes with a

capability to communicate with each other. In contrast to an integrated Prolog/DBMS system, in a coupled system, the Prolog component has no knowledge of the internal database storage structure. Therefore, the Prolog component accesses the data through the DBMS. Since the DBMS facilities are used in this system, functions such as concurrency control are supported. Coupled systems are simpler to implement because operations that modify the database (adds, deletes, and updates) can take advantage of the built-in DBMS facilities. Retrieval operations can use the set retrieval capabilities of the DBMS, but buffering capability must be made available so that the Prolog component can receive records in a one-tuple-at-a-time method. The set retrieval functionality of the DBMS, however, causes the implementation of recursion to be extremely awkward.

High communication costs and speed may be factors in a coupled implementation. When two or more processes must communicate via an interprocess communication mechanism, performance may be inferior to that of a system composed of only one process whose components communicate through shared memory.

The levels of physical coupling can be defined by the division of processing and control among the components. When implementing a coupled system, the capability for future additional subsystem interfacing must be considered. The type of coupling may determine the ease with which other systems may be connected at a later time. Three types of physical coupling are dominant component, equal partners and supervisor.

With the "dominant component" type of physical coupling,

processing and control are handled by only one of the two components. In this master/slave relationship, either the logic programming language or the DBMS simply acts as a server process to accept and satisfy the requests of the other component. This requires the addition of communication and control operations to the component that is chosen to be dominant. All user interaction must take place through the dominant subsystem. The connection of additional components may be difficult, as this will require modifications to the interfacing routines of the dominant component. Figure 2.1 illustrates this type of system [ZoGr87]. PROSQL is an example of 2.1b [ChWa86].

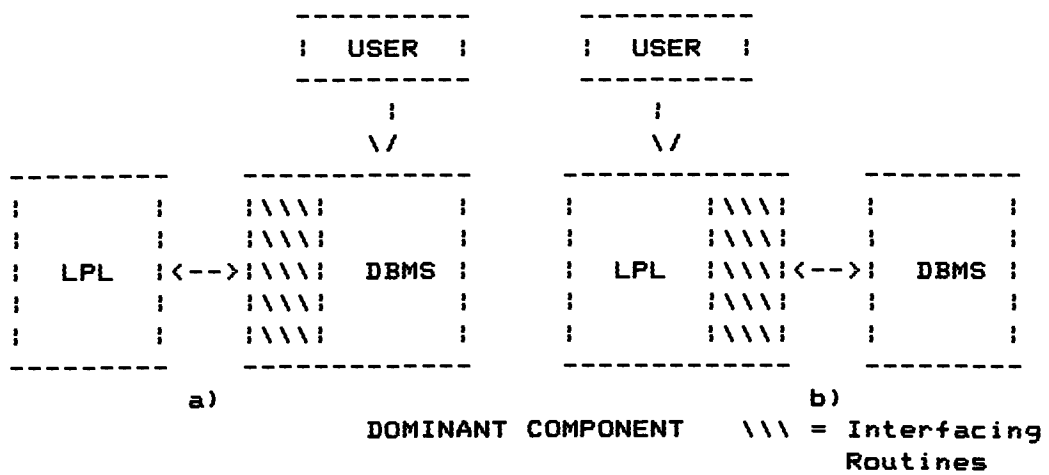


Figure 2.1

The "equal partners" coupling allows each component to assume both master and slave roles. Processing and control are distributed among the components, and therefore, each component must contain interfacing routines. User interaction may be accomplished through either subsystem. This type of

implementation permits incorporation of other components; however, because the two systems are effectively totally independent of each other, problems such as inconsistency, incompatibility and redundancy may arise [ZoGr87]. An "equal partners" system is depicted in Figure 2.2 [ZoGr87].

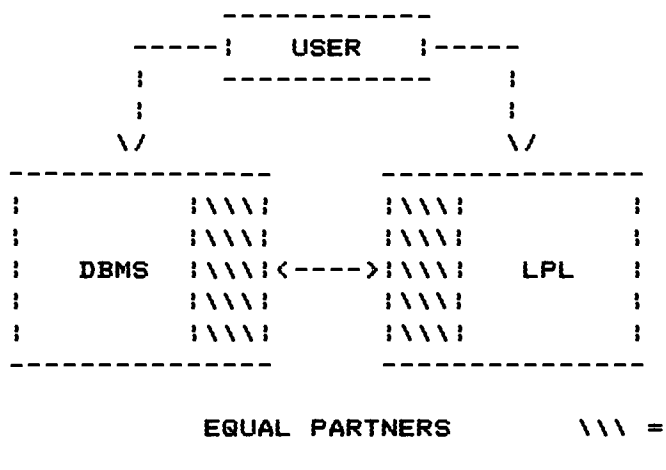


Figure 2.2

The "supervisor" implementation incorporates an independent subsystem where the control routines are concentrated. All processing is handled by the logic programming and DBMS components, which communicate directly with and only with the supervising subsystem. All interfacing routines reside in the supervisor subsystem, and the user only communicates with the supervisor. The supervisor performs all necessary steps for interfacing the systems (e.g. translations). A challenge to the implementer of this type of system is to make full use of the capabilities of each system while avoiding redundancies. The DIFEAD system is an example of this approach [ZoGr87]. Figure 2.3

illustrates the "supervisor" implementation [ZoGr87].

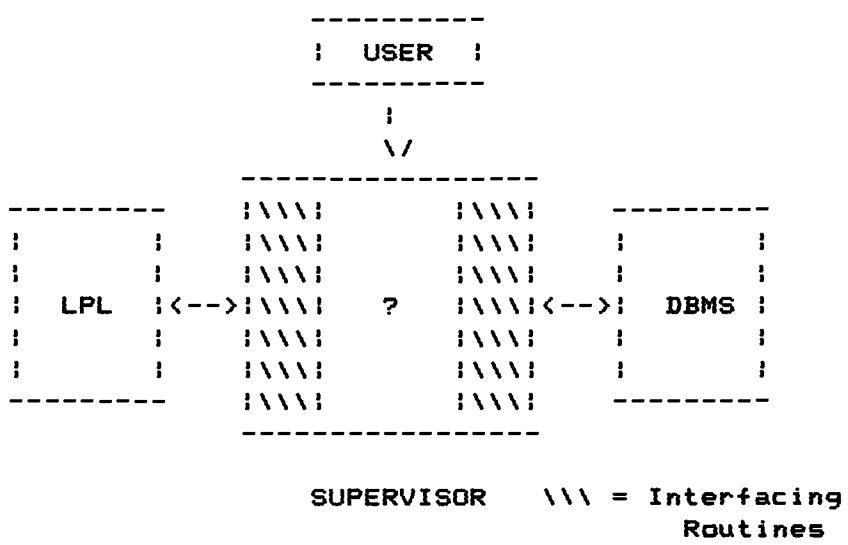


Figure 2.3

Chapter 3

Previous Work

Literature exists on numerous systems that attempt to interface logic programming (including Prolog and various expert systems) with relational DBMSs. The majority of implementations that were found in the literature are prototype or experimental systems. However, commercial software packages are available for personal computers such as Arity/SQL that attempt to offer such a system.

The following section provides an overview of several systems that are described in the literature. The systems reviewed here were chosen to provide a spectrum of designs and implementations. Along with the descriptions, any applications based on the system are also described, if they exist. This section is divided into two parts, mainframe based systems, and microcomputer systems.

3.1 Mainframe/minicomputer systems

The three systems discussed here are experimental or prototype systems. They were chosen to illustrate different approaches to coupling and implementation.

3.1.1 PROSQL

PROSQL is an experimental system that couples Prolog with SQL/DS, an IBM relational database system. SQL/DS is a complete DBMS, as it supports access paths, concurrency, security, recovery, data sharing, and efficient retrieval from a relational database in secondary storage [ChWa86].

PROSQL is a coupled system, whose interface is implemented by the capability to call SQL statements from Prolog. The two components (SQL/DS and Prolog) may run on independent virtual machines linked by a communication channel over which messages are passed in files. SQL/DS functions as a database server to Prolog. PROSQL was designed as a flexible system that allows both loose and tight coupling; however, at the time of the writing of [ChWa86], only loose coupling had been implemented.

The basis for PROSQL is an extension to Prolog that provides the special predicate *SQL*. The complete implementation of PROSQL includes a communication link between Prolog and SQL/DS virtual machines, programs for transforming PROSQL statements to SQL statements, programs to process SQL statements in SQL/DS, programs for loading assertions from SQL/DS into Prolog, and programs for compiling recursive rules into programs.

The PROSQL system is accessed from within Prolog by execution of the special predicate *SQL*. The argument of the *SQL* predicate must be a string that represents a valid SQL statement; however, Prolog variables may be included in the argument. For example,

*SQL('CREATE VIEW T AS SELECT * FROM P').*

would call SQL to create a view named *T* in the external database. This method of database access from Prolog indicates a loose user interface.

Tight coupling is accomplished in PROSQL by using an SQL retrieval statement that includes the keyword *INTO*, as in the following example:

```
SQL('SELECT NAME,SAL INTO *X,*Y
     FROM EMP WHERE MGR=JONES').
```

When the SQL component finds a tuple that satisfies the request, the Prolog component will both load the tuple into its workspace and bind the component(s) of the tuple to the Prolog variable(s) in the *INTO* clause which are preceded by "*". SQL/DS could continue to generate tuples to satisfy the *SELECT* query, in preparation for another possible request of the same *SELECT* call therefore supporting the Prolog concept of backtracking.

Loose coupling statements in PROSQL do not include the *INTO* keyword, as shown by the following example:

```
SQL('SELECT * FROM EMP WHERE SAL > 50000').
```

SQL retrieves the set of tuples that satisfy the request, and they are deposited into the Prolog workspace as assertions with the

predicate specified in the *FROM* clause of the SQL statement.

PROSQL supports the following database operations from Prolog:

1. Data definition (*CREATE, DROP TABLE*)
2. Insertion
3. Retrieval
4. Transaction control (*COMMIT*)

Rules can only be stored in the internal database of the Prolog component of PROSQL because SQL/DS is not designed for rule storage. Recursion is supported by PROSQL. Rather than creating multiple SQL requests to satisfy recursive retrievals, making communication costly, PROSQL uses a compiled approach. Recursive rules are compiled by PROSQL into an iterative program that is executed by SQL/DS. However, it is noted in the literature that the compilation of recursive rules into an iterative program is a "non-trivial problem" [ChWa86].

The SQL/DS component supports multiple-user concurrency control within single SQL operations. However, if a PROSQL user wishes to ensure database consistency through a PROSQL logical transaction consisting of several SQL requests, PROSQL supports this with the statements:

SQL('AUTOCOMMIT OFF') and

SQL('COMMIT WORK').

3.1.2 DIFEAD

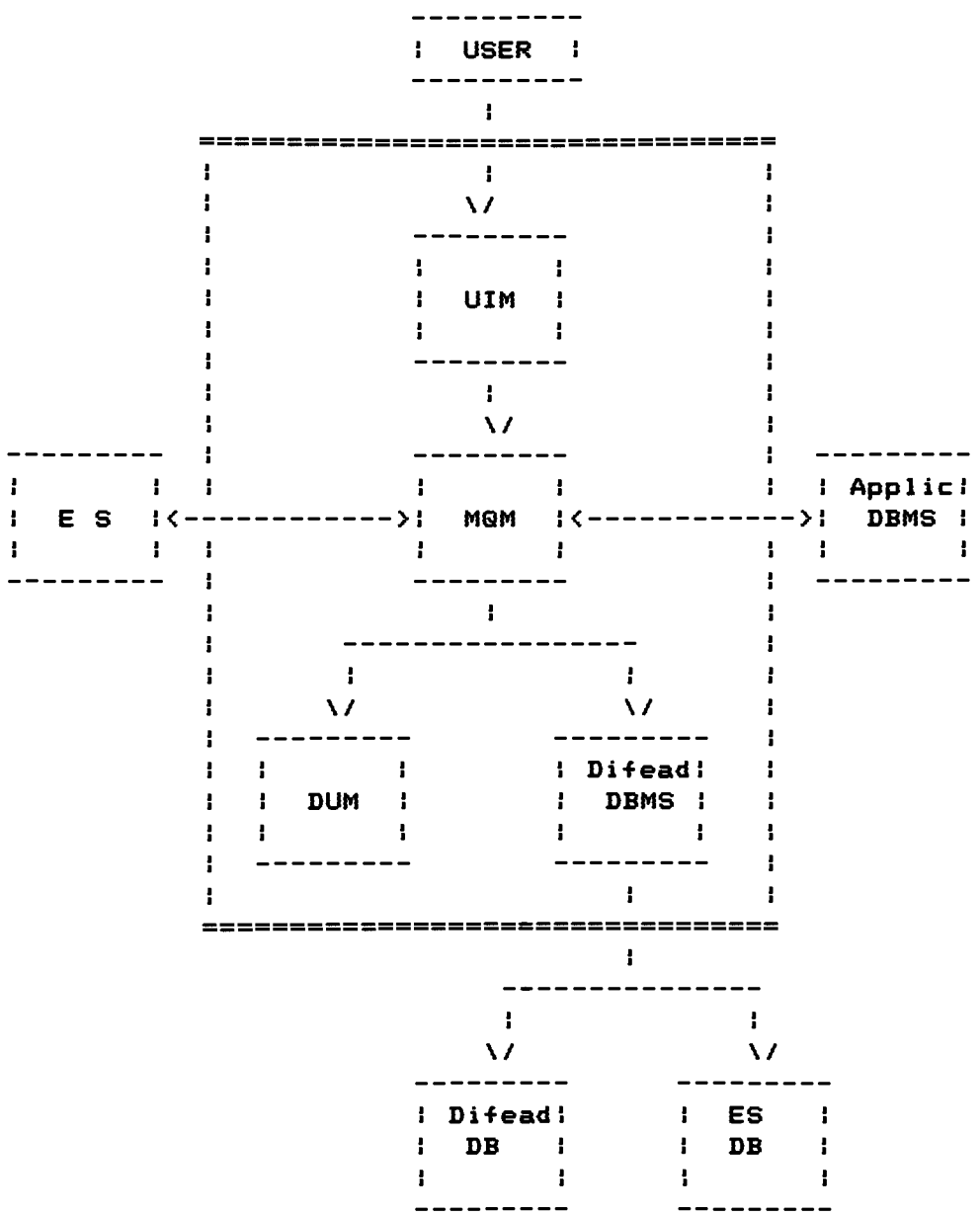
DIFEAD [ZoGr87] is an acronym for "Dictionary Interface for Expert Systems and Databases". DIFEAD is a prototype system that uses an independent subsystem for control of the processing that is distributed to the database and expert system components. The DIFEAD system was implemented on a Digital VAX-11/780 computer. The expert system used in the prototype is a Medical Diagnosis System (MDS), written in Lisp. The database coupled to it is a clinical database under the management of the Troll/Use relational DBMS which runs in the Unix environment.

The database and expert system components of the DIFEAD system are coupled by way of an independent, controlling subsystem called the Metalevel Component (MLC) in a "supervisor" configuration. Both the expert system and database components communicate with the MLC through Unix pipes. The structure of DIFEAD is shown in Figure 3.1.

The MLC depends on two types of data to control the interaction between the database and expert system. Dictionary data give basic information about the database and expert system, such as application domain and size of database and expert system. Control data enable DIFEAD to control the interaction of the database and expert system, such as the availability and location of data in the database for each rule in the expert system.

The MLC is itself implemented as a database. This concept is similar to that of the data dictionary subsystem found in most DBMSs. In addition to the MLC data, DIFEAD has three components:

1. User Interface Module (UIM) - Handles all user interfacing, including input validation and reformat of data for user presentation.
2. Metadata Query Module (MQM) - Deals with all communication between the database and the expert system. This includes database query construction, and translation of database query replies.
3. Data Update Module (DUM) - Responsible for all update operations of the MLC database(s) as well as the application database(s).



DIFEAD

Figure 3.1

The DIFEAD implementation did not require any modifications to either Lisp or the Troll/Use DBMS, since all control operations reside in the DIFEAD modules.

Users access the Medical Diagnosis System/Clinical database through a menu-driven interface that is the UIM component of DIFEAD. DIFEAD was designed to handle multiple expert systems and multiple databases. The user must specify which system is to be accessed. The user also has the option of accessing the database directly.

Inferencing is done by the expert system until additional data are required. At that point, the MQM determines if the required data are available in the database, or must be obtained from the user. All user questions and input are handled by the menu-driven user interface. Input validation is handled by the UIM, leaving the expert system free of this task.

DIFEAD provides data to the expert system only when necessary, thereby, making the expert system more efficient. DIFEAD also increases data integrity by supplying the expert system with data from the database whenever possible rather than from user input. Because of the architecture of DIFEAD, multiple and pre-existing expert systems and databases can be incorporated into a single system.

One advantage of DIFEAD, which is common with database/expert systems such as this, is the data independence via the separate storage of the knowledge and data. This allows for more flexible systems to be created. Specific to DIFEAD, another advantage is in its data-sharing capabilities. The DIFEAD system was designed to allow more that one expert system to communicate with and obtain data from multiple databases. DIFEAD also handles textual data, as stored in the database, and allows the expert system to

refer to such texts by referring to them by database keys. This avoids the loading and storing of texts in the expert system.

The current implementation of DIFEAD couples one expert system with one database. The application, Medical Diagnosis, holds a great number of possibilities for expanding the system. One enhancement cited in the literature is to add a second expert system, Thyroid Function Test, which accesses the same clinical database. Future directions include integration of other application databases that run under control of DBMSs other than Troll/Use, such as dBaseII. The goal of the designers is for DIFEAD to handle a heterogeneous collection of systems in an area such as medicine, where the distributed systems may be linked via network.

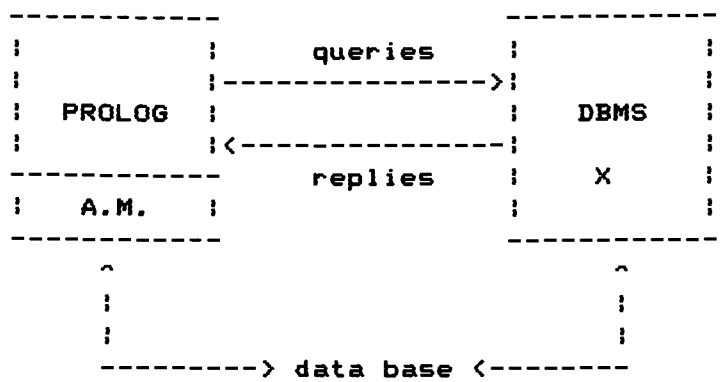
3.1.3 EDUCE

The EDUCE system [Bocc86] interfaces Prolog and the relational DBMS INGRES by a combination of integration and coupling. The reference cited above does not indicate the type of hardware that the system was implemented on; however, EDUCE was implemented in a Unix environment. It is the result of the need for a logic programming system that is capable of handling a very large knowledge base. In a coupled Prolog/relational DBMS system, implementation of recursion is difficult because of the multiple tuples returned by a DBMS query. The EDUCE system uses an integrated implementation to overcome the problems caused by recursion. The integrated architecture allows Prolog to retrieve

tuples one-at-a-time, directly from the database, thereby allowing an easier implementation of recursion. EDUCE supports the storage of both facts and rules in the internal INGRES database.

Coupling is implemented in EDUCE through a construction of a QUEL (the query language for INGRES) query in the Prolog component, which is sent via Unix pipe to INGRES. This query construction is done by the execution of the built-in predicate *query*, whose argument is a string of characters that describes the action to be performed. The predicate *query* was written in C and embedded in the Prolog interpreter. After sending a query to the INGRES component, the Prolog component waits for a reply and an acknowledgement of completion. In the case of a multiple tuple reply from INGRES, Prolog regards the pipe as a queue, and removes one tuple at a time.

Tight integration is implemented in EDUCE, by embedding the INGRES database access method (AM) into the Prolog component. Therefore, this (Prolog + AM) component has direct access to the physical database storage, thereby circumventing the DBMS. The EDUCE structure diagram is given in Figure 3.2.



EDUCE

Figure 3.2

Because of the lack of DBMS control functionality in the Prolog component, the tight integration is restricted to retrieval operations. This restriction was placed on the system to avoid the problem of concurrent updates. In the EDUCE system, the Prolog used was greatly modified to include several built-in predicates such as *query*, and to include the DBMS access method. The INGRES portion of EDUCE was not modified.

The EDUCE system appears to users as a Prolog system extended by the built-in predicates used for database manipulation and queries. One of several database retrieval predicates achieves database transparency via a tight integration and a structure similar to that of predicates that query Prolog's internal database.

Some of the capabilities of EDUCE can be illustrated by describing the built-in database predicates of the Prolog component of EDUCE. There are five groups of predicates:

1. Predicates that allow users to set-up, destroy and connect to databases.
2. Predicates that provide data-dictionary information.
3. Predicates to manipulate the database schema (i.e. create/destroy tables)
4. Predicates to add, delete and update facts and rules
5. Predicates for retrieval, of which there are three forms:

   retrieve - provides loose coupling retrieval

   save - provides loose coupling to retrieve tuples and store them in another table

   retr - provides a tightly integrated access to the database

The example below illustrates how the *retr* predicate may be used to provide database transparency. This example assumes the existence of the physical database table *parent*, with the attributes *parentname* and *childname*.

To evaluate:

?- parent(A,B).

the following rule must exist to interface to the physical database relation:

parent(X,Y) :- retr(parent(X,Y)).

where the argument to the predicate *retr* is a reference to the physical database relation.

As previously stated, in EDUCE, rules as well as facts may be stored in the external (INGRES) database. This is accomplished by the existence of a special relation *rulerel* in the INGRES database. Facts are stored in database tables according to their function name, while all rules reside together in *rulerel*. Rules that are stored in this manner in the database may be retrieved, added, deleted and updated, just as though they are facts. Also, a rule stored in the database may be used just as any other rule can. In a relational database, ordering of tuples in a table is immaterial. However, in Prolog the order of residence of rules is important, as it determines the execution sequence and the results of a Prolog operation. Therefore, an ordering must be imposed on the rule tuples that reside in *rulerel*. This is accomplished by an integer attribute, in addition to the rule body, that specifies the position of the rule in the order of execution or "firing". This appears to be an encumbrance when adding or deleting rules, since the numbering must be maintained, but it is a solution to the ordering problem.

Recursive queries are handled by use of the *retr* predicate in all such definitions. As mentioned, the *retr* predicate uses close integration via the DBMS access method to provide one-tuple-at-a-time responses to queries.

The EDUCE system supports concurrent user access to the database. This capability is due to the access control functionality built into the INGRES DBMS, with the restriction that only retrieval operations may bypass the DBMS.

3.2 Microcomputer systems

Several microcomputer software packages are commercially available for MS-DOS personal computer systems that facilitate the coupling of Prolog and a database. However, the majority of such systems are implemented as very loose coupling. The following section describes an experimental system as well as a system that is currently available for microcomputers.

3.2.1 Berghel's system

The system described in [Berg85] is an experimental implementation of a loose coupling between Prolog and a relational DBMS. It was implemented on an MS-DOS microcomputer and interfaced two commercially available software packages, micro-Prolog (version 3.0 - Logic Programming Associates) and dBaseII (version 2.4 - Ashton-Tate). Micro-Prolog was chosen because it was the only Prolog available for microcomputers at the time the project was started in 1982. dBaseII was selected mainly because of its popularity on microcomputer systems. dBaseII is considered to be "minimally relational" by Codd's definition since it does not support the entire relational algebra or integrity

constraints [Codd70]. Although Berghel's system lacks the elegance and efficiency of a tightly coupled or integrated system, it may be useful in many personal computer applications. Its intended use is as a tool for the research environment. The system takes advantage of relational DBMS facilities, such as file management, I/O management and arithmetic, where Prolog falls short. However, it also takes advantage of Prolog's capabilities for more sophisticated querying of databases.

