Tetrahedron Computer Methodology, Vol.3, No. 6C, pp. 739-756, 1990
Printed in Great Britain

0898-5529/90 $5.00+.00
©1992 Pergamon Press Ltd

A PROLOG Approach to Analysing Protein Structure

Geoffrey J. Barton[1,2] and Christopher J. Rawlings[2]

[1]University of Oxford; Laboratory of Molecular Biophysics
The Rex Richards Building; South Parks Road; Oxford OX1 3QU - UK (geoff@uk.ac.ox.biop)
and
[2]Biomedical Computing Unit; Imperial Cancer Research Fund
Lincoln's Inn Fields; London

Received: 26 September 1991; Revised: 14 January 1991; Accepted: 15 January 1991

Key Words: logic programming, PROLOG, protein structure, secondary structure, topology, relational database, knowledge base

Abstract: *This paper provides a detailed description of a database of protein structure implemented in the logic programming language PROLOG. The database allows flexible access to structural information at the atom, residue, secondary structure and topology levels of the protein structural hierarchy. An extended version of the Kabsch and Sander algorithm for secondary structure definition has been implemented in PROLOG, forming an integral part of the database. For protein structure analysis, the PROLOG system shows significant advantages in flexibility over conventional programming languages such as Fortran, and Relational Database Management Systems using SQL.*

INTRODUCTION

Comparative analysis of the protein three-dimensional structures determined by X-ray crystallography or NMR, furthers our understanding of the factors influencing the native protein fold and suggests rules to guide the prediction of structure and function from the amino acid sequence. Traditionally, these analyses have been performed with bespoke Fortran programs that access flat files of coordinate data. Attention has focussed recently on using commercial Relational Database Management Systems (RDBMS) to store the coordinate and derived structural information (*e.g.*, torsion angles, accessibilities, secondary structure, *etc.*) whilst making use of the SQL language to query the data.[1,2] However, both traditional Fortran and RDBMS are limited by the inflexibility of the data storage format and the query language. Whilst in principal all queries can be answered through a conventional programming language such as Fortran or C, the effort involved in coding the question can be formidable. Furthermore, if the first question leads on to another, then a similar programming project must be undertaken to answer the follow-up query. In contrast, the RDBMS systems provide ready access to simple queries of the data without the need for complex programming. Unfortunately, the query language SQL can only represent simple tabular data structures, and the underlying relational model, though well suited to tables of names and addresses, wages, *etc.*, does not cope well with the naturally sequential protein structural data.

Gray *et al.* [3] have described an object-oriented database of protein structure. Their system explores the

advantages of a system developed in the logic programming language Prolog. The relational data structures and unification based retrieval used by Prolog provide flexible ways of accessing structural data, and Prolog's symbolic (rule-based) programming style enables many aspects of structure analysis to be represented directly.

Our work on representing protein structures in Prolog described in this paper has developed out of the successful initial investigations into using logic programming techniques to represent and search for topological motifs.[4] Subsequent studies have made use of symbolic logical descriptions of protein structures during protein structure prediction,[5] as a common platform in the PAPAIN project to develop a knowledge-based environment for the interpretation of protein sequence data,[6] and for the prediction of protein topology.[7] Here, we extend the original representation of protein structure which was centered around protein topology, to include more detailed structural information. Particular attention has been paid to the representation of hydrogen bonding patterns, and the systematic classification of protein secondary structure using Kabsch and Sander's definitions.[8]

The Brookhaven Protein Data Bank

All protein structural data was extracted from the Brookhaven Protein Data Bank (PDB).[9] The PDB acts as a repository for macromolecular structure information, primarily the results of X-ray crystallographic analysis. Each molecular structure (*e.g.*, a protein structure) is stored in an individual flat file. The file entry contains the x, y, z coordinates in a Cartesian coordinate system for every atom in the structure. Additional information regarding the experimental procedure used is also included together with the source of the biological material that has been analysed.

Introduction to Prolog, System, Syntax and Hardware

All programming was performed using the commercially available Quintus Prolog system (Release 2.4.2). General details of the Prolog language may be found in the book *Programming in Prolog* (Clocksin, W. F. and Mellish, C. S, Springer-Verlag, FRG, 1981). Briefly, Prolog stores facts and deductive rules in a database, where the facts are equivalent to a collection of relations in a relational database. The Prolog interpreter when presented with a query (or goal as it is often called) searches the database of facts and rules to see if any of the facts and rules match the query. If there are facts that match, then the variables in the query are unified with the constant symbols in the fact(s). If there is a rule whose consequent (the head of the rule) matches the query, then the interpreter proceeds by attempting to recursively prove the subgoals of the rule. Specific examples relevant to the protein structure information stored in our database are described below. The database system was originally developed on a Sun 3/50 workstation with 4MBytes of memory. The current installation is running on a Sun SPARCstation 1 with 12MBytes of memory.

Chain, Residue and Protein Naming Conventions

A protein coordinate entry (file) as listed in the Brookhaven Protein Data Bank (PDB) is uniquely identified by a 4 or 5 character code starting with an integer, for example, 1FB4 is the code for an immunoglobulin Fab fragment. However, any given entry may contain multiple protein chains, and these are usually the basic unit of protein structure. If more than one chain exists in an entry, the PDB assign a single character code to each chain. For example, the entry 1FB4 contains two chains labelled L and H.

One often wishes to reference an individual chain rather than the whole protein entry. Accordingly, when converting the PDB to Prolog clauses, a two-level naming convention was adopted. Each Entry is identified by the PDB code (*e.g.* 1fb4) and each chain belonging to an entry is identified by the PDB code with the chain identifier code appended (*e.g.* 1fb4l). If an entry has a single chain that does not have a chain identifier, then the chain is simply identified by the PDB code alone. The PDB code and chain codes are linked by the Prolog clause `chain/2` as described below.

Residue numbers are represented by two-element lists (*e.g.* `[37,a]`, `[37,b]`, `[38,-]`) in order to accommodate the insertion characters that allow homologous structures to be numbered in a manner consistent with the 'parent' structure. For example, all serine proteinase enzymes (trypsin, elastase *etc.*) are numbered according to the structure of chymotrypsinogen, which was the first member of the family to be solved by X-ray crystallography. Although this simplifies cross-referencing between different proteins in the same family, the flexibility creates difficulties when searching for residues N amino acids before, or after the current position.

