Report Number: WUCS-87-13

1987-06-01

Towards a Generalized Database System with Multiple Interfaces

Steve B. Cousins and Guillermo R. Simari

We have applied logic programming to the problem of designing knowledge representation systems. This report describes a Generalized Database System, PRODB, that has been implemented in Prolog. It also describes two extensions to the basic PRODB core. First, knowledge representation and consistency-checking features have been added to PRODB to enhance its ability to consistently represent knowledge, especially in an Engineering domain. Second, extensions to Prolog's definite clause grammar mechanism have been used to create interfaces to a knowledge base directly from grammars describing the input languages. The interface to the system is derived directly from the grammars, so this... **Read complete abstract on page 2.**

Towards a Generalized Database System with Multiple Interfaces

Steve B. Cousins and Guillermo R. Simari

Complete Abstract:

We have applied logic programming to the problem of designing knowledge representation systems. This report describes a Generalized Database System, PRODB, that has been implemented in Prolog. It also describes two extensions to the basic PRODB core. First, knowledge representation and consistency-checking features have been added to PRODB to enhance its ability to consistently represent knowledge, especially in an Engineering domain. Second, extensions to Prolog's definite clause grammar mechanism have been used to create interfaces to a knowledge base directly from grammars describing the input languages. The interface to the system is derived directly from the grammars, so this part of the system is easy to tailor. In addition, we are able to use different grammars at different times in order to have multiple interfaces to the same knowledge base system.

TOWARDS A GENERALIZED DATABASE SYSTEM WITH MULTIPLE INTERFACES

Steve B. Cousins and Guillermo R. Simari

WUCS-87-13

June 1987

Center for Intelligent Computer Systems
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

We have applied logic programming to the problem of designing knowledge representation systems. This report describes a Generalized Database System, PRODB, that has been implemented in Prolog. It also describes two extensions to the basic PRODB core. First, knowledge representation and consistency-checking features have been added to PRODB to enhance its ability to consistently represent knowledge, especially in an Engineering domain. Second, extensions to Prolog's definite clause grammar mechanism have been used to create interfaces to a knowledge base directly from grammars describing the input languages. The interface to the system is derived directly from the grammars, so this part of the system is easy to tailor. In addition, we are able to use different grammars at different times in order to have multiple interfaces to the same knowledge base system.

TABLE OF CONTENTS

1. Introduction

PRODB, a full relational database system written in PROLOG,[1] has been extended in two novel ways. First, the database system has been expanded to include new ways to consistently represent knowledge. Second, multiple interfaces to the system have been devised using an extension of PROLOG's definite clause grammar mechanism. We refer to this extended system as a knowledge base system or a "generalized database system".

The term "generalized database system" describes the result of combining database technology and artificial intelligence techniques. Other terms used for this new type of system are "expert database systems" and "intelligent databases".[2]

Relational database systems have become mature enough to meet the requirements of traditional data processing applications, especially in the business community, but engineering applications are a different matter. Relational databases represent an improvement over other standard data models, but engineering applications have special requirements. For example, engineering applications require the handling of multiple data types and complex relationships among the components of a model, the storing and handling of knowledge such as design rules, the creation and manipulation of complex objects with a rich semantic composition, and the supporting of multiple versions of the same database.[3]

In fact, the key characteristic of these engineering applications is the evolving nature intrinsic to them. That distinguishing attribute is part of any representation, or conceptual model, chosen for the problems to be solved. By coupling artificial intelligence techniques and database management systems technology, new systems for developing applications which require knowledge-directed processing of shared information are being developed. PRODB is one such system.

The information produced in today's systems has grown too fast for humans to handle with the necessary accuracy and speed. This leads to a poor use of the pool of atomic knowledge. The real issue is the requirement of moving one step further in abstraction to the point where humans beings can understand and properly handle the knowledge again. Various database abstractions have been created and implemented, but while they have solved some problems, they were not enough for the general engineering design task. The stored ground atomic facts represented extensional knowledge adequately, but the corresponding intensional knowledge has been lost in these models. This intensional knowledge can be captured and interpreted, but we must develop, refine, and use the correct tools to do so. Systems to solve these problems must provide for two aspects: the efficient and reliable handling of ground data (the DBMS technology contribution), and a knowledge representation and inferencing environment (the AI contribution).[4]

Along with the problems of representing knowledge are the problems of accessing the knowledge that is represented. In using systems based on the relational model, engineers tend to use more, smaller relations and hence have to keep more knowledge about where the data is stored in their heads. Furthermore, while advanced knowledge representation systems often give many more ways to access data, keeping all of these

access methods in mind is tedious for people who already have a full-time job in engineering.

