A Study on Development of a Deductive Object-Oriented Database and Its Application to Genome Analysis

Susumu GOTO

January 1994

Abstract

A deductive object-oriented database (DOOD) is the integration of a deductive database (DDB) and an object-oriented database (OODB). DDB and OODB are next generation databases proposed for overcoming the shortcomings of relational databases (RDB).

The DDB is an extension of the RDB. It is based on the first order predicate logic, and provides a declarative query (and programming) language. An advantage of the DDB is that the recursive query processing can be easily expressed. However, it is associated with the following problems.

- Many methods have been proposed for recursive and negative query processing. However, there are few systems implementing these methods and almost no practical applications. Therefore, it is not clear whether the methods are applicable to and useful for practical applications.
- The DDB can handle nested relations by using functions. However, nested relations are not sufficient for representing the data in practical applications.

The OODB provides a data model that can represent a variety of data used in practical applications. Recently, many OODB management systems have been developed and used in practical fields, such as computer-aided design, manufacturing and engineering. However, the following weaknesses have been noted.

- A lack of declarative query languages: Application programs should be written in procedural languages, such as C++ and Smalltalk.
- Difficulty in representing and managing incomplete knowledge: Users must write application programs in procedural languages to handle incomplete knowledge, and it is difficult to manage them in the database.

The DOOD was proposed to integrate the DDB and OODB. It was intended to possess the advantages of both the DDB and the OODB; that is, declarativeness and an object-oriented data model. Current problems with the DOOD are that:

- there is no consensus for a framework for integrating these two databases;
- it is not clear what kind of techniques in the DDB are necessary for the integration.

It was with the above issues in mind that this project was initiated. First, we developed a prototype of a query evaluator of a deductive database and applied it to the following practical problems.

- a secondary structure search in RNA sequences of human genome, and
- configuration detection problems in CAD systems of petrochemical plants.

The genome of an organism consists of its set of chromosomes, which includes all of its genetic information. Genome analyses require at least the following three functions: 1) management of a huge volume of data produced by experiments, 2) representation of users' biological knowledge, and 3) flexible retrieval of data using the knowledge. Such flexible retrieval is also required in CAD systems.

We evaluated query processing methods for recursive queries and negative queries from the viewpoint of expressive power and efficiency, using the above applications. We also designed and implemented classes in an OODB for genome data (data in GenBank nucleotide sequence databank) and used them for retrieval using various search conditions.

Based on the above evaluation, we discussed what kind of framework would be necessary for integrating a DDB and an OODB. The main finding was the necessity of OODB functions for data management and DDB functions for query description.

We developed a system in which an OODB is used to create instances and define methods and a deductive language is integrated to the OODB. Users can handle objects and call methods using the language. They can also represent biological knowledge and can retrieve data using this knowledge.

We applied this system to genome analyses. The combination of several searches, such as a keyword search and a similarity search, can be performed easily. Integration of several genome databanks, which were developed independently, can be easily achieved. On the other hand, it is difficult to manage

users' knowledge in the OODB within the proposed framework. This is one unresolved problem.

In this work, the following results were obtained.

1. We showed that the recursive queries and negative queries in the DDB can be used to solve many practical problems and that efficient query processing can be achieved by selecting appropriate methods.

2. We proposed a subgoal ordering method in deductive databases. The method first determines how bindings are passed in the query evaluation, and then transforms the given rules. We demonstrated the efficiency of the method by performing evaluation with some test data.

3. We discussed what kind of framework is necessary for integrating a DDB and OODB in terms of the practical problems, and proposed a framework in which an OODB could be used to manage data and a DDB used to represent queries.

4. We implemented a system based on that framework and applied it to genome analyses. We showed that combination of a variety of searches for genome analyses and integration of several databanks were available using this system. Unresolved problems comprise management of users' knowledge in the database for flexible analyses and establishment of efficient query evaluation techniques.

This thesis consists of six chapters. Chapter 1 is the introduction, which describes the background, aims and organization of this thesis.

Chapter 2 defines the basic concepts and describes the results of related work. First, the concepts of DDB and OODB are given and, then related work concerning the DOOD are summarized. Next, the human genome is introduced, and the features of genome data and required functions for genome analyses are described.

In chapter 3, a prototypical query evaluator "DEE" is introduced. The results of practical applications of DEE are shown. The applications include configuration detection problems of piping-and-instrument diagrams in petrochemical plants and secondary structure search in RNA sequences. We solved an inefficiency of the KRS method, which is one of the methods we implemented in DEE, by developing a subgoal ordering method.

In chapter 4, we propose a framework of a DOOD, and a resulting system developed is described.