Berghel's system is a loosely coupled system. The coupling of the components is accomplished by a file interface program that converts dBaseII files to micro-Prolog files and vice-versa. Only facts may be exchanged between the two systems; therefore, all rules must be stripped from the micro-Prolog database before a conversion to dBaseII files is done.

The file interface program requires format information about the dBaseII files and micro-Prolog files. Micro-Prolog files are divided into two sections. The first section contains all the Prolog "sentences" or records in the form *(P a1 a2 ... an)* where *P* is an n-ary predicate and *a1...an* are arguments. The second section contains dictionary information in the format *(DICT P)*, where there is an entry for each existing predicate in the system. dBaseII files are also composed of two sections. The first section contains the file description, and the second section stores the actual data.

When converting from dBaseII files to micro-Prolog files, the following three steps are taken. An analogous procedure is used for the conversion of micro-Prolog files to dBaseII files.

1. A command file is written in dBaseII, specifying the data to be extracted (relations and attributes).
2. Execution of the command file produces two output files for each relation extracted from dBaseII:
   SDF file, which contains the data and
   DBF file, which contains structural information.
3. The file conversion program in executed with the *SDF* and *DBF* files as input. Micro-Prolog sentences are created from the dBaseII relations, along with the appropriate *(DICT P)* entry for each relation.

This implementation requires no modifications to either micro-Prolog or dBaseII. The file conversion program was written in Pascal.

For the construction of the command file, the user must specify the portion of the database to be converted. The user may partition the database to create arbitrary new predicates. A situation where this would be useful is if the user perceives a relationship between attribute *i* of *R1* and attribute *j* of *R2*. Berghel's system allows the user to describe this relationship and to construct Prolog sentences that reflect it. The user also has the capability of specifying which tuples of a relation are to be converted, in the case where the entire relation is not desired. The end result is a micro-Prolog database tailored to the user's needs.

Once the data are transferred from dBaseII to micro-Prolog,

the full extent of micro-Prolog's programming capabilities can be used to query and perform operations on the data. These capabilities are assumed to be much stronger than those available in dBaseII. Afterwards, if desired, the micro-Prolog database may be converted back to dBaseII files, using a procedure similar to the one previously described.

3.2.2 Arity/SQL

Arity Corporation's Arity/SQL [Rett87b] provides a database interface environment for Arity Prolog. It is sold as an "add-on" package to the Prolog interpreter and compiler. This Prolog/SQL system is commercially available for PC-DOS microcomputers. The Arity/SQL package was made available by the Arity Corporation in 1986.

Arity/SQL provides a loose coupling between Arity/Prolog and the Arity/SQL database. This coupling is implemented by extending Prolog to include a set of predicates that allow the user to include SQL statements in Prolog programs. The Arity/SQL package was developed for the specific purpose of coupling with Prolog; however, Arity/SQL statements may be input and executed by the user standalone, in what is called "example mode."

"Example mode" allows a user to access Arity/SQL databases directly, which is a feature that is useful for testing queries before they are used in Prolog programs. However, the more significant method of accessing the SQL database is through the Prolog interface. From a Prolog program, SQL statements can be

executed by using the built-in predicate *exec_sql*. The argument to the *exec_sql* predicate is a string representing an SQL statement. The results of the SQL statement or query issued in this manner is displayed at the terminal, with all database error handling done by Arity/SQL. If the user, or application writer wishes to have query results deposited into the internal Prolog database, the *exec_sql* predicate can be given a second argument, which is the name of a key to store the data under in the Prolog database. This loose coupling allows the user to take advantage of the complete functionality of Prolog with the data.

SQL statements can be included in Prolog programs in three ways:

1. They may be "hard-coded" in the Prolog program.
2. An application program may accept input from a user to "build-up" an SQL statement "on the fly", using a template.
3. They may be read directly from a disk file.

Arity/SQL syntax is similar to that of dBaseII and standard SQL and supports the same basic operations. Arity/SQL allows for the placing of type constraints on attributes in relations. As Prolog is a "typeless" language, wise use of the data-typing facilities of Arity/SQL can enforce typing in Arity/Prolog applications. Arity/SQL supports views on base tables, multiple indexes on a base table to increase performance, and *ROLLBACK/COMMIT* for data integrity and recovery. It provides

functions such as *MIN*, *MAX*, *COUNT*, *AVERAGE* and *SUM*. The database catalog is stored as SQL tables, and may be queried using SQL commands [Rett87b].

Chapter 4

Design

4.1 Overview

The Prolog/relational database interface that was constucted, dbProlog, consists of two parts:

1. Extensions to Prolog, implemented as Prolog "procedures," to accomplish the communication between the Prolog subsystem and the interface subsystem.
2. A "Database Driver" program to translate from the predicates of the Prolog extension to the query language of the DBMS, translate replies from the DBMS to a format usable by Prolog, and control communications between the DBMS subsystem and the Prolog subsystem. The driver program, which is specific to the DBMS being used, provides the mapping between Prolog and the DBMS, it could be replaced by a similar one to drive another DBMS.

A diagram of the system architecture is given in Figure 4.1.

The primary design goal of this implementation was that both loose and tight coupling be supported by dbProlog. Loose coupling is supported by the capability of converting database tuples into

DBPROLOG SYSTEM

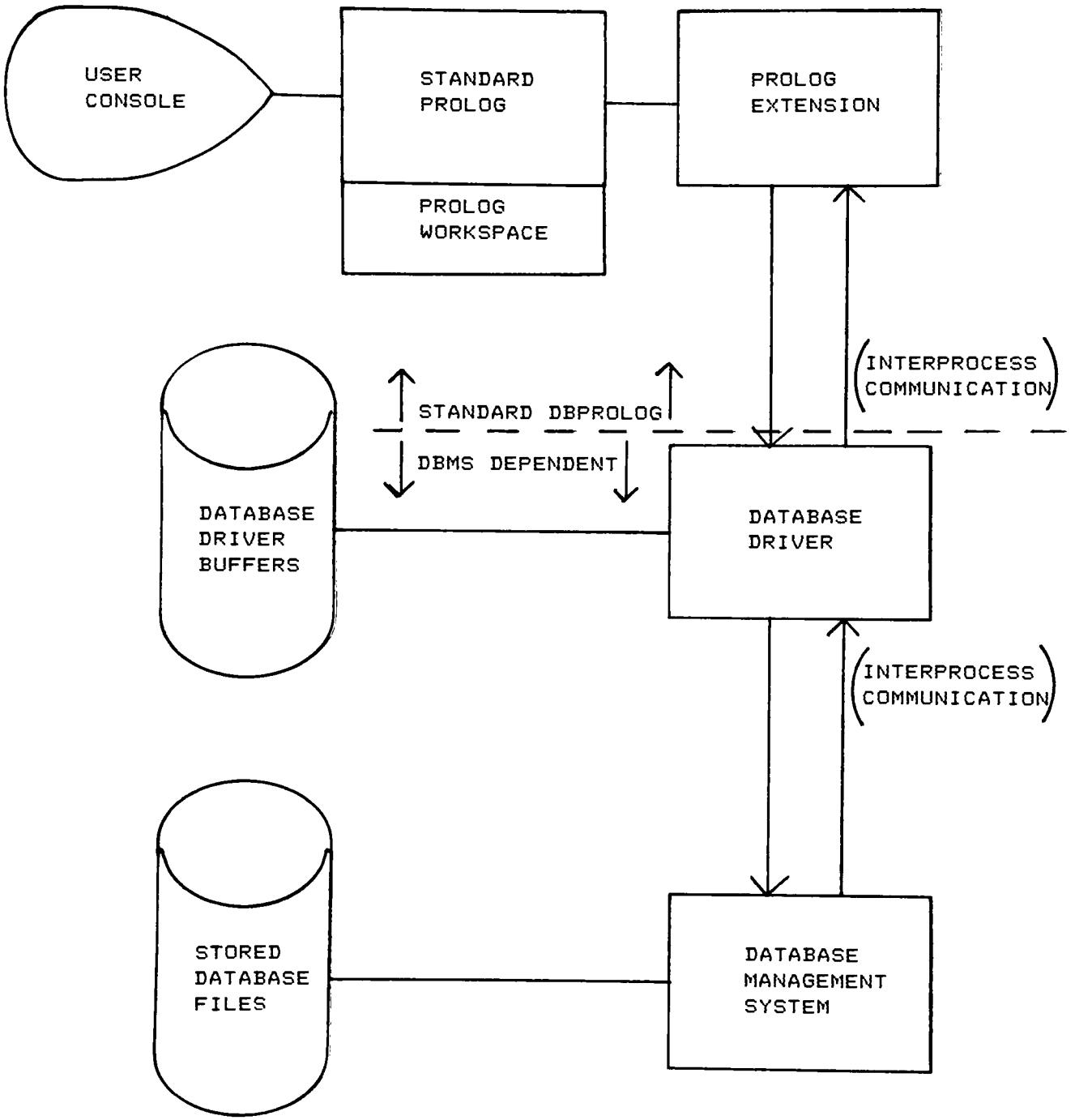


Figure 4.1

Prolog facts and loading them into the workspace of Prolog. Tight coupling is supported by the capability of retrieval and unification of database tuples with Prolog statements, causing possible instantiation of Prolog variables. Another goal was that the interface should allow manipulation of the external database (EDB), so that tuples may be asserted and retracted, just as facts are asserted and retracted from Prolog's internal database (IDB). A related goal was to provide transparent retrieval from the EDB. Tuples should be retrieved from the EDB in a manner that appears the same as fact retrieval from the IDB. The most important goal was to leave the DBMS and Prolog interpreter intact (as much as possible). That is, the above goals were met without changes to either the DBMS code or the Prolog interpreter.

4.2 Software/hardware

The dbProlog system coupled C-Prolog to a relational DBMS that was implemented as the project for the Database Implementation course, ICSS-739. This relational DBMS was written in C and may be described as "minimally relational" since it supports the basic operations of *SELECT*, *PROJECT* and *JOIN*. Other features of this DBMS include creation and deletion of base tables, inserting of tuples, indexing, and selective deletion of tuples. The DBMS does not implement the concept of views.

C-Prolog was chosen because of its availability, familiarity and, since it is written in C, to provide easier interfacing and process communications via the Unix operating system.

A reason for the DBMS choice was the availability of the source code, in case any modifications were needed. The services provided by the DBMS that were used by the interface are standard operations that should be provided by any relational database. Therefore, the choice of which relational DBMS for functionality is arbitrary.

The driver program was written in C to facilitate Unix process communication system calls and to be compatible with C-Prolog and the DBMS with which it will be communicating.

The dbProlog system was implemented on the AT&T Unix[1] PC for the following reasons:

* Portability to other Unix systems
* Availability of C-Prolog and C programming language (in which the DBMS is written)
* Availability of Unix process communication facilities

4.3 User Interface

Before describing the user interface of the dbProlog system, the term "user" must be defined. In the context of the dbProlog system, the "user" is the Prolog programmer. This is the person who has direct access to the dbProlog interface commands (the extension to Prolog) and the awareness of the existence of the interface and the EDB. The Prolog programmer may write application programs to be used by other users. These "secondary"

[1]Unix is a trademark of AT&T

users need not know of the interface or of the EDB. Therefore, when the term "user" is mentioned, it is in reference to the "primary" user, the Prolog programmer.

The dbProlog user interface consists of a set of Prolog predicates that communicate with the interface, which in turn communicates with the DBMS. The predicates are described below.

* * * * * * * * * *

dbretrieve(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate returns one tuple to Prolog from the external base table *Relname* that satisfies the constraints placed on the attributes that are instantiated (*Attr$_n$*, etc.). If successful, the variables in the group *Attr$_1$,...,Attr$_n$* are appropriately instantiated, and *Status* is set to a positive value or zero indicating the total number of database tuples that satisfy the request. If the retrieval fails, *Status* is returned as "-1", and *Message* is instantiated to an appropriate error message. The first call to *dbretrieve* will place all retrieved tuples into a buffer and return the first tuple to Prolog for processing. Backtracking into a subsequent *dbretrieve* call will retrieve the next tuple from the buffer. This predicate may be used to provide tight coupling, transparent retrievals from the database. The following is an example of how this may be done. The clause following the *dbretrieve* call causes an error message to be displayed if an error occurs.

```
parent(X,Y) :-
     dbretrieve(parent(X,Y),Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

* * * * * * * * * *

dbload(Relname(Attr$_1$,...,Attr$_n$),Idbname,Status,Message).

This predicate returns all tuples to Prolog from the external base table *Relname* that satisfy the constraints placed on the attributes that are instantiated. This group of tuples are then asserted into the internal database of Prolog with the functor name *Idbname*. If successful, *Status* will return a positive value or zero indicating the number of tuples retrieved and asserted. If the operation fails, *Status* is returned as "-1", and *Message* will be instantiated to an appropriate error messsage. An example of usage follows:

```
loadparents :-
     dbload(parent(X,Y),parent_idb,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

* * * * * * * * * *

dbassert(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate is used to insert one tuple into the external base table *Relname*. All *Attr$_n$* variables must be instantiated before this predicate may be executed. Successful completion is indicated by instantiation of the *Status* variable to "0". Failure is indicated by a *Status* value of "-1" and the instantiation of *Message* to an appropriate error message. An example of usage follows.

```
insert(parent(X,Y)) :-
        dbassert(parent(X,Y),Stat,Msg),
        ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

* * * * * * * * * *

dbretract(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate is used to delete a group of tuples from the external base table *Relname* that satisfy the constraints placed on the attributes that are instantiated. All *Attr$_n$'s* need not be instantiated. Success is indicated by a *Status* value of "0". Failure is indicated by a *Status* value of "-1" and the instantiation of *Message* to an appropriate error message. An example of usage follows.

```
delete(parent(X,Y)) :-
        dbretract(parent(X,Y),Stat,Msg),
        ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

4.4 Detailed design

The purpose of this section is to present the design of the Prolog/relational DBMS interfacing system that was implemented (dbProlog). Leveled data flow diagrams are included to provide a graphical description of the interaction of components. A data dictionary is also included to define the data flows, and mini-specifications to define some of the lower level modules.

The complete system consists of three subsystems: Prolog, the dbDriver program, and the DBMS. Figure 4.2 gives the context diagram for the subsystems and the data flows between them. Each subsystem is designed to run as a separate process under Unix, communicating via Unix FIFO's. Only the portions of the subsystems that are part of the interface will be described in detail in this section.

The Prolog subsystem consists of two parts, standard Prolog and an extension to standard Prolog. Standard Prolog refers to the two major conponents of the Prolog interpreter: the inference engine and the internal database manager with all associated built-in predicates. Since one objective of the design of the system was to not make changes to standard Prolog, the low level design of this component will not be described. The extension consists of the Prolog code to conduct DBMS interaction, i.e.

making requests to the DBMS and processing DBMS responses, via the interface. All user interaction with dbProlog is done through the Prolog extensions of the Prolog component. The extensions are to be implemented using standard Prolog statements and procedures, rather than modifying the Prolog interpreter. The design of the Prolog subsystem is depicted in Figures 4.3, 4.4, and 4.5.

The dbDriver subsystem comprises the major portion of the interface. Its functions include translation and communication between the DBMS and Prolog. It may also provide buffering of DBMS responses. The dbDriver was written in C and has a clearly modular structure to provide easy interchange of translation modules so that interfaces to various DBMSs can be easily and quickly constructed, using the same overall design and communication routines. When interfacing to other DBMSs, the Prolog/dbDriver interface should remain the same. The design of the dbDriver subsystem is given in Figures 4.6, 4.7, and 4.8.

A detailed design of the DBMS subsystem is not included in this document, since the purpose of the system is to allow the use of a non-specific DBMS. The DBMS subsystem may be regarded as a server system that accepts requests from a user or another process and satisfies requests by sending a response back to the requestor. The functions of the DBMS required by the dbProlog system should be those commonly available in any relational DBMS, such as *insert*, *delete*, and *retrieve*. Some things, such as views, are common but not implemented in the DBMS used here. The high-level description of the DBMS is given in Figure 4.9.

4.4.1 The Prolog extension

The data flow diagrams for the Prolog extension are given in Figures 4.3, 4.4, and 4.5. Following the diagrams are the specifications of the low level modules. Figure 4.3 represents the entire Prolog component while Figures 4.4 and 4.5 depict the subcomponents that send requests to the interface and receive responses from the interface, respectively.

1 PROLOG SUBSYSTEM
GOAL
RESOLUTION
1-1
INFERENCE ENGINE
TUPLE-FACT
STATUS MESSAGE
TUPLE-DESC
1-3
SEND REQUEST TO I/F
REL-INFO
SUBGOAL
FACT
1-2
PROLOG IDB MANAGER
TUPLE-FACT
1-4
RECEIVE RESPONSE FROM I/F
PROLOG IDB
INT-REQ-PKT
INT-RESP-PKT
I/F REQUEST FIFO
I/F RESPONSE FIFO

FIG. 4.3

1-3 SEND REQUEST TO I/F

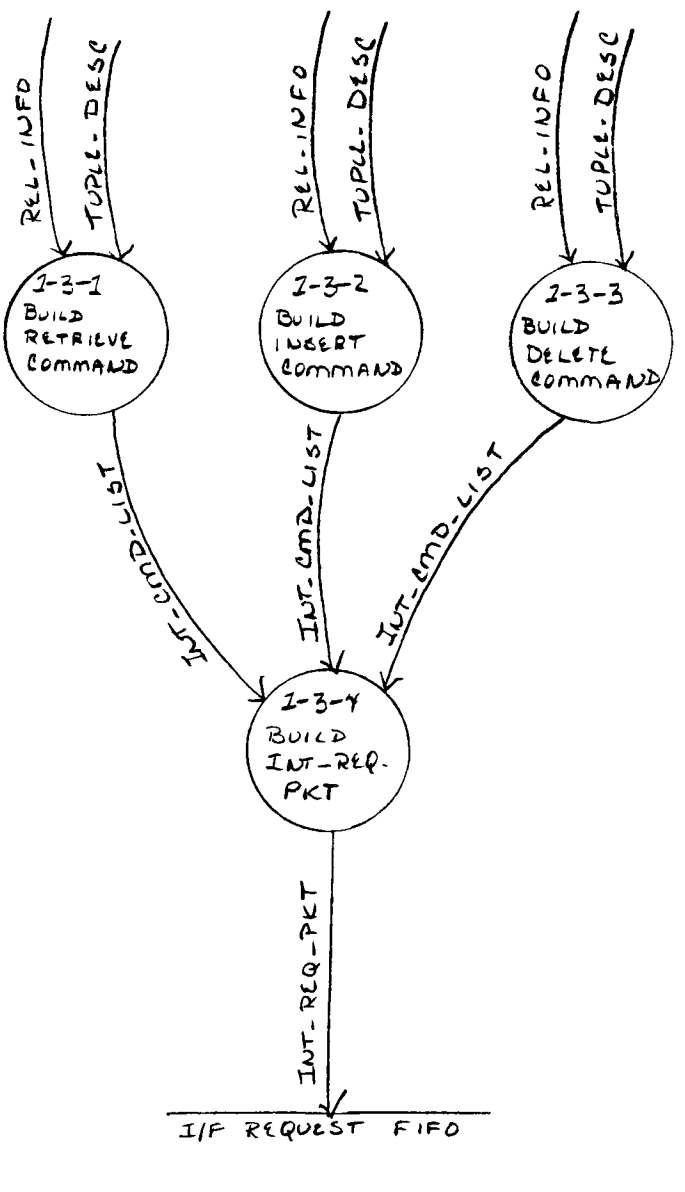


FIG. 4.4

1-4 RECEIVE RESPONSE FROM I/F

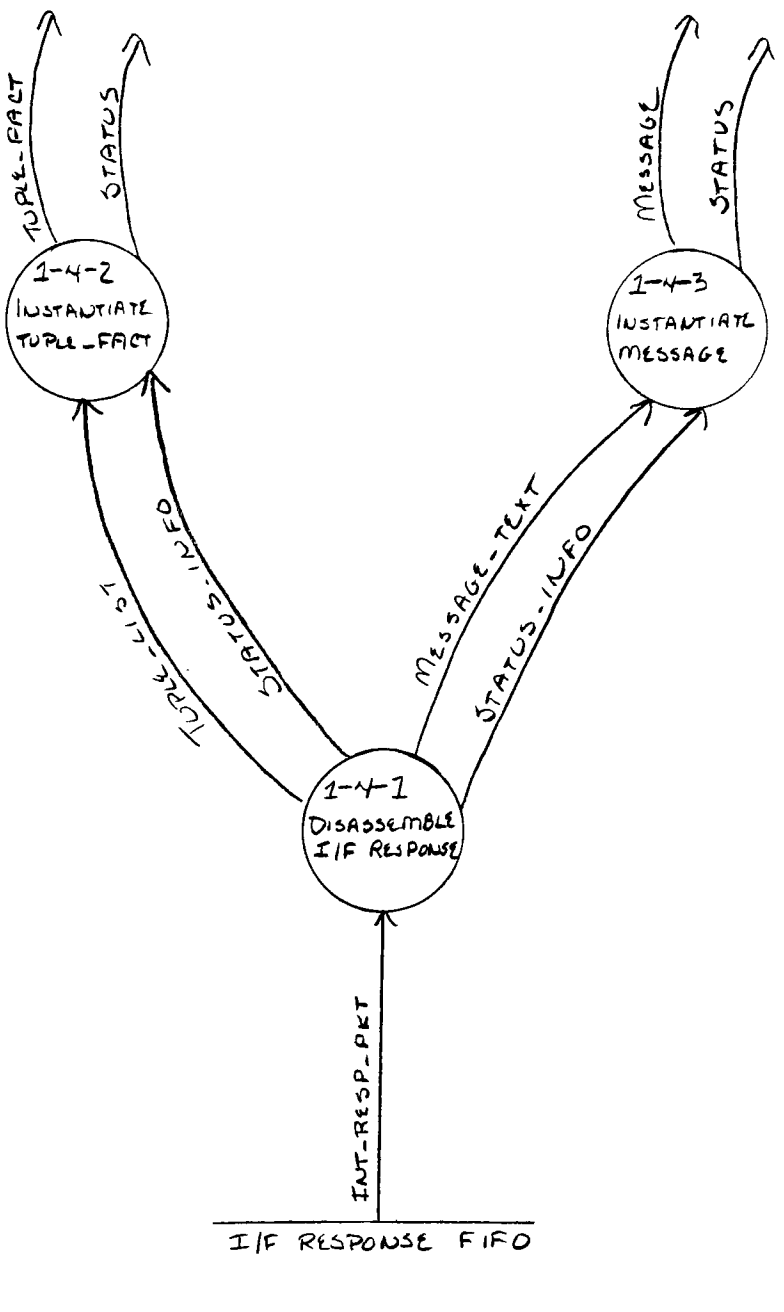


FIG. 4.5

Specifications of Prolog extension modules

1-3-1. Build RETRIEVE Command

Input: *Tuple_desc*, *Rel_info*

Output: *Int_cmd_list*

- Strip *Relname* from *Tuple_desc*
- Retrieve *Rel_info* fact with matching *Relname*. If none exist, return error and fail "no such base table available".
- For each instantiated *Attr_val* of *Tuple_desc*, create one *Attr_eq_stmt* and construct *Cond_list*
- Instantiate *Int_cmd_word* to "RETRIEVE"
- Construct *Int_cmd_list*

1-3-2 Build INSERT Command

Input: *Tuple_desc*, *Rel_info*

Output: *Int_cmd_list*

- Strip *Relname* from *Tuple_desc*
- Retrieve *Rel_info* fact with matching *Relname*. If none exist, return error and fail "no such base table available".
- Check that each *Attr_val* of *Tuple_desc* is instantiated, if not, return error and fail.
- Instantiate *Int_cmd_word* to "INSERT"
- Construct *Int_cmd_list*

1-3-3 Build DELETE Command

Input: *Tuple_desc*, *Rel_info*

Output: *Int_cmd_list*

- Strip *Relname* from *Tuple_desc*

- Retrieve *Rel_info* fact with matching *Relname*, if none exist, return error and fail "no such base table available".