The research reported here focuses on the issues mentioned above. In the following section we will describe PRODB, our prototype generalized database system implemented in Prolog. Prolog was chosen as the implementation language because of the close relation between Logic Programming and the Relational Data Model and the fast prototyping environment that the language provides.[5] After we have described the basis for our work, we will report on novel aspects of the knowledge representation system and the interface system. Finally, we will briefly mention some directions this work is taking.

2. Description of PRODB

PRODB is being used to explore the technical problems that appear when developing database and knowledge based applications in a process engineering design environment. The system provides a relational data structure and the standard relational algebra operators which extend the Prolog capabilities to those of a relational database system. In addition, the user is allowed to define new domains using Prolog predicates and domain integrity is maintained. That extension gives considerable freedom for the definition of the attributes of the relations. This will be extended to the complete definition of abstract data types, i.e. complete encapsulation of the extensional predicate and the operators applicable to the elements in the domain.

Also, it is possible in PRODB to define assertions which are predicates involving the attributes in a relation that every tuple in the relation must satisfy. Every time a tuple is inserted in a relation the assertions related to that relation are checked. If the tuple does not conform to any of them the insertion is rejected. Modifications are handled as delete/insert pairs.

Actions are daemons associated with the update operations and can be defined by the user as Prolog predicates that will be triggered by those operations. Foreign keys can be defined and referential integrity is enforced. When an update operation is issued over a relation, the system checks the foreign keys defined for that relation and, in case that referential integrity is violated, a user defined daemon is triggered. A postcondition for that user procedure should be that the referential integrity is restored.

Time-stamps are associated with every object in the database and there are special predicates to handle them, but they are protected from changes. The system can handle transactions and these transactions can be open, committed and rolled-back providing a good error recovery facility.

The system is capable of handling several databases at the same time and all the relational operations can take arguments from different databases. Also the referential integrity can be enforced across databases. Different scenarios can be developed in that way and in conjunction with the transaction facility the seed for an exploration

capability is in place.

2.1. Relations and Schemas

In the relational model a relation is defined as a subset of the Cartesian product of a fixed number of domains. In the systems currently in use, domains are reduced to a small, fixed, number of data types such as real numbers, integers, strings of characters, etc. This is one of their shortcomings when they are used in engineering applications. In PRODB, relations can be defined over any domain that can be defined using Prolog predicates. This characteristic gives the user a great deal of freedom and the opportunity of introducing more semantics from the beginning of the design process.

The description of a relation is contained in its schema. A schema is a finite, ordered set of attribute names. In correspondence with every name there is a domain. A subset of these attribute names should be defined as forming the (primary) key attributes of the relation. Primary keys (keys from now on) must be defined and the uniqueness of the values in the relation is maintained, i.e. no two tuples can have the same values for the key attributes. Key definition is an important issue because keys provide the only way of addressing tuples in the context of the relational model, and the values of the keys are used for referencing the tuples identified by their values.

In PRODB the schema is defined when the relation is created and should contain the attribute names, the related domains, and the specification of the subset of the attribute names that form the key. Also a format that will be used in printing the relation contents can be optionally specified. An example of a possible schema follows. The schema is shown in the same way as the PRODB command print_schema prints it.

```
parts    (created : 07/09/86 22:50:16)
         [[p_nr,pcode],[pname,string]]
         [[pcolor,color],[weight,real],[city,string]]
         [4,8,7,6,7]
         weight >= 10
         if_removed : record_deletion
```

In this schema for the relation parts, the first line contains the name of the relation and the time of creation. The second and third line are dedicated to the specification of the attribute names and its related domains pairwise. The second line is the description of the key attributes, containing in this case the attributes p_nr and pname. The domain of p_nr is pcode, the domain of pname is string. The domain pcode is user defined and string is system defined. A possible definition for the domain pcode could be,

```
pcode(Value) :-
      integer(Value),
      ( Value >= 10, Value =< 19 )
      ;
      ( Value >= 40, Value =< 49 ).
```

This is a Prolog predicate saying that Value should be an integer between 10 and 19 or between 40 and 49. The fourth line of the schema definition is the printing format specification. Fifth and sixth lines contain an assertion and an action; those topics will be introduced below.