SIMPLE PROLOG CLAUSES DESCRIBING BROOKHAVEN ENTRIES

A Fortran program (BRKSEQ), reads the PDB files and processes the necessary information to describe each protein entry by up to eleven different types of Prolog clauses:

```
header(Ident,List).
compnd(Ident,List).
source(Ident,List).
resolution(Ident,R).
chain(Ident,Chcode).
nchains(Ident,N).
chain_range(Chcode,[Cstart,CstartIN],[Cend,CendIN]).
chain_length(Chcode,Len).
residues(Chcode).
no_mainchain(Chcode).
no_sidechains(Chcode).
```

not all clauses need be present for a particular entry, as shown for the Immunoglobulin structure 2fb4.

```
header(2fb4,[immunoglobulin,18-apr-89,2fb4]).
compnd(2fb4,[immunoglobulin,fab]).
source(2fb4,[human,(homo,sapiens),myeloma,patient,kol,serum]).
resolution(2fb4,   1.900).
nchains(2fb4,    2).
chain(2fb4,2fb4l).
chain_range(2fb4l,[   1,-],[ 214,-]).
chain_length(2fb4l,  216).
residues(2fb4l).
chain(2fb4,2fb4h).
chain_range(2fb4h,[   1,-],[ 221,-]).
chain_length(2fb4h,  229).
residues(2fb4h).
```

The `header`, `compnd`, `source` and `resolution` clauses are extracted directly from the information stored at the beginning of every PDB file. The `Ident` is the PDB identification code for the protein (*e.g.* 9lyz), `List` is a Prolog list containing textual information, and `R` is the resolution of the structure in Angstroms. The remaining seven clauses are derived from an analysis of the PDB ATOM records. The `chain` clauses link the PDB identification code to the chain code `Chcode` whilst `nchains` simply lists how many chains are present in the PDB entry. The `nchains` clause is included for simplicity, though is strictly unnecessary since a Prolog rule could be used to count the number of `chain` clauses present for each protein. For every `chain` clause, there is one `chain_range` clause which specifies the starting and ending residue numbers of the chain. Similarly, there is a `chain_length` clause that states the number of residues present in the chain (this clause is essential due to the alphanumeric residue numbering scheme used by the PDB). The `residues` clause identifies a chain as having amino acid residues other than UNK (or X), whilst the presence of `no_mainchain` or `no_sidechains` clauses for a chain shows that the protein entry is incomplete (some PDB entries only contain mainchain, or $C_\alpha$atoms).

Using the PDB Information

There are 7102 clauses defining the November 1990 release of the Brookhaven PDB. These provide a very simple database that may be read directly into the Prolog system and interrogated by writing simple Prolog queries. For example, the following query, also known as a *goal* could be typed in at the Prolog prompt (`| ?-`).

```
| ?- resolution(PID,R),R < 2, R > 0,chain(PID,CID),residues(CID).
```

The query will return the protein identifier (PID), chain identifier (CID) and crystallographic resolution (R) for each chain in the databank whose entry is less than 2 A resolution and has amino acid residues deposited.

The Prolog interpreter attempts to satisfy this query as follows. Firstly Prolog looks in its database for facts called `resolution`. The first fact is found and the variables PID and R unified with the arguments of the clause. The value of R is then tested to see if it is less than 2. If it is, then the test is made to see if R is greater than zero. If this succeeds, then a `chain` clause is looked up in the database that unifies with the current value of PID. Finally, if the `chain` clause is found, a `residues` clause is looked up in the database that unifies with the value of CID.

The query can fail at any stage. For example, if no `residues` fact is found that contains the current CID, then the goal fails. Prolog then starts a process of *backtracking* to search for a possible solution. The interpreter would first look for another `chain` fact. If present, then this would unify CID with the value shown in the `chain` fact, again the database would be checked for a corresponding `residues` clause. If all elements of the query succeed, then the values of PID,CID and R are displayed. The entire query may be forced to search for alternative solutions by typing a semicolon. For example, the following are the first three solutions to the query shown.

```
| ?- resolution(PID,R),R < 2, R > 0,chain(PID,CID),residues(CID).

PID = CID = 1alc,
R = 1.7 ;              %First solution found - type ';'
                       %       to force backtracking
PID = 1amt,
R = 1.5,
CID = 1amta ;         %Second solution - type ';' again for next solution

PID = 1amt,
R = 1.5,
CID = 1amtb           % ... and so on ...
```

If we often want to select protein chains by the criteria shown in this query, it is simple to build the query into a general purpose Prolog rule. For example, the rule called `select_chains`:

```
                                Comments start with a %
select_chains(Rmin,Rmax,CID):-
        resolution(PID,R),      % look up resolution
        R < Rmax,               % resolution below Rmax
        R > Rmin,               % resolution above Rmin
        chain(PID,CID),         % find a chain identifier for this PID
        residues(CID).          % check the chain has residues
```

We can now type:

```
| ?- select_chains(0,2,CID).
```

at the prolog prompt to find out which chains satisfy our criteria. Having established this new rule, we can then use it in further queries. For example:

```
| ?- select_chains(2,3,CID),chain_length(CID,Len),Len >= 150.
```

will return the chain identifier and length for chains belonging to entries that are of between 2 and 3 A resolution and where the chain length is at least 150 amino acids.

Getting All Solutions

As stated, the example queries require interactive prompting by typing a semicolon, to return more than one solution. However, there are two methods by which Prolog can return all possible solutions to a particular query. The first is to put the built-in predicate (also called a procedure) `fail.` at the end of the query. This forces backtracking whenever it is encountered. In order that the results of each solution to the query are recorded, some write statements must be added. For example:

```
| ?- select_chains(2,3,CID),chain_length(CID,Len),Len >= 150,
        write(CID),nl,
        write(Len),nl,fail.
```

will print the values of chain identifier and length that satisfy the query. The `nl` predicate simply places a carriage-return character after each write. Part of the output of this query looks like this:

```
8at1c
310
8atca
310
8atcc
310
```

A more useful type of output would be to write out the results in a form that can be directly re-used by Prolog as a set of clauses. For example:

```
| ?- select_chains(2,3,CID),chain_length(CID,Len),Len >= 150,
        writeq(special_chains(CID,Len)),writeq(.),nl,fail.
```

where the `writeq` predicate makes sure that all syntax is correctly observed. Part of the output resulting from this query is shown below and can be read back into Prolog for further processing:

```
special_chains(5acn,754).
special_chains(5adh,374).
special_chains(5at1a,310).
special_chains(5at1c,310).
special_chains(5tln,316).
```

In this example, there is no real advantage in writing the results in this form since the original query is very quickly evaluated. However, for a complex query where Prolog may take several minutes to find all solutions, it makes sense to save intermediate results in this form. This is particularly true in the analysis of protein structure, where one question frequently leads to another and the results of the first question are necessary to solve the following question.

An alternative way to return all solutions to a query is to make use of the Prolog built-in procedure `bagof`. This is an extremely flexible procedure that returns the results of a query in the form of a Prolog list. For example, the previous query could be expressed as:

```
| ?- bagof(special_chains(CID,Len),
          (select_chains(2,3,CID),chain_length(CID,Len)),L)
```

and will return the variable `L` as a list containing `special_chains(CID,Len)` clauses. Normally, this is the method of choice since it gathers all solutions into an easily manipulated list structure. This example required 0.08 seconds to scan a database containing 501 entries on a Sun SPARCstation 1.