- For each instantiated *Attr_val* of *Tuple_desc*, create one *Attr_eq_stmt* and construct *Cond_list*.

- Instantiate *Int_cmd_word* to "DELETE"

- Construct *Int_cmd_list*

1-3-4 Build *Int_req_pkt*

Input: *Int_cmd_list*

Output: *Int_req_pkt*

- Create *Req_pkt_hdr* with info such as length of message, etc.

- Combine *Req_pkt_hdr* with *Int_cmd_list* to make *Int_req_pkt*

- Put *Int_req_pkt* onto I/F Request FIFO.

1-4-1 Disassemble I/F Response

Input: *Int_resp_pkt*

Output: *Status_info*, (*Tuple_list* or *Message_text*)

- Get *Int_resp_pkt* from FIFO

- Read *Resp_type* of *Int_resp_pkt* to determine type of message (Tuple or Message)

- If Tuple, build *Tuple_list* from *Resp_text* and pass it and *Status_info* to tuple receiver

- If Message, build *Message_text* from *Resp_text* and pass it and *Status_info* to Message handler

1-4-2 Instantiate *Tuple_fact*

Input: *Tuple_list*, *Status_info*

Output: *Tuple_fact*, *Status*

- Use *Tuple_list* to instantiate variables of *Tuple_desc* in interface-calling command to create a Prolog *Tuple_fact*

- Convert *Status_info* to *Status*

- Pass *Tuple_fact* and *Status* to caller (*dbretrieve* or *dbload*)

1-4-3 Instantiate *Message*

Input: *Message_text*, *Status_info*

Output: *Message*, *Status*

- Use *Message_text* to instantiate *Message* of interface-calling command

- Convert *Status_info* to *Status*

- Pass *Message* and *Status* to caller (*dbretrieve*, *dbassert*, *dbretract*, *dbload*)

4.4.2 The dbDriver program

The data flow diagrams for the dbDriver program are given in Figures 4.6, 4.7, and 4.8. Following the diagrams are the specifications of the low level modules. Figure 4.6 represents the complete dbDriver component while Figures 4.7 and 4.8 depict the subcomponents that translate commands and responses between Prolog and the DBMS.

2-1 COMMAND TRANSLATOR

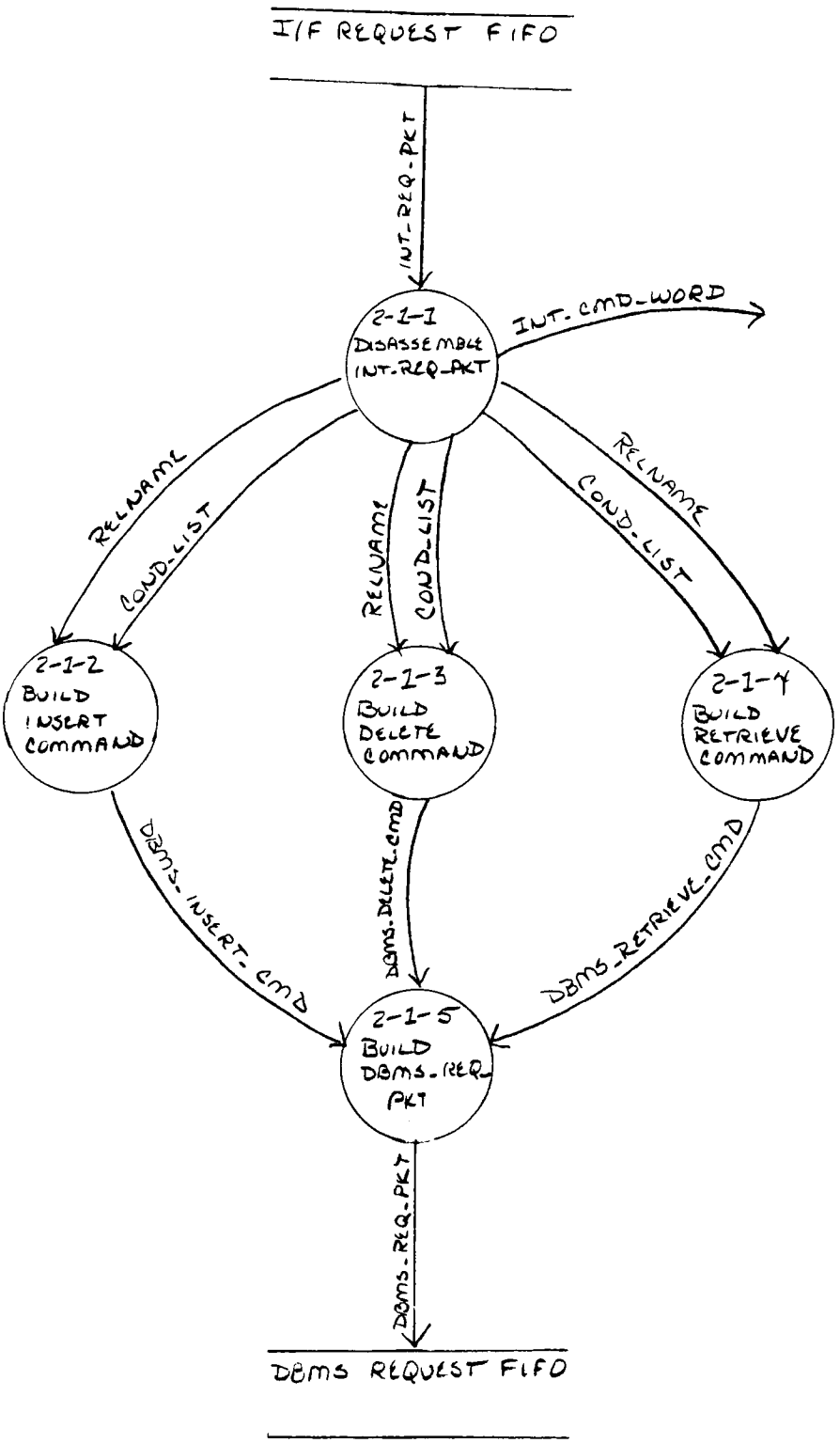


FIG. 4.7

2-2 RESPONSE TRANSLATOR

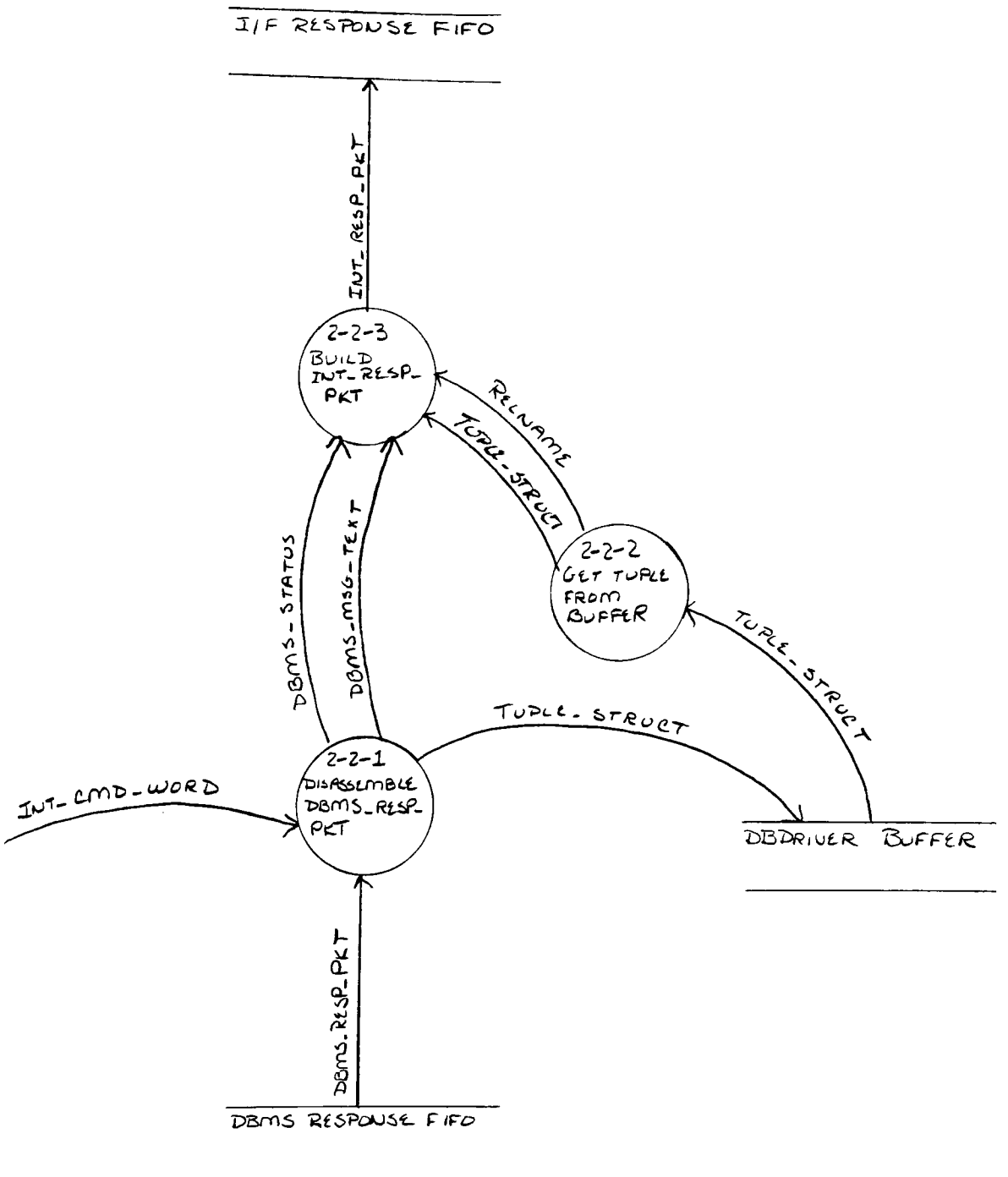


FIG. 4.8

Specifications of dbDriver modules

2-1-1 Disassemble *Int_req_pkt*

Input: *Int_req_pkt*

Output: *Relname*, *Cond_list*, *Int_cmd_word*

- Get *Int_req_pkt* from I/F request FIFO
- Strip *Int_cmd_word* to determine appropriate DBMS command builder
- Send *Int_cmd_word* to DBMS response translator module
- Strip *Relname* and *Cond_list* and send to appropriate DBMS command builder

2-1-2 Build INSERT command (DBMS specific routine)

Input: *Relname*, *Cond_list*

Output: *DBMS_insert_cmd*

- Using *Relname* and *cond_list*, build syntactically correct text string representing DBMS INSERT command

2-1-3 Build DELETE command (DBMS specific routine)

Input: *Relname*, *Cond_list*

Output: *DBMS_delete_cmd*

- Using *Relname* and *Cond_list*, build syntactically correct text string representing DBMS DELETE command

2-1-4 Build RETRIEVE command (DBMS specific routine)

Input: *Relname, Cond_list*

Output: *DBMS_retrieve_cmd*

- Using *Relname* and *cond_list*, build syntactically correct text string representing DBMS RETRIEVE command

2-1-5 Build *DBMS_req_pkt*

Input: *DBMS_cmd*

Output: *DBMS_req_pkt*

- Construct *Req_pkt_hdr* and combine with *DBMS_cmd* text
- Put resulting *DBMS_req_pkt* on DBMS request FIFO

2-2-1 Disassemble *DBMS_resp_pkt* (DBMS specific routine)

Input: *Int_cmd_word*, *DBMS_resp_pkt*

Output: *DBMS_msg_text*, *DBMS_status*, *Tuple_struct*

- Get *DBMS_resp_pkt* from DBMS response FIFO
- Determine what type of DBMS response (tuple, error message, or completion message), using *Int_cmd_word* to anticipate type of response
- If response represents a tuple reformat it into *Tuple_struct* and place in tuple buffer
- If response represents completion message, send appropriate DBMS_status to *Int_resp_pkt* builder
- If response represents error message, send *DBMS_msg_text* and appropriate *DBMS_status* to *Int_resp_pkt* builder

2-2-2 Get tuple from buffer

Input: *Tuple_struct*

Output: *Relname*, *Tuple_struct*

- Retrieve *Tuple_struct* from tuple buffer
- Strip *Relname* from *Tuple_struct* and send to *Int_resp_pkt* builder

2-2-3 Build *Int_resp_pkt*

Input: (*Relname*, *Tuple_struct*) or

DBMS_msg_text and/or *DBMS_status*

Output: *Int_resp_pkt*

- Build *Resp_pkt_hdr* and combine with rest of response
- Put *Int_resp_pkt* on I/F response FIFO

3 DBMS

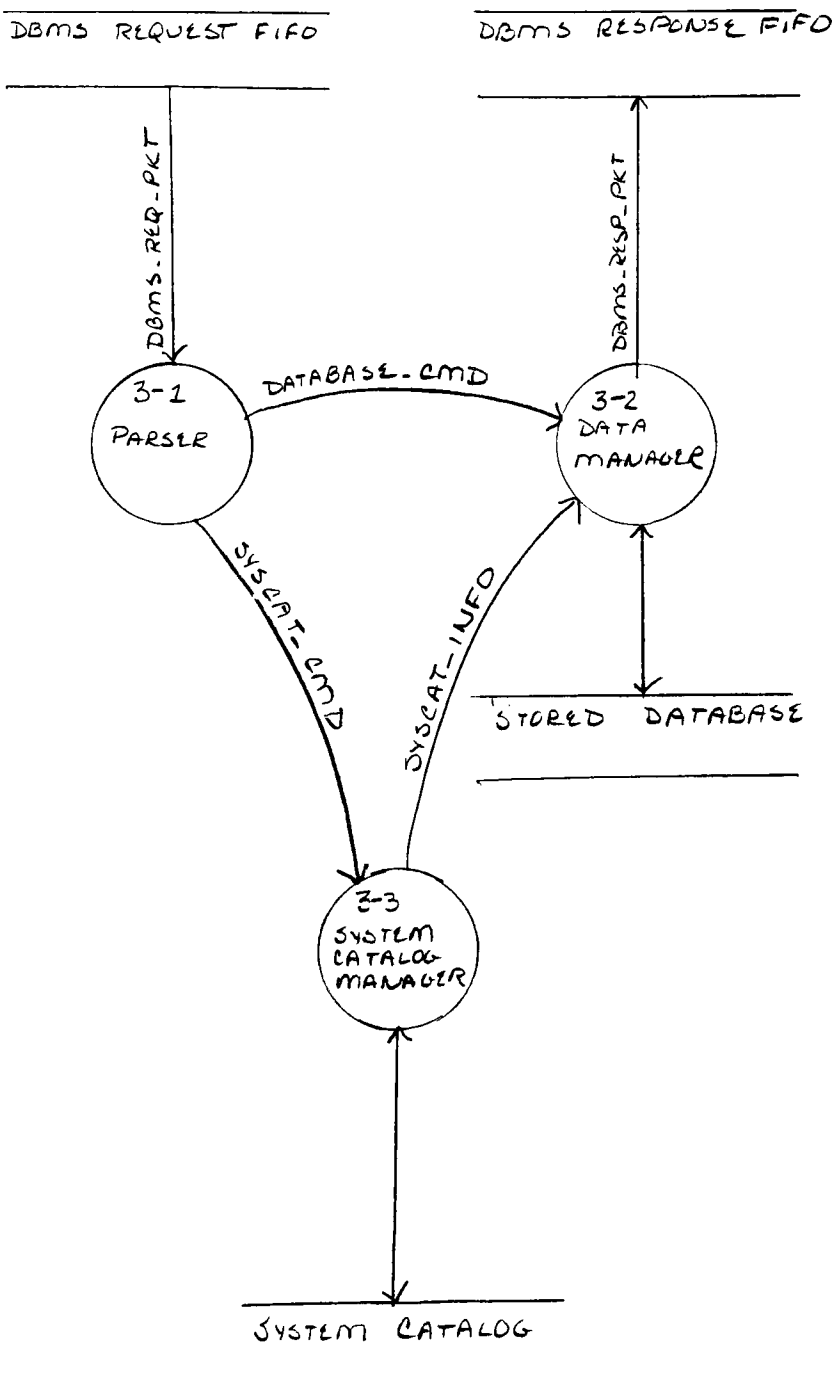

DBMS REQUEST FIFO
DBMS RESPONSE FIFO
DBMS_REQ_PKT
DBMS_RESP_PKT
3-1
PARSER
DATABASE_CMD
3-2
DATA
MANAGER
SYSCAT_CMD
SYSCAT_INFO
STORED DATABASE
3-3
SYSTEM
CATALOG
MANAGER
SYSTEM CATALOG

FIG. 4.9

4.4.3 Data dictionary

The following data dictionary entries define the data elements that are used in the dbProlog design description. Each entry includes an indication of the component that the data element is used in, its data type and a description of the data item.

Attr_eq_stmt := (Prolog, dbDriver) Character string,

Represents an equality statement for an attribute variable,

Ex: *'Attr_name = Attr_val'*

Attr_name := (Prolog, dbDriver) Character string,

Represents the name of an attribute of a relation

Attr_val := (Prolog, dbDriver) Atom,

Represents the value of a relation attribute,

may be instantiated or uninstantiated

Cond_list := (Prolog, dbDriver) List of character strings,

Represents list of *Attr_eq_stmt*'s to provide all conditions

of a retrieval,

$[Attr\_eq\_stmt_1|...|Attr\_eq\_stmt_n]$

DBMS_cmd := (dbDriver) Character string,

DBMS_insert_cmd or *DBMS_delete_cmd* or *DBMS_retrieve_cmd*

DBMS_delete_cmd := (dbDriver) Character string,

Represents text of DBMS command to be executed for tuple delete

DBMS_insert_cmd := (dbDriver) Character string,

Represents text of DBMS command to be executed for tuple insert

DBMS_msg_text := (dbDriver) Character string,

Represents tuple or message text from DBMS

DBMS_req_pkt := (drDriver) Structure,

Packet that contains request from the dbDriver to the DBMS

<Req_pkt_hdr,DBMS_cmd>

DBMS_resp_pkt := (dbDriver) Structure,

Packet that contains response to the dbDriver from the DBMS

<Resp_pkt_hdr,DBMS_msg_text>

DBMS_retrieve_cmd := (dbDriver) Character string,

Represents text of DBMS command to be executed for tuple retrieve

DBMS_status := (dbDriver) Integer,

Represents status of DBMS response to request

Int_cmd_list := (Prolog, dbDriver) Structure,

Represents all necessary information to build DBMS command,

<Int_cmd_word,Relname,Cond_list>

Int_cmd_word := (Prolog, dbDriver) Character string,

Represents keyword of DBMS command,

<'RETRIEVE'> or <'INSERT'> or <'DELETE'>

Int_req_pkt := (Prolog, dbDriver) Structure,

Represents request packet sent to the dbDriver program,

<Req_pkt_hdr,Int_cmd_list>

Int_resp_pkt := (Prolog, dbDriver) Structure,

Represents response packet received from the dbDriver program,

<Resp_pkt_hdr,Resp_type,Resp_body>

Message := (Prolog) Character string,

Represents message text returned to interface-calling predicate

Message_text := (Prolog) Character string,

Represents text of message that is returned to Prolog by the dbDriver

Rel_info := (Prolog) Structure,

Represents relation name and all attribute names of a relation,

$\langle Relname, [Attr\_name_1 | ... | Attr\_name_n] \rangle$

Relname := (Prolog, dbDriver) Character string,

Represents name of a relation

Req_pkt_hdr := (Prolog, dbDriver) Structure,

Header information of Prolog/dbDriver communication packets

Resp_body := (Prolog, dbDriver) Character string,

DBMS response data, representing Prolog fact or a message

Resp_pkt_hdr := (Prolog, dbDriver) Structure,

Header information of dbDriver/Prolog communication packets

Resp_type := (Prolog, dbDriver) Character,

Indicates type of interface/DBMS response,

<Tuple> or <Status> or <Message>

Status := (Prolog) Integer,

Value of status returned to interface-calling predicate

Status_info := (Prolog) Integer,

Represents status of DBMS processing results,

Tuple_attr := (dbDriver) Character String,

Represents value of one of the attributes of one tuple from the database

Tuple_desc := (Prolog) Structure,

Represents a fact or tuple in Prolog database,

Attr_val's need not all be instantiated,

$\langle Relname(Attr\_val_1,...,Attr\_val_n)\rangle$

Tuple_fact := (Prolog) Structure,

Represents a fact or tuple in Prolog database,

Attr_val's must be instantiated,

$\langle Relname(Attr\_val_1,...,Attr\_val_n)\rangle$

Tuple_list := (Prolog) List,

List of Prolog atoms that represent an EDB tuple,

$[Relname|Attr\_val_1|...|Attr\_val_n]$

Tuple_struct := (dbDriver) Structure,

List of attribute *Tuple_attr*'s that make up a tuple of a base table of the DBMS,

$[Tuple\_attr_1|...|Tuple\_attr_n]$

Chapter 5

Implementation

The implementation of dbProlog differed little from the original design. In fact, the only major difference was that interprocess communication was implemented with Unix messages rather than FIFOs. The implementation of dbProlog will be outlined in this section along with a discussion of deviations from the original design.

Section 5.1 describes the approach taken in the implementation of this system while Section 5.2 describes the major components of the resulting system structure. Section 5.3 explains the interprocess communication schemes used and gives a brief overview of the Unix message facility. The execution sequence of the interface is outlined in Section 5.4, and tuple buffering is also discussed. Section 5.5 describes how error handling was incorporated into the system.

5.1 Implementation Order and Approach

A "fast prototype" approach was taken in implementing the Prolog/DBMS interface to build the most functionality into the system in the least amount of time. Functionality and ease of implementation were given priority over system speed and efficiency. The C-Prolog interpreter was not modified at all, at

the expense of having to create suboptimal methods of communication between the Prolog and dbDriver components. However, by avoiding modification of the C-Prolog interpreter, system portability was left intact. The Unix System V message facility was used for interprocess communication, and file I/O was also used for communication to and from the Prolog process. C-Prolog and the DBMS still may be run stand-alone with no difference in their individual functionalities. The only modifications made to the original DBMS were extensions that made it closer to commercial DBMS products and modifications to support communication with the dbDriver process.

The initial implementation of dbProlog supported only one active DBMS call at any time. Therefore, any single Prolog statement could specify only one reference to a database table. Clearly, in this implementation, joins of EDB tables and recursion were not supported. This implementation did support all of the various DBMS operations that were outlined in Chapter 4 (*dbretrieve*, *dbload*, *dbassert*, and *dbretract*). Because this implementation was restricted to one DBMS call at a time, external tuple buffering was not required, so internal message queues were used to buffer retrieved tuples until their transfer to Prolog.

The initial implementation was extended to support the capability for multiple DBMS calls from Prolog to be active at any time. This provides dbProlog with the functionality needed to execute joins and recursive calls on DBMS tables. To do this, buffering of retrieved tuples is required to keep message queues free for subsequent calls while retaining the tuples for

backtracking. Buffer use is controlled by Prolog's internal activation stack.

For convenience, two dbProlog predicates were added to the system. *dbcreate* allows the generation of a new table in the DBMS without the user having to exit Prolog or run the DBMS standalone. *dbdrop* allows the user to delete a DBMS table and corresponding definitions from the Prolog and DBMS components. Their formats are described below:

* * * * * * * * * *

dbcreate(Relname,Attrlist,Stat,Msg).