In chapter 5, the results of application of the system to genome analyses are shown.

Chapter 6 is the conclusion of the thesis.

Acknowledgements

I am most grateful to Professor Kazuo Ushijima, my supervisor at Kyushu University, for guidance, support and constructive criticism. Professor Ushijima supported my application to study at the Human Genome Center, Institute of Medical Science, University of Tokyo, as a research student.

I also express my thanks to Associate Professor Toshihisa Takagi at the Human Genome Center. He led a research group on deductive databases when he was at Kyushu University, and supported my work on deductive databases. Since he moved to the Human Genome Center, I have studied at the center as a research student. We had a numerous discussions and he gave me many useful comments. Professors Minoru Kanehisa and Yoshiyuki Sakaki at the Human Genome Center offered me a comfortable research environment and supported me there.

Professor Akifumi Makinouchi and Professor Toru Hidaka at Kyushu University reviewed early versions of this thesis and gave me many valuable comments.

Dr. Takahiko Suzuki at the Human Genome Center kindly helped me in implementing the query evaluator of the deductive database when he was at Kyushu University. I would also like to thank Mr. Norihiro Sakamoto for helping me to implement the deductive object-oriented database. He designed and implemented a lot of classes for genome data in an object-oriented database. Ms. Catherine Payne kindly helped me in preparing the manuscript.

Finally, I thank all my friends in the computer software laboratory at Kyushu University, in the Human Genome Center, and in the Institute for Chemical Research, Kyoto University, for their encouragement and support.

Contents

Chapter 1

Introduction

1.1 Background

A deductive object-oriented database (DOOD) is the integration of a deductive database (DDB) and an object-oriented database (OODB). The DDB and the OODB are next generation databases proposed for overcoming the shortcomings of a relational database (RDB).

The formal foundations of the RDB were first outlined by Codd [Cod79]. A RDB is a collection of individual facts (data), equipped with the capability to efficiently manipulate (update) its contents and to answer queries about it. Many RDB systems were developed in the 1980's.

However, in spite of their great usefulness, the capabilities of the RDB are severely limited by their inability to handle deduction and incomplete information. It has also been recognized that the relational data model is too simple to model complex nested entities, such as designed and engineering objects and complex documents.

The discovery of the shortcomings of the RDB led to active research of the DDB and the OODB. In addition to storing individual facts (extensional data), the DDB can store and manipulate deductive rules of reasoning (intensional data) and can answer queries based on logical derivation coupled with a mechanism for handling incomplete information [Ull89, PP90]. The OODB is a collection of objects whose behavior, states, and relationships are defined in accordance with an object-oriented data model, which provides complex data types and other useful features, such as encapsulation, inheritance and

polymorphism [Kim90].

The research on these two types of databases was conducted independently until the late 1980's. In the course of the research, some shortcomings were become obvious. The shortcomings of the DDB are as follows.

- Many methods have been proposed for recursive and negative query processing. However, there are few systems implementing the methods and almost no practical applications. Therefore, it is not clear whether the methods are applicable to and useful in practical cases.
- The DDB can handle nested relations by using functions. However, nested relations are not sufficient for representing the data in the practical applications.

The problems found with the OODB include:

- a lack of declarative query languages — application programs should be written in procedural languages, such as C++ and Smalltalk;
- difficulty in representing and managing incomplete knowledge — application programs written in procedural languages are necessary for handling incomplete knowledge and it is difficult to manage them in the database;
- insufficiency of formal foundations.

In an effort to unravel these shortcomings, it was proposed that the DDB and OODB be integrated, resulting in the DOOD, or Deductive Object-Oriented Database. From the late 1980's, there has been much active research on supporting a deductive language by an object-oriented data model; in other words, designing deductive object-oriented languages [YN89, Yok92]. Current problems regarding the DOOD are that:

- there is no consensual framework for integrating these two databases; and
- it is not clear what techniques in the DDB are necessary for the integration.

The Human Genome Project aims to find all human genes and related information that can be used for accurate diagnosis of inherited diseases, for understanding how humans develop from single cells to adults, and why this process sometimes goes wrong [CCS92, KNKT91]. To collect such information, the project is directed towards:

- experimentally determining all of the DNA sequences of the human genome and the genome of other organisms, and
- acquiring knowledge of the structures, functions, and evolution of the human genome.

A huge volume of data has been produced, and as research progresses, the amount of data will increase exponentially. Therefore, database development will be a major focus of the project.

Several databanks were developed to store these data [Tak92, Kam92]. However, users of the databanks are confronted with the following problems.