DERIVED INFORMATION

So far we have described an extremely simple database of protein structure-related information and shown how Prolog can be used to query the data. This simple database forms a hub of basic facts around which the more detailed structural data of our system are organised.

Although PDB coordinate files contain the x, y, z coordinates of individual atoms in a protein, one frequently does not require this level of detail. It is often sufficient to consider the structure at the level of the amino acid residues. For example, one may be interested in the secondary structure of a particular residue, (α–, β–or turn), the accessibility to solvent, the ϕ/ψ angles and so on. The Kabsch and Sander secondary structure definition program DSSP,[8] reads a Brookhaven coordinate file and generates a file of definitions for the secondary structure of the protein. In addition to defining the structural state of each residue, DSSP also generates a table containing the main chain hydrogen bonds, accessibility, ϕ/ψ angles and $C_\alpha$coordinates together with other relevant information. Accordingly, the table provides an invaluable reference source of residue-level information for the protein. Simply converting these tables to Prolog clauses allows the power of Prolog to be used in queries of the data. A sample of the table output by DSSP is shown here:

```
  1    1 A V              0   0  139    0, 0.0   2,-0.6   0, 0.0 130, 0.0
0.000 360.0 360.0 360.0 130.6    6.9   17.8    4.7
  2    2 A L        -     0   0   19  125,-0.1 122, 0.0 126,-0.1   5, 0.0
-0.711 360.0-142.7 -80.5 121.9   10.6   17.9    4.2
  3    3 A S      > -     0   0   44   -2,-0.6   4,-2.0   1,-0.1  -1, 0.0
-0.301  27.8-105.5 -71.3 164.8   12.3   19.8    7.0
  4    4 A P  H  > S+     0   0   98    0, 0.0   4,-2.4   0, 0.0  -1,-0.1
0.880 125.4  55.4 -62.3 -35.0   15.3   22.0    6.2
  5    5 A A  H  > S+     0   0   59    1,-0.2   4,-2.4   2,-0.2  -3, 0.0
0.907 104.9  52.7 -65.0 -41.5   17.5   19.4    7.7
  6    6 A D  H  > S+     0   0   15    1,-0.2   4,-2.3   2,-0.2  -1,-0.2
0.918 109.0  50.3 -56.3 -42.8   15.9   16.9    5.2
  7    7 A K  H  X S+     0   0   46   -4,-2.0   4,-2.5   1,-0.2  -2,-0.2
0.881 109.2  50.2 -64.7 -36.2   16.7   19.2    2.4
  8    8 A T  H  X S+     0   0   90   -4,-2.4   4,-2.0   1,-0.2  -1,-0.2
0.910 111.6  49.6 -63.3 -42.7   20.4   19.6    3.5
```

The program BRKSAND converts this table format into Prolog clauses, one for each residue. For example:

```
ks(    1,2hhb,2hhba,[  1,-],V,-,[-,-,-],[  0,   0,-],
 139,[  0, 0.0],[  2,-0.6],[  0, 0.0],[ 130, 0.0],[ 0.000, 360.0, 360.0],[
 360.0, 130.6],[   6.9,   17.8,    4.7]).
ks(    2,2hhb,2hhba,[  2,-],L,-,[n,-,-],[  0,   0,-],
  19,[ 125,-0.1],[ 122, 0.0],[ 126,-0.1],[  5, 0.0],[-0.711, 360.0,-142.7],[
 -80.5, 121.9],[  10.6,   17.9,    4.2]).
ks(    3,2hhb,2hhba,[  3,-],S,-,[n,-,-],[  0,   0,-],
  44,[  -2,-0.6],[  4,-2.0],[  1,-0.1],[  -1, 0.0],[-0.301,  27.8,-105.5],[
 -71.3, 164.8],[  12.3,   19.8,    7.0]).
ks(    4,2hhb,2hhba,[  4,-],P,H,[p,-,-],[  0,   0,-],
  98,[  0, 0.0],[  4,-2.4],[  0, 0.0],[  -1,-0.1],[ 0.880, 125.4,  55.4],[
 -62.3, -35.0],[  15.3,   22.0,    6.2]).
ks(    5,2hhb,2hhba,[  5,-],A,H,[p,-,-],[  0,   0,-],
  59,[  1,-0.2],[  4,-2.4],[  2,-0.2],[  -3, 0.0],[ 0.907, 104.9,  52.7],[
 -65.0, -41.5],[  17.5,   19.4,    7.7]).
ks(    6,2hhb,2hhba,[  6,-],D,H,[p,-,-],[  0,   0,-],
  15,[  1,-0.2],[  4,-2.3],[  2,-0.2],[  -1,-0.2],[ 0.918, 109.0,  50.3],[
 -56.3, -42.8],[  15.9,   16.9,    5.2]).
ks(    7,2hhb,2hhba,[  7,-],K,H,[p,-,-],[  0,   0,-],
  46,[  -4,-2.0],[  4,-2.5],[  1,-0.2],[  -2,-0.2],[ 0.881, 109.2,  50.2],[
 -64.7, -36.2],[  16.7,   19.2,    2.4]).
ks(    8,2hhb,2hhba,[  8,-],T,H,[p,-,-],[  0,   0,-],
```

The general form of a ks fact is as follows:

```
ks(N,PID,CID,RNUM,AA,SS,CLAD,BPSHEET,ACC,NHO1,ONH1,NHO2,ONH2,TKA,PHIPSI,XYZ).
```

where:

```
N          unique atom identifier in this entry   (integer)
PID        Protein identifer code                 (character string)
CID        Chain identifer code                   (  "          "  )
RNUM       residue number  List: [integer,character insert code]
AA         amino acid code (Uppercase character - cys in bridge
                                                      is lowercase)
SS         secondary structure (Uppercase character)
CLAD       [chirality,ladder1,ladder2]  where
                                        chirality = n (-ve), p (+ve)
                                        ladder1/2 = uppercase character
BPSHEET    [BP1,BP2,SHeet] BP1/2 give values of N for bridges, Sheet is
                                        a character that labels the sheet
ACC        accessibility (Angstroms**2) (int)
NHO1/2,
ONH1/2     [Ndiff,energy]  Ndiff is the difference in N value to the Hbond
           partner, energy is the energy (kcal/mol)  (int,real)
TKA        Angles: [TCO,KAPPA,ALPHA]  (All real numbers)
PHIPSI     [PHI,PSI]  phi,psi angles (both real)
XYZ        [X,Y,Z]  coordinates of this CA atom  (all reals)
```

For details of the specific structural meaning of each of these values, see Kabsch and Sander.[8]

The first argument in the ks/16 clause is a unique numeric residue number, where the numbers run consecutively from the beginning of the protein entry. This numbering scheme overcomes the difficulties associated with the standard PDB numbering system, by allowing simple questions of the form, 'what is the residue 4 amino acids N-terminal of residue 37' to be readily answered.