2.2. Operations

The system currently provides the set theoretic operations union, intersection, difference and Cartesian product, and the special relational operations join, project and select. In the execution of each of the them a new relation is created as the result of the operation. That is, the system is closed under the algebraic operations.

The set operations, and project, are defined as usual in relational algebra. Join and select are generalized by allowing the use of arbitrary predicates as the selectors of the tuples that will be stored in the resulting relation. That predicates can even involve the execution of operations over other relations. For example, if the contents of parts is,

```
| ?- print_rel(parts).

DB: spj  Relation : parts (created : 07/09/86 22:50:16)

-----------------------------------------------
p_nr |    pname |   color | weight |    city
-----+----------+---------+--------+---------
  p1 |      nut |     red |     12 |  london
  p2 |     bolt |   green |     17 |   paris
  p3 |    screw |    blue |     17 |  athens
  p4 |    screw |     red |     14 |  london
  p5 |      cam |    blue |     12 |   paris
  p6 |      cog |     red |     19 |  london
  p7 |    brake |   black |     30 |  madrid
  p8 |     bolt |    blue |     30 | cordoba
-----------------------------------------------

8 tuples in the relation
```

the following is a select operation over that relation parts, using the system defined predicate "member" and ">".

```
| ?- select(spj : parts,
            spj : red_green_parts,
            [ [member, [color],[[red, green]] ],
              [ >, [weight], [ 14 ] ] ]).

  Done!

DB: spj  Relation : red_green_parts (created : 07/10/86 09:59:01)

  ------------------------------------------
  p_nr |    pname |   color | weight |    city
  -----+----------+---------+--------+---------
    p2 |     bolt |   green |     17 |   paris
    p6 |      cog |     red |     19 |  london
  ------------------------------------------

  2 tuples in the relation

  CPU : 0.0666657 secs.
```

At this point is worth mentioning a very nice result of the coupling of Prolog and the DBMS. Is possible to define Prolog predicates, very complex indeed, involving several of the relational algebra operations mixed with other Prolog predicates.

One immediate application of that ability is the definition of virtual data, i.e. view definition. Certainly that gives the user a very dynamic environment for data manipulation, providing the full power of Prolog, a complete logic programming language, to work with a relational database data structure.

2.3. Assertions

The system gives the capability of defining assertions over the contents of the relations. Assertions are a way of controlling the integrity of the relations and are predicates that, when defined, every tuple in the relation is required to satisfy. Every time a tuple is inserted in a relation the assertions over that relation are checked and if the tuple does not conform to any of them, it is rejected. This ensures that given a consistent state of a database no operation will lead to an inconsistent state. But the addition of new assertions over the relations in a database can make the database inconsistent.

In the previous example of the schema for the relation parts, the fifth line contains a simple assertion over the attribute weight. That assertion ensures that every tuple in the relation will have a value for the weight attribute greater than 10. For example, if a new assertion is added specifying that no part can have a value for the weight attribute bigger than 20, the relation parts becomes inconsistent because it contains a tuple, [47,brake,black,30,madrid], whose weight attribute is 30. Here is another example of an assertion over the relation parts:

```
parts_assertion(X,Y,Z) :-
        (member(X,[nut,cog,screw]), Y == red)
                -> Z == london
                ; true.
```

It establishes that all the red nuts, red cogs and red screws come from London, and has no restriction over other parts, or the same parts of different color.

It is possible to check the consistency of the database against the set of assertions and the offending tuples are displayed. The consistency of the database is performed automatically every time the database is loaded and every time the set of assertions is modified. A variation of the consistency checking command allows a user to check the relations individually and eliminate the offending tuples.

