Database Integration and Querying in the Bioinformatics Domain

Bob Myers[1,2,5], Trevor Dix[1,2], Ross Coppel[1,3], David Green[1,4]
1:Victorian Bioinformatics Consortium.
2:School of Computer Science and Software Engineering, Monash University, Australia
3:Dept. of Microbiology, Monash University, Australia.
4:Faculty of IT, Monash University, Australia.
*5:*bob.myers@med.monash.edu.au

Abstract

Given the exponential growth in the amount of genetic data being produced, it is more important than ever for researchers to have effective tools to help them manage this data. This paper describes a system that enables users, generally biologists, to construct components to answer specific questions in their field. The system allows the creation of modules and submodules via top-down decomposition. Concepts and terms can be defined through conversation. These are then used when composing base-level functions to produce code for modules and for interfacing modules.

1. Introduction

For more than a decade the quantity of genetic data produced each year has been growing exponentially. GenBank's database has grown from just over 217 million base pairs in 1994 to more than 44 billion in 2004 [1], increasing on average by a factor of 1.7 each year. PubMed now contains over 15 million citations [2] and is growing by approximately 450,000 each year. Not only is the quantity of data increasing but so too is the number of databases, web-sites and other information sources that a researcher must cope with. Also, given the ever increasing number and speed of modern DNA sequencing machines, micro-arrays and other devices, it seems unlikely that the flood will abate any time soon. Faced with such an information overload, researchers need computerized tools to assist them in making best use of the available data.

The basic problem is to build a system that enables users, generally biologists, to construct components to answer specific questions in their field. The following list provides examples of questions in the context of the malaria parasite *Plasmodium falciparum*:

- Find a list of all proteins that have predicted trans-membrane anchor sequences and are expressed in asexual blood stages.
- Find secreted proteins that have disulphide bonds.
- Find proteins with repeats.
- Find allelic variants of this surface protein that have been identified in Thailand.
- Find homologs in other species and home-in on conserved sequences.

The authors created a prototype of a system intended to fulfill some of these needs. Based on a workflow style graphical interface where the user drags modules from a palette of tools onto a drawing frame and then connects the outputs of some modules to the inputs of other, as shown in figure below.

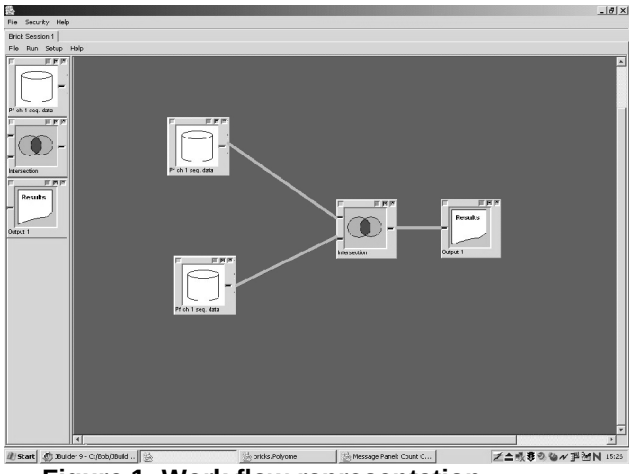

Figure 1: Work flow representation

Sufficient modules were programmed to enable the system to answer some real-life biological questions in the domain mentioned above. In the course of developing the

system some significant problems were encountered including:

- Modules tend to be extremely complex, requiring much time and expertise to build.
- Some of the modules were very fragile. A slight change in a data source, for example, could cause the module to fail.
- A new module has to be created for each new data source.

This prototype is described in more detail in section 2.

To overcome these problems it was decided to break the large modules down into many small units, each one representing a very basic piece of functionality; for example, read an HTML page, send an SQL query to a relational database, find a substring of some text etc. Because each module is now trying to do much less, it requires fewer lines of code and is less complex. There are fewer ways in which a small module can go wrong, making it easier to trap errors. There will of course be more modules. However, the system allows decomposition of modules into submodules. Indeed most common workflow modules will be sequential, permitting a functional definition for each in terms of input(s) and output(s).