If the ks clauses for a protein are loaded into the Prolog system, then queries may be made. For example, to find the names of residues with accessibilities $< 100A^2$ and with positive ϕ angles one could type:

```
| ?- ks(_,_,_,_,AA,_,_,_,ACC,_,_,_,_,_,_,[PHI,_],_),ACC < 100, PHI >0.
```

Clearly, this is a rather cumbersome interface to the data. In order to simplify this sort of query, a set of high level procedures have been written to allow different aspects of the data to be extracted. The procedure get/4 allows a particular field to be returned:

```
| ?- get(N,CID,Entry,Value)
```

N and CID are the unique residue number and chain identifier. Entry is one of the strings: ks_number, protein_identifier, residue_number, amino_acid, amino, accessibility, hbondNO1, hbondON1, hbondNO2, hbondON2, hbonds, tka, phipsi, xyz, x,y,z, sec. These values follow the arguments of the ks procedure, hbonds causes backtracking through all four hbond types that are stored, whilst amino_acid converts the code letters for residues involved in disulphide bridges into the normal amino acid codes. Value holds the returned value of the procedure. For example, to perform the same query as above we would type:

```
| ?- get(N,CID,accessibility,ACC),ACC < 100, get(N,CID,phipsi,[PHI,_]),PHI > 0.
```

which reads, 'Look up an accessibilty value, if this value is less than 100 then look up the value of ϕ for the residue, if this is greater than 0, then display the result.'

Frequently, one requires a *range* of values that start and end at particular residues in the protein chain. The `getR` procedure performs this task. For example:

```
| ?- getR(2,10,1fb4l,amino_acid,AA)
```

which will return the value for `AA` of:

```
AA = [S,V,L,T,Q,P,P,S,A]
```

The `getR` procedure can be used for far more complex queries. For example, the following query will find the sequence, accessibility and summary of secondary structure for all pentapeptides that have a mean accessibilty of less than $10A^2$.

```
CID = 1fb4l,                    %just set the chain to 1fb4l for now
kchain_range(S,E,CID),          %find the start (S) and end (E) residue numbers
SM1 is S - 1,                   %find S - 1 number
LST is E - 4,                   %find the last possible START for a pentapeptide
next1(SM1,START,LST),           %on backtracking this procedure returns START as
                                %successive values from SM1 to LST.
END is START + 4,               %specify the last residue of the pentapeptide
getR(START,END,CID,accessibility,ACC),  %get the accessibilities for this
                                        %pentapeptide
mean(MEAN,ACC),                 %find the mean accessibility
MEAN < 10,                      %test if mean is < 10 for this pentapeptide,
getR(START,END,CID,amino_acid,AA),      %look up the amino acid sequence
getR(START,END,CID,sec,SEC),            % and summary of secondary structure
write_list([CID,START,END,ACC,AA,SEC,MEAN]). %write out the information we want
```

This query produces the following output:

```
1fb4l 32 36 [9,34,0,0,1] [I,T,V,N,W] [S,-,-,E,E] 8.79999
1fb4l 33 37 [34,0,0,1,0] [T,V,N,W,Y] [-,-,E,E,E] 7.0
1fb4l 34 38 [0,0,1,0,34] [V,N,W,Y,Q] [-,E,E,E,E] 7.0
1fb4l 35 39 [0,1,0,34,2] [N,W,Y,Q,Q] [E,E,E,E,E] 7.39999
1fb4l 72 76 [2,34,0,4,1] [A,S,L,A,I] [E,E,E,E,E] 8.2
1fb4l 85 89 [10,9,0,11,0] [S,D,Y,Y,C] [S,E,E,E,E] 6.0
1fb4l 86 90 [9,0,11,0,0] [D,Y,Y,C,A] [E,E,E,E,E] 4.0
1fb4l 87 91 [0,11,0,0,0] [Y,Y,C,A,S] [E,E,E,E,E] 2.2
1fb4l 88 92 [11,0,0,0,1] [Y,C,A,S,W] [E,E,E,E,E] 2.4
1fb4l 89 93 [0,0,0,1,0] [C,A,S,W,N] [E,E,E,E,E] 0.2
1fb4l 90 94 [0,0,1,0,34] [A,S,W,N,S] [E,E,E,E,T] 7.0
1fb4l 98 102 [14,5,11,11,2] [S,Y,V,F,G] [E,E,E,E,B] 8.59999
1fb4l 134 138 [3,7,0,0,0] [A,T,L,V,C] [E,E,E,E,E] 2.0
1fb4l 135 139 [7,0,0,0,2] [T,L,V,C,L] [E,E,E,E,E] 1.8
1fb4l 136 140 [0,0,0,2,0] [L,V,C,L,I] [E,E,E,E,E] 0.4
1fb4l 137 141 [0,0,2,0,13] [V,C,L,I,S] [E,E,E,E,E] 3.0
1fb4l 177 181 [2,0,1,0,5] [A,A,S,S,Y] [E,E,E,E,E] 1.6
1fb4l 178 182 [0,1,0,5,2] [A,S,S,Y,L] [E,E,E,E,E] 1.6
```

and required 18 seconds to complete on a Sun SPARCstation 1. Loading the protein into the Prolog system (consultation) required an additional 14 seconds.

SECONDARY STRUCTURE DEFINITION

The Kabsch and Sander[8] secondary structure definition algorithm follows a strict hierarchy of structural sub-types. However, the `ks` clauses store only the summary of the structure thus losing the underlying information that goes to make up the defined secondary structure. For example, to define an α–helix, the DSSP algorithm first finds all main chain hydrogen bonds between residues i and $i+4$. These residues are then said to form a *four turn.* A *minimal helix* is then defined in terms of two overlapping four turns, finally a helix is defined as at least two overlapping minimal helices. When the secondary structural state of a residue is simplified to a single character such as 'H', the reason for that residue being defined as 'H' (*i.e.* hydrogen bonds, turns and minimal helices) is lost. Similarly, the possibility of a residue belonging to more than one structural class, for example both 3_{10}–and α–helix, is not allowed by the `ks` summary.

Prolog is a rule based language and the Kabsch and Sander definitions are a set of simple hierarchical rules. Accordingly, it is a relatively straightforward matter to implement the secondary structure definition rules in Prolog. Having the structure definitions expressed as Prolog rules enables queries to be formulated at each level of the definition hierarchy.