2.4. Actions

Actions are predicates, or daemons, that are associated with the relations and are executed in one of the following triggering situations: when a tuple is inserted, when a tuple is removed, when a tuple is modified, when a tuple is requested via the get_tuple operation.

Actions must be user or system predefined predicates, and are considered as part of a given database. The execution of an action can affect other relations in any loaded database and can contain any Prolog or PRODB predicate.

User-defined predicates normally take one argument: the tuple. However, in the if_requested case the predicate should take two arguments, the first is the the tuple stored in the relation and the second is a place where the tuple modified will be returned. If no modification is done it is not necessary to return the original tuple. The get_tuple predicate will return the first argument, i.e. the actual tuple stored in the relation, if the second argument is not instantiated by the action predicate. This provides a way of returning values calculated using the values in the original tuple.

2.5. Time Handling

Time is an important issue to consider in relation to engineering databases. In PRODB a timestamp is associated with every relation schema and every tuple contained in the database. This timestamp is visible through predicates such as "tprint_rel" which prints the contents of the relation in the same way as "print_rel" but with an extra column containing the timestamp value. No predicate can change timestamp values.

Several predicates can be used to access timestamps. A special "time_select" operator is provided. This operator performs selections over the tuples in a given relation based in the value of the timestamp, a time value provided by the user and several time comparison operators provided by the system. For example,

```
time_select(parts, newest_parts, '07/09/86 10:30', *> ).
```

will store in a new relation called newest_parts those tuples in the relation parts with timestamp representing a time after (*>) the specified date.

There are also defined three comparison predicates "older", "newer" and "same_date" which decide whether a tuple is older, newer or has the same timestamp that another tuple in the same or different relation.

PRODB provides several aggregate predicates. There are two of them that are time related, oldest and newest. The first will return a tuple in a relation that has the oldest timestamp. The latter will return a tuple with the newest timestamp. Obviously the result can be not unique, i.e. many tuples can share the same timestamp. The set of all the tuples with the same timestamp can be obtained using the "time_select" predicate.

2.6. Transactions

A transaction is a sequence of operations that is considered as an atomic operation for the system. That is, given a database state if the user starts a transaction the system will not consider the operations issued inside the transaction as a change in the original state until the transaction is closed. In particular, inside a transaction the referential integrity checking will be suspended. At present time this capability is useful in relation with the roll_back predicate. Transactions cannot be nested.

The PRODB predicates related to transactions are: "open_transaction" for starting a transaction, "close_transaction" for ending a transaction and committing all operations since the open_transaction command, and "roll_back" for canceling the effect of the commands issued since the open_transaction command.

3. Extensions to the Representation System

The current implementation of the system includes the major features in the relational model with some extensions, already mentioned before and summarized here, such as arbitrary domains over which relations can be defined, primary and foreign key definitions, use of arbitrary predicates in the selection and join operations, definition of actions (daemons) that get invoked when certain operations are executed, definition of assertions over the tuples in a relation and the consistency checking of the relations stored in the database, transactions, and the inclusion of time as information associated with the objects in the database along with predicates for handling tuples using that information.

As an exploratory vehicle the system should continue its evolution. That evolution will be prompted by theoretical reasons and for requirements derived from its use in

projects in progress. The following paragraphs will address that expansion in the immediate future.

3.1. Integrity Checking

A database is a discrete and static model of some reality. A database evolves from one state to another when the information is updated. That evolution, if it is left without control, can start in a consistent state and end in an inconsistent one, i.e. not every possible state of the database is a valid representation of the reality modelled. The reason for this is that, even assuming the best representation possible, the model is constructed with tools that have a representational grain bigger than the real object of the modelling. The intention here is to provide better tools for the conceptual modelling task.[4]

Integrity constraints are a fundamental issue in the definition of the model as they define the intension of relations, i.e. they describe time independent properties of relations. [6] PRODB allows the definition of local assertions over a given relation, and this assertions are consulted every time a tuple is added or modified. In that way the local granularity is reduced by rejecting more incorrect states.

The next step in the integrity constraint implementation will be the capability of defining global constraints involving more than one relation. Toward that goal the definition of foreign keys is already implemented.