The creation of modules and submodules is typical of top-down decomposition. This allows consideration of (sub)modules in isolation, thus keeping complexity manageable. The coding within (sub)modules is a mixture of top-down and bottom-up. The system allows concepts, terms etc to be defined through conversation. However the base-level functions are composed in a bottom-up manner applying to the defined terms. Section 3 presents the conversational process within the system.

2. Related Methods

The problem of data integration has been approached in many ways, some of which are described below. There is a degree of overlap between some of these approaches, and many solutions to the problem of data integration consist of combinations of these approaches.

- Data Warehousing: Data is extracted from its original location and added to a common, centralized database. The main problems with this approach is that the quantity of data is so large that it would require massive amounts of computing power to handle it, and there are so many different types of information sources that a vast programming effort would be required to write the extraction routines.

- Standardised Databases: Several attempts have been made to come up with a standard (relational) database structure to suit all purposes within the Bioinformatics field. Examples of this are GUS[3] (used for PlasmoDB, AllGenes, EPConDB, GeneDB etc.), ACEDB[4] (used for WormBase, DictyDB etc) and GMOD[5] (used for WormBase, FlyBase, MGI, SGD, Gramene, Rat Genome Database, EcoCyc, and TAIR). The main problem with this approach is finding a format that suits everyone, otherwise leading to a proliferation of standards, defeating the original purpose.

- Federated Databases: Data is not amalgamated into a data warehouse, instead it remains in its original location and is retrieved on demand. This approach requires interfaces to each data source to be constructed. This is manageable if all the data sources are of a similar type, e.g. relational databases, but it can become impractical if the data sources vary wildly in type and structure, requiring expertise in both data analysis and the subject matter of the data sources.

- Web Services: Applications are made available for automatic use by other systems via a network. The main problem here is that each data source would need to provide its own services and make them generally available.

- Database Wrappers: Similar to the idea of federated databases, this consists of a piece of software that interfaces between the user and the target database. This software receives queries and returns replies in a standard format, enabling the user to query multiple data sources in the same way regardless of the target's type. This concept requires the software to be specifically built for each target, requiring expertise in data analysis and the subject matter.

- Semantic Web: Consists of annotating data sources in a standardized, machine readable way, so that systems can extract meaning from them (e.g. the types of entity represented in the data and how they relate to each other). Contrast this to the World Wide Web, where the data sources (web pages) have content, but little machine readable meaning. Few bioinformatic data sources have done this so far, although a few do provide data in XML as well as HTML.

Our initial model for an information system was based on a combination of the above approaches, and was developed from the observation that people often draw flow diagrams to describe the information flow and processing that they wish to perform. The system interface consists of a palette of tools and a drawing pane, as shown

in figure 1 above. This work-flow type of interface has been used in a number of other systems, for example, DiscoveryNet[6].

Each tool is written as a separate module to perform a specific task, like read from a specific database or combine two sets of data etc. Each tool may have data-streams as inputs and outputs, and a set of parameter type inputs used to tailor the specifics of its functionality. To perform some work the user drags tools from the palette onto the drawing frame, sets the values of any parameters and then joins the output data-streams of modules to the input data-streams of others to produce a work-flow type diagram. When the work-flow is executed, all modules with no input data-streams (generally functions like reading from data sources) are run first, each in a separate execution thread, and their output data-streams are passed onto the appropriate modules as indicated by the arcs in the diagram. When a module has received a sufficient amount of its input data-streams, it too runs and so on until all modules have been executed. The terminal modules of the work-flow are generally those that produce some sort of output for the user (writing a file, producing a printout etc) and so do not pass their output data streams onto another module. The example of a simple work-flow in figure 1 shows two databases being read, their outputs combined and displayed on the screen.

The initial system was set up with several representative modules, including:

- Run a TMPRED query (Prediction of Transmembrane Regions and Orientation) at EMBNet in Switzerland (http://www.ch.embnet.org/software/TMPRED_form.html).
- Run a SignalP query (presence and location of signal peptide cleavage site prediction in amino acid sequences) at the Center for Biological Sequence Analysis at the Technical University of Denmark (http://www.cbs.dtu.dk/services/SignalP-2.0/).
- Count amino acids in a protein and selecting proteins with counts in specified ranges.
- Create the intersection of two sets of proteins.
- Read a number of protein sequences from a FASTA format file.
- Format protein data for screen output.