Helix definitions

In order to illustrate the definition of secondary structure using Kabsch and Sander's rules the complete Prolog code for the definition of helix structure is shown here. The basic building block for the definition of a helix is the `kturn` rule. This looks up the hydrogen bonding information stored in the `ks` clauses and if an $i, i+n$ pattern with an energy of at least -0.5 KCal is seen, then a turn is defined.

```
/* kturn rule*/

kturn(N,N3,CID,Type):-
     ks(N,_,CID,_,_,_,_,_,_,_,[B1,E1],_,[B2,E2],_,_,_),
     turn_type(NN,Type),
     N3 is N + NN,
     B1A is N + B1,
     B2A is N + B2,
     check_bond(N3,[B1A,E1],[B2A,E2]).

/* turn_type facts */

turn_type(3,three_turn).
turn_type(4,four_turn).
turn_type(5,five_turn).
turn_type(6,six_turn).

/* check_bond rules */

check_bond(N3,[N3,E1],_):-
           E1 =< -0.5.
check_bond(N3,_,[N3,E2]):-
           E2 =< -0.5.
```

The `kturn` rule operates by first looking up the unique residue number, offset and energy of the two hydrogen bonds listed as O-HN hydrogen bonds for that residue. A turn type is then looked up (*e.g.* three_turn) to obtain the required offset for a hydrogen bond (in this example, 3). The `check_bond` rule is then used to see if a hydrogen bond exists between the current residue and one three further down the polypeptide chain.

Given the `kturn`, the definition of a minimal helix is readily expressed as:

```
minimal_helix(N,N2,CID,Type):-
    kturn(N,Nend2,CID,Type),
    NM1 is N - 1,
    kturn(NM1,_,CID,Type),
    N2 is Nend2 - 1.
```

Prolog rules then define the start and end points of a helix:

```
helix_start(N,CID,Type):-
      minimal_helix(N,_,CID,Type),
      NM1 is N - 1,
      \+ in_minimal_helix(NM1,CID,Type).      % \+ means 'not'

helix_end(N,CID,Type):-
      minimal_helix(_,N,CID,Type),
      NP1 is N + 1,
      \+ in_minimal_helix(NP1,CID,Type).

in_minimal_helix(N,CID,Type):-
      nonvar(N),                          %must call with N instantiated
      minimal_helix(N1,N2,CID,Type),      %i.e. N must have a value before
      N >= N1,                            %                       the call.
      N =< N2,
      !.                                  %succeed only once.
```

The rule `in_minimal_helix` succeeds once if the residue number is found within a minimal helix. The `helix_start` rule reads 'residue N is the first residue in a helix if it is the first residue in a minimal helix, and residue $N-1$ is not in a minimal helix.' Similarly for the `helix_end` rule.

Three `helix_type` facts link the different types of turn with 3_{10}–, α–and five-helix names.

```
helix_type(three_turn,three_ten).
helix_type(four_turn,alpha).
helix_type(five_turn,five).
```

Finally, we can write the `helix` rule by making use of the `helix_start` and `helix_end` rules.

```
helix(N1,N2,CID,Htype):-
  helix_start(N1,CID,Type),
  find_helix_end(N1,CID,Type,N2),
  helix_type(Type,Htype).

find_helix_end(N,CID,Type,N2):-
  helix_end(N2,CID,Type),
  N2 > N,
  !.
```

The `helix` rule reads 'find the start of a helix, find the first end of helix that follows this start, then look up the helix type.'

The Prolog rules for helix, make up the computer program that defines helix structures. However, a feature of Prolog is that any rule can be replaced by a collection of facts. For example, the `kturn` rule can be replaced or augmented by facts that are specific to a particular protein:

```
kturn(416,419,1fb4h,three_turn).
kturn(418,421,1fb4h,three_turn).
kturn(418,422,1fb4h,four_turn).
kturn(430,434,1fb4h,four_turn).
kturn(430,435,1fb4h,five_turn).
kturn(431,434,1fb4h,three_turn).
```

these facts may then be used in exactly the same way as the general purpose rule with the same name and number of arguments. In this way, if a rule is particularly time consuming to evaluate, it need only be evaluated once for the protein, then stored as a set of facts. Other rules that subsequently make use of the rule need only look up the corresponding facts in the database, rather than repeat the time-consuming rule evaluation. This principal is similar to storing intermediary results in a conventional Fortran or C program. The difference in Prolog is that the routines that access the pre-calculated data are identical to those that access the routines that initially calculated the data.

Beta Structure Definitions

Kabsch and Sander define all beta structure in terms of 'bridges' which are either parallel or antiparallel. Where two or more bridges of the same type are consecutive, the structure is termed a ladder. Finally, overlapping ladders are amalgamated into sheets. Additional complications arise because ladders may have discontinuities in them, and ladders may consists of just a single bridge. These aspects of protein structure make the coding of beta-structure in Prolog a little less straightforward than for helix.

In finding the sheets in a protein, the following steps are performed:

- Define the parallel and antiparallel bridges.
- Define start and end points of ladders.
- Build ladders of that consist of two or more consecutive bridges.
- Identify ladders that consist of a single bridge (bridge_ladder).
- Locate ladder pairs that are linked by a β–bulge.
- Identify β–sheets in terms of the ladders that make them up.
- Identify β–strands by examining the sheets.

The general Prolog rules developed for β–structure are:

```
bridge(I-J,CID1,CID2,Btype)
ladder_start(A-B,CID1,CID2,Btype)
ladder_end(A-B,CID1,CID2,Btype)
ladder(BridgeList,CID1,CID2,Ltype)
bridge_ladder(ladder([A-B],CID1,CID2,Type))
bulge_linked_ladders(Ladder1,Ladder2)
l_sheet(PID,LadderList,Type)
strand(NumberList,CID)
s_sheet(PID,StrandList,Type)
```

where `I-J` is a pair of residue identifiers from chains `CID1`, `CID2`, respectively. `Btype` is the bridge type, either `parallel` or `antiparallel`. `BridgeList` is a Prolog List containing `bridge/4` facts. Similarly, LadderList is a list of `ladder/4` definitions, whilst NumberList is a list of residue numbers. The `bulge_linked_ladders/2` rule identifies two ladders that are linked by a β–bulge.