3.2. Abstract Data Types

Relations are defined in the relational data model over domains that are sets of values. With the addition of the capability of defining arbitrary domains together with the use of arbitrary predicates in the selection and join operations the next step will be the introduction of abstract data types.

An abstract data type (ADT) is a set of values and a set of operations over those values. Prolog provides a way of defining operations but the encapsulation implicit in the definition of ADT's is not present per se. Tying operations to domains introduces more semantic constraints allowing better representational capabilities. [7]

3.3. Object Oriented Programming and Databases

In the Object Oriented Programming paradigm the computing process is seen as the development of a system composed of objects through sequences of changing states. Each object has a class, and a class is defined by specifying a set of supported operations, a set of methods and a set of instance variables. The instance variables are used in representing the state of the object. Objects are manipulated by sending messages containing the name of the operation to be executed and certain objects. The object responds to the message by possibly changing its state and by returning a result object. Methods are associated with operations and are executed when a message

designating that operation is received by the object.

There is an important relationship among classes that characterizes this approach. Similar objects are grouped together in classes, then classes are in turn also grouped together in superclasses, an so on. This hierarchical organization is extended with the possibility of inheritance of all the attributes corresponding to objects at a higher level.

This paradigm of computation can bring a good approach to solving problems such as version management, integrity, entity definitions, evolution of the model, etc.

The first step in that direction has been taken through the incorporation of a set of frame based object definition and handling procedures.

4. Multiple Interfaces

This system can allow the user to communicate with the knowledge base with a variety of languages. Each input language is specified by a context-free grammar. We consider the grammar to be knowledge about the language, and use this knowledge to help the user to communicate with the knowledge base. Using a grammar, the interface sub-system generates an interface to the knowledge base. Currently, this interface is a simple menu-driven one, and a command-line interface is being developed.

4.1. Interface Architecture

The architecture of our interface system differs significantly from that of traditional knowledge bases. Current interfaces to knowledge bases and databases fall into two categories, command-line interfaces and highly graphical interfaces in which data is manipulated directly. Our interface falls somewhere in between these extremes, taking benefits from both.

In a command-line interface, a user types a sentence in the interface language on a terminal of some sort. The language may be formal and well-defined (e.g. SQL) or may be informal such as a natural language interface. In any case, the language will recognize some sentences and will not be able to recognize others. The system *parses* the sentence and if it recognizes the sentence, it performs an appropriate action. If the user makes a mistake in typing, she must retype the sentence, or at best she can call the sentence back and edit it.

Graphical, "direct manipulation" interfaces as Shneiderman calls them8 provide very effective access to information in knowledge bases. Unfortunately, such interfaces are usually difficult to develop. More importantly, they usually need special hardware such as a bit-mapped display. We take the position that everyone *should* have a bit-mapped workstation on his desk, but since that is not the case for the majority of computer users, the interfaces presented here can take advantage of the bit-mapped display or can run on more traditional terminals.

The interfaces to PRODB add a layer of user support where basic command provide simple functions such as backspace. Our interfaces assist the user in constructing a syntactically-correct sentence, either by giving her step-by-step choices, or by checking the syntax as the sentence is typed. The sentences that pass this first layer of support are always syntactically correct.

This extra interface layer is generated directly from a context-free grammar for the knowledge base's input language. This is the same grammar that we use to translate the input language into Prolog commands, but without the translational component. We provide multiple interfaces by providing multiple languages (and corresponding grammars) which are translated into basic knowledge-base commands. Our approach does not require special hardware, but can take advantage of it.

4.2. Automatic Menu Generation

This system is similar to the NLMenus system of Texas Instruments.[9,10] Our simple interface will display up a menu of the next syntactically correct words in the input language as the interface user composes input to the program. The user then chooses one of the next legal words and the system comes up with a new set of possible next words.

We generate these menus automatically by calculating the set of next legal words (the "Next" set) for the grammar and the input selected so far.[11] This interface allows the user to construct only legal sentences of the input language, as well as providing him choices of next legal words.

It is possible that some of the menus generated could contain hundreds of items, but since we can specifically tailor our languages, we can prevent this from happening. Furthermore, we are working on ways to extend the system further so that large menus can be easily broken into hierarchies.