The size and functionality of the modules were chosen based on the level of breakdown that a biologist would use when thinking of a real-world problem. For example, a question might be: *"Which secreted proteins have disulphide bonds?"*, and a biologist could break this down into the following parts:

a) Get protein sequences from file x.fasta
b) Count the number of Cysteine amino acids in each protein, keep only the proteins with 2 or more.
c) Feed those proteins into the SignalP service with parameters:
   i. Truncate at 50 residues
   ii. Organism group = Eukaryotes
   iii. Method = HMM

   and keep only those proteins that return a positive result.
d) Feed those proteins into TMPRED with parameters:
   i. Minimum length of hydrophobic part of trans-membrane helix = 17
   ii. Maximum length of hydrophobic part of trans-membrane helix = 33

   and keep only those that return a negative result.
e) Display those proteins.

It is easy to see how the modules listed above correspond to the steps in the process, which would be represented graphically as in figure 2:

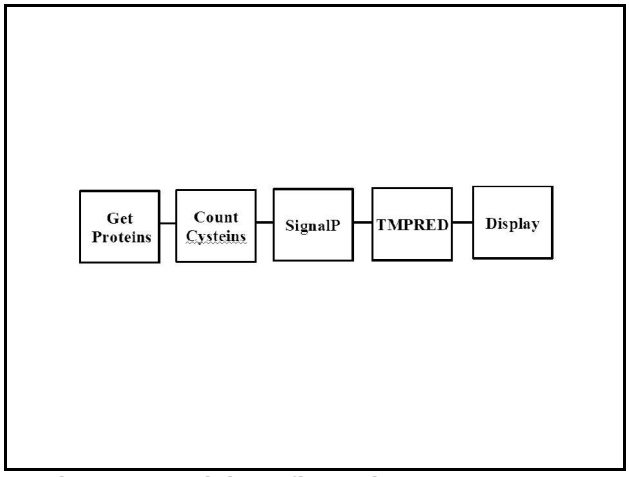

Figure 2: Serial configuration

It might also be advantageous to perform some of the steps in parallel to take advantage of distributed processing if tasks run on remote processors (as in the case of SignalP and TMPRED), in which case the process could be represented as in figure 3:

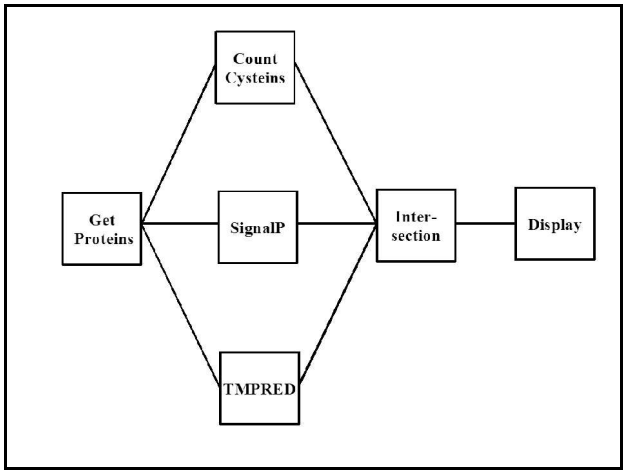

Figure 3: Parallel configuration

Although the system worked well, producing the expected results in a reasonable time-frame, a number of problems were found:

- **Complexity**: Many of the modules were very complex, requiring much time, effort and knowledge about the biological subject matter and computing techniques, making them too difficult for a biologist to produce.

- **Fragility**: Some of the modules (particularly those that interfaced with external systems) were very susceptible to minor changes in the target system, causing it to either fail or yield incorrect results. Because of the complexity of the module they also contain many possible points of failure. The number and variety of ways that a module could fail made trapping the errors and taking appropriate action difficult.

- **Variability**: A new module would need to be created for each new data source, and even sometimes for different questions posed to the same data source. Given the explosion in the number and types of bioinformatic data sources it would require a very significant effort to keep up.

These problems indicate that the development and ongoing maintenance load required for such a system would be impractical to maintain.