Examples of these facts specific to the protein 1fb4 are shown here:

```
bridge(200-205,1fb4l,1fb4l,antiparallel)
bridge(228-340,1fb4h,1fb4h,parallel)

ladder([4-101,4-102,4-103],1fb4l,1fb4l,parallel)
ladder([35-90,36-89,37-88,38-87,39-86],1fb4l,1fb4l,antiparallel)

l_sheet(1fb4,
       [ladder([4-101,4-102,4-103],1fb4l,1fb4l,parallel),
        ladder([8-106,9-106,10-107,11-108,12-109,12-110],1fb4l,1fb4l,parallel),
        ladder([35-90,36-89,37-88,38-87,39-86],1fb4l,1fb4l,antiparallel),
        ladder([36-49,37-47,38-46],1fb4l,1fb4l,antiparallel),
        ladder([85-107,86-106,87-105],1fb4l,1fb4l,antiparallel),
        ladder([89-102,90-101,91-100,92-99,93-98],1fb4l,1fb4l,antiparallel)
       ],mixed)

s_sheet(1fb4,
       [strand([4],1fb4l),
        strand([8,9,10,11,12],1fb4l),
        strand([35,36,37,38,39],1fb4l),
        strand([46,47,48,49],1fb4l),
        strand([85,86,87,88,89,90,91,92,93],1fb4l),
        strand([98,99,100,101,102,103],1fb4l),
        strand([105,106,107,108,109,110],1fb4l)
       ],mixed)
```

Two alternative representations of a β–sheet are shown. The first consists of the list of `ladder` clauses that are used in the definition of the sheet. This representation immediately shows which residues are involved in the sheet and what their hydrogen bonding partners are. The sheet type is also defined as one of pure parallel, pure antiparallel, or mixed, depending upon the type of ladders in the sheet. Having established the sheet definition, it is possible to define the β–strands that make up the sheet and consequently to define the second alternative sheet fact. This shows a list of strand definitions in place of the ladder definitions.

The time required to calculate all secondary structure definitions for a protein is dependent upon the number of residues present, and the total secondary structure content. Some typical examples are: 1fb4 (Immunoglobulin) a β protein of 445 residues in total takes 50 seconds (including consultation time); 1mbn (Myoglobin) an all α protein of 153 residues takes 25 seconds on a SPARCstation 1.

INTERFACE TO TOPOL TOPOLOGY REASONING RULES

The TOPOL system for reasoning about protein topology in Prolog[4] makes use of the secondary structure definitions as deposited in the Brookhaven Protein Data Bank. In order that the TOPOL rules may be applied to Kabsch and Sander derived secondary structure definitions and to allow access to the angle and accessibility information, several rules were developed.

Kabsch and Sander helix definitions allow a residue to belong to more than one type of helix. Thus, it often occurs that a region of α–helix will overlap with a region of 3_{10}–helix at the C-terminal end of the α–helix. TOPOL expects residues to belong to only one secondary structure. Accordingly, where overlaps occur, the helix definitions are compressed into a single `thelix` definition. For example:

```
thelix(186,191,1fb4l,[
                      helix(189,191,1fb4l,three_ten),
                      helix(186,191,1fb4l,alpha)
                     ])
```

shows the starting and ending residue numbers for the concatenated helix, the chain identifier, then a list of the helix definitions that overlap.

Strands are simply identified by their start and end residue numbers, rather than a list of all residues in the strand. In addition, regions of polypeptide that are neither in strand, nor helix are defined as the structure `tloop`. The TOPOL clauses `follows/2`, `is_parallel_to/2` and `is_antiparallel_to/2` are then defined in terms of the tstrand, tloop and thelix clauses. For example:

```
follows(tstrand(175,184,1fb4l),tloop(172,174,1fb4l))
```

specifies that the given `tstrand` follows the `tloop` in the structure, as will be self evident from the residue numbers.

The full interface to TOPOL includes calls from Prolog to Fortran routines to fit straight lines through helices and strands and to calculate overlaps, distances and angles. The details of this interface are under further development and will be described elsewhere.

EXTENSIONS TO KABSCH AND SANDER SECONDARY STRUCTURE DEFINITIONS

Extending β–strands

The basic Kabsch and Sander definitions for beta structure require that a residue must be involved in two hydrogen bonds (or bordered by residues involved in two hydrogen bonds) in order to be classified as in a 'bridge'. Only residues in bridges can be built into ladders, and hence into sheets. However, residues frequently make a single hydrogen bond at the end of a ladder and would traditionally be considered as part of the sheet. The `ladder_extension` clause:

```
ladder_extension(L2,ladder(L1,CID1,CID2,Type)).
```

succeeds if the ladder may be extended to either the 'left' or 'right' end. L2 is the list L1 with the additional hydrogen bonded residue pair appended to the appropriate end or ends.

When Type = antiparallel, the extension is straightforward, *e.g.*: 20-119,21-118 might be extended to 19-120,20-119,21-118, or similarly at the right hand end. However, when Type = parallel, there is a problem, since the additional residue at the end of the ladder will be hydrogen bonded to a residue already in the ladder. As a consequence, the extended ladder list (L2) will contain two references to a single residue on one strand. *E.g.* 20-90,21-91,22-92 is extended at the right hand end to: 20-90,21-91,22-92,22-93, or possibly: 20-90,21-91,22-92,23-92; and similarly for the left hand end.

Sub-classifying β–strands

β–strands may be sub-classified according to their position within the sheet. For example, a strand may be hydrogen bonded on both sides (a mid-strand), or only on one side (an edge-strand). In addition, the strand may be parallel or antiparallel to both its neighbours, or parallel to one and antiparallel to the other. The `strand_type` clause subclassifies β–strands according to these criteria:

```
strand_type(tstrand(X,Y,CID),Sub_type)      % head of the general rule

strand_type(tstrand(323,323,1fb4h),edge_antiparallel)  % a specific example
```

Scanning More than One Protein

The examples have so far assumed that the `ks` clauses for only one protein are resident in the Prolog database. However, a query should be able to be applied to more than one protein at a time. One approach to this problem would be to load `ks` clauses into memory for all proteins to be studied. An alternative approach is to arrange for Prolog to access the clauses as they reside on disk, rather than reading them into memory. This

dilemma is central in the design of large Prolog systems and is a subject of continuing research, some solutions are raised in the discussion.

The limited memory of the Sun-3/50 workstation on which our system was originally developed eliminated the possibility of reading data on all proteins into the Prolog system. Accordingly, a simple solution was adopted whereby each protein is loaded in turn for analysis. In order to economize on disk space, the secondary structure definitions for each protein are not pre-calculated, but performed 'on the fly' as the query is executed. The `scan_with/2` facts, and `get_protein/2` rules manage this operation. For example:

```
scan_with(helix,ScanList), Plist = [1fb4,1sgt,4fxn],member(PID,Plist),
   get_protein(PID,ScanList),
   list of goals using helix definitions go here,
   fail.
```

The `scan_with` fact returns a list of procedures that are to be executed by the `get_protein` procedure. In this example, the `ScanList` returned would take the value of [kturn, minimal_helix, helix_start, helix_end,helix], specifying the rules for structural units that are to be used. `Plist` is simply a list of the identifiers for the proteins that are to be analysed. The `member/2` rule returns successive members of the `Plist` on backtracking, and thus feeds each value of PID in turn to the `get_protein/2` procedure.