This predicate creates the EDB relation *Relname* from the information given in *Attrlist*. The format of *Attrlist* is

$$[attr(Name_1,Type_1)/.../attr(Name_n,Type_n)]$$

where *Name* specifies the name of the attribute and *Type* specifies the type of attribute. In this implementation only two values for *Type* are supported, "int" and "char". If the request is successful, and the corresponding table is created by the DBMS, a value of "0" is returned for *Stat* and *Msg* is not instantiated. Otherwise, a value of "-1" is returned for *Stat* and the appropriate DBMS error message is returned in *Msg*. In addition, the appropriate data dictionary information is asserted into the workspace of Prolog.

* * * * * * * * * *

dbdrop(Relname,Stat,Msg).

This predicate sends a request to the DBMS to remove the EDB table *Relname*. If successful, *Stat* is returned the value "0" and *Msg* is not instantiated. Otherwise *Stat* is instantiated to "-1", and *Msg* is instantiated to the appropriate DBMS error message. *dbdrop* also retracts the corresponding data dictionary information.

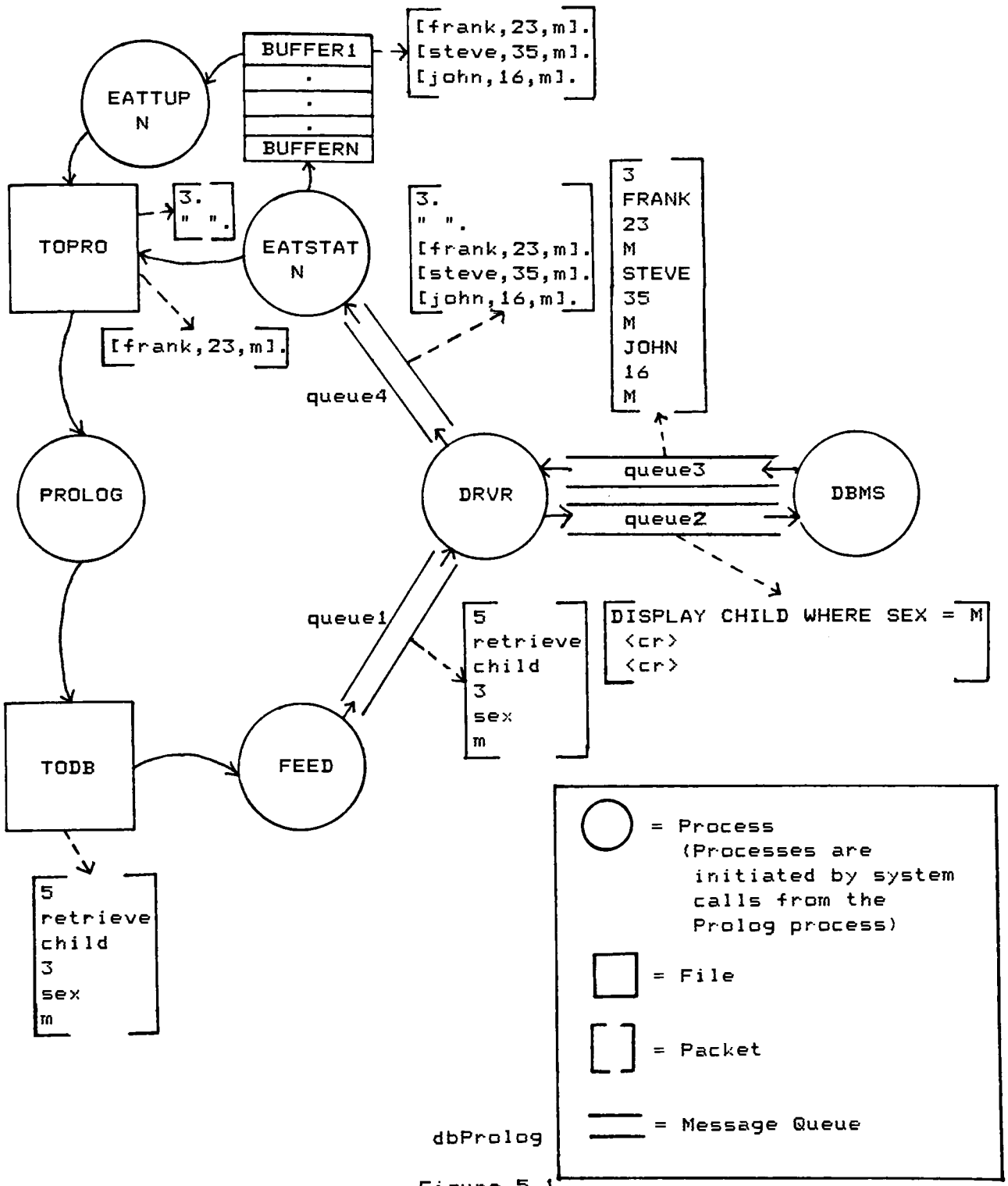
5.2 System structure

The overall structure and data flows of dbProlog are shown in Figure 5.1. The complete system consists of three components: the DBMS, Prolog and the dbDriver processes. There are four dbDriver processes: *drvr*, *feed*, *eatstat*, and *eattup*. The DBMS and dbDriver processes are all initiated by Prolog system calls. The processes communicate via four Unix message queues, which operate in a first-in, first-out fashion, and two standard files *todb* and *topro*. Buffer files are used to store tuples that are returned as a result of DBMS retrieval calls.

In Figure 5.1 the formats of the packets at each state of execution are illustrated for the dbProlog call:

dbretrieve(child(Name,Age,m),Stat,Msg).

assuming the proper data dictionary entry in the Prolog workspace. We will refer to Figure 5.1 throughout this chapter in explaining



dbProlog

Figure 5.1

the implementation of dbProlog.

5.2.1 Prolog

The Prolog component consists of six predicates that are used by the Prolog programmer to access an external DBMS. Each of the predicates initiate an action by the dbDriver component that forwards a request to the DBMS. These predicates reference other lower level predicates that are not accessible to the programmer. The lower level predicates check the validity of requests, construct interface packets, and process responses.

Each EDB table accessed by the Prolog component must be defined in the "data dictionary" that resides in Prolog's workspace. These definitions provide Prolog with information to construct generic calls to the DBMS via the dbDriver, and to interpret and use the results of a DBMS call. An example of a data dictionary entry is the following:

baseRel(child,3,[name,age,sex]).

where the arguments are defined as:

* *child* = name of EDB table
* *3* = number of attributes in the *child* table
* *[name,age,sex]* = list of attribute names of the EDB table *child*. These names must correspond exactly in order and spelling to the names of the EDB table attributes.

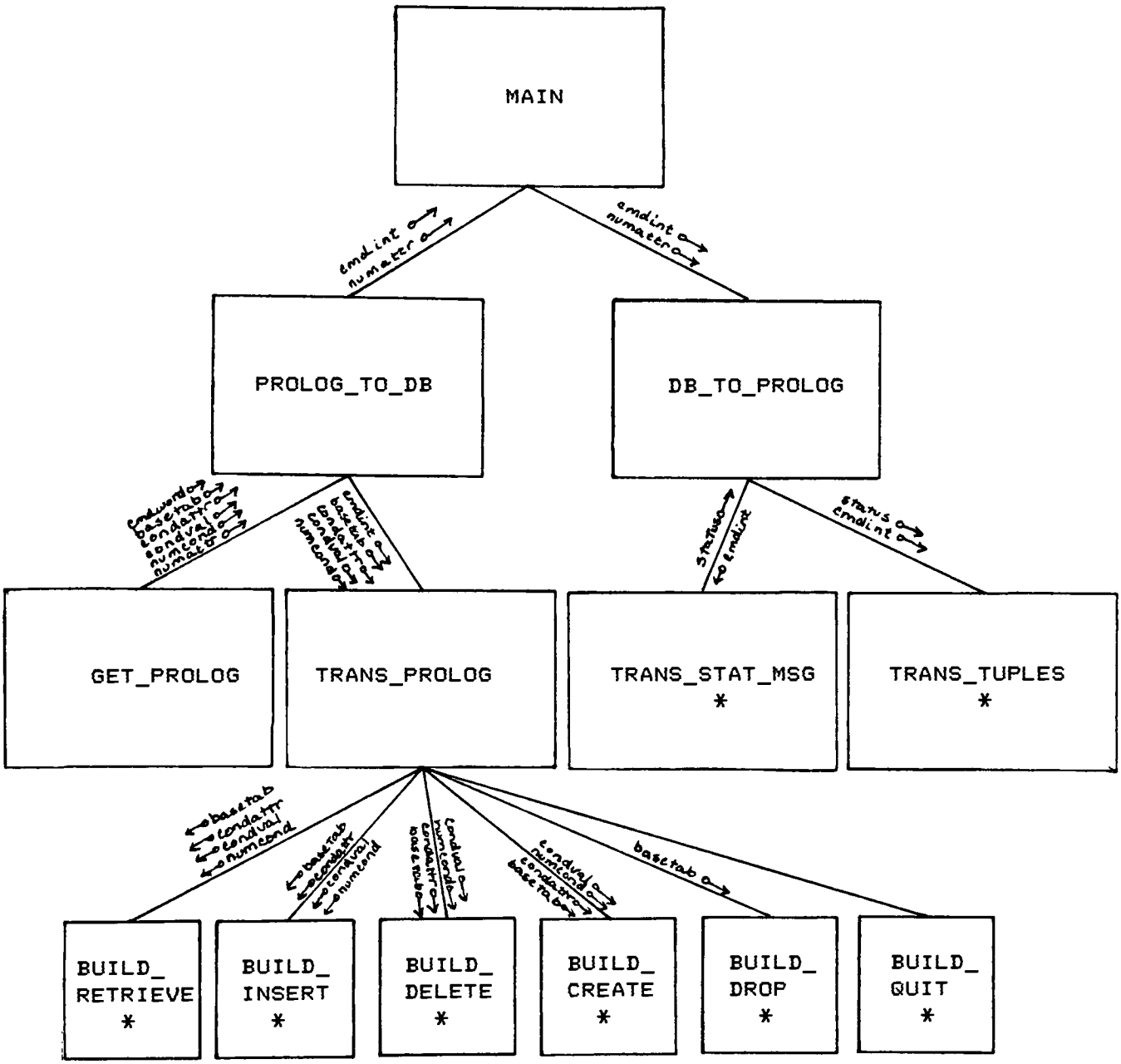
Communication between the Prolog component and the dbDriver component is accomplished by the use of the files *todb* and *topro*, which are accessed on the dbDriver side by the processes *feed*, *eatstat* and *eattup*. The Prolog component accesses these files through *tell* and *see* statements in predicates *$sendReq*, *$waitReply*, and *$getTuple*.

The *$sendReq* predicate writes a generic representation of a DBMS request to the file *todb*. Prolog then issues a system call to execute the process *feed*, which reads the file and places the contents of the file onto *queue1*, which in turn is read by the *drvr* process.

$waitReply issues the system call that activates the *eatstat* process, which transfers messages containing *Status* and *Msg* from *queue4* to the file *topro*. These data are then obtained by Prolog by reading the file *topro*. Likewise, *$getTuple* issues a system call to activate the process *eattup* to transfer a single DBMS tuple representation (or an end-of-file marker) to the file *topro*, which in turn is read by Prolog.

5.2.2 dbDriver

The dbDriver component is composed of four processes to facilitate communication and translation between Prolog and the DBMS. The main component is the process *drvr*, whose structure is shown in Figure 5.2. The two functions of *drvr* are:



DRVR

Figure 5.2

1) Receive generic DBMS request from Prolog, translate it to specific DBMS syntax, and send it to the DBMS.

2) Receive response from DBMS, translate it to Prolog acceptable format, and send it to Prolog.

The modules in Figure 5.2 identified by "*" are DBMS specific. Therefore, to interface to a different DBMS, these modules would have to be custom-written to adhere to the syntax of commands and the output format of the specific DBMS. Care was taken to design the interfaces between the custom modules and the standard modules so that calls to these modules should remain the same for any DBMS.

The prolog_to_db module of *drvr* blocks on a "get" from *queue1*. When a message appears on *queue1*, it is a DBMS request from Prolog. The type of DBMS command is determined from the command word of the message (*retrieve*, *delete*, etc.) and the appropriate DBMS command-building module is called using the rest of the data from the message. The result is a syntactically correct command for the DBMS, which is placed into *queue2*.

A parameter describing the type of command sent to the DBMS is passed to the db_to_prolog module, where it is used to anticipate the response from the DBMS. For example, if the command is *delete*, *drvr* only anticipates a status value and possibly a message, whereas if the command is *retrieve*, the *drvr* must anticipate receiving tuples from the DBMS.

drvr blocks on a "get" from *queue3* to await the reply from the DBMS. When it is received, it is appropriately translated to

a format acceptable to Prolog and placed on *queue4*.

The other processes that are part of the dbDriver component are the *feed*, *eatstat*, and *eattup* processes. The purpose of these processes is to facilitate communication with Prolog and to control the buffering of DBMS tuples. As described in Section 5.2, all of these processes are initiated by system calls from Prolog and are not continuously running when the interface is in execution. The *feed* process reads the *todb* file written by Prolog and transfers its contents to *queue1*, which is read by the *drvr* process. The *eatstat* process reads the translated DBMS reply from *queue4*, writes the status and message to *topro* and writes the tuples retrieved (if any) to a buffer file. The *eattup* process reads the first tuple from the appropriate buffer and writes it to the *topro* file, which is then read by Prolog. This tuple is then removed from the buffer.

All communication to and from the dbDriver component is done via Unix message system calls. The *drvr* process is the creator of these message queues.

5.2.3 DBMS

An extensive description of the DBMS will not be given here, as the DBMS is meant to be an interchangeable component. However, it is important that the DBMS support standard relational DBMS functions such as retrieval, insertion, and deletion of tuples, and creation and elimination of base tables.

The DBMS communicates only with the dbDriver component. This

communication is done by using the Unix message facility. The DBMS waits for a request by blocking on a "get" from *queue2*. When a request arrives from the *drvr* on *queue2*, the DBMS executes the request or determines an error condition. After the DBMS constructs its reply, the output is directed to *queue3*, which is read by the *drvr*.

The original version of the DBMS was constructed for the Database Implementation course (ICSS-739). It was modified for this thesis to be closer in functionality to most commercial DBMSs and to support interprocess communication. The original DBMS supported simple *delete* and *retrieve* requests, where only one or two conditions were allowed. Therefore, conditional requests could only be specified as follows (relop is =, !=, <, or >):

* <field1> *relop* <value1>
* <field1> *relop1* <value1> *and* <field2> *relop2* <value2>
* <field1> *relop1* <value1> *or* <field2> *relop2* <value2>

For dbProlog, the DBMS must have the capability to accept as many equality conditions as there are attributes in a table, such as:

<field1> = <value1> *and* <field2> = <value2> *and* ... *and*
<fieldn> = <valuen>

The DBMS was modified to support such statements. Most commercial DBMS products, such as SQL, do support such multi-condition commands [Date86].

The DBMS was also modified to support a "process mode" in addition to the standard "interactive mode" of operation. When executing in process mode, the DBMS input is accepted from a message queue instead of standard input, and all output is redirected to a message queue rather than standard output. In process mode, all prompts are supressed, and DBMS replies are preceded by an integer value, which gives the status of the response (success, error or number of tuples retrieved).

5.3 Interprocess communication

The purpose of this section is to describe the process communication mechanisms used in the dbProlog system. Packet formats will be described, and an overview of the Unix System V message facility will be given. In this implementation, packets are defined as a group of related messages, and messages are defined as the smallest units of information to be passed between processes (such as a command word, base table name, attribute name or value, integer, string representation of a DBMS command or DBMS response). If a packet is transferred between processes in the form of a file, as they are to and from Prolog, messages are delimited by a carriage return. If the message facility is used, as between all other processes, each message is a separate Unix message.

5.3.1 Interface language and packet formats

The interface language between the Prolog and dbDriver components is standard and independent of the specific DBMS used. However, because of DBMS syntax and output differences, the interface between the dbDriver and the DBMS must be tailored to the DBMS. In the following descriptions of the interfaces and packet examples, the reader is encouraged to refer to Figure 5.1.

5.3.1.1 Prolog -> dbDriver

Each interface packet that is transferred from Prolog to the dbDriver consists of a number of individual messages put on the queue. A standard packet consists of the following:

* Header - indicates number of messages that compose the packet
* DBMS command verb - generic DBMS command, such as *retrieve*, *load*, *insert*, *delete*, *create*, or *drop*
* DBMS base table name
* Number of attributes in DBMS base table
* Attribute/value list - messages alternating attribute names and corresponding values

A sample packet and its explanation follows:

```
5
retrieve
child
3
sex
m
```

This packet indicates a retrieval from the DBMS of all tuples in the table *child* where *sex* = "m". The "5" in the first message indicates that there are five succeeding messages that compose this packet. The "3" of the fourth message is used by the dbDriver to construct a Prolog-acceptable format of the DBMS tuples. The term "generic DBMS commands", implies that these same words are used regardless of the actual syntax of the DBMS commands. The format of the Prolog -> dbDriver packets is standard, regardless of DBMS used.

5.3.1.2 dbDriver -> DBMS

Each dbDriver -> DBMS interface packet consists of a message (or messages) that contains a string representing a syntactically correct DBMS command. This command is built from the information received from the Prolog -> dbDriver packet and from knowledge of the data manipulation and data definition languages of the DBMS.

For the DBMS used in this implementation, a sample dbDriver -> DBMS packet corresponding to the Prolog -> dbDriver

packet above would be:

DISPLAY CHILD WHERE SEX = M

<Carriage Return>

<Carriage Return>

(This DBMS requires two carriage returns to signify the end of this "DISPLAY" command).

5.3.1.3 DBMS -> dbDriver

The format of the DBMS -> dbDriver packets is dependent upon the DBMS used. In this implementation, there are two types of packets. The first type is sent to the dbDriver in response to any request. It consists of a message containing an integer status value (indicating success or failure of call) and possibly a DBMS error message string. A status value greater than or equal to zero indicates a successful DBMS call. A negative status indicates that an error condition was detected in processing the request, in which case it is followed by a message containing a string representing the DBMS error message.

The second type of DBMS -> dbDriver interface packet is only used for responding to retrieval requests. It consists of a status message, followed by a number of messages where each one represents a field value of a database tuple. In this type of packet, a negative status value indicates failure of the DBMS request and is followed by an error message string. A

non-negative status value indicates the number of database tuples retrieved as a result of the request. Some sample DBMS -> dbDriver packets are shown:

```
-1
"Record with this primary key already exists"
```

The above packet indicates failure of an insert call, because of the existence of a record with the same primary key value.

```
3
FRANK
23
M
STEVE
35
M
JOHN
16
M
```

Above packet is a typical packet returned as a result of a retrieval operation, where three tuples are returned ([FRANK,23,M], [STEVE,35,M] and [JOHN,16,M]).

5.3.1.4 dbDriver -> Prolog

The dbDriver -> Prolog packet format is independent of the DBMS used. There are two packet types. The first packet type consists of two messages. The first message represents the status value in a form that C-Prolog accepts. The second message contains a DBMS error message, if any, or a blank message if not, in C-Prolog format.

The second type of packet is used only for retrieval responses. The first message indicates status, or how many tuples have been retrieved and transmitted in the packet. Following the status are the actual tuples, if any, one tuple representation per message, in the form of a C-Prolog list. Samples of packets follow:

```
"0".
[Record with this primary key already exists].

(error condition detected)

"3".
[frank,23,m].
[steve,35,m]
[john,16,m].

(tuple retrieval)
```

5.3.2 Unix System V message facility

The Unix System V message facility provides the capability for communication between executing processes via the transfer of discrete blocks of data called "messages." It is one of three types of interprocess communication supported by Unix System V. The others are semaphores and shared memory [Unix87].

The message facility is implemented as a group of system calls. These include calls to initiate a message queue (*msgget*), those to change the characteristics of a message queue (*msgctl*), those to put a message onto a queue (*msgsnd*), and those to get a message from a queue (*msgrcv*). The *msgsnd* and *msgrcv* system calls can be called to cause the calling process to either block or not block depending on the status of the message queue.

Each message queue has an associated data structure to control its access and operation. Part of this data structure holds the operation permissions, which can be used to restrict the read/write access by owner, group or world. The *msgctl* system call can be used to return queue status information, set permissions or other parameters associated with a queue, or remove a message from the system.

To place a message on a queue, a process must know the unique identifier of the queue. This identifier can be obtained through the *msgget* system call, where an integer key, which was previously agreed upon between the processes that need to communicate, is passed to the system call, and the queue identifier is returned.

If the *IPC_CREATE* flag (a parameter of the *msgget* call) is on, and a corresponding queue identifier does not exist for the key value, a new message queue will be created, otherwise only the queue identifier will be returned.

Of the three types of interprocess communication supported by System V, messages appeared to provide the simplest mechanism for passing the types of packets required for the dbProlog system. However, system limitations, such as a limit on the number of messages permitted to exist on the system at one time, caused some problems in the implementation. These problems will be discussed in Section 6.2.

5.4 Execution of interface

All operation and control of the Prolog/DBMS interface is done through Prolog. However, the DBMS may still be run stand-alone if desired.

5.4.1 Initiation/termination of interface

The initiation of the dbProlog system by a user is accomplished by entering the following goal for Prolog to solve:

dbProlog(start).

The solving of this goal by Prolog causes the following operations to occur:

* The *drvr* process is initiated, and all message queues are created.
* The DBMS process begins execution in "process mode".
* The value for "maximum number of tuple buffers" is asserted into Prolog's workspace.
* The value for "current tuple buffer" is asserted into Prolog's workspace.

While the dbProlog system is in execution, the DBMS may be run stand-alone by entering the Prolog goal:

dbProlog(dbms).

Interface execution is terminated by entering the Prolog goal:

dbProlog(stop).

Solving this goal causes the following to happen:

* The DBMS process terminates.
* The *drvr* process terminates.
* All Prolog workspace assertions regarding buffer management are removed.
* All dbProlog access predicates are removed from Prolog's workspace.
* All *baseRel* facts are removed from Prolog's workspace.

5.4.2 Flow of control

Each DBMS Prolog request initiates a sequence of steps involving the translation of the request to DBMS syntax and the translation of the DBMS reply to Prolog syntax. This sequence of steps may be traced through the system structure in Figure 5.1 and is summarized as follows:

1) Prolog writes DBMS request information to file *todb*.
2) Prolog calls *feed*, which writes file *todb* to *queue1*.
3) *drvr* accepts request and translates it into syntax of DBMS.
4) *drvr* puts translated DBMS request onto *queue2*.
5) DBMS receives request and processes it.
6) DBMS puts reply on *queue3*. Reply consists of status, error message (if any) and tuple representations (if any).
7) *drvr* accepts reply from DBMS and translates it to a standard form that is acceptable to Prolog.
8) Prolog initiates *eatstat* to get the first two records from *queue4* and places them in *topro*. Prolog reads these records, which are the status and error message records from the file. *eatstat* transfers all tuple records (if any) into a buffer file.
9) If Prolog must retrieve tuples it initiates *eattup*. *eattup* transfers the first record from the specified

buffer into the file *topro* to be read by Prolog. *eatstat* then does any necessary buffer management. Prolog makes subsequent calls to *eattup* for each tuple that must be retrieved, or until an end-of-file marker is received.

5.4.3 Use of multiple "tuple buffers"

The concept of a circular queue of "tuple buffers" was used to support multiple, currently active DBMS retrieval calls. This provides the capability for database table joins and recursive database retrieval calls to be done from Prolog through the dbProlog interface. Tuple buffers hold the tuples that are produced by a response of the DBMS, since Prolog works only with one tuple at a time. When Prolog requires a tuple, one is removed from the buffer and transferred to Prolog for satisfying a goal, unification, instantiation of variables, or loading into Prolog's internal database. Prolog's activation stack is used for keeping track of the tuple buffer that corresponds to a specific DBMS request. Facts asserted into Prolog's workspace are used to identify the next free buffer to be used for a DBMS request.