- Although the data are often complicated and nested, most of the databanks store the data using RDBs or flat files. Therefore, it is difficult to retrieve these complicated data.
- Because the databanks were developed and managed independently, it is difficult to retrieve newly constructed relationships between data from several databanks.
- There is no way to analyze the data flexibly. The databanks provide only keyword searches and similarity searches. The users must write programs to combine searches and further analyze the retrieved data.

In this project, we discuss what kind of database functions are required to deal with practical problems. We proposed a framework including these functions, developed a system based on the framework, and evaluated it by applying it to genome analyses.

1.2 Thesis Organization

This thesis consists of six chapters. Chapter 2 defines the basic concepts and describes the results of related work. First, the concepts of the DDB and the

OODB are defined and related work concerning the DOOD are summarized. Next, the Human Genome Project is introduced, and the features of genome data and required functions for genome analyses are described.

In chapter 3, a prototype of a query evaluator DEE (Deductive Engine for Engineering databases) is introduced. The results of practical applications of DEE are shown. The applications include configuration detection problems of piping-and-instrument diagrams in petrochemical plants and secondary structure search in RNA sequences. We showed that the recursive queries and negative queries in the DDB can be used to express many practical problems and that efficient query processing can be achieved by selecting appropriate methods.

Through these applications, an inefficiency with the current method was discovered. Thus, we propose a subgoal ordering method to improve on this inefficiency in deductive databases. The method first determines how bindings are passed in the query evaluation, and then transforms the given rules. We demonstrated the efficiency of the method by performing evaluation with some test data.

In chapter 4, we propose a framework for a DOOD, in which an OODB is used to manage data and a DDB is used to represent queries. A system developed from the framework is described.

In chapter 5, the results of applying the system to genome analyses are shown. Combinations of a variety of searches for genome analyses and integration of several databanks is available by using the system. Unresolved problems are management of users' knowledge of the database for flexible analyses and establishment of efficient query evaluation techniques.

Chapter 6 provides the conclusion of this thesis.

Chapter 2

Preliminaries

This chapter gives an overview of deductive object-oriented databases (DOODs) and genome analyses. Sections 2.1, 2.2 and 2.3 describe deductive databases (DDBs), object-oriented databases (OODBs) and DOODs, respectively. In section 2.4, genome information, current databanks for the information and their analyses are presented.

2.1 Deductive Databases

2.1.1 Basic Concepts

A DDB consists of the following components [PP90, Ull89]:

- EDB = a set of facts that represent tuples in relations: This is the *extensional database*.
- IDB = a set of rules that represent definitions of relations: This is the *intensional database*.
- IC = a set of conditions that must be satisfied by rules or facts: This is the *integrity constraint*.

For example, the following two facts (EDB) represent that *taro* is the father of *jiro*, and *hanako* is the mother of *jiro*.

$$father(taro, jiro). \quad mother(hanako, jiro).$$

The following two rules (IDB) represent a parents relation *par*, i.e. if X is the father of Y then X is a parent of Y, or if X is the mother of Y then X is a parent of Y. (The symbol :– means implication).

$$\begin{array}{lll} par(X,Y) & :- & father(X,Y). \\ par(X,Y) & :- & mother(X,Y). \end{array}$$

From these facts and rules, we can derive two new facts (derived facts):

$$par(taro, jiro). \quad par(hanako, jiro).$$

Syntax of Rules and Facts

Rules and facts in DDBs have the following syntax.

A *term* is defined recursively as follows:

- A *constant* is a term. It is a symbol denoted by a string that starts with a lower case letter.
- A *variable* is a term. It is a symbol denoted by a string that starts with an upper case letter.
- If f is an n-ary function symbol and $T_i (1 \leq i \leq n)$ are terms, then

  $$f(T_1, \ldots, T_n)$$

  is a term (*function*).

Terms without variables are called *ground terms*.

If p is an n-ary predicate symbol and $T_i (1 \leq i \leq n)$ are terms, then $p(T_1, \ldots, T_n)$ is an *atom*. An atom whose arguments are all ground terms is a *ground atom*.

A literal is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom.

A *rule* has a form

$$H{:-}L_1, L_2, \ldots, L_m.$$

where H is an atom, and $L_i (1 \leq i \leq m)$ are literals. H is called the *head* and $L_1, L_2, \ldots, L_m$ is called the *body* of the rule.

A *fact* is a ground atom.

A *ground instance* of a rule r is a rule obtained from r by substituting all its variables with ground terms.

A *query* is $:-Q$, where Q is a positive literal.

A rule is *recursive* if a same predicate symbol appears in both its head and body.