3. A Solution: PolyOme

To overcome the problems listed in section 2, it was decided to breakdown the high level modules, like "Run a SignalP query..." into base-level modules like "Get an HTML page" or "Run a remote CGI script". Smaller, simpler modules with limited functionality have the advantage of being:

- easier, quicker and cheaper to build,

- reusable, only need to be written once but can be used in many combinations with other small modules, and

- more robust, simpler modules have fewer ways in which they can go wrong, making error trapping easier.

However, having many small modules introduces another problem. It became correspondingly more difficult for the user to join together a large number of small modules compared to a small number of large ones. Essentially, the skill that the developer of the large module had used in writing it, now had to be shown by the user in linking the smaller modules together. Clearly another method of linking the modules was required.

After considering a number of GUI type interfaces, it was decided to use a text based interface, with English words as input. A text based interface would be superior in several ways:

a) It would be more flexible, allowing the system to cope with a wide variety of questions, statements and data sources.
b) Details of the workings of the functional modules could be hidden from the user.
c) The system could resolve ambiguities or request further information from the user via the same interface, establishing a conversation with the user.
d) The system could use the same process to handle non English type inputs such as HTML, XML, comma delimited data etc. enabling it to not only handle user input, but also to process the information returned from a data source.

It was decided not to use a full-blown Natural Language Processing (NLP) system because it would be unnecessarily complicated. The domain is simpler, not needing full NLP. Also, full NLP would not permit the handling of HTML, XML and other non English information.

PolyOme is essentailly a mechanism by which the user can compose submodules from base-level units and higher level modules from submodules by means of a conversational process. The general architecture of PolyOme consists of the user interface, a database, a

library of basic functions and a set of processes, see figure 4.

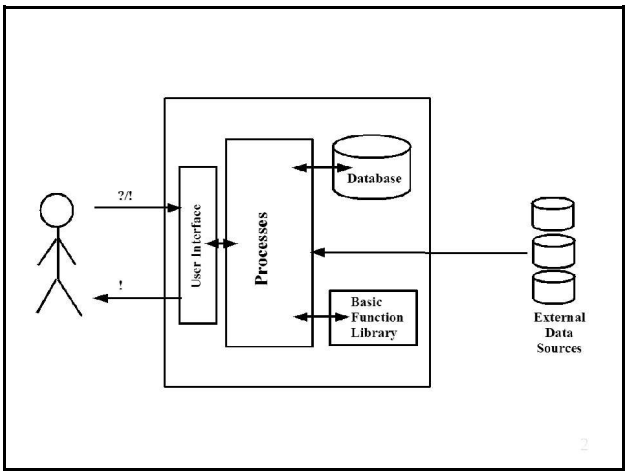

Figure 4: PolyOme general architecture.

The system is implemented as a single user, single database structure for the sake of simplicity.

- **User Interface:** The user interface is implemented as a simple two part screen, the user types input at the bottom and the system's reply appears at the top, as shown in figure 5.

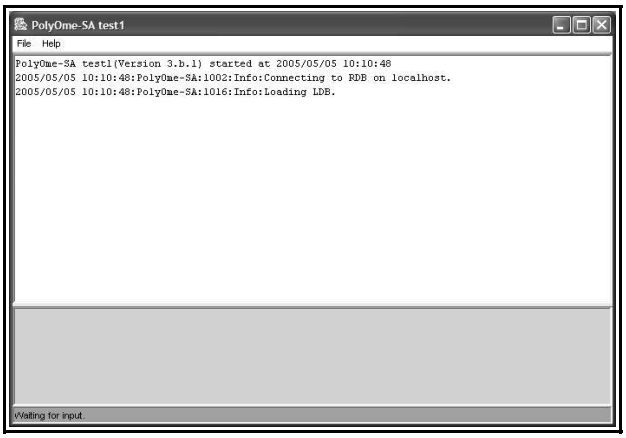

Figure 5: User interface

- **Database:** The database consists of two parts, a relational database (RDB) and a logic database (LDB). The LDB is used to contain facts about the subject matter and the RDB is used to record part-of speech tags and translation rules. Although the database could have been constructed using either a LDB or a RDB alone, a combination of the two was chosen so that each could be used for the tasks to which it is best suited. A LDB (implemented here in Prolog) is good for storing a wide variety of information, having no strict field/table structure, and the logic engine is good at inferring information from the stored facts. Both of these are difficult to do in a relational database. However, Prolog tends to be slow, so tasks that require little or no use of the inferencing engine and have well defined data structures would be better performed by the RDB.