By using an abstract menu predicate that takes a list of items for the menu and returns one choice, we can use this system on a variety of hardware. On a simple terminal, a menu choice might be made by using cursor keys to select a choice. On a more advanced terminal, a mouse interface can be added. On workstations, we can tailor the menu predicate to the system's native window system.

This system is good for novices but may be slow for experienced users. Novice and casual users of the system are led by the hand through the construction of legal queries. We believe that this leads to faster learning of the query language. For experienced users, however, it is a standard argument against menu systems that they are much slower than the keyboard. In our applications the target audience is the casual user so it is not clear that this objection applies. However, we do have another interface which gives som of the advantages of this system but which is more suited to experienced users.

4.3. Heuristic Command Line Completion

Experienced users know the interface language syntax and tend to be slowed down by menus. For these users, we are developing a keyboard interface that works in a similar way to the menu system. As the user types, the system will calculate the Next set just as it did in the menu system. If the user types a word that is not in the Next set, the system will notify him that he has made a mistake by ringing the bell on the terminal. This way, the experienced user is also prevented from making syntax errors.

In addition, the system can provide another benefit to the keyboard user to reduce his typing. When the Next set is relatively small, a couple of characters may be enough of a prefix to uniquely identify the choice in the Next set. In this case, the user can type a space and the system will complete the word for him. This type of command completion has been used in other systems, but to our knowledge, has not been implemented in general for context-free grammar based systems.

5. Future Work

The PRODB system will continue evolving towards the integration of Object Oriented Programming concepts and Logic Programming in the framework of extending the Relational Model. We are developing a sound theoretical basis to support that merging as well as a pragmatic expresiveness in the data modelling (and manipulation) language. Contained in these ideas is the objective reducing of the impedance mismatch between the operative language and the representational language.

There are three things in the near future for the interface system. First, the command line interface can be enhanced by a spelling correcter. It should be possible to take advantage of the small size of the Next sets so that when a word is typed that is not in the set, the word can be modified to fit the set on the assumption that it had been spelled incorrectly. Secondly, a graphical interface in which there are icons that can be chosen and buttons that can be "pressed" with the mouse is really a kind of grand menu. Therefore, we would like to map words in the input language to icons on the screen and allow the menu items to be chosen by pointing to pictures instead of words. Finally, the two current interfaces might be combined, so that if a user delayed in typing or hit a special key, a menu containing the next set would come up. Intermediate users would then be able to try the command interface, but would get help when the syntax of a command slipped from mind.

6. References

1. G. R. Simari, "PRODB, An Experimental Generalized Database System (User Manual)," *Washington University*, Dept. of Computer Science, Tech. Rep. WU-87-13, (June 1987).

2. J. M. Smith, "Expert Database Systems: A Database Perspective," *Proc. First International Workshop on Expert Database Systems*, pp. 3-15 (1986).

3. David J. Hartzband and Fred J. Maryanski, "Enhancing Knowledge Representation in Engineering Databases," *Computer* **18**(9) pp. 39-48 (Sep 1985).

4. Michel L. Brodie, "On the Development of Data Models," in *On Conceptual Modelling*, ed. Joachim W. Schmidt,Springer-Verlag (1984).

5. Veronica Dahl, "On Database Systems Development Through Logic," *ACM Transactions on Database Systems* **7**(1) pp. 102-123 (Mar 1982).

6. J. Nicolas, "Logic for Improving Integrity Checking in Relational Databases," *Acta Informatica* **18** pp. 227-253 (1982).

7. S. L. Osborn and T. E. Heaven, "The Design of a Relational Database System with Abstract Data Types for Domains," *ACM Transactions on Database Systems* **11**(3) pp. 357-373 (Sep 1986).

8. Ben Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, Reading, Mass (1987).

9. Harry R. Tennant, Kenneth M. Ross, and Craig W. Thompson, "Usable natural language interfaces through menu-based natural language understanding," pp. 154-160 in *Proc. CHI '83 Conference on Human Factors in Computing Systems*, (1983).

10. , *Explorer Natural Language Menu System User's Guide*, Texas Instruments (1985).

11. Steve Cousins, "Automatic Menu Generation," *SIGART Newsletter*, (October, 1987).