Relationship to Relational Databases

A set of facts can be treated as a *relation* of RDBs. For example, the *par* facts:

$$par(taro, jiro). \quad par(hanako, jiro).$$

can be considered as a relation PAR:

$PARENT$	$CHILD$
$taro$	$jiro$
$hanako$	$jiro$

Query optimization techniques in RDBs can be applied to query processing in DDBs when certain conditions are satisfied.

For a rule without functions (i.e. a Datalog rule [Ull89]), there exists a relational algebra formula equivalent to the rule when the rule is not recursive. A relational algebra formula composed of selection, projection, join and union operations can be transformed to equivalent rules. For example, an operation that derives a grandparent's ($gpar$) relation from a parent's (par) relation can be defined both in relational algebra and in the form of rules.

- The relational algebra formula is:

  $$GPAR = \pi_{AC}(PAR \bowtie \varrho_{B|A}(\varrho_{C|B}(PAR)))$$

  where attribute A is for parents and B for children, and $\varrho_{A|B}$ is an attribute renaming operator from B to A.

- The rule is:

  $$gpar(X, Z) :- par(X, Y), par(Y, Z).$$

Recursive Query Processing

The following rules define an ancestor's (anc) relation:

$$\begin{array}{lll} anc(X,Y) & :- & par(X,Y). \\ anc(X,Y) & :- & par(X,Z), anc(Z,Y). \end{array}$$

The second rule is recursive. A query evaluation of a database that contains recursive rules (a recursive database) has the following characteristics.

- If the database is negation free, semantics of the database can be determined by using the least fixpoint operation [ABW88].
- Completeness of the query evaluation is an important requirement. For example, if a Prolog interpreter is used for the query evaluation of a recursive database, the evaluation may not terminate and derive all answers for a given query.
- Some recursive rules have no equivalent relational algebra formulae. Query optimization techniques for the relational algebra cannot be applied to such rules.

Efficient query processing in recursive databases is an important research subject of DDBs. Many optimization techniques for recursive query processing have been proposed [BR86, SZ86, BR87, Nau87, Nau88, NRSU89b, NRSU89a, KRS90]. Some of them can be applied to general recursive rules while others are limited to their subclasses.

Negation

Negative conditions in the bodies of rules make DDBs more flexible. The following is an example of a database including a negative condition. (The symbol "$\neg$" denotes a negation.)

$$\begin{array}{l} businessman(iacocca). \\ avoidmath(X) :- businessman(X), \neg goodmathematician(X). \end{array}$$

From the database, a fact $avoidmath(iacocca)$ can be derived. This is based on the closed world assumption (CWA) [Rei78]. In this case, because the fact $goodmathematician(iacocca)$ cannot be derived from the database, $\neg goodmathematician(iacocca)$ is derived.

For a simple database such as the above example, it is intuitively clear what can be derived from the database and what cannot. For the following database it is not clear what can be derived from the database:

$$
\begin{aligned}
&even(Y) \text{:}-next(X,Y), \neg even(X).\\
&even(zero).\\
&next(zero, one).\\
&next(one, two).\\
&next(two, three).
\end{aligned}
$$

This example defines *even* numbers in a set $\{zero, one, two, three\}$. There are many studies on the semantics of databases that contain negation. Using those semantics, we can decide what is derivable from the database shown in the example. The semantics of databases that contain both recursive rules and negation tends to be complex. Query processing of those databases is also complex compared to that of databases without recursion and negation.

2.1.2 The Model of a Definite Database

We can assume without loss of generality that the set of predicate symbols of base facts (EDB) and the set of predicate symbols of the heads of the rules (IDB) are disjoint. A predicate in a database is either a *base predicate* (a predicate symbol of EDB) or a *derived predicate* (a predicate symbol of IDB). A database is definite (a *definite database*) if the rules in it do not contain negation in their body literals. The semantics of a definite database is determined by the *least model* of the database [ABW88], as in Prolog. A database is a *Datalog database* if it contains no functions. A rule is *range restricted* if every variable in the rule appears in its body.

The least model of the database can be computed by the fixpoint operation, if the database is a definite Datalog database composed of range restricted rules [ABW88].

The Fixpoint Operation

Let P be a definite database and S be a set of facts. The fixpoint operator T_P for S is defined as follows:

$$
\begin{aligned}
T_P(S) = S \cup \{h \mid\ & h\text{:}-l_1, \ldots, l_n. \text{ is a ground instance of a rule in } P\\
& \text{and } l_i \in S (1 \leq i \leq n)\}
\end{aligned}
$$

Let T_P^1 be a set of all facts in P, and $T_P^{n+1} \stackrel{\text{def}}{=} T_P(T_P^n)$. The fixpoint operator $T_P^\uparrow$ for P is defined as

$$T_P^\uparrow(P) \stackrel{\text{def}}{=} \lim_{n \to \infty} T_P^n.$$

For a definite Datalog database P with only range restricted rules, there exists a finite n such that

$$T_P^{n+1} = T_P^n.$$

The least model of P coincides with the fixpoint, and can be computed by iterative application of the T_P operator.