The first attempt at multiple tuple buffers was made using Unix message queues for buffering. It was quickly discovered that there are system limits on both the number of bytes allowed in a message queue and on the number of messages allowed system wide in all queues at any time. Because of the possible quantities of messages that could be created as a result of a DBMS request,

these limitations made it impossible to use the message facility for buffering. To overcome this problem, files were used to buffer all DBMS tuple responses, thereby leaving all message queues empty upon completion of each DBMS request and response. File buffering was satisfactory for this implementation; however, the file I/O adds to execution overhead.

Management of buffers presented several problems. Placing restrictions on the number of buffers available requires efficient reuse of buffers that are no longer needed. This problem and its subsequent solution are discussed in Section 6.3.

5.5 Error handling

dbProlog was designed to detect some error conditions in the Prolog component, so that execution time is not wasted in sending invalid DBMS requests to the dbDriver and DBMS. Error detection is divided into what can be done in the Prolog component and what is done by the dbDriver and DBMS components. The six dbProlog DBMS access predicates include *Status* and *Message* as returned values from the interface. This is to allow the application programmer to do application specific error handling.

5.5.1 Prolog

The Prolog component must have knowledge of the external database. This is accomplished by the assertion of a *baseRel* fact for every EDB table that is to be accessed. When the user or

application program references a base table in a dbProlog predicate, the base table name and number of attributes can be checked against the *baseRel* assertions in Prolog's workspace. If a corresponding *baseRel* entry does not exist for the table name, or the number of attributes does not agree with the request, the dbProlog predicate returns immediately with a status value indicating failure, along with an error message. The user or application programmer must be careful to provide dbProlog with all the correct *baseRel* descriptions of any EDB tables to be accessed by asserting them into Prolog's workspace or including them in an application program.

5.5.2 dbDriver/DBMS

Any errors detected by the driver are treated as "fatal" conditions, as all input errors are detected by either Prolog or the DBMS. The purpose of the dbDriver is to serve only as translator between the two systems. The dbDriver expects only correct input from both Prolog and the DBMS, and it fails otherwise. Proper operation of Prolog and the DBMS will produce nothing but correct input to the dbDriver.

All error handling done by the DBMS is reported. Therefore, any error that would be detected by the DBMS in interactive mode, will be detected through the interface, and the user will be notified. Some of the errors detected by the DBMS used here are:

* Duplicate primary key

* Insert into a table that does not exist
* Create of table that already exists
* Drop of table that does not exist
* Invalid integer field

When an error is detected by the DBMS, a status value of -1 is transmitted to the dbDriver along with the appropriate error message. This information is then converted to Prolog format and transferred to Prolog.

Chapter 6

Results

6.1 Goals/objectives met

The major goal of the dbProlog design was to provide both tight and loose coupling between Prolog and a DBMS. Loose coupling is accomplished when tuples from an external database are loaded into Prolog's internal database. In dbProlog, the *dbload* predicate accomplishes this. Tight coupling is supported in dbProlog by the *dbretrieve* predicate. Tight coupling occurs when fields of an EDB tuple instantiate variables in a Prolog predicate.

Although not stated as a goal, it is important that dbProlog supports Prolog backtracking. After a tuple is retrieved and variables are instantiated, backtracking may cause another tuple to be retrieved, which will re-instantiate the variables. If application programming is done such that EDB retrievals are transparent to the user (see operation examples), then the Prolog build-in operations of *bagof* and *setof* work as expected.

Another goal was to support assertion and retraction of EDB tuples, just as IDB tuples are asserted and retracted. These capabilities were implemented via the dbProlog predicates of *dbassert* and *dbretract*.

Another objective was to leave the C-Prolog interpreter

unmodified. Fortunately, this objective was met; however, it required creating primitive means of communication between Prolog and the dbDriver processes. Likewise, modification of the DBMS was avoided; however, it became apparent early in the design that the DBMS needed enhancements to allow multiple conditions to be specified for tuple selection. These modifications were minor and gave the DBMS capabilities that are comparable to other commercial DBMSs. Another modification to the DBMS was the development of a "process mode" of execution. When executing in process mode, the DBMS returns a status value in addition to an error message or tuples. Also, prompts and headers are suppressed and communication is via the message facility rather than standard input and output.

Overall, the implementation was successful; however, several problems were encountered. Most of the difficulties were related to the use of the Unix message system. In addition, the development of a tuple buffer management scheme was challenging. These areas are discussed in Sections 6.2 and 6.3

6.2 Message system limitations

Message system limitations caused problems that led to modifications of the original dbProlog implementation. The first symptom appeared to be processes hanging while trying to "put" a message onto a full queue. Processes that were blocked on a "put" to a full queue appeared not to unblock after messages were taken from the queue. The initial solution was to make these calls

non-blocking, by inserting a sleep system call inside a "while" loop, executing until the "put" is successful. It later became apparent that the real cause of this problem was the system limit on the number of messages in all queues.

The second symptom also appeared to be processes hanging (i.e. infinite execution of while/sleep loops) when executing a "put" to a message queue. However, processes were still hanging even when the target message queue was first flushed. It was then discovered that there is a limit to the number of messages that can exist in all queues system-wide at any given time. This required an alternate solution to the multiple DBMS call capability, which led to the file buffering solution. In other words, the limit on the number of messages in all queues prevented the queues from being used for buffering.

Since most DBMSs retrieve groups of tuples that fit a set of retrieval criteria in blocks, and Prolog only works with facts (or tuples) in a one-at-a-time fashion, buffering of the retrieved EDB tuples is required. The first dbProlog implementation permitted only one DBMS request to be active at any time. Therefore, any Prolog statement could have at most one reference to an EDB table. In this implementation, the dbDriver -> Prolog message queue was used for buffering. As Prolog required another tuple, one was retrieved from the queue. When a new request to the DBMS was issued, the queue was flushed before storing the tuples from the new request.

When the capability of allowing multiple DBMS requests was implemented, the first attempt used multiple message queues for

buffering. This system quickly failed as a result of exceeding system message limits. Allowing multiple DBMS requests requires separate buffering of tuples from separate DBMS requests. When backtracking requires a tuple from a previous request, the tuple must be retrieved from the correct buffer. Therefore, buffering was done outside of the message queues so that after each DBMS response, all message queues are left empty in preparation for the next request. To keep the implementation simple, the buffers were implemented by external flat files managed by the *eatstat* and *eattup* processes.

6.3 Use and management of buffers

The greatest challenge of the multiple DBMS call implementation was the management of buffers. The two major issues were:

1) Prolog must retrieve from the correct buffer for a DBMS call; therefore, the *drvr* and *eatstat* processes must deposit DBMS responses into the correct buffer.
2) Accumulated buffer files that are no longer needed must be removed or reused.

The solution to 1 was straightforward. Prolog stores the number of the next buffer to be used by an asserted fact in its workspace. When Prolog initiates the *eatstat* process to receive the DBMS response, the call includes an integer parameter to

assign any retrieved tuples to a specific buffer. This buffer number is stored as an instantiated variable on Prolog's stack with other information relating to that specific DBMS request. Therefore, if backtracking occurs, requiring another tuple from a previously issued DBMS request, the buffer number variable will be reinstated automatically by Prolog's stack and backtracking mechanisms. To retrieve another tuple, Prolog initiates the *eattup* process with the buffer number as a parameter, to retrieve from the correct buffer.

Problem 2 was much more difficult to solve. It requires determining when a buffer is no longer needed and may be reused or deleted. The most obvious solution was the "don't worry about it" approach, which would allow an infinite number of buffers to be created during each run of dbProlog. Clearly, this is not a good solution, but it could be improved slightly, using the knowledge that once the last tuple is taken from a buffer, the buffer is no longer needed for that DBMS request. It then could be used for the next DBMS request issued.

To implement an efficient solution, one must know when a buffer is no longer needed as a result of backtracking. If two successive DBMS retrieval requests are made (such as that found in a join statement) the buffer of the first request must be retained, as backtracking may require retrieval from it. However, if a buffer is no longer needed because the user entered a carriage return to stop backtracking, (rather than ";" followed by a carriage return to backtrack and retrieve another tuple) and end-of-file has not been reached in the buffer file, processing

stops and the buffer cannot be deleted programmatically. In other words, Prolog cannot tell the difference between this type of stopping of a retrieval and the suspension of a retrieval to make another DBMS request (which may later backtrack to the first buffer). If the retrieval is stopped by the user, the buffer can be reused. However, if it is merely suspended, the buffer cannot be reused.

This problem was overcome by implementing a circular queue of buffers. This is not a foolproof solution, but it is a practical one. The Prolog component stores a fact indicating the maximum number of buffers allowed to exist in the system. This number should be realistic with respect to system resources, yet large enough to support anticipated levels of multiple DBMS calls and recursion. The buffers are assigned one-at-a-time, in consecutive order. After the last buffer is used, the first buffer is reused. To maximize buffer usage, if a buffer is the most recently used and its end-of-file is reached, it is reassigned as the buffer for the very next DBMS request.

Problems will occur, such as data being lost and wrong data retrieved if a Prolog program exceeds the anticipated number of buffers. This may happen as a result of a large number of concurrently active DBMS calls where a buffer that is in use is reused. A solution for this was not addressed in this implementation, but it can be viewed as a condition similar to that of exceeding Prolog's internal stack. A possible solution is outlined in Section 7.2.2.

6.4 Operation

This section provides a description of the operation and use of dbProlog by examples. For a complete description of dbProlog use, consult the "dbProlog User Manual" found in Appendix C.

6.4.1 Data dictionary and application program examples

dbProlog requires a "data dictionary" to be resident in Prolog's workspace. These entries must match exactly with the external database that is being accessed. The following is an example of data dictionary entries for two EDB tables *parent* and *child*:

```
baseRel(parent,2,[pname,cname]).
baseRel(child,3,[name,age,sex]).
```

Application program statements provide the link between dbProlog and an application end-user. The application programmer uses the dbProlog predicates to provide a simpler, application specific interface to the user. Through proper programming, an external DBMS retrieval may appear to the end-user as a retrieval from Prolog's internal database. The application programmer may wish to do all error handling to present the end-user with a simpler interface. In each of the following sample application program statements, a dbProlog predicate is called that effects a DBMS operation (see predicate descriptions in Chapters 4 and 5).

The statements following the dbProlog predicate check the returned status value and display an error message if the status indicates that an error condition is present.

```
parent(ParentName,ChildName) :-
     dbretrieve(parent(ParentName,ChildName),Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).

child(Name,Age,Sex) :-
     dbretrieve(child(Name,Age,Sex),Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *parent* and *child* predicates call the *dbretrieve* predicate to retrieve tuples from the corresponding base table. The arguments may or may not be instantiated to values to specify retrieval criteria.

```
insert(TupleDesc) :-
     dbassert(TupleDesc,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *insert* predicate defined here accepts one argument which is a description of a tuple to be inserted into a base table (such as *parent(frank,john)*). This description is passed to the *dbassert* predicate which requires all arguments to be instantiated so that the tuple can be inserted into the base table.

```
loadtuples(TupleDesc,IdbRel) :-
     dbload(TupleDesc,IdbRel,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *loadtuples* predicate accepts two arguments. The first is a tuple description which describes the criteria for retrieval from the DBMS (such as *parent(lorr,X)*). The second argument is the functor name which the tuples will be asserted under in Prolog's workspace. It may or may not be the same as the base table name. The arguments are passed to the *dbload* predicate.

```
delete(TupleDesc) :-
     dbretract(TupleDesc,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The delete predicate defined here accepts one argument which is a description of a tuple to be deleted from a base table (such as *parent(roger,joe)*). This description is passed to the *dbretract* predicate.

```
create(Relname,Attrlist) :-
     dbcreate(Relname,Attrlist,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *create* predicate implemented here accepts two arguments that define a base table and corresponding record structure. The first argument is the name of the base table that is to be created and the second argument is the description of the record format, which is a list of *attr* structures such as:

```
[attr(name,char),attr(age,int),attr(sex,char)]
```

(see *dbcreate* description in Section 5.1). The arguments are passed to the *dbcreate* predicate.

```
drop(Relname) :-
     dbdrop(Relname,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *drop* predicate takes one argument which is simply the name of the base table to be removed from the external database. This name is passed to the *dbdrop* predicate.

6.4.2 Sample dbProlog conversation

This section provides a sample dbProlog conversation that uses the application program statements and data dictionary definitions that were given in the previous section.

---

1) Starting dbProlog:

```
;int file contains pre-loaded dbProlog code

$ prolog int

C-Prolog version 1.4
[ Restoring file int ]

yes

; The application program code file, dbapp, is loaded

| ?- [dbapp].
dbapp consulted 2748 bytes 0.783333 sec.

yes

; When dbProlog(start) is executed, the two files
; dbProlog and tools are reconsulted in case the
; interface is being restarted during a Prolog session

| ?- dbProlog(start).
dbProlog reconsulted 0 bytes 4.06667 sec.
tools reconsulted 0 bytes 0.850002 sec.

yes
```

---

2) Running DBMS stand-alone:

```
| ?- dbProlog(dbms).

) DISPLAY PARENT SORTED

RELATION : PARENT

PNAME )) DEE
CNAME )) JOHN

PNAME )) FRANK
CNAME )) DEE

PNAME )) LORR
CNAME )) DEE

((Press RETURN key to continue))


PNAME )) STEVE
CNAME )) JOHN

PNAME )) THERESA
CNAME )) FRANK

) QUIT

no
```

---

3) Transparent retrieval:

```
| ?- parent(X,Y).

X = theresa
Y = frank ;

X = frank
Y = dee ;

X = lorr
Y = dee ;

X = dee
Y = john ;

X = steve
Y = john ;

no
```

---

4) Transparent join of tables:

```
;the predicate "grand(X,Z)" is defined in the Prolog
; workspace as:
; grand(X,Z) :- parent(X,Y),parent(Y,Z).


| ?- grand(X,Y).

X = theresa
Y = dee ;

X = frank
Y = john ;

X = lorr
Y = john ;

no
```

---

5) Creating view of base table

```
; the predicate "minor(Name,Age)" is defined in the Prolog
; workspace as:
; minor(Name,Age) :- child(Name,Age,Sex),Age < 21.


; ?- minor(Name,Age).

Name = john
Age = 16 ;

Name = maltic
Age = 9

yes
```

---

6) Recursive DBMS calls:

```
; the predicate "ancestor(X,Y)" is defined in the Prolog
; workspace by the following two rules:
; ancestor(X,Y) :- parent(X,Y).
; ancestor(X,Z) :- parent(X,Y),ancestor(Y,Z).


| ?- ancestor(X,Y).

X = theresa
Y = frank ;

X = frank
Y = dee ;

X = lorr
Y = dee ;

X = dee
Y = john ;

X = steve
Y = john ;

X = theresa
Y = dee ;

X = theresa
Y = john ;

X = frank
Y = john ;

X = lorr
Y = john ;

no
```

---

7) Use of Prolog built-in function *bagof*:

```
| ?- bagof(Name,Age^child(Name,Age,Sex),L).

Name = _0
Age = _1
Sex = f
L =  [dee,|[tumby,|[snowbear,|[chrissy,|[sue,|[beta,|[carrie,|[debb,|
[fawna,|[gertie,|[kelly,|[lana,|[maura,|[ursula,|[vanna,|[xavier,|
[zenity]]]]]]]]]]]]]]] ;

Name = _0
Age = _1
Sex = m
L =  [john,|[frank,|[steve,|[maltic,|[sebastian,|[mark,|[brian,|[allan,|
[efa,|[harold,|[isa,|[jocko,|[norm,|[oscar,|[perry,|[quincy,|[roger,|
[sam,|[tubs,|[wally,|[yoda]]]]]]]]]]]]]]]] ;

no
```

---

8) Loading of external database tuples into the internal database of Prolog (loose coupling):

```
| ?- loadtuples(male(X),male_idb).

X = _0

yes
| ?- male_idb(X).

X = frank ;

X = john

yes
```

---

9) Insert and delete of EDB tuples:

```
| ?- insert(child(bunky,11,m)).

yes
| ?- child(bunky,Age,_).

Age = 11 ;

no
| ?- delete(male(roger)).

yes
| ?- male(roger).

no
```

---

10) Creation and deletion of EDB tables:

```
| ?- create(car,[attr(make,char),attr(year,int),attr(color,char)]).

yes
| ?- insert(car(chevy,1987,grey)).

yes
| ?- car(Make,Year,Color).

Make = chevy
Year = 1987
Color = grey ;

no
| ?- drop(car).

yes
| ?- insert(car(chevy,1987,grey)).
-1
Invalid Request
Make = _0
Year = _1
Color = _2
yes
```

---

11) Error handling:

```
; error detected by dbProlog

| ?- create(newtab,[attr(field,integer)]).
-1
Invalid attribute/type list
yes


; error detected by DBMS

| ?- insert(child(bob,m,12)).
-1
Invalid INT field
yes
```

---

12) Termination of dbProlog processing:

```
| ?- dbProlog(stop).

yes
| ?- halt.

[ Prolog execution halted ]
```

Chapter 7

Conclusions

7.1 Review of system

dbProlog is an interface between C-Prolog and a relational DBMS. To the application programmer, dbProlog is a group of Prolog predicates that effect communication between the Prolog process and a DBMS server process. DBMS requests are initiated through the Prolog predicates to execute standard operations that are supported by nearly all relational DBMSs. The DBMS executes the request, and a response is sent back to Prolog. Properly written application programs make the interface transparent to the end-user and make access to an external database look the same as access to the internal database of Prolog. The major component of dbProlog is a driver process that is customizable to the requirements of whatever DBMS is used. The key function of the driver is to translate requests and responses between the DBMS and Prolog processes.

7.1.1 Uses for dbProlog

A natural use for dbProlog is for the development of expert systems that use data which already exist in a relational database. Similarly, dbProlog could be used to write user

interfaces or "front-ends" to databases. dbProlog may also be useful when the fact base required by a Prolog program is too large for Prolog's internal workspace. By storing the data in an external database, the data may be better managed by the DBMS facilities, and Prolog's workspace remains free.

7.1.2 Advantages of dbProlog

The dbProlog system for interfacing Prolog with a DBMS, reinforces the relational database concept of "data independence." Data independence can be defined as "the immunity of applications to change in storage structure and access strategy" [Date86]. dbProlog provides the Prolog programmer with a consistent form of access to any relational database, assuming the correct driver exists. Therefore, Prolog application programming may be done independently of the specific DBMS being used. Also, DBMSs may be interchanged (with appropriate driver modifications) with no effect on Prolog Programs, if base table contents and attributes remain the same.

dbProlog supports rapid development and growth of applications. An application may be initially written using data from Prolog's internal database. However, if the amount of application data increases or must be obtained from a database, the dbProlog predicates can be used for retrieval from the external database.

dbProlog provides the capability for a recursive definition or view of a relational database table. Recursion is not usually

supported by relational DBMSs. Also, dbProlog provides join capabilities that may not be supported by a DBMS. Most DBMSs only support joins of two tables. In dbProlog, the number of tables that can be joined depends only on buffer restrictions.

dbProlog is useful for application areas where multiple users require concurrent access to a database. Since dbProlog accesses the database through standard DBMS facilities, all security, integrity and concurrency control mechanisms of the DBMS are used. (The DBMS used in this implementation does not have security or concurrency control.) Therefore, many users, possibly through various application types, may access the DBMS concurrently with dbProlog users.

7.1.3 Disadvantages of dbProlog system

The interface between C-Prolog and dbProlog is the same regardless of DBMS used. However, because of syntax and output differences of various DBMSs, the dbProlog "DBMS driver" must be customized to a specific DBMS product. An effort was made to keep the driver component modular so that translation modules for various DBMSs could be interchanged easily.

Another disadvantage is that several layers of translation software must be executed to perform a dbProlog DBMS request. This and the interprocess communication add execution overhead to the dbProlog system, making the system much slower than pure Prolog.

7.2 Future improvements and enhancements

To make dbProlog a viable application tool, enhancements are necessary, mainly in the areas of run-time efficiency and buffer management.

7.2.1 Run-time efficiency

Optimization of the execution speed of dbProlog is needed. Improving the mechanism of communication to and from C-Prolog would greatly improve run-time efficiency. The implementation of direct FIFO or message queue communication between Prolog and the driver processes would require modifications to the C-Prolog interpreter, but this would eliminate the need for the auxiliary driver processes, reducing the overhead of their initiation and execution. Functionality would have to be added to the driver, however, to perform operations such as "get next tuple from buffer 2" etc.

Use of Unix System V shared memory facility instead of message queues may also improve system speed. However, this would only be possible for systems executing on the same CPU.

7.2.2 Better buffer management

To eliminate the restraint on the maximum number of buffers allowed, the system must be able to determine when a buffer is no

longer needed, so that it may be used for another request. For reasons outlined in Section 6.3, when a new DBMS request is issued, dbProlog has no way of knowing whether previously used buffers are still needed. If a single Prolog goal generates more than one DBMS request, more than one buffer will be required. In dbProlog, there are two conditions that indicate a buffer is no longer needed:

1) End-of-file is reached on buffer
2) User enters carriage return to stop backtracking

In this implementation, 1 is managed efficiently. To handle situation 2, the carriage return entered by the user must be intercepted by the system and all buffers cleared before processing stops. If instead the user continues backtracking by entering ";", all buffer files will eventually reach end-of file and, therefore, be reused. This could be implemented by modifications to the C-Prolog interpreter or addition of a "meta-level interpreter" to manage buffers. The meta-level interpreter may appear as C-Prolog to the user. It would pass all user input to C-Prolog for processing; however, before passing carriage returns during backtracking, the interpreter would perform buffer management.

7.3 Extensions and future work

There are several areas that could be investigated to take advantage of dbProlog's versatility. However, dbProlog should be enhanced to improve its run-time efficiency and buffering limits, before being used in information intensive applications.

A natural extension to the dbProlog system implemented here is interfacing it with a commercial DBMS product. This would require customizing the dbProlog translation and communication modules for compatibility with the DBMS.

Development of dbProlog applications, such as small expert systems, should be straightforward and no more difficult than writing expert systems in C-Prolog. Also, database user interfaces should be easily constructed with dbProlog.

One may also wish to modify dbProlog to interface with an alternate Prolog implementation. A compiled Prolog may improve run-time performance. However, dbProlog was constructed exclusively for C-Prolog, and such a modification could be extensive.

Interfacing Prolog over a network to a DBMS running on a separate machine could also be considered. In addition to dbProlog modifications to accomodate the network communication protocol, the implementer would have to determine which machine would be best for the placement and execution of the driver component.

7.4 Concluding remarks

Many things were learned by trying to interface two distinct systems, which are extremely powerful themselves, to produce a unified system. The most important lesson learned (and the biggest feeling of accomplishment) was that of managing an entire project from beginning to end. A self-imposed schedule with milestones was closely followed. Being a one person "project team" took me through the various stages of systems development: research, design, development of system, implementation, testing, and most importantly, post-implementation analysis. Through research I learned that for the number of people attempting to develop such a system, there are nearly as many distinct approaches.

I have learned much about Prolog. I discovered the true power of Prolog, as found in mechanisms such as backtracking and ! (cut) by verifying they still work as intended in dbProlog. I also learned much about the Unix System V operating system, especially the interprocess communication facilities and their limitations.

After many months of work on this thesis and implementation, I must admit that I have enjoyed it, as I have enjoyed the many courses that I have taken that prepared me for this. One of the things that I had most wanted to receive as a result of this work I have already received: a great sense of accomplishment. Gaining recognition from my peers in the field who share similar

interests would also be a welcome reward. At the very least, I hope this will fulfill degree requirements for a Master's degree in Computer Science.

Bibliography

[BaBo83] Bates, M., Bobrow, R. *Natural Language Interfaces: What's Here, What's Coming, and Who Needs It.* In Artificial Intelligence Applications for Business. Proceedings of the NYU Symposium (New York University, May, 1983) Ablex Publishing Corporation, 1984, 179-193.