- **Basic Function Library:** The basic function library consists of a number of Prolog functions, and provides the basic capabilities of the system, such as reading relational databases, reading HTML pages etc. Some of these are programmed as Java classes, which are then registered with the Prolog engine as 'built-in' functions, and some are written as pure Prolog.

  These functions are purely for internal use within the system. They are hidden from the user by the User Interface and the process ProcessMsg, which is described below.

  The following two Java functions provide the system with the ability to read any ODBC data source and to add that information into the LDB.

  readODBC/4, which reads an ODBC data source. Given the data source name, table name and a list of fields it returns a list of values corresponding to the rows and columns extracted from the table.

  factise/3, which converts a list of row and column values (as returned by readODBC/4 for example) into facts which are added to the LDB.

  An example of a pure Prolog function is xisa/2:

  ```
  xisa(X,Y):-isa(X,Y).

  xisa(X,Y):-isa(X,Z),xisa(Z,Y)
  ```

  The function isa(X,Y) corresponds to the English statement "X is a Y". xisa(X,Y) extends this to say that if "X is a Z" and "Z is a Y" then "X is a Y". For example, if it is known that isa(human,mammal) and isa(mammal,vertebrate) (i.e. "human is a mammal and a mammal is a vertebrate) then the question "isa (human,vertebrate)" gets the answer "No", however asking "xisa(human,vertebrate)" gets the answer "Yes".

  Several other functions have been written in Java to assist in the system's processing. These include:

checkPoint/1, to take a snapshot of the LDB.

rollBack/1, to restore the LDB from a previous checkPoint snapshot.

These functions are required to enable the system to test sets of updates to the LDB and back them out if required. checkPoint/1 is also used to ensure LDB persistence from one user session to the next.

- **Processes:** The main process, Converse, is responsible for conducting a dialog with the user, see figure 6 below.

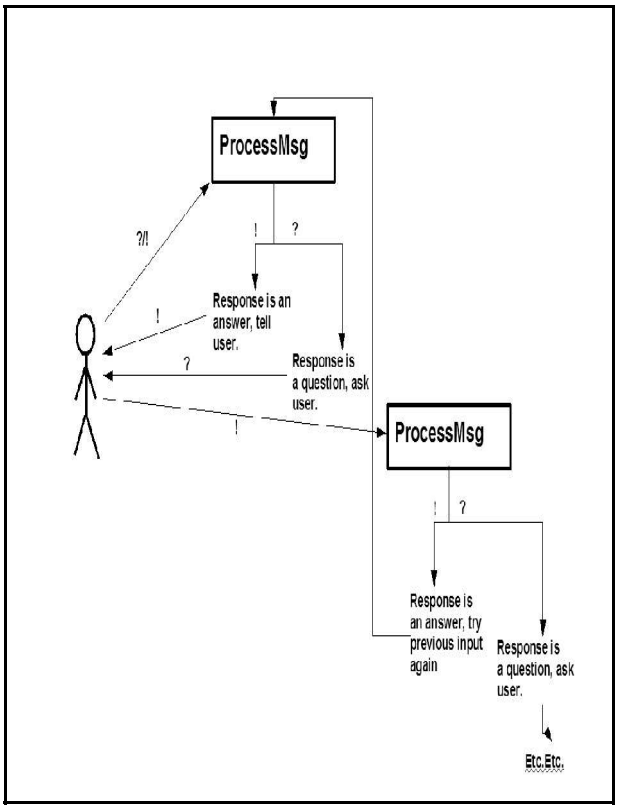

Figure 6: Converse.

The user enters some input, which may be a question or a statement, and this is passed to a sub-process; ProcessMsg (the workings of which are discussed below). Having processed the input, ProcessMsg returns either a statement or a question. If it is a statement, it is passed to the user and the conversation ends. If it is a question, it is also passed back to the user, but in this case the system places the original input on a stack and waits for a reply from the user. When the system receives a reply it is passed to a new invocation of ProcessMsg. This second invocation of ProcessMsg can also return either a statement or a question. If a question is returned, the (second) input is stacked, the question is sent to the user, etc on down through another level. If a statement is returned, this too is sent to the user and the system pulls the top message from the stack and passes it to ProcessMsg where it is processed again from scratch.