Semi-naive Evaluation

Naive application of the fixpoint operator contains redundant operations. If a ground instance of a rule $h{:-}l_1, l_2, \ldots, l_n.$ generates h in the k-th iteration, h is repeatedly generated in the $k+1, \ldots, n$-th iterations. *Semi-naive* evaluation [Ull89] is used to avoid such redundancy, by insisting that each time we apply a rule, at least one of the body literals uses a fact that was discovered on just the previous iteration.

Let T_P^0 be an empty set, and

$$\Delta T_P^{n+1} \stackrel{\text{def}}{=} T_P^{n+1} - T_P^n.$$

T_P^n can be redefined as follows:

$$T_P^{n+1} = T_P^n \cup \{h| \quad h{:-}l_1, l_2, \ldots, l_n. \text{ is a ground instance of a rule in } P \text{ and } l_1, l_2, \ldots, l_n \in T_P^n \text{ and some } l_i \in \Delta T_P^n\}.$$

For any k, semi-naive evaluation does compute the same T_P^k that is computed by the fixpoint operation, and thus can be used to compute the least model of P.

2.1.3 Query Processing Methods

Query processing in the database P is an operation that computes the answer set S for a given query $:{-}Q$. The set S satisfies the following conditions.

1. An element of S is in the *intended model* [PP90] of P. For a definite database, the intended model is the least model of P.

2. An element of S is a ground instance of P.

3. There are no facts that satisfy conditions 1 and 2 other than elements in S.

It is not necessary to compute the entire intended model of a program in order to evaluate a given query. Various query processing techniques are proposed for definite Datalog databases composed of range restricted rules.

Top-down Evaluation and Bottom-up Evaluation

Query processing methods in DDBs can be classified into either *top-down evaluation* or *bottom-up evaluation*.

Top-down evaluation starts with a given query as a goal. *SLD resolution* [Llo87] used in Prolog interpreters can be used as a simple method of top-down evaluation. SLD resolution may not terminate if the database is recursive.

SLD-AL resolution [Vie87], which is an improved version of SLD resolution, guarantees termination of evaluation in definite Datalog databases by using tables of intermediate results.

On the other hand, bottom-up evaluation is based on the fixpoint operation, and has the following advantages:

- It can be implemented easily.
- It always terminates and computes the answer set correctly for definite Datalog databases composed of range restricted rules.
- Some optimization techniques in RDBs are applicable; thus large numbers of facts can be handled efficiently.

However, even using semi-naive evaluation, fixpoint operation is not efficient for evaluating a given query, because there are many facts in the least model that are not relevant to the query. Therefore, many evaluation techniques, such as the Magic Set method have been proposed [BR86, SZ86, BR87, Nau87, Nau88, NRSU89b, NRSU89a, KRS90, KP88, STU89].

2.2 Object-Oriented Databases

An OODB is a collection of objects whose behavior, states, and relationships are defined in accordance with an object-oriented data model [Kim90]. An object-oriented data model captures the logical organization of real-world objects (entities), puts constraints on them, and defines relationships among them.

2.2.1 Object-Oriented Data Model

Although there are no standard concepts for an object-oriented database, we can see its core model [Kim90], which is common in almost all OODB systems. The core model includes:

- Objects and object identifiers
- Attributes and methods
- Encapsulation and message passing
- Classes
- Class hierarchies and inheritance

Objects and object identifiers

All of the entities in object-oriented databases are objects. They can be simple objects or complex objects. The simplest are objects such as integers, characters, strings and floats. Complex objects are built from the simpler ones, and include tuples, sets, bags, lists, arrays, and other nested objects.

The object identifier of an object is unique system-wide. It promotes implementation of object-sharing and object-updates.

Attributes and methods

The attributes of an object represent the state of the object, and using the methods associated with the object we can manipulate the state of the object. The specification for an attribute may include integrity constraints. Integrity constraints include the uniqueness of the value, admissibility of the null value,

the domain of the attribute, and so on. Because the domain of an attribute may be an arbitrary class, the value of an attribute of an object can also be an object. This gives rise to nested objects.

Encapsulation and message passing

Messages are sent to an object to access the values of the attributes and the methods encapsulated in the objects. There is no way to access an object except through the public interface specified for it.