[Berg85] Berghel, H. L. *Simplified integration of Prolog with RDBMS.* In DATABASE (Vol. 16 No. 3). Spring 1986, 3-12.

[Boas86] van Emde Boas, Ghica;van Emde Boas, Peter. *Storing and Evaluating Horn-clause Rules in a Relational Database.* In IBM Journal of Research and Development (Vol. 30, No. 1) January 1986

[Bocc86] Bocca, Jorge. *EDUCE: A Marriage of Convenience: Prolog and a Relational DBMS.* In IEEE 1986 Symposium on Logic Programming (Salt Lake City, Utah, September 22-25) 1986, 36-45.

[Brat86] Bratko, Ivan. Prolog Programming for Artificial Intelligence, Addison-Wesley, 1986.

[BrJa84] Brodie, Michael L.; Jarke, Matthias. *On Integrating Logic Programming and Databases.* In EXPERT DATABASE SYSTEMS Proceedings From the First International Workshop (Editor: L. Kerschberg, Kiawah Island, S.C. October 24-27) Benjamin Cummings, 1984, 191-207.

[Chan78] Chang, C.L. *DEDUCE 2: Further Investigations of Deduction in Relational Data Bases.* In LOGIC AND DATABASES Proceedings of the Symposium on Logic and Databases (Editors: Gallaire, Herve;Minker, Jack) Plenum Press, 1978, 201-236.

[ChWa86] Chang, C.L.; Walker, A. *PROSQL: A Prolog programming Interface with SQL/DS.* In EXPERT DATABASE SYSTEMS Proceedings From the First International Workshop (Editor: L. Kerschberg, Kiawah Island, S.C. October 24-27) Benjamin Cummings, 1984, 233-246.

[ClMe84] Clocksin, W.F.; Mellish, C.S. Programming in Prolog, Springer-Verlag, 1984.

[Cloc87] Clocksin, W. *A Prolog Primer.* In BYTE, (Vol. 12, No. 9), August 1987, 147-158.

[Codd70] Codd, E.F. *A Relational Model of Data for Large Shared Databanks.* In Communications of the ACM (Vol. 13, No. 6) 1970, 377-387.

[Dahl82] Dahl, V. *On Database Systems Development through Logic.* In ACM Trans. Database Systems 7, 1982, 102-123.

[Date83] Date, C.J. An Introduction to Database Systems Vol. II, Addison-Wesley, 1983.

[Date86] Date, C.J. An Introduction to Database Systems Vol. I, Addison-Wesley, 1986.

[FuDa77] Futo, I.; Darvas, F.; Szeredi, P. *The Application of Prolog to the Development of QA and DBM Systems.* In LOGIC AND DATABASES Proceedings of the Symposium on Logic and Databases (Editors: Gallaire, Herve; Minker, Jack) Plenum Press, 1978, 347-376.

[GaMi84] Gallaire, H.; Minker, J.; Nicolas, J. M. *Logic and Databases: A Deductive Approach.* In Computing Surveys, (Vol. 16, No. 2), June 1984, 153-185.

[GaMi77] Gallaire, H.; Minker, J.; Nicolas, J. M. *An Overview and Introduction to Logic and Data Bases.* In LOGIC AND DATABASES Proceedings of the Symposium on Logic and Databases (Editors: Gallaire, Herve; Minker, Jack) Plenum Press, 1978, 3-28.

[Gray85] Gray, P.M.D. *Efficient Prolog Access to CODASYL and FDM Databases.* In Proceedings of the ACM - SIGMOD 1985 International Conference on Management of Data, (Austin, Texas, May 28-31) 1985, 437-443.

[HoLe85] Hornsby, C., Leung, C.H.C. *The Design and Implementation of a Flexible Retrieval Language for a Prolog Database System.* In ACM SIGPLAN Notices (Vol. 20, No. 9) September, 1985, 43-51.

[JaVa83] Jarke, M.;Vassiliou, Y. *Databases and Expert Systems: Opportunities and Architectures for Integration.* In New Applications of Databases, Workshop Proceedings at Cambridge England (Editors: Gardarin, G.; Gelenbe, E, September 2-3, 1983), Academic Press, 1984, 185-201.

[JaVa84] Jarke, M.; Vassiliou, Y. *Coupling Expert Systems With Database Management Systems.* In Artificial Intelligence Applications for Business. Proceedings of the NYU Symposium (New York University, May 1983) Ablex Publishing Corporation, 1984, 65-85.

[Kell82] Kellogg, C. *Knowledge Management: A Practical Amalgam of Knowledge and Database Technology.* In Proceedings of The National Conference on Artificial Intelligence, AAAI-82 (Carnegie Mellon University-University of Pittsburgh, Pittsburgh, PA, August 18-20, 1982) 306-309.

[KlSz85] Kluzniak, F., Szpakowicz, S. Prolog for Programmers. Academic Press, 1985.

[Komo84] Komorowski, Henryk J. *Rapid Software Development in a Database Framework - A Case Study.* In IEEE International Conference on Data Engineering (Los Angelis CA, April 24-27) 1984, 394-398.

[Luca86] Lucas, R. *An Expert System to Detect Burglars using a Logic Language and a Relational Database.* In Proceedings of the Fifth British National Conference on Databases (BNCOD 5, University of Kent at Canterbury, July 14-16, 1986) Cambridge University Press, 1986, 43-54.

[MaJo83] Marque-Pucheu, G.; Martin-Gallausiaux, J.; Jomier, G. *Interfacing Prolog and Relational Data Base Management Systems.* In New Applications of Databases, Workshop Proceedings at Cambridge England (Editors: Gardarin, G.; Gelenbe, E. September 2-3, 1983), Academic Press, 1984, 225-244.

[Marc86] Marcus, Claudia Prolog Programming, Arity Corporation, Addison-Wesley, 1986.

[Oagl88a] Oagley, Diane M. dbProlog User Manual. Rochester Institute of Technology, Rochester, NY, June 1988.

[Oagl88b] Oagley, Diane M. DBMS User Manual. Rochester Institute of Technology, Rochester, NY, June 1988.

[Pere85] Pereira, Fernando; Warren, David; Bowen, David; Pereira, Luis. C-Prolog User's Manual, Version 1.4. (Editor: Pereira, Fernando), April 24, 1985.

[Pere87] Pereira, Fernando. *The Indiscipline of Prolog Programming.* In AI EXPERT (Vol. 2, No. 6) June 1987, 7-9.

[PCGJ84] Parker, D.S.; Carey, M.; Golshani, F.; Jarke, M.; Sciore, E.;Walker, A. *Logic Programming and Databases.* In EXPERT DATABASE SYSTEMS Proceedings From the First International Workshop (Editor: L. Kerschberg, Kiawah Island, S.C. October 24-27) Benjamin Cummings, 1984, 35-48.

[Rett87a] Rettig, Marc. *Marrying Logic Programming and Databases.* In AI EXPERT (Vol. 2, No. 6) June, 1987, 15-19.

[Rett87b] Rettig, Marc. *Prolog and SQL: A Happy Union.* In AI EXPERT (Vol. 2, No. 7) July, 1987, 19-24.

[ScWa86] Sciore, Edward; Warren, David Scott. *Towards an Integrated Database-Prolog System.* In EXPERT DATABASE SYSTEMS Proceedings From the First International Workshop (Editor: L. Kerschberg, Kiawah Island, S.C. October 24-27) Benjamin Cummings, 1984, 293-305.

[StSh86] Sterling, Leon; Shapiro, Ehud. The Art of Prolog, MIT Press, 1986.

[Ullm82] Ullman, J.D. Principles of Database Systems, Second Edition, Computer Science Press, 1982.

[Unix87] UNIX System V Programmers Guide, AT&T, Prentice Hall, Inc., 1987.

[VaCl83] Vassiliou, Y.; Clifford J.; Jarke, M. *How does an expert system get its data?* (Extended Abstract) In Ninth International Conference on Very Large Data Bases (Florence, Italy) 1983, 70-72.

[Walk84] Walker, Adrian, *Databases, Expert Systems, and PROLOG.* In Artificial Intelligence Applications for Business. Proceedings of the NYU Symposium (New York University, May, 1983) Ablex Publishing Corporation, 1984, 87-109.

[YeTh86] Yeo, C.; Thorpe, J.; Longstaff, J. *Knowledge Base Enhancements to Relational Databases.* In Proceedings of the Fifth British National Conference on Databases (BNCOD 5, University of Kent at Canterbury, July 14-16, 1986) Cambridge University Press, 1986, 87-103.

[Zani86] Zaniolo, Carlo. *Prolog: A Database Query Language for All Seasons.* In EXPERT DATABASE SYSTEMS Proceedings From the First International Workshop (Editor: L. Kerschberg, Kiawah Island, S.C. October 24-27) Benjamin Cummings, 1984, 219-232.

[ZoGr87] Zobaidie, A.Al; Grimson, J.B. *Expert Systems and Database Systems: How can they serve each other?* In Expert Systems (Vol. 4, No. 1.) February 1987, 30-37.

Appendix A

The Relational Database Model

This section will provide a brief overview of the relational database model as defined by E. F. Codd [Codd70]. Some issues such as recovery, integrity, concurrency, and security are covered lightly, or not at all. The interested reader may consult [Date86], [Date83], and [Ullm82].

1. Definition of the relational model

Relational databases are databases that can be perceived by the user as a collection of rectangular tables. The table columns correspond to attributes, and the rows correspond to tuples. Each tuple of a relation (table) must contain the same number of values and be defined over the same set of attributes. Also, each relation must have a set of attributes which uniquely identifies each tuple in the relation, called the primary key. Information is neither contained in the ordering of the tuples in a relation, nor in the ordering of attributes in a tuple. Therefore, order of tuples and/or attributes is not important [Date86], [Ullm82].

A relational DBMS must support the operations of *SELECT*, *PROJECT* and *JOIN* without requiring the predefinition of physical access paths to the data. Other functions are also commonly supported, such as the capability to create and destroy tables.

In addition, relations must be normalized in order to avoid maintenance anomalies. The are several levels of normalization which will be explained in the following section [Date86].

2. Functional dependencies and normal forms

Before defining the normal forms of the relational model, some definitions must be given [Berg85], [Date86].

Functional dependence - An attribute *B* is functionally dependent on attribute *A* if and only if for each tuple in the relation, the value of *A* uniquely determines the value of *B*. (Ex: *Social Security Number* and *Name* are functionally dependent on each other.)

Transitive functional dependence - A transitive functional dependency of attribute *C* upon attribute *A* exists if and only if *C* is functionally dependent on *B* and *B* is functionally dependent on *A*.

Fully functional dependence - Attribute *C* is fully functionally dependent upon *A* and *B* if and only if *C* is functionally dependent upon their combination, but not upon either individually.

Determinant - A determinant is any attribute upon which some other attribute is fully functionally dependent.

Candidate key - A candidate key is a set of attributes of a relation that can be used as a unique identifier for the tuples in the relation.

Primary key - Primary keys are a special case of candidate keys. For a given relation, one of the candidate keys is chosen to be the primary key, and then the remainder (if any) are called the alternate keys.

Non-key attribute - A non-key attribute is any attribute that does not participate in the primary key of the relation.

Foreign key - A foreign key is an attribute (or attribute combination) in one relation *R2* whose values are required to match those of the primary key of some relation *R1* (*R1* and *R2* are not necessarily distinct).

The purpose of normal forms, is to make the logical structure of the database more desirable, and less prone to maintenance anomalies. "The fundamental point is that a given relation, even though it is normalized, may still posess certain undesirable properties; normalization theory allows us to recognize such cases and show how such relations can be converted to a more desirable form" [Date86]. The normal forms are defined as follows [Date86]:

1NF - A relation *R* is in first normal form (1NF) if and only if all underlying domains contain atomic values only. That is, at every row and column position, in every table, there is always exactly one data value, never a set of values.

2NF - A relation *R* is in second normal form (2NF) if and only if it is in 1NF and every non-key attribute is fully dependent on the primary key.

3NF - A relation *R* is in third normal form (3NF) if and only if it is in 2NF and every non-key atribute is nontransitively dependent on the primary key.

BCNF - A relation *R* is in Boyce/Codd normal form (BCNF) if and only if it is in 3NF and every determinant is a candidate key.

Fourth and fifth normal forms also exist, however, these are outside the scope of this paper. The interested reader may pursue them in [Date86].

3. Operations and levels of relational databases

For a database system to be labeled relational, it must at least support a specified subset of the relational algebra. Other operations usually supported are those that create and destroy relations and views. Typically, the same base subset of

operations can be found in any relational system.

The operations of the relational algebra that must be supported by a relational system are *SELECT*, *PROJECT* and *JOIN*. Even though they are less than the full algebra, there are few practical problems that can be solved with the algebra that cannot be solved with these operations [Date86]. These operations are defined as follows [Date86]:

SELECT - Extracts specified tuples from a specified relation. *SELECT* yields a horizontal subset of a given relation, that is, that set of tuples for which a certain comparison is satisfied. The result is also a relation.

PROJECT - Extracts specified attributes from a specified relation. *PROJECT* yields a vertical subset of a given relation, that is, that subset obtained by selecting attributes, in a specified left-to-right order, and then eliminating redundant tuples within the attributes selected, if necessary. The result is not necessarily a relation.

JOIN - Builds a relation from two specified relations consisting of all possible concatenated pairs of tuples, one from each of the two specified relations, such that in each pair the two tuples satisfy some specified condition. The result is also a relation.

Examples of other operations usually supported by relational DBMSs are:

CREATE/DROP Table - These operations are used to define new relations and to eliminate existing relations, respectively.

CREATE/DROP View - A view is a virtual table. It does not exist in its own right but appears to the user as if it does. Views are defined on base tables to be used as different ways to "look" at the base tables. These operations are used to define new views and eliminate existing views, respectively.

Not all database systems that are defined as relational support all of the same operations and concepts of the relational model. Relational systems can be further classified within the relational spectrum. The first requirement for a relational database is that it must be tabular; however, this alone does not make a system relational. The categories are defined as follows [Date86]:

Minimally relational - DBMS that supports *SELECT*, *PROJECT*, and *JOIN*, but does not support any of the other relational algebra operations.

Relationally complete - DBMS that supports all the operations of the relational algebra, including *UNION*, *INTERSECT*, *DIFFERENCE*, *PRODUCT*, and *DIVIDE* in addition to those supported by minimally relational systems.

Fully relational - DBMS that supports all aspects of the relational model. This includes all operators of the relational algebra and support for domains and integrity rules (see next section).

4. Integrity rules

Integrity rules enforce the logical consistency of the database. The relational model specifies two types of integrity, entity integrity and referential integrity. Other integrity rules may be specified for a database with regards to its specific application. An example of an application-specific integrity rule would be that the *SALARY* attribute of the *EMPLOYEE* relation cannot exceed $100,000. The two integrity rules defined by the relational model are [Date86]:

Entity integrity - No attribute participating in the primary key of a base relation is allowed to accept null values.

Referential integrity - If base relation *R2* includes a foreign key *FK* matching the primary key *PK* of some base relation *R1*, then every value *FK* of *R2* must either:

a) be equal to the value of *PK* in some tuple of *R1* or

b) be wholly null (i.e. each attribute value participating in that *FK* value must be null). *R1* and *R2* are not necessarily distinct.

Appendix B

Prolog

Prolog is a very high-level language based on the Horn clause subset of first-order predicate logic. It is a language for symbolic non-numeric computation and thus is an appropriate language for solving problems that involve objects and relations among objects. Prolog has a built-in theorem prover, or inference engine, which operates in a top-down, depth-first manner. Prolog also maintains a built-in database that stores the components of a Prolog program. The intent of this section is to highlight various points of the Prolog language. The interested reader may consult the references for more information on Prolog itself ([ClMe84], [Brat86], [StSh86]).

Components and execution of Prolog programs

Programming in Prolog can be stated simply as "you state or assert what is true and ask the system to draw conclusions" [BrJa84]. Another way to say this is "Execution of a Prolog program involves a depth-first search with backtracking and uses the unification process based on the resolution principle" [JaVa84]. A Prolog program is described in the following paragraphs [Brat86].

A Prolog program consists of clauses. There are three types of Prolog clauses: facts, rules, and questions. Prolog clauses

consist of a head and a body. The body is a list of goals (or questions) separated by commas. Commas are understood as conjunctions. Facts declare things that are always unconditionally true. Facts are clauses that have an empty body. An example of a fact is:

father(frank,john).

Rules declare things that are true depending on a given condition. Rules have a head and a non-empty body. An example of a rule is:

mother(X,Y) :- parent(X,Y),female(X).
(where *X* and *Y* are variables).

Questions provide the means for a user to ask the program what things are true. Questions only have a body, and consist of one or more goals. An example of a question is:

mother(lorraine,X),father(frank,X).
(where *X* is a variable).

In Prolog, a relation can be specified by stating a number of facts with the same predicate and arity. The arity of a Prolog clause is the number of arguments in the head of the clause. This is similar to a relational database table, except that there are no type restrictions on the elements in Prolog. A relation can also be specified in Prolog by stating rules that define it.

Querying relations by means of questions resembles querying a database. Prolog's answer to a question consists of a set of objects that satisfy the question.

In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possible backtracking. All this is done automatically by the Prolog system and is, in principle, hidden from the user.

Simple objects in Prolog are atoms, variables and numbers. Structures are used to represent objects that have several components. Variables are understood to be universally quantified, and in the course of an execution of a Prolog program, a variable can be substituted by another object. This substitution is called "instantiation".

Prolog supports the concept of recursion in that the body of a rule may contain a reference to the rule itself.

Prolog has a library of built-in procedures. These may provide facilities that cannot be obtained by definitions in pure Prolog, or they may provide programming conveniences. Prolog also has a built-in database where facts and rules are stored. Built-in predicates, such as *assert* and *retract* are available to manage the clauses in Prolog's database.

dbProlog

User Manual

by

Diane M. Oagley

May 28, 1988

Table of Contents

1. Preface

This manual describes the operation of dbProlog, a prototype system that provides a C-Prolog user access to data in an external relational database. dbProlog was implemented as a result of a Master's Thesis by Diane M. Oagley [Oagl88a]. The relational database management system used in this implementation was constructed as a project in the course ICSS-739 (Database Implementation), and is described in the document "DBMS User Manual" [Oagl88b]. dbProlog is based on the C-Prolog interpreter version 1.4 [Pere85]. dbProlog consists of modules written in C-Prolog and modules written in the language C. No modifications were made to the C-Prolog interpreter. The system runs under the Unix[1] System V operating system, and has been tested under version 3.51. This manual assumes the user has knowledge of the Prolog language and especially C-Prolog. References such as [ClMe84] and [Pere85] may be consulted. dbProlog is described in detail in the thesis report [Oagl88a].

dbProlog may be used in the development of expert systems that use data which already exist in a relational database. Similarly, dbProlog could be used to construct user interfaces or "front-ends" to databases. dbProlog may also be useful when the fact base required by a Prolog program is too large for Prolog's

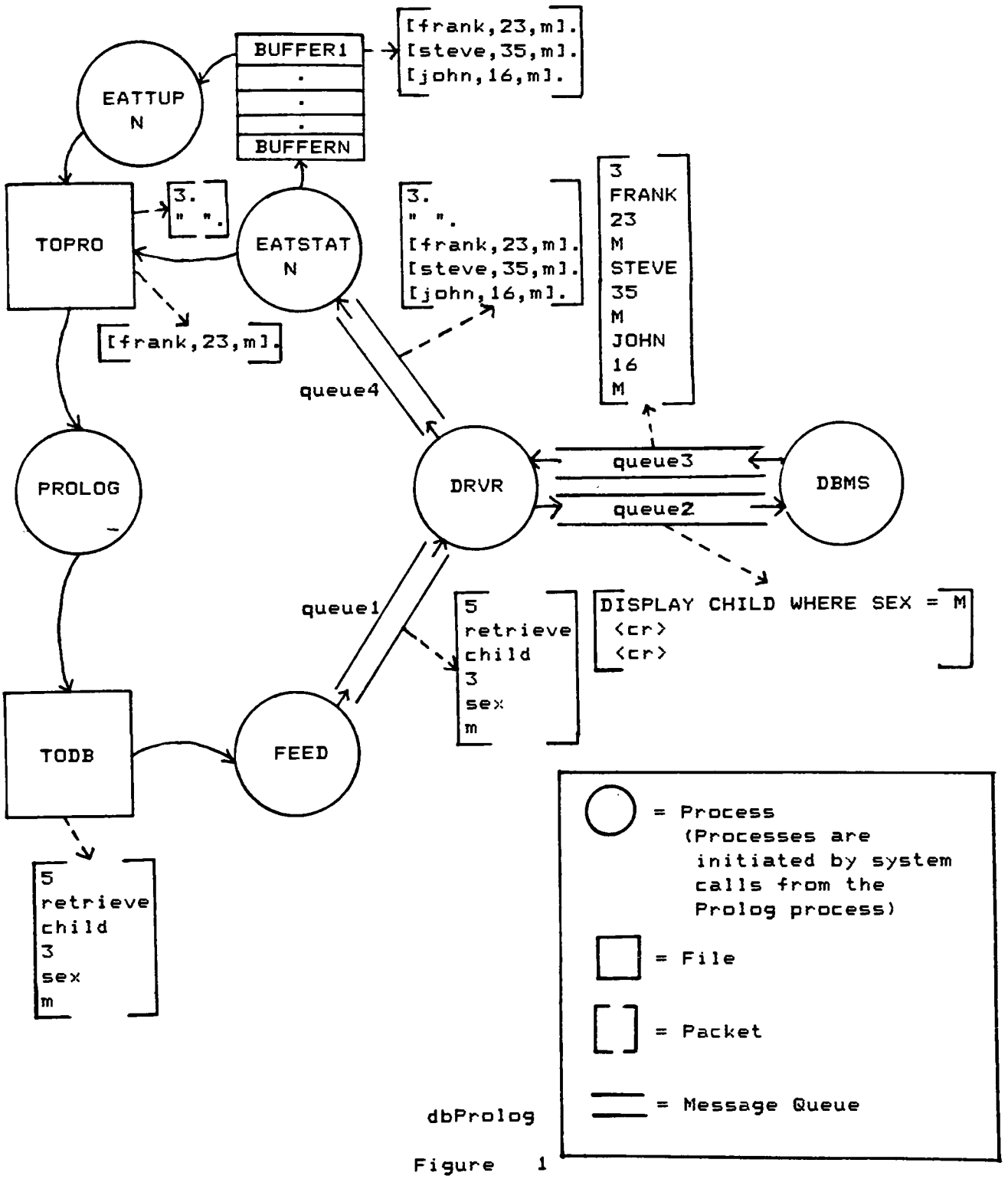
[1]Unix is a trademark of AT&T

internal workspace. By off-loading the data to an external database, the data may be better managed by the DBMS facilities, and Prolog's workspace remains free.

2. Overview

dbProlog is an interface between C-Prolog and a relational DBMS. To the application programmer, dbProlog is a group of Prolog predicates that effect communication between the Prolog process and a DBMS server process. DBMS requests are initiated through the Prolog predicates to execute standard operations that are supported by nearly all relational DBMSs. The DBMS executes the request, and a response is sent back to Prolog. Properly written application programs make the interface transparent to the end-user and make access to an external database look the same as access to the internal database of Prolog. The major component of dbProlog is a driver process that is customized to the requirements of whatever DBMS is used. The key function of the driver is to translate requests and responses between the DBMS and Prolog processes.