A call to `get_protein` first loads the `ks` clauses for the specified protein, then loads the general rules for helix definition (kturn, minimal_helix, *etc*....). All solutions to these general rules are then found for the protein and the specific structural facts asserted into the Prolog database. Having loaded all specific facts for the protein, the particular goals that require the secondary structure definitions are executed.

ALL ATOM REPRESENTATION IN PROLOG

Although many questions may be answered by regarding the protein structure at the residue level, some analyses require access to the individual atomic coordinates. For example, the location of close approaches between residue sidechains to identify hydrophobic or electrostatic interactions. The analysis of all atoms creates several additional complications:

- The need to cope with a greatly increased diversity of atom labelling.
- Keep record of which atoms belong to which residue and distinguish between atoms that are in the protein chain, and those that are not.
- Cope with non-protein atoms and groups that are often part of a Brookhaven coordinate entry: *e.g.*, water molecules, haeme, carbohydrate *etc*....
- Combinatorial problems: *E.g.* searching for all close approaches is time consuming because there roughly 10x as many atoms as $C_\alpha$atoms...
- Storage and memory problems: Full coordinate sets take up a lot of space.

A simple strategy for the representation of all-atom sets in Prolog was adopted, whereby each atom is represented by a Prolog fact of the form:

```
brk(I,RN,IN,ATYPE,CID,RTYPE,ATTYPE,XYZ)
```

where `I` is the atom number, `RN` is the residue number (*e.g.* 2), `IN` is the residue number insertion code (*e.g.* "-" for no insertion code); `ATYPE` is either `atm`, or `het` , for protein ATOM or HETATM records; `CID` is the chain identifier code (*e.g.* "1fb4l"); `RTYPE` is the amino acid type in three letter code (*e.g.* val); `ATTYPE` is the atom type as a list including the atom insertion code (*e.g.* `[cg1,-]`) and `XYZ` is the atomic coordinates as a list. In the current implementation, the temperature factor, occupancy and footnote fields are not included.

The PDB CONECT records are converted to `bond` clauses where each clause has the form:

```
bond(I,J,Type)
```

signifying a bond between atoms I and J of type Type. Type may be one of the following:
covalent
hbond_da (I is donor, J is acceptor in hydrogen bond)
saltb_neg (I is negative partner in salt bridge)
hbond_ad (I is acceptor)
saltb_pos (I is positive partner)

This format of a PDB entry may be used directly for analysis in Prolog. For example, given the rule rdist/3 which returns the linear distance between two points in space, we can readily calculate distances between any pair of atoms, simply by typing:

```
| ?- brk(I,RN1,IN1,ATYPE1,CID1,RTYPE1,ATTYPE1,XYZ1),
     brk(J,RN2,IN2,ATYPE2,CID2,RTYPE2,ATTYPE2,XYZ2),
     J > I,
     rdist(XYZ1,XYZ2,Distance).
```

which returns as the first solution:

```
I = RN1 = RN2 = 1,
IN1 = IN2 = -,
ATYPE1 = ATYPE2 = atm,
CID1 = CID2 = 5chaa,
RTYPE1 = RTYPE2 = cys,
ATTYPE1 = [n,-],
XYZ1 = [40.935,13.504,1.417],
J = 2,
ATTYPE2 = [ca,-],
XYZ2 = [40.345,14.599,2.14],
Distance = 1.43871
```

It is a simple matter to restrict the distance search to all atoms of a particular type. For example, to search for close approaches between cys sulphur atoms:

```
| ?- brk(I,RN1,IN1,ATYPE1,CID1,cys,[sg,_],XYZ1),
     brk(J,RN2,IN2,ATYPE2,CID2,cys,[sg,_],XYZ2),
     J > I,
     rdist(XYZ1,XYZ2,Distance),
     Distance < 5.

I = 6,
RN1 = 1,
IN1 = IN2 = -,
ATYPE1 = ATYPE2 = atm,
CID1 = CID2 = 5chaa,
XYZ1 = [37.649,15.819,1.913],
J = 893,
RN2 = 122,
XYZ2 = [36.339,14.497,2.687],
Distance = 2.01565
```

or perhaps, to identify close approaches between water molecules and glutamate residues and write out the findings in a Prolog clausal form.

```
brk(I,RN1,IN1,het,CID1,hoh,ATTYPE1,XYZ1),
     brk(J,RN2,IN2,atm,CID2,glu,ATTYPE2,XYZ2),
     rdist(XYZ1,XYZ2,Distance),Distance < 3,
     writeq(water_glu(water(I,[RN1,IN1],ATTYPE1,CID1),
                      glu(J,[RN2,IN2],ATTYPE2,CID2),Distance)),
     nl,fail.
```

```
water_glu(water(3603,[554,-],[o,-],5chaa),glu(123,[20,-],[oe1,-],5chaa),2.93873)
water_glu(water(3606,[557,-],[o,-],5chaa),glu(2264,[70,-],[cb,-],5chab),2.98971)
water_glu(water(3638,[589,-],[o,-],5chaa),glu(1898,[21,-],[ca,-],5chab),2.76785)
water_glu(water(3638,[589,-],[o,-],5chaa),glu(1899,[21,-],[c,-],5chab),2.90702)
water_glu(water(3638,[589,-],[o,-],5chaa),glu(1902,[21,-],[cg,-],5chab),2.49055)
water_glu(water(3644,[595,-],[o,-],5chaa),glu(492,[70,-],[cb,-],5chaa),2.85485)
water_glu(water(3648,[599,-],[o,-],5chaa),glu(551,[78,-],[cb,-],5chaa),2.77389)
water_glu(water(3663,[614,-],[o,-],5chaa),glu(2263,[70,-],[o,-],5chab),2.87087)
water_glu(water(3680,[631,-],[o,-],5chaa),glu(1895,[20,-],[oe1,-],5chab),2.3201)
water_glu(water(3723,[674,-],[o,-],5chaa),glu(120,[20,-],[cb,-],5chaa),2.8711)
water_glu(water(3723,[674,-],[o,-],5chaa),glu(125,[21,-],[n,-],5chaa),2.8872)
water_glu(water(3724,[675,-],[o,-],5chaa),glu(2121,[49,-],[oe2,-],5chab),2.20559
)
```

Consulting (loading into the Prolog system) the 3719 `brk/8` clauses for protein `5cha` took 46 seconds. The query then required 75 seconds to run. When the `brk/8` clauses were compiled into the Prolog system, the execution time was reduced to 30 seconds. Unfortunately compilation required 162 seconds, leading to a net loss in overall execution time.

The ease with which these simple queries can be executed in Prolog, belies the complications that would be necessary to provide such flexibility in a conventional Fortran or C program. As for Prolog, the conventional program would first have to read in the complete dataset into the chosen internal representation of the data. A general purpose command parser would need to be written to enable the operator to tell the program which comparison was required. A general selection routine would also be required to enable the operator to choose which subset of atoms are required for the comparison. Whilst all these routines could certainly be provided in a Fortran program, Prolog provides a far more concise route to such analyses.