Classes

A class is also an object. A class is often used as the receiver of a message to create an instance of the class. The concept of a class is important because it can be considered a link between object-oriented systems and databases for the following reasons:

1. It directly captures instance-of relationships.
2. It is useful for specification of a search domain and aggregation.
3. Semantic constraints can be specified by the class for the domain of an attribute.
4. It is not necessary to define attributes and methods for each instance in a class.

Class Hierarchies and Inheritance

Each class in an object-oriented database has a superclass, except for a root class. A class inherits attributes and methods from its superclasses. There are two advantages of inheritance: it is a powerful modeling tool, because it gives a concise and precise description of the world; and it helps in factoring out shared specifications and implementations in applications.

2.2.2 Features of Object-Oriented Database Systems

Atkinson, et al. proposed a definition for an OODB system [ABD+90]. In addition to the core model, they proposed that the following are mandatory features of the system.

- Computational completeness
- Extensibility
- Persistence
- Secondary storage management
- Concurrency
- Recovery
- Ad hoc query facility

2.3 Deductive Object-Oriented Databases

Throughout the 1980's, DDBs and OODBs received a great deal of attention, but for the most part these two fields evolved independently. Recently, however, much effort has been devoted to integration of the two promising techniques.

Why is the integration of DDB and OODB necessary?

As we stated in Section 2.1, DDBs have a high inference ability and formal foundations in semantics of databases as their advantages. However, they cannot model all real-world entities.

On the other hand, OODBs can model real-world entities as complex nested objects, and have been applied to many types of practical problems. However, they lack consensus for concepts of data modeling and formal foundations. They also suffer from a poor inference ability.

Thus, for the next generation of databases, integration of deductive databases and object-oriented databases is considered [Yok92].

Approach to integration

Many approaches to the integration of DDBs and OODBs have been researched [YN89]. They can mainly be divided into two fields: extension of DDBs to handle various data types and formalization of OODB semantics.

The extension of DDBs can be classified as follows:

- logical extension: introduction of null values, belief values and/or existential qualifiers
- data model extension: introduction of complex nested objects, methods (procedures) for data and/or object identity.

Following are examples of research conducted on the extension of deductive databases:

- O-logic [KW89]
- F-logic [KL89]
- DOT [TNF91]
- LDL [BNR+87]
- HiLog [CKW89]
- LLO [LO91]

Research on deductive object-oriented databases is very active and many languages or semantics have been proposed. However, a consensus has not been reached on deductive object-oriented databases and there are few systems and applications.

2.4 Genome Analyses

2.4.1 Genome

The complete set of instructions for making an organism is called its genome. It contains the master blueprint for all cellular structure and activities for the lifetime of a cell or organism. Found in every nucleus of a person's 10×10^{12} cells, the human genome consists of tightly coiled threads of deoxyribonucleic acid (DNA) and associated protein molecules, organized into structures called chromosomes.

For each organism, the components of the genome encode all the information necessary for building and maintaining life. Understanding how DNA performs this function requires some knowledge of its structure and organization.

DNA

In humans, as in other higher organisms, a DNA molecule consists of two strands that wrap around each other to resemble a twisted ladder whose sides are connected by "rungs" of nitrogen-containing chemicals called bases. Four different bases are present in DNA — adenine (A), thymine (T), cytosine (C), and guanine (G). The particular order of the bases is called the DNA sequence; the sequence specifies the exact genetic instructions required to create a particular organism with its own unique traits (Fig. 2.1).

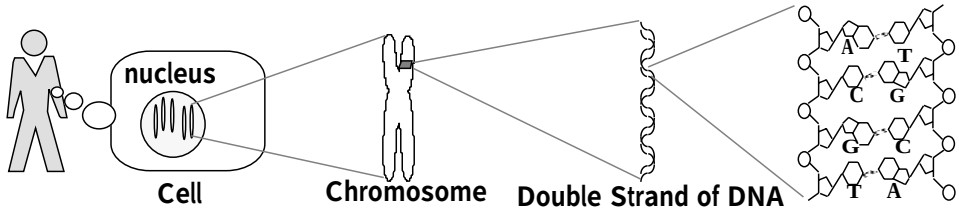


Figure 2.1: Chromosome and DNA

The two DNA strands are held together by weak bonds between the bases on each strand, forming base pairs (bp). Genome size is usually stated as the total number of base pairs — the human genome contains roughly 3×10^9 bp.

Genes

A gene is a specific nucleotide sequence that has genetic information about the structure of a biochemical substance such as a protein, or regulatory functions. Humans have at least 100,000 genes.