In this way the system conducts a conversation with the user, successively seeking clarification to anything that it does not know how to handle.

The sub-process ProcessMsg is responsible for taking one input at a time, converting it into a set of calls to functions from the Basic Function Library and executing these functions. This is done in the following steps:

- The words are split up and associated with all known part-of-speech tags from the RDB. For example, the input:

  "rip is a protein"

  gives the combinations of patterns:

  (rip,noun),(is,verb),(protein,noun)

  (rip,verb),(is,verb),(protein,noun)

- Each pattern is then matched, word by word, to a set of translation rules from the RDB. Each translation rule associates a pattern of words to one or more basic functions. E.g:

  (%sub%,noun),(is,verb),(%obj%,noun) = assert(isa(%sub%,%obj%))

  The left side of the rule specifies the word patterns to which it matches (note the use of named variables, %sub% and %obj%) and the right side specifies the corresponding basic functions.

  It is possible to have several patterns for an input, and for each of these to match to several translation rules. Hence an algorithm is applied to all the possible interpretations of the input to decide which is the most appropriate. For example, the second pattern would not match this rule. In fact it is unlikely that a rule for a verb/verb/noun pattern would exist, so this interpretation would be discarded.

After matching, the actual values from the pattern are substituted for the appropriate variables in the right side of the rule, giving a prolog statement:

assert(isa(rip,protein))

Note that for some patterns it is possible that a series of more than one rule may apply, i.e. rule1 matches the first 4 words of the pattern and rule2 matches the next 5 and so on.

- The Prolog clauses are then passed to the Prolog engine for processing. In this example it adds one fact to the LDB.

Note that the system is not attempting to understand the input. It is merely determining the most likely association between the input and a set of basic functions based on the rules it has in its database. The system does not impose a specific syntax on the user. It is also possible for the system to accept other forms of input, for example, structured text such as XML or HTML, by adding appropriate translation rules to the database.

4. Future Work

The essence of the converstaional engine has been presented. The following are yet to be investigated fully:

- Scaling: It is not yet clear how well the Prolog engine will cope with a significant increase in the size of the LDB.
- Scoring algorithm: As more and more translation rules are added to the database the scoring algorithm may need to be changed in order to adequately differentiate between them.
- Automatic generation of translation rules: For the system to 'grow and learn' it will be necessary for ProcessMsg to be able to add appropriate translation rules to the database.
- Fault tolerance: Improve the mechanism for handling failure within a basic function.
- Bulk load part-of-speech tables and LDB: To give the system a head start it is intended to bulk-load several parts of the database from various online dictionaries and ontologies.
- Build primitive function library: What the system is able to do is limited mainly by the contents of the basic function library. It will be necessary to write an extensive set of routines.
- Make the system available to a number of biologists to assess its usefulness.

5. Conclusion

The outline of a system for the integration and querying of data sources in the bioinformatic domain has been given. It has a simple, intuitive user interface, and is capable of being extended to cope with new data sources. A mechanism for the composition of high level modules from base-level units using a conversational process has also been presented. The full system is still under development. It already performs reasonably well on simple tasks and statements like learning facts from conversation to add to the LDB where appropriate, being able to read data from a (relational) data source and integrate it into the LDB, and to answer simple user queries on that data. Further development is required to demonstrate the system's full potential and to investigate the effects of increases of scale.

6. References

[1] NCBI, "GenBank Growth", http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html, 2005

[2] NCBI, "Entrez PubMed", http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=PubMed, 2005

[3] CBIL and others, "The GUS Platform", http://www.gusdb.org, 2005

[4] Sanger-Institute, "The Sanger Institute: AceDB Database", http://www.acedb.org, 2005

[5] GMOD, "Generic Model Organism Database Construction Set", http://www.gmod.org/, 2005

[6] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo, "The discovery net system for high throughput bioinformatics," Bioinformatics, vol. 19, pp. 225i-231, 2003.