DISCUSSION

In this paper we have described the use of Prolog to represent and manipulate protein structure and illustrated the use of the system to refine the Kabsch and Sander definitions of β–structure. As it stands, the system is a practical tool that offers flexible access to structural information at all levels of the protein structural hierarchy. The system is fast enough to enable scans to be made of subsets of the database at the residue level, however, for simple queries, the time required to load each protein into Prolog (consultation) dominates the scan time. For example, although 18 minutes was required to scan 94 proteins for the amino acid sequence Gly-Gly and return the secondary structure summary and accessibility for the residues, 94% of the time was used for consultation. The time required to consult all 525 proteins in the current databank at the residue level is approximately 90 minutes, whilst scanning with all atoms would require approximately 5 hours consultation time on a SPARCstation 1.

There are a number of possible ways in which the consultation time could be reduced. The most common, and that usually offered by Prolog vendors is an interface to a relational database such as Oracle. These interfaces

are described as providing a loose coupling to Prolog, since they effectively replace the standard file-based methods of retrieving Prolog facts (ground clauses) into the Prolog internal database. Access to SQL is provided from within Prolog, and these hybrid database/Prolog systems have been shown to be effective when Prolog is used to simplify access to an underlying database, or where the database retrievals are infrequent. A number of examples of this approach can be found.[10, 11]

Although the loosely coupled interface to an RDBMS provides an engineered solution to the problem of managing large collections of data from a Prolog programming environment, a much better solution is to use a tightly coupled approach. Tight coupling between logic programming languages and large storage management systems exist in the class of systems called deductive databases[12] or expert databases.[13] These systems are programming and data management systems based on principles of symbolic logic and do not require the user to access data via a standard query language such as SQL. The database and the deductive engine co-exist using common storage and execution models. This is a much more satisfactory approach, and in our view the best suited to applications in protein sequence and structure analysis.

Object-oriented databases (OODB) are perhaps a better known way to combine a computational paradigm with a database. OODBs combine the object-oriented programming style of specifying methods, and passing messages to activate methods stored in objects to manipulate data stored as properties of objects. In OODBs the object classes, objects and methods are maintained in a persistent storage system. A good example of the use of OODBs in the domain of protein structure is that of Gray *et al.* [3] who implemented their OODB in Prolog augmented with a custom-built object storage module. Although OODBs and deductive databases aim to deliver similar functionality to the user, the deductive database approach is more suitable for the development of knowledge-based systems because both data representation and the computational paradigm are based on well-founded theories of symbolic logic. Data and rules of deduction can also be freely intermixed whereas no equivalent theoretical basis exists for OODB and the traditional distinction between data and program in imperative programming languages is preserved.

Putting the database scanning problem to one side, the examples shown in this paper illustrate that Prolog is a useful tool for the analysis of protein structure. Once the relevant Prolog clauses have been loaded, queries regarding one protein can be evaluated in a few seconds. Indeed, it is possible to browse the protein structure, examining distances, angles, hydrogen bonds *etc.* with a simplicity that would be difficult to rival by conventional programming means. Unfortunately, in order to take advantage of these benefits with the current implementation, one needs to learn Prolog, and even seasoned "C" or Fortran programmers usually find this a barrier. The provision of a toolkit of high level functions specifically aimed at protein structure analysis, for example, torsion angle/distance calculation, extraction of helices *etc.* greatly reduces this barrier. Alternatively, the use of higher level languages developed with a particular problem domain in mind (*e.g.* Daplex,[3]) can ease the transition to Prolog-like systems. Ultimately, developments to graphical interfaces which can allow inexperienced users access to data structures describing complex concepts such as protein topology[14] will provide flexible access to Prolog-level queries.

REFERENCES

1. Islam, S. A.; Sternberg, M. J. E. "A Relational Database of Protein Structures Designed for Flexible Enquiries about Conformation". *Protein Eng.* **1989**, *2*, 431–442.

2. Huysmans, M.; Richelle, J.; Wodak, S. J. "SESAM: A Relational Database for Structure and Sequence of Macromolecules". *Proteins: Struct., Funct., Genet.* **1991**, *11*, 59–76.

3. Gray, M. D.; Paton, N. W.; Kemp, G. J. L.; Fothergill, J. E. "An Object Orientated Database for Protein Structure Analysis'. *Protein Eng.* **1990**, *3*, 235–243.

4. Rawlings, C. J.; Taylor, W. R.; Nyakairu, J.; Fox, J.; Sternberg, M. J. E. "Reasoning about Protein Topology Using the Logic Programming Language PROLOG". *J. Mol. Graph.* **1985**, *3*, 151–157.

5. Clark, D. A.; Barton, G. J.; Rawlings, C. J. "A Knowledge-Based Architecture for Protein Sequence Analysis and Structure Prediction". 1990, *J. Mol. Graph.* **8**, 94–107.

6. Clark, D. A.; Rawlings, C. J.; Barton, G. J. "Knowledge-Based Orchestration of Protein Sequence Analysis and Knowledge Acquisition for Protein Structure Prediction". *Proc. Am. Assoc. Artif. Intel.* **March 1990**.

7. Clark, D. A.; Shirazi, J.; Rawlings, C. J. "Protein Topology Prediction through Constraint-Based Search and the Evaluation of Topological Folding Rules". *Protein Eng. in press.*

8. Kabsch, W.; Sander, C. "Dictionary of Protein Secondary Structure: Pattern Recognition of Hydrogen-Bonded and Geometrical Features". *Biopolymers* **1983**, *22*, 2577–2637.

9. Bernstein, F. C.; Koetzle, T. F.; Williams, G. J. B.; Meyer, E. F.; Brice, M. D.; Rodgers, J. R.; Kennard, O.; Shimanouchi, T.; Tasumi, M. "The Protein Data Bank: A Computer-Based Archival File for Macromolecular Structures". *J. Mol. Biol.* **1977**, *112*, 535–542.

10. Gray, P. M. D.; Lucas, R. J. *Prolog and Databases.* Ellis Horwood, Chichester, 1988.

11. Deen, S. M.; Thomas, G. P. *Data and Knowledge Base Integration.* Pitman, London, 1990.

12. Grant, J.; Minker, J. "Deductive database theories'. *Knowledge Engineering Review* **1989**,*4*, 267–304.

13. Kerschberg, L. *Expert Database Systems.* Benjamin/Cummings, Menlo Park, California, 1986.

14. Seifert, K. L.; Rawlings, C. J. *GRIPE: A graphical Interface to a knowledge based system which reasons about protein topology*, pages 391–406. Cambridge Press, 1988.