A gene consists of two types of regions; exons and introns. Exons are protein coding regions where amino acids are designated by triplets of bases (codon). A coding region of DNA is transcribed to a mRNA (messenger RNA). The mRNA sequence is translated into an amino acid sequence. When it folds, it shows its biological function (Fig. 2.2). Introns are non-coding regions and their functions are not known.

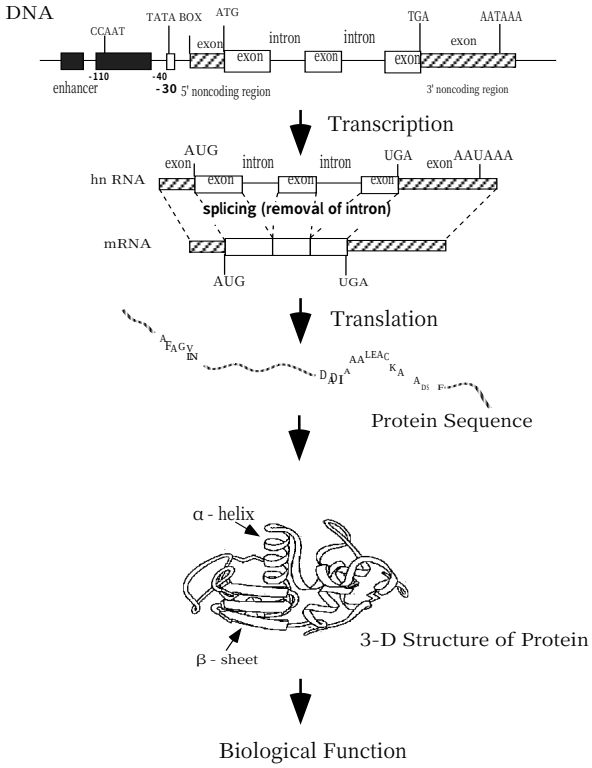


Figure 2.2: From a DNA sequence to the function of the corresponding protein

Chromosomes

Human genes are scattered in 24 types of structures, which we call chromosomes. Human chromosomes consist of 22 autosomes and two sex chromosomes (X and Y). They include 50% protein and 50% DNA. Damage of chromosomes (DNA) may cause various hereditary diseases.

2.4.2 The Human Genome Project

The Human Genome Project is directed towards:

- mapping all the human genes: determination of all genes on the DNA sequence;
- improving mapping and sequencing technologies (ultimately all these procedures should be performed automatically); and
- contributing to human biology and medical science.

Mapping and Sequencing the Human Genome

Mapping strategies

Human genome maps are divided into two types; genetic linkage maps and physical maps. A genetic linkage map shows the relative chromosomal position of DNA markers, determined by their patterns of inheritance. The physical map is determined by the chemical characteristics, such as restriction enzyme[1] cutting sites and genes, of chromosomal locations. The resolution of the physical map varies from the lowest level, determined by banding patterns, to the highest level, determined by complete DNA sequences. About 2,300 genes have been currently located.

Genetic linkage maps depict the relative chromosomal locations of DNA markers by their patterns of inheritance. The maps are used to determine the locations of the genes responsible for hereditary diseases. DNA markers are the sites where we can easily identify detectable differences between individuals. Allelic genes and DNA sequence, which are easily traceable even though their functions are not known, can be used as the markers. Examples of these types of markers are:

- restriction fragment length polymorphisms, and
- variable number of repeat units.

As a unit of distance between markers, centimorgans (cM) are used, so named for the American geneticist Thomas Hunt Morgan. They represent the frequencies of recombinations of two markers. 1 cM roughly corresponds to 1 Mbp. Current resolution of the genetic map is about $7 \sim 10$ cM. It will be improved to $2 \sim 5$ cM.

[1] A restriction enzyme recognizes a specific short sequence ($4 \sim 8$ letters) and cuts the DNA sequence at that point

Chromosomal maps and cDNA maps are examples of low-resolution physical mapping. Chromosomal maps are locations determined by a labeled marker that fluoresce or is radioactive and are constructed for the genes that exist in all individuals. Their current resolution is about 100,000 bp. Improvement of the resolution will be useful in analyzing chromosomal abnormalities. A cDNA map shows the positions of expressed DNA regions (exons) relative to particular chromosomal regions or bands. It is useful for focusing on the genes that are responsible for hereditary diseases when their approximate location has been mapped by genetic linkage techniques.

The two approaches to high-resolution physical mapping are termed "top-down" and "bottom-up." A macro-restriction map is constructed by the top-down approach. It depicts the sites where restriction enzymes cut. This approach yields maps with more continuity and fewer gaps between fragments than the bottom-up approach, but the map resolution is lower and may not be useful for finding particular genes. The bottom-up approach involves cutting the chromosome into small pieces, each of which is cloned and ordered. The ordered fragments form contiguous DNA blocks (contigs) and produce contig maps.