The overall structure and data flows of dbProlog are shown in Figure 1. The complete system consists of three components: the DBMS, Prolog and the dbDriver processes. There are four dbDriver processes: *drvr*, *feed*, *eatstat*, and *eattup*. The DBMS and dbDriver processes are all initiated by Prolog system calls. The processes communicate via four Unix message queues, which operate in a first-in, first-out fashion, and two standard files *todb* and *topro*. Buffer files are used to store tuples that are returned as a result of DBMS retrieval calls. In Figure 1 the formats of the



dbProlog

Figure 1

packets at each state of execution are illustrated for the dbProlog call:

dbretrieve(child(Name,Age,m),Stat,Msg).

The Prolog component consists of six predicates that are used by the Prolog programmer to access an external DBMS. Each of the predicates initiate an action by the dbDriver component that forwards a request to the DBMS. These predicates reference other lower level predicates that are not accessible to the programmer. The lower level predicates check the validity of requests, construct interface packets and process responses.

Each dbProlog request that causes a retrieval of tuples from the DBMS requires a buffer to hold the tuples, so they may be processed one-at-a-time by Prolog. More than one DBMS retrieval request may be active at any time, in programs where predicates reference more than one database table, or programs that contain recursive predicates. In such cases, several buffers may be required at the same time. dbProlog supports this by maintaining a "circular queue of buffers." The Prolog component stores a fact indicating the maximum number of buffers allowed to exist in the system. This is a tuneable system parameter and should be realistic with respect to system resources, yet be large enough to support anticipated levels of multiple DBMS calls and recursion. The buffers are assigned one-at-a-time, in consecutive order and are named "BUFF01", etc. After the last buffer is used, the first buffer is reused. To maximize buffer usage, if a buffer is the

most recently used and its end-of-file is reached, it is reassigned as the buffer for the very next DBMS request. Problems will occur, such as data being lost and wrong data retrieved, if a Prolog program exceeds the anticipated number of buffers. This may happen as a result of a large number of concurrently active DBMS calls where a buffer that is in use is reused prematurely.

3. Predicates and Operation

3.1 Data Dictionary

dbProlog requires a "data dictionary" to be resident in Prolog's workspace. These entries must match exactly with the base tables of the external database that is being accessed. The following is an example of a data dictionary entry:

baseRel(child,3,[name,age,sex]).

where the arguments are defined as:

* *child* = name of EDB table
* *3* = number of attributes in the *child* table
* *[name,age,sex]* = list of attribute names of the EDB table *child*. These names must correspond exactly in order and spelling to the names of the EDB table attributes.

3.2 Predicates

dbProlog provides the Prolog programmer with a set of Prolog predicates that effect communication with the DBMS. The predicates are described below.

* * * * * * * * * *

dbretrieve(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate returns one tuple to Prolog from the external base table *Relname* that satisfies the constraints placed on the attributes that are instantiated (*Attr$_n$*, etc.). If successful, the variables in the group *Attr$_1$,...,Attr$_n$* are appropriately instantiated, and *Status* is set to a positive value or zero indicating the total number of database tuples that satisfy the request. If the retrieval fails, *Status* is returned as "-1", and *Message* is instantiated to an appropriate error message. The first call to *dbretrieve* will place all retrieved tuples into a buffer, and return the first tuple to Prolog for processing. Backtracking into a subsequent *dbretrieve* call will retrieve the next tuple from the buffer.

* * * * * * * * * *

dbload(Relname(Attr$_1$,...,Attr$_n$),Idbname,Status,Message).

This predicate returns all tuples to Prolog from the external base table *Relname* that satisfy the constraints placed on the attributes that are instantiated. This group of tuples are then asserted into the internal database of Prolog with the functor name *Idbname*. If successful, *Status* will return a positive value or zero indicating the number of tuples retrieved and asserted.

If the operation fails, *Status* is returned as "-1", and *Message* will be instantiated to an appropriate error messsage.

* * * * * * * * * *

dbassert(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate is used to insert one tuple into the external base table *Relname*. All *Attr$_n$* variables must be instantiated before this predicate may be executed. Successful completion is indicated by instantiation of the *Status* variable to "0". Failure is indicated by a *Status* value of "-1" and the instantiation of *Message* to an appropriate error message.

* * * * * * * * * *

dbretract(Relname(Attr$_1$,...,Attr$_n$),Status,Message).

This predicate is used to delete a group of tuples from the external base table *Relname* that satisfy the constraints placed on the attributes that are instantiated. All *Attr$_n$'s* need not be instantiated. Success is indicated by a *Status* value of "0". Failure is indicated by a *Status* value of "-1" and the instantiation of *Message* to an appropriate error message.

* * * * * * * * * *

dbcreate(Relname,Attrlist,Status,Message).

This predicate creates the external base table *Relname* from the information given in *Attrlist*. The format of *Attrlist* is

$$[attr(Name_1,Type_1)/.../attr(Name_n,Type_n)]$$

where *Name* specifies the name of the attribute and *Type* specifies the type of attribute. In this implementation only two values for *Type* are supported, "int" and "char". If the request is successful, and the corresponding base table is created by the DBMS, a value of "0" is returned for *Status* and *Message* is not instantiated. Otherwise, a value of "-1" is returned for *Status* and the appropriate error message is returned in *Message*. In addition, the corresponding *baseRel* fact is asserted into the workspace of Prolog.

* * * * * * * * * *

dbdrop(Relname,Status,Message).

This predicate sends a request to the DBMS to remove the external base table *Relname*. If successful, *Status* is returned the value "0" and *Message* is not instantiated. Otherwise *Status* is instantiated to "-1", and *Message* is instantiated to an appropriate error message. *dbdrop* also retracts the *baseRel* fact corresponding to the external base table *Relname*.

3.3 Application programs

Application program statements provide the link between dbProlog and an application end-user. The application programmer uses the dbProlog predicates to provide a simpler, application specific interface to the user. Through proper programming, an external DBMS retrieval may appear to the end-user as a retrieval from Prolog's internal database. The application programmer may wish to do all error handling to present the end-user with a simpler interface. In each of the following sample application program statements, a dbProlog predicate is called that effects a DBMS operation. The statements following the dbProlog predicate check the returned status value and display an error message if the status indicates that an error condition is present.

```
parent(ParentName,ChildName) :-
     dbretrieve(parent(ParentName,ChildName),Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).

child(Name,Age,Sex) :-
     dbretrieve(child(Name,Age,Sex),Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *parent* and *child* predicates call the *dbretrieve* predicate to retrieve tuples from the corresponding base table. The arguments may or may not be instantiated to values to specify retrieval criteria.

```
insert(TupleDesc) :-
     dbassert(TupleDesc,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *insert* predicate defined here accepts one argument which is a description of a tuple to be inserted into a base table (such as

parent(frank,john)). This description is passed to the *dbassert* predicate which requires all arguments to be instantiated so that the tuple can be inserted into the base table.

```
loadtuples(TupleDesc,IdbRel) :-
     dbload(TupleDesc,IdbRel,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *loadtuples* predicate accepts two arguments. The first is a tuple description which describes the criteria for retrieval from the DBMS (such as *parent(lorr,X)*). The second argument is the functor name which the tuples will be asserted under in Prolog's workspace. It may or may not be the same as the base table name. The arguments are passed to the *dbload* predicate.

```
delete(TupleDesc) :-
     dbretract(TupleDesc,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The delete predicate defined here accepts one argument which is a description of a tuple to be deleted from a base table (such as *parent(roger,joe)*). This description is passed to the *dbretract* predicate.

```
create(Relname,Attrlist) :-
     dbcreate(Relname,Attrlist,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *create* predicate implemented here accepts two arguments that define a base table and corresponding record structure. The first argument is the name of the base table that is to be created and the second argument is the description of the record format, which is a list of *attr* structures such as:

```
[attr(name,char),attr(age,int),attr(sex,char)]
```

(see *dbcreate* description in Section 5.1). The arguments are passed to the *dbcreate* predicate.

```
drop(Relname) :-
     dbdrop(Relname,Stat,Msg),
     ((Stat < 0,write(Stat),nl,printstring(Msg));true).
```

The *drop* predicate takes one argument which is simply the name of the base table to be removed from the external database. This name is passed to the *dbdrop* predicate.

3.4 Startup/termination of dbProlog and stand alone DBMS

The dbProlog system is initiated by entering the following goal for Prolog to solve:

```
dbProlog(start).
```

The solving of this goal by Prolog causes the following operations to occur:

* The *drvr* process is initiated, and all message queues are created.
* The DBMS process begins execution in "process mode".
* The value for "maximum number of tuple buffers" is asserted into Prolog's workspace.
* The value for "current tuple buffer" is asserted into Prolog's workspace.

While the dbProlog system is executing, the DBMS may be run stand alone by entering the Prolog goal:

```
dbProlog(dbms).
```

The user will then be given the interactive DBMS prompt.

Interface execution is terminated by entering the Prolog goal:

dbProlog(stop).

Solving this goal causes the following to happen:

* The DBMS process terminates.
* The *drvr* process terminates.
* All Prolog workspace assertions regarding buffer management are removed.
* All dbProlog access predicates are removed from Prolog's workspace.
* All *baseRel* facts are removed from Prolog's workspace.

3.5 Sample dbProlog conversation

This section provides a sample dbProlog conversation that uses the application program statements and data dictionary definitions that were given in the previous section, or are given with the example.

---

1) Starting dbProlog:

```
;int file contains pre-loaded dbProlog code

$ prolog int

C-Prolog version 1.4
[ Restoring file int ]

yes

; The application program code file, dbapp, is loaded

| ?- [dbapp].
dbapp consulted 2748 bytes 0.783333 sec.

yes

; When dbProlog(start) is executed, the two files
; dbProlog and tools are reconsulted in case the
; interface is being restarted during a Prolog session

| ?- dbProlog(start).
dbProlog reconsulted 0 bytes 4.06667 sec.
tools reconsulted 0 bytes 0.850002 sec.

yes
```

---

2) Running DBMS standalone:

```
| ?- dbProlog(dbms).

) DISPLAY PARENT SORTED

RELATION : PARENT

PNAME )) DEE
CNAME )) JOHN

PNAME )) FRANK
CNAME )) DEE

PNAME )) LORR
CNAME )) DEE

((Press RETURN key to continue))


PNAME )) STEVE
CNAME )) JOHN

PNAME )) THERESA
CNAME )) FRANK

) QUIT

no
```

---

3) Transparent retrieval:

```
| ?- parent(X,Y).

X = theresa
Y = frank ;

X = frank
Y = dee ;

X = lorr
Y = dee ;

X = dee
Y = john ;
```

```
X = steve
Y = john ;

no
```

---

4) Transparent join of tables:

```
; the predicate "grand(X,Z)" is defined in the Prolog
; workspace as:
; grand(X,Z) :- parent(X,Y),parent(Y,Z).


| ?- grand(X,Y).

X = theresa
Y = dee ;

X = frank
Y = john ;

X = lorr
Y = john ;

no
```

---

5) Creating view of base table

```
; the predicate "minor(Name,Age)" is defined in the Prolog
; workspace as:
; minor(Name,Age) :- child(Name,Age,Sex),Age < 21.


| ?- minor(Name,Age).

Name = john
Age = 16 ;

Name = maltic
Age = 9

yes
```

---

6) Recursive DBMS calls:

```
; the predicate "ancestor(X,Y)" is defined in the Prolog
; workspace by the following two rules:
; ancestor(X,Y) :- parent(X,Y).
; ancestor(X,Z) :- parent(X,Y),ancestor(Y,Z).


| ?- ancestor(X,Y).

X = theresa
Y = frank ;

X = frank
Y = dee ;

X = lorr
Y = dee ;

X = dee
Y = john ;

X = steve
Y = john ;

X = theresa
Y = dee ;

X = theresa
Y = john ;

X = frank
Y = john ;

X = lorr
Y = john ;

no
```

---

7) Use of Prolog built-in function *bagof*:

```
| ?- bagof(Name,Age^child(Name,Age,Sex),L).

Name = _0
Age = _1
Sex = f
L =  [dee,|[tumby,|[snowbear,|[chrissy,|[sue,|[beta,|[carrie,|
[debb,|[fawna,|[gertie,|[kelly,|[lana,|[maura,|[ursula,|[vanna,|
[xavier,|[zenity]]]]]]]]]]]]]] ;

Name = _0
Age = _1
Sex = m
L =  [john,|[frank,|[steve,|[maltic,|[sebastian,|[mark,|[brian,|
[allan,|[efa,|[harold,|[isa,|[jocko,|[norm,|[oscar,|[perry,|
[quincy,|[roger,|[sam,|[tubs,|[wally,|[yoda]]]]]]]]]]]]]]] ;

no
```

---

8) Loading of base table tuples into internal database of Prolog:

```
| ?- loadtuples(male(X),male_idb).

X = _0

yes
| ?- male_idb(X).

X = frank ;

X = john

yes
```

---

9) Insert and delete of external base table tuples:

```
| ?- insert(child(bunky,11,m)).

yes
| ?- child(bunky,Age,_).

Age = 11 ;

no
| ?- delete(male(roger)).

yes
| ?- male(roger).

no
```

---

10) Create and delete of external base tables:

```
| ?- create(car,[attr(make,char),attr(year,int),attr(color,char)]).

yes
| ?- insert(car(chevy,1987,grey)).

yes
| ?- car(Make,Year,Color).

Make = chevy
Year = 1987
Color = grey ;

no
| ?- drop(car).

yes
| ?- insert(car(chevy,1987,grey)).
-1
Invalid Request
Make = _0
Year = _1
Color = _2
yes
```

---

11) Error handling:

```
; error detected by dbProlog

| ?- create(newtab,[attr(field,integer)]).
-1
Invalid attribute/type list
yes


; error detected by DBMS

| ?- insert(child(bob,m,12)).
-1
Invalid INT field
yes
```

---

12) Termination of dbProlog processing:

```
| ?- dbProlog(stop).

yes
| ?- halt.

[ Prolog execution halted ]
```

4. Error and Warning Messages

Most of the error messages that appear in dbProlog are generated by the DBMS component. These messages are listed in the document "DBMS User manual" [Oag188b]. The following messages are those that are generated exclusively by the Prolog component.

---

1) "Invalid Request"

This error message is used in cases where a specific error cannot be identified. Two of the most common cases are:

a) (dbretrieve, dbload, dbassert, dbretract) The basetable is not defined in dbProlog's data dictionary, or an inconsistency has been found between the definition, and the record description entered.

b) (dbassert) One of the fields is not instantiated to a value, and therefore cannot be inserted into the database

---

2) "Basetable already exists in dbProlog"

(dbcreate) The specified base table already is defined in

the data dictionary of dbProlog.

---

3) "Invalid attribute/type list"

(dbcreate) The list of attributes and their types is invalid.

---

4) "Basetable does not exist - no delete performed"

(dbdrop) The basetable is not defined in the data dictionary of dbProlog, and therefore not drop command is sent to the DBMS.

References

[ClMe84] Clocksin, W.F.; Mellish, C.S. Programming in Prolog, Springer-Verlag, 1984.

[Oagl88a] Oagley, Diane M. dbProlog: A Prolog/Relational Database Interface. Master's Thesis, Rochester Institute of Technology, Rochester, NY, June 1988.

[Oagl88b] Oagley, Diane M. DBMS User Manual. Rochester Institute of Technology, Rochester, NY, June 1988.

[Pere85] Pereira, Fernando; Warren, David; Bowen, David; Pereira, Luis. C-Prolog User's Manual, Version 1.4. (Editor: Pereira, Fernando), April 24, 1985.

DBMS

User Manual

by

Diane M. Oagley

May 28, 1988

Table of Contents

1. Preface

This manual describes the operation of a relational database management system that was implemented as a project in the course ICSS-739 (Database Implementation). It runs under the Unix operating system, and has been tested under AT&T Unix[1] System V, version 3.51. This guide assumes that the user is familiar with relational databases and the execution of programs under the Unix operating system. The user may consult the references for additional information about these topics.

Relational database theory provides a simple way of organizing data. In a relational database, data is arranged in tabular form. Each table consists of a group of records that share the same group of attribute definitions. Typical relational DBMS operations include creation and deletion of tables, views, and index files, insertion of data into tables, deletion of data from tables and retrieval of data from and about tables.

---

[1]Unix is a trademark of AT&T

2. Overview

This relational DBMS consists of a group of commands to create and maintain a simple relational database consisting of a set of base tables. The standard mode of operation is interactive - as a conversation between the user and the DBMS. However, the DBMS may also be run in a "process" mode where input and output is directed through the Unix interprocess message facility (with message key values of 2 and 3 for input and output message queues respectfully), prompts are suppressed, and every DBMS response returns an initial "status" value, indicating success or failure of the operation.

The DBMS requires and maintains three system tables which are contained in the files SYS_TABLES, SYS_ATTRIBUTES, and SYS_TABLES_IND. These contain information about the base tables defined by the user. If these files do not exist when the DBMS is initiated, they will be created.

Before a base table is referenced, it must be defined to the DBMS. Its characteristics are stored in the system tables. By defining a base table, its associated record structure is defined. A base table record structure may have up to six (6) attributes defined and they may be of type character or integer (see DEFINE command). When a base table is defined, one of the attributes must be identified as a primary key. The value of this field must be unique for every record entered into the table. Record sorting

is done on the primary key attribute. The following size restrictions are placed on the field lengths and names in the DBMS:

Base table name - 10 characters

Attribute name - 10 characters

Character field - 35 characters

Integer field - 10 digits

Number of attributes/base table - 6

The base table file is created at the time the first record is inserted. An accompanying index file is also created which contains only key values and addresses into the base table. The index file is sorted on the primary key value, and is used for a sorted display of data in the base table. All tables are stored as Unix files with names *TABLENAME.R* (base table file) *TABLENAME.I* (index file).

This implementation does not support views (virtual tables). Also, there is no "modify" functionality. If the user wishes to change the values of one or more fields in a record, the record must be deleted and re-added with the proper values.

Conventions used in this manual:

<cr> = carriage return

RELOP = relational operator (=,!=,<,>)

{ } = optional input

[|] = choose one of the values

Regular font = reserved word of command language

Italics font = name or value designated by the user

... = optional repeating item

Some of the commands used in the DBMS allow embedded carriage returns, so that the user input may extend over more than one line. Such commands require two carriage returns in succession to signify the end of command. Carriage returns will be indicated in the command syntax where necessary. The user must also be aware that all command and input are entered in uppercase.

3. Syntax and Operation

3.1 Startup

The database management system is initiated in interactive mode by the command

dbms

or

dbms -i

at the Unix shell prompt. Process mode is initiated by the command:

dbms -p

If the system is initiated in interactive mode, the DBMS prompt will be displayed (">").

3.2 Defining Base Tables

The DEFINE command is used to establish a base table with its corresponding record structure. One and only one of the fields must be identified as the primary key by including the word "PRIMARY."

```
DEFINE TABLENAME
FIELD FIELDNAME1 TYPE [CHAR|INT] {PRIMARY}
{ FIELD FIELDNAMEN TYPE [CHAR|INT] {PRIMARY} <cr>
... }
<cr>
```

3.3 Describing Base Tables

The DESCRIBE command returns the description of a base table that has been previously defined to the system. The attribute names are displayed by the DBMS, with their type (Character or Integer) and the primary key field is identified. The syntax is:

```
DESCRIBE TABLENAME <cr>
```

3.4 Inserting Records into Base Table

The INSERT command is used to add records to a base table. The insert command appears as follows:

```
INSERT INTO TABLENAME <cr>
```

If the tablename is valid, the system will respond by prompting the user for each field value in the base table record. The user responds by entering the field value, followed by a carriage return, as shown in the following example:

```
> INSERT INTO CHILD <cr>
**PRIMARY KEY** NAME (CHAR) >> MALTIC <cr>
AGE (INT) >> 9 <cr>
SEX (CHAR) >> M <cr>
```

The user's entries will be checked for validity, such as proper integer field format and duplicate primary key. Error messages will be displayed if any condition is violated. Field values may not have embedded white space (such as "JOHN DOE"). Whitespace signifies the end of a field value, and all characters entered after the whitespace are ignored.

3.5 Retrieving Records from Base Table

There are several formats of the DISPLAY command. They will be given below with a description of their use. Each of the DISPLAY commands are audited for existence of base table and specified attributes. Retrieved tuples are displayed with their attribute names.

---

1)

DISPLAY *TABLENAME* <cr>

This command will retrieve and display all records in a base table, in the order that they were added to the table.

---

2)

DISPLAY *TABLENAME* SORTED <cr>

This command will cause all records to be retrieved, and displayed in the sorted order of the primary key field.

---

3)

DISPLAY *TABLENAME* WHERE *FIELDNAME RELOP VALUE* <cr>
<cr>

This command will cause retrieval of only the tuples that satisfy the condition specified in the clause *"FIELDNAME RELOP VALUE."* The following is an example of usage:

> *DISPLAY CHILD WHERE AGE < 12 <cr>*
> *<cr>*

---

4)

DISPLAY *TABLENAME* WHERE *FIELDNAME1 RELOP1 VALUE1* AND
FIELDNAME2 RELOP2 VALUE2 <cr>
<cr>

This command causes retrieval of all tuples that meet the conjunction of the condition clauses. Both conditions must be true in each tuple retrieved. An example of usage follows:

```
> DISPLAY CHILD WHERE AGE = 12 AND
> SEX = M <cr>
> <cr>
```

---

5)

DISPLAY *TABLENAME* WHERE *FIELDNAME1 RELOP1 VALUE1* OR
FIELDNAME2 RELOP2 VALUE2 <cr>
<cr>

This command is used to retrieve tuples that satisfy at least one of the specified conditions. An example follows:

```
> DISPLAY CHILD WHERE NAME = JOHN
> OR NAME = FRANK <cr>
> <cr>
```

---

6)

DISPLAY *TABLENAME* WHERE *FIELDNAME1 = VALUE1*
AND FIELDNAME2 = VALUE2
(AND FIELDNAMEN = VALUEN ...) <cr>
<cr>

This form of the DISPLAY command only permits equality conditions to be specified, however an arbitrary number of equality conditions may be specified. The number of conditions is limited by the number of attributes in the table. All tuples that satisfy all conditions will be retrieved. An example follows:

```
> DISPLAY CHILD WHERE NAME = STEVE AND
> AGE = 35 AND
> SEX = M <cr>
> <cr>
```

---

7)

DISPLAY *TABLENAME FIELDNAME1 (... FIELDNAMEN)* <cr>
<cr>

This DISPLAY command produces a projection of the base table. All tuples are retrieved, but only the specified fields of each tuple are displayed in the order given in the command. An example follows:

```
> DISPLAY CHILD NAME AGE <cr>
> <cr>
```

---

8)

DISPLAY *TABLENAME1 TABLENAME* 2 WHERE
FIELDNAME1 = FIELDNAME2 <cr>
<cr>

This command produces a "join" of the two tables, on the fieldnames specified. *FIELDNAME1* is the name a field in *TABLENAME1*, and *FIELDNAME2* is the name of a field in *TABLENAME2* (*TABLENAME1* and *TABLENAME2* need not be distinct). A "join" is the result of taking the record-by-record Cartesian product of the two base tables, and eliminating all records where the values of the specified fieldnames in each table are not the same. The

remaining records are those where the fieldnames have the same value in both tables. The resulting records are displayed as a "natural join", where the joined field is only displayed once per record. The records are displayed with their appropriate base table name and attribute name. An example of the command, and sample output follows:

```
> DISPLAY PARENT CHILD WHERE CNAME = NAME
> <cr>

PARENT.PNAME = FRANK
PARENT.CNAME = JOHN
CHILD.AGE = 16
CHILD.SEX = M
```

3.6 Deleting Records from Base Tables and/or Deleting Entire Base Tables

There are several formats of the DELETE command. The DELETE commands are audited for appropriate base table and attribute names.

---

1)

DELETE *TABLENAME* <cr>

This command removes all tuples from the specified base table. However, the base table name still exists in the system tables and may be referenced in later commands.

---

2)

DELETE *TABLENAME* ALL <cr>

In addition to removing all base table tuples, this command also removes the base table name and definition from the system tables.

---

3)

DELETE *TABLENAME* WHERE *FIELDNAME RELOP VALUE* <cr>

This command deletes all tuples of the base table that meet the condition specified in the *"FIELDNAME RELOP VALUE"* clause. An example:

> DELETE CHILD WHERE AGE > 102 <cr>

---

4)

DELETE *TABLENAME* WHERE *FIELDNAME1 = VALUE1*
AND FIELDNAME2 = VALUE2
(AND FIELDNAMEN = VALUEN ...) <cr>

This command deletes all records that meet all the equality conditions specified. An example follows:

> *DELETE CHILD WHERE NAME = ROGER AND AGE = 91 <cr>*

3.7 Termination of System

The QUIT command stops all DBMS processing and returns the user to the Unix shell prompt. All system tables, base tables, and index files are saved for later access.

> *QUIT* <cr>

4. Error and Warning Messages

4.1 Error messages

A list of DBMS error messages is given, with short explanations. Where applicable, the command name is given that the error message applies to.

---

1) "Input line too long -- try again"

Entered command overflows input buffer. Check command syntax.

---

2) "Invalid command syntax -- try again"

A general error was found in the entered command. Check command syntax.

---

3) "Invalid command -- No primary key"

(DEFINE) No fieldname was identified as the primary key of the basetable.

---

4) "Invalid command -- PRIMARY specified more than once"

(DEFINE) Parser found the word PRIMARY used to describe more than one field in the base table definition.

---

5) "Cannot create system files"

(System initiation) Fatal condition. System cannot run without system tables. Check Unix file system.

---

6) "Relation name greater than 10 characters"

(DEFINE) Check that base table name is 10 characters long or less.

---

7) "Field name greater than 10 characters"

(DEFINE) Check that all field names are 10 characters long or less.

---

8) "Duplicate field name defined for relation"

(DEFINE) A specific field name was found more than once in the base table definition.

---

9) "Too many attribute (>6) defined"

(DEFINE) Base table definition includes more than six attributes.

---

10) "Relation already exists"

(DEFINE) User attempted to define a base table using a name

that already has been defined, and exists in the system tables.

---

11) "Relation does not exist"

(INSERT, DESCRIBE, DISPLAY, DELETE) User attempted operation against base table that has not been defined to the system.

---

12) "Invalid INT field"

(INSERT, DISPLAY, DELETE) User entered field value that is not consistent with integer format.

---

13) "Invalid CHAR field"

(INSERT, DISPLAY, DELETE) User entered field value that is not consistent with character format.

---

14) "Record with this primary key already exists"

(INSERT) User attempted to enter record with primary key value equal to the primary key of an existing record in the base table.

---

15) "Specified attribute does not exist in relation"

(DISPLAY, DELETE) An attribute name was specified that does not exist in the base table.

---

16) "USAGE: 'dbms', 'dbms -i', or 'dbms -p'"

(System initiation) User entered invalid DBMS startup command.

---

17) "FATAL error -- must recover SYS files"

(System error) System files may have been corrupted. It is advisable to restore a backup copy of the system tables and database files.

4.2 Warning messages

Warning messages do not indicate errors, but conditions that the user should be aware of.

---

1) "Entry truncated at space"

(INSERT) Field value entered with embedded space. All characters after space will be ignored.

---

2) "Relation has no data"

(DISPLAY) There are no tuples to be displayed from the specified base table.

---

3) "There are no data files to delete"

(DELETE) Specified base table contains no records that can be deleted.

---

4) "There are no records that satisfy condition"

(DISPLAY, DELETE) Base table contains no records that satisfy condition specified in condition clause.

---

5) "There are no records that satisfy JOIN condition"

(DISPLAY) There are no records to display resulting from the specified join of the base tables.

References

[Codd70] Codd, E.F. *A Relational Model of Data for Large Shared Databanks.* In Communications of the ACM (Vol. 13, No. 6) 1970, 377-387.

[Date83] Date, C.J. An Introduction to Database Systems Vol. II, Addison-Wesley, 1983.

[Date86] Date, C.J. An Introduction to Database Systems Vol. I, Addison-Wesley, 1986.

[Ullm82] Ullman, J.D. Principles of Database Systems, Second Edition, Computer Science Press, 1982.

[Unix87] UNIX System V Programmers Guide, AT&T, Prentice Hall, Inc., 1987.

Appendix E

Prolog Code

The following section contains the Prolog code that makes up the Prolog extension. It includes definitions of all the six dbProlog predicates plus lower level functions required by dbProlog.

```
%************************************************************%
%                                                            %
% dbProlog - Prolog rules for interfacing standard           %
%            Prolog to a relational database via a           %
%            "DBDriver" program.                             %
%                                                            %
% Diane M. Oagley                                            %
% Master's Thesis                                            %
% 15 January 1988                                            %
%                                                            %
%************************************************************%
%                                                            %
% Version History                                            %
%                                                            %
% v.1    15 January 1988                                     %
% v.2    27 January 1988 (output written to "todb")          %
% v.3    12 February 1988 (send table arity to driver)       %
% v.4    03 March 1988 (re-instantiation of retrieve)        %
% v.5    16 March 1988 (multiple reply queues-recursion)     %
%                                                            %
%************************************************************%
%                                                            %
% Notes:                                                     %
%                                                            %
% 1.  The only functions of dbProlog that are "user          %
%     accessible" are:  dbretrieve, dbload, dbassert,        %
%     and dbretract.  All other routines (with the           %
%     exception of some "utility" routines such as           %
%     "printstring" and "writeCondList" are only to be       %
%     used internally to dbProlog.  These dbProlog           %
%     "internal" routines are preceded by "$".               %
% 2.  Any external database referred to by the dbProlog      %
%     user must have an entry residing in the Prolog         %
%     internal database in the form:                         %
%         baseRel(parent,2,[pname,cname]).                   %
% 3.  User must program any handling of Stat and Msg         %
%     returned from dbProlog call.                           %
%     Example:                                               %
%       parent(Pname,Cname) :-                               %
%          dbretrieve(parent(Pname,Cname),Stat,Msg),         %
%          ((Stat < 0,write(Stat),nl,printstring(Msg));      %
%           true).                                           %
% 4.  The "feed" program is used to transmit each            %
%     request from the file "todb" to the message queue      %
%     that is read by the dbDriver program.                  %
% 5.  The "eat" program transmits each record from the       %
%     dbDriver message queue to the file "topro".  This      %
%     file must only contain one record at a time in         %
%     case backtracking is stopped, so that the file         %
%     contents will not be mistaken for a command input.     %
% 6.  In any defined rule, only one clause may be an edb     %
%     reference.  This is an area for future enhancement.    %
%     Example:                                               %
%       father(Name) :- parent(Name,Cname),male(Name).       %
%     Where "parent" is an external database reference       %
%     and "male" is an internal database reference.          %
```

```
% 7.  The predicates dbretrieve, dbload, dbassert, and   %
%     dbretract will always succeed, therefore, the      %
%     value of "Stat" must be checked by the application %
%     program.                                           %
%                                                        %
%********************************************************%



%********************************************************%
% dbretrieve -                                           %
%     Converts TupleDesc into a standard format for a    %
%     retrieval request to the DBDriver program.         %
%     TupleDesc is in the form of a prolog predicate     %
%     such as "parent(john,Y)" which would be converted  %
%     into a request of a retrieval of a tuple from the  %
%     external database relation "parent" where the      %
%     value of the first attribute is "john".            %
%     Dbretrieve1 waits for the reply which is in the    %
%     form of a status value "Stat" and possibly a       %
%     message "Msg".  Dbretrieve2 gets each dbms record  %
%     and performs instantiation of any uninstantiated   %
%     variables in "TupleDesc".                          %
%********************************************************%

dbretrieve(TupleDesc,Stat,Msg) :-
     dbretrieve1(TupleDesc,Stat,Msg,QueueNum,StatCall,TupCall),
     (Stat < 0;
      dbretrieve2(TupleDesc,QueueNum,TupCall)).

dbretrieve1(TupleDesc,Stat,Msg,QueueNum,StatCall,TupCall) :-
     $checkReq(retrieve,TupleDesc,RelInfo,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,TupCall),
       $constructReq(TupleDesc,RelInfo,RelName,CondList,TupleList,TabArity),
       $sendReq(retrieve,RelName,CondList,TabArity),
       $waitReply(Stat,Msg,StatCall));
      true),
     !.

dbretrieve2(TupleDesc,QueueNum,TupCall) :-
       TupleDesc =.. [TupleHead|TupleList],
       repeat,
         $getTuple(TupleList,Eof,TupCall),
         ((Eof == true,$freeQueue(QueueNum),!,fail);
           true).


%********************************************************%
% dbload -                                               %
%     Converts TupleDesc into a standard format for a    %
%     retrieval request to the DBDriver program.         %
%     TupleDesc is of the same format as in the          %
%     dbretrieve predicate.  After retrieval, dbload     %
%     asserts each tuple retrieved from the external     %
%     database into the internal prolog database under   %
```

```
%     the functor name given by the user (IdbRel).        %
%     dbload also instantiates the Stat and optionally,   %
%     the Msg variables to returned values, and instan-   %
%     tiates the variables in TupleDesc to the values     %
%     in the last tuple asserted into the internal        %
%     database.                                           %
%*********************************************************%

dbload(TupleDesc,IdbRel,Stat,Msg) :-
     $checkReq(load,TupleDesc,RelInfo,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,TupCall),
       $constructReq(TupleDesc,RelInfo,RelName,CondList,TupleList,TabArity),
       $sendReq(load,RelName,CondList,TabArity),
       $waitReply(Stat,Msg,StatCall));
      true),
     ((GoodReq,Stat > 0,
       $assertTups(TupleList,IdbRel,TupCall));
      true),
     !.

%*********************************************************%
% dbassert -                                             %
%     Converts TupleDesc into a standard format for an    %
%     insert request to the DBDriver program.  All        %
%     atoms of TupleDesc must be instantiated, or an      %
%     error condition will occur.  dbassert will attempt  %
%     the insertion on one tuple into the external        %
%     database relation, as expressed in TupleDesc.       %
%     dbassert instantiates the Stat and Msg variables    %
%     to the values returned from the DBDriver program.   %
%*********************************************************%

dbassert(TupleDesc,Stat,Msg) :-
     $checkReq(insert,TupleDesc,RelInfo,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,TupCall),
       $constructReq(TupleDesc,RelInfo,RelName,CondList,TupleList,TabArity),
       $sendReq(insert,RelName,CondList,TabArity),
       $waitReply(Stat,Msg,StatCall));
      true),
     !.

%*********************************************************%
% dbretract -                                            %
%     Converts TupleDesc into a standard format for a     %
%     delete request to the DBDriver program.  All        %
%     atoms of TupleDesc need not be instantiated, to     %
%     specify conditions for deletion.  dbretract causes  %
%     the deletion of tuples in the external database     %
%     relation as expressed in TupleDesc.  dbretract      %
%     instantiates the Stat and Msg variables to the      %
%     values returned from the DBDriver program.          %
%*********************************************************%

dbretract(TupleDesc,Stat,Msg) :-
```

```
     $checkReq(delete,TupleDesc,RelInfo,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,TupCall),
       $constructReq(TupleDesc,RelInfo,RelName,CondList,TupleList,TabArity),
       $sendReq(delete,RelName,CondList,TabArity),
       $waitReply(Stat,Msg,StatCall));
      true),
     !.

%*********************************************************%
% dbcreate -                                              %
%     Constructs information required to create a new     %
%     edb table in the dbms from the user specified       %
%     information such as name of the table, list of      %
%     attributes and type (char or int).  dbcreate        %
%     instantiates the Stat and Msg variables to the      %
%     values returned from the DBDriver program.          %
%     The idb description (RelInfo) of the edb table      %
%     is asserted into the Prolog database.               %
%*********************************************************%
dbcreate(TabName,AttrList,Stat,Msg) :-
     $checkCreate(TabName,AttrList,NameList,TabArity,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,_),
       $sendReq(create,TabName,AttrList,TabArity),
       $waitReply(Stat,Msg,StatCall),
       ((Stat == 0,assert(baseRel(TabName,TabArity,NameList)));true));
      true),
     !.

%*********************************************************%
% dbdrop -                                                %
%     Constructs information required to drop an          %
%     edb table that currently is defined in the dbms.    %
%     The base table name is required to send a command   %
%     to the dbms.  dbdrop instantiates the Stat and      %
%     Msg variables to the values returned from the       %
%     DBDriver program.  dbdrop also retracts the idb     %
%     description (RelInfo) of the edb base table.        %
%*********************************************************%
dbdrop(TabName,Stat,Msg) :-
     $checkDrop(TabName,GoodReq,Stat,Msg),
     ((GoodReq,
       $getQueue(QueueNum,StatCall,_),
       $sendReq(drop,TabName,_,_),
       $waitReply(Stat,Msg,StatCall),
       retract(baseRel(TabName,_,_)));
      true),
     !.


%*********************************************************%
% buildRelInfo -                                          %
%     Converts TupleDesc into the standard format of      %
%     the baseRel predicate for validation in the         %
%     prolog idb.                                         %
```

```
%*********************************************************%
$buildRelInfo(TupleDesc,baseRel(TabName,TabArity,TabAttrList)) :-
     functor(TupleDesc,TabName,TabArity),
     !.

%*********************************************************%
% getRelInfo -                                           %
%     Checks for the existence of a baseRel entry in the %
%     prolog idb that corresponds to the TupleDesc that  %
%     was entered by the user.                           %
%*********************************************************%
$getRelInfo(baseRel(TabName,TabArity,TabAttrList),GoodRel) :-
     ((baseRel(TabName,TabArity,TabAttrList),
     GoodRel = true);
     (GoodRel = false)).

%*********************************************************%
% checkReq -                                             %
%     Checks that the request entered by the user is a   %
%     valid request.  If invalid, sets Stat and Msg      %
%     appropriately.                                     %
%*********************************************************%
$checkReq(insert,TupleDesc,RelInfo,GoodReq,Stat,Msg) :- !,
     $buildRelInfo(TupleDesc,RelInfo),
     $getRelInfo(RelInfo,GoodRel),
     ((GoodRel,
       $checkNonvar(TupleDesc),
       GoodReq = true);
      (GoodReq = false,
       Stat is -1,
       Msg = "Invalid Request")).

$checkReq(_,TupleDesc,RelInfo,GoodReq,Stat,Msg) :- !,
     $buildRelInfo(TupleDesc,RelInfo),
     $getRelInfo(RelInfo,GoodReq),
     ((not GoodReq,
       Stat is -1,
       Msg = "Invalid Request");
      true).

%*********************************************************%
% checkCreate -                                          %
%     Checks that the table create request entered is a  %
%     valid request.  If invalid, sets Stat and Msg      %
%     appropriately.                                     %
%*********************************************************%
$checkCreate(TabName,AttrList,NameList,TabArity,GoodReq,Stat,Msg) :-
     $getRelInfo(baseRel(TabName,_,_),RelExists),
     ((RelExists,
       Stat is -1,
       GoodReq = false,
       Msg = "Basetable already exists in dbProlog");
       ($constructNList(AttrList,NameList,TabArity,GoodReq),
        ((not GoodReq,
          Stat is -1,
          Msg = "Invalid attribute/type list");
```

```
          true))).

%*********************************************************%
% checkDrop -                                             %
%     Checks that the table drop request entered is a     %
%     valid request.  If invalid, sets Stat and Msg       %
%     appropriately.                                      %
%*********************************************************%
$checkDrop(TabName,GoodReq,Stat,Msg) :-
     $getRelInfo(baseRel(TabName,_,_),GoodReq),
     ((not GoodReq,
       Stat is -1,
       Msg = "Basetable does not exist - no delete performed");
      true).

%*********************************************************%
% constructReq -                                          %
%     Gathers all the information required to make a      %
%     request to the DBDriver program (and edb).          %
%*********************************************************%
$constructReq(TupleDesc,baseRel(TabName,TabArity,TabAttrList),
     TabName,CondList,TupleList,TabArity) :-
     TupleDesc =.. [TupleHead|TupleList],
     $nextAttr(TabAttrList,TupleList,[],NewList),
     rev(NewList,CondList),
     !.

%*********************************************************%
% sendReq -                                               %
%     Writes edb request information to message file      %
%     used as input to the DBDriver program.              %
%*********************************************************%
$sendReq(drop,RelName,_,_) :- !,
     tell(todb),
     write(2),nl,
     write(drop),nl,
     write(RelName),nl,
     told,
     system("feed").

$sendReq(create,RelName,TypeList,TabArity) :- !,
     tell(todb),
     listlen(TypeList,Len),
     Numrec is (Len * 2) + 3,
     write(Numrec),nl,
     write(create),nl,
     write(RelName),nl,
     write(TabArity),nl,
     writeTypeList(TypeList),
     told,
     system("feed").

$sendReq(quit,_,_,_) :- !,
     tell(todb),
     write(1),nl,
     write(quit),nl,
```

```
      told,
      system("feed").

$sendReq(Cmd,RelName,CondList,TabArity) :- !,
      tell(todb),
      listlen(CondList,Len),
      Numrec is (Len * 2) + 3,
      write(Numrec),nl,
      write(Cmd),nl,
      write(RelName),nl,
      write(TabArity),nl,
      writeCondList(CondList),
      told,
      system("feed").

%************************************************************%
% waitReply -                                                %
%     Reads Stat and Msg that is returned from the           %
%     DBDriver program, and asserts them into the idb        %
%************************************************************%
$waitReply(Stat,Msg,StatCall) :-
      system(StatCall),
      see(topro),
      read(Stat),
      read(Msg),
      seen.

%************************************************************%
% getTuple -                                                 %
%     Reads a list that represents a tuple returned from     %
%     the edb via the DBDriver program, from the message     %
%     file.  The TupleList is then instantiated with the     %
%     variables.                                             %
%************************************************************%
$getTuple(TupleList,Eof,TupCall) :-
      system(TupCall),
      see(topro),
      read(FromdbList),
      seen,
      ((FromdbList == "eof",
        Eof = true);
       (Eof = false,
        $instList(FromdbList,TupleList))),
      !.

%************************************************************%
% instList -                                                 %
%     Instantiates the variables in a dbProlog tuple         %
%     the actual attribute values of an edb tuple.           %
%************************************************************%
$instList([],[]) :- !.

$instList([DbListHead|DbList],[TListHead|TList]) :-
      var(TListHead),
      TListHead = DbListHead,
      $instList(DbList,TList),!.
```

```
$instList([DbListHead|DbList],[TListHead|TList]) :-
     nonvar(TListHead),
     $instList(DbList,TList),!.

%************************************************************%
% nextattr -                                                 %
%     Constructs a list of conditions that is specified      %
%     by the instantiated elements of TupleDesc.  This       %
%     list will become the CondList that will be send to     %
%     DBDriver program.                                      %
%************************************************************%
$nextAttr([],_,OldList,NewList) :-
     NewList = OldList, !.

$nextAttr([Attr|RestOfAttr],[Elem|RestOfElem],OldList,NewList) :-
     nonvar(Elem),
     NextList = [cond(Attr,Elem)|OldList],
     $nextAttr(RestOfAttr,RestOfElem,NextList,NewList).

$nextAttr([Attr|RestOfAttr],[Elem|RestOfElem],OldList,NewList) :-
     var(Elem),
     $nextAttr(RestOfAttr,RestOfElem,OldList,NewList).

%************************************************************%
% constructNList -                                           %
%     Constructs a list of attribute names (NameList)        %
%     given a list of attribute name/type pairs in the       %
%     form "attr(Name,Type)" in AttrList.  It also           %
%     returns the number of elements in the list and         %
%     whether the list is constructed correctly.             %
%************************************************************%
$constructNList(AttrList,NameList,Arity,GoodList) :-
     listlen(AttrList,Arity),!,
     ((Arity < 1,
       GoodList = false);
      (($buildNameList(AttrList,[],RevList),
        rev(RevList,NameList),
        GoodList = true);
        GoodList = false)).

%************************************************************%
% buildNameList -                                            %
%     Separates the elements of AttrNameList in the form     %
%     "attr(Name,Type)" to produce a list of attribute       %
%     names only.                                            %
%************************************************************%
$buildNameList([],NameList,NameList) :- !.

$buildNameList([attr(Name,Type)|AttrList],OldNList,NameList) :-
     (Type == int;Type == char),
     append([Name],OldNList,NewNList),
     $buildNameList(AttrList,NewNList,NameList).

%************************************************************%
% assertTups -                                               %
```

```
%     Receives the tuples that are returned to prolog    %
%     from DBDriver as a result of the dbload predicate, %
%     and asserts these tuples into the idb, with the    %
%     functor name expressed in IdbRel.                  %
%*********************************************************%
$assertTups(TupleList,IdbRel,TupCall) :-
     repeat,
       $getTuple(TupleList,Eof,TupCall),
       (Eof == true;
        ($idbAssert(IdbRel,TupleList),
         fail)).

%*********************************************************%
% getQueue -                                             %
%     Obtains the next available message queue number    %
%     from "circular" pool of message queue numbers.     %
%     Returns queue number in string form, to be used    %
%     to send to driver, and in the form for calling     %
%     the "eat" program.  Updates queue value to the     %
%     next available, and re-asserts it.                 %
%*********************************************************%
$getQueue(QueueNum,StatCall,TupCall) :-
     retract(currQueue(QueueNum)),
     name(QueueNum,QueueStr),
     append("eatstat ",QueueStr,StatCall),
     append("eattup ",QueueStr,TupCall),
     maxQueue(Qmax),
     ((QueueNum == Qmax,assert(currQueue(1)));
      (NewQnum is QueueNum + 1,
       assert(currQueue(NewQnum)))).

%*********************************************************%
% freeQueue -                                            %
%     Makes queue specified by QueueNum available for    %
%     use, after retrieve call fails.                    %
%*********************************************************%
$freeQueue(QueueNum) :-
     retract(currQueue(_)),
     assert(currQueue(QueueNum)).

%*********************************************************%
% checkNonvar -                                          %
%     Checks that each attribute specified in TupleDesc  %
%     is not a variable.  Used only for dbassert calls.  %
%*********************************************************%
$checkNonvar(TupleDesc) :-
     TupleDesc =.. TupleList,!,
     $listNonvar(TupleList).

%*********************************************************%
% listNonvar -                                           %
%     Checks that each member of list is not a variable. %
%*********************************************************%
$listNonvar([]) :- !.

$listNonvar([Head|Rest]) :- !,
```

```
     nonvar(Head),
     $listNonvar(Rest).

%************************************************************%
% IdbAssert -                                                %
%     Asserts a tuple into the idb of prolog.                %
%************************************************************%
$idbAssert(Functor,TupleList) :-
     DbTuple =.. [Functor|TupleList],
     assert(DbTuple).

%************************************************************%
% writeCondList -                                            %
%     Writes attribute name and value pairs, one per         %
%     line.                                                  %
%************************************************************%
writeCondList([]) :- !.

writeCondList([cond(First,Second)|RestOfList]) :-
     write(First),nl,
     write(Second),nl,
     writeCondList(RestOfList).

%************************************************************%
% writeTypeList -                                            %
%     Writes attribute name and type pairs, one per          %
%     line.                                                  %
%************************************************************%
writeTypeList([]) :- !.

writeTypeList([attr(First,Second)|RestOfList]) :-
     write(First),nl,
     write(Second),nl,
     writeTypeList(RestOfList).

%************************************************************%
% printstring -                                              %
%     Primitive to write a prolog character string.          %
%************************************************************%
printstring([]).

printstring([H|T]) :-
     put(H), printstring(T).


%************************************************************%
% listlen -                                                  %
%     Primitive to determine the length of a list            %
%************************************************************%
listlen([],0).

listlen([Head|Tail],Len) :-
     listlen(Tail,Len2),
     Len is Len2 + 1.
```