Sequencing technologies

Present sequencing technologies are based on gel electrophoresis, by which one molecule of DNA can be identified. Two popular methods have been developed.

- Maxam-Gilbert sequencing uses chemicals to cleave DNA at specific bases, resulting in fragments of different length.
- Sanger sequencing involves using an enzymatic procedure to synthesize DNA chains of varying length with four different reactions, stopping the DNA replication at a position occupied by one of four bases, and then determining the resulting fragment length.

Completing maps and sequences: Finding specific genes

Gaps tend to remain in maps constructed by current mapping techniques. Chromosome walking is often used to fill in gaps.

About 1,000 markers have been identified on the genetic map. The distance between adjacent markers is roughly 3×10^6 bp, in which about 100 genes reside.

It is easy to identify the gene responsible for a hereditary disease in a case where it was caused by a large mutation in the DNA sequence. It can be detected by observing chromosomes through a microscope, then searching a genetic map. Identifying a small mutation is more difficult. If a DNA fragment adjacent to the gene responsible for a hereditary disease can be obtained, the fragment can be used as a probe and the responsible site can be detected by using an ordered genetic library.

Model organism research is also active. Organisms for studies of genomes include bacteria, yeasts, fruit flies and roundworms. Evolutionally preserved DNA sequences can be compared through research. They can be applied to the discovery of the genes responsible for human hereditary diseases.

2.4.3 Genome Databanks

Collecting and storing data

It is necessary to collect, store and distribute data obtained by genome research [KNKT91, Eri92, Tak92]. Thus, it is very important that a useful database design be developed, because it should be able to accurately represent map information (linkage, physical location, disease gene) and sequences (genomes, cDNAs, proteins), as well as link them to each other and to bibliographic text databases of scientific and medical literature. In addition to collecting and storing data, software for database access and manipulation is required.

Interpreting data

Two major methods for interpreting genome data are homology search and motif extraction. Searching a database for a particular DNA sequence may uncover the homologous (similar) sequences in a known gene in a model organism, revealing insights into the functioning of the corresponding human gene.

A motif is the tertiary structure pattern of a protein and is responsible for the function of the protein. Because motifs are much more evolutionarily

conserved than amino acid sequences (or DNA sequences), they are useful for homology search.

The major genome databanks are summarized in Table 2.1.

Table 2.1: Major Genome Databanks

Name	Organization	Data
GenBank	NIH (U.S.A.)	DNA sequences
EMBL	EMBL (Germany)	DNA sequences
DDBJ	NIG (Japan)	DNA sequences
PIR	NBRF (U.S.A.)	Amino acid sequences
SWISSPROT	Univ. Geneva (Switzerland)	Amino acid sequences
PROSITE	Univ. Geneva (Switzerland)	Protein motif
PDB	BNL (U.S.A.)	3-D structure of proteins
GDB/OMIM	JHU (U.S.A.)	Genetic and physical maps
GBASE	Jackson Lab. (U.S.A.)	Mouse genome maps
MEDLINE	NLM (U.S.A.)	Bibliography

NIH: National Institute of Health
EMBL: European Molecular Biology Laboratory
NIG: National Institute of Genetics
NBRF: National Biomedical Research Foundation
BNL: Brookhaven National Laboratory
JHU: Johns Hopkins University
NLM: National Library of Medicine

Mapping databanks

The Genome Database (GDB) [PMFR92] provides location, ordering, and distance information for human genetic markers, probes, and contigs linked to known human hereditary disease. The Online Mendelian Inheritance in Man (OMIM) database is a catalog of inherited human traits and diseases.

The Human and Mouse Probes and Libraries Database and the GBASE mouse database include data on RFLPs[2], chromosomal assignments, and probes from the laboratory mouse.

[2]RFLP is the abbreviation for restriction fragment length polymorphism. It is the variation between individuals in DNA fragment sizes cut by specific restriction enzymes.

Sequence databanks

GenBank [BCF+92], the European Molecular Biology Laboratory (EMBL) sequence database [HFSC92], and the DNA Database of Japan (DDBJ) [Miy90] house over 70 Mbp of sequences from more than 2,500 different organisms.

The major protein sequence databases are the Protein Identification Resource (PIR) [BGMT92] and SWISSPROT [BB92]. In addition to sequence information, they contain features of protein structures. PROSITE [Bai92] is a databank for information on protein motifs. The Protein Data Bank (PDB) [BKW+77] is the protein tertiary structure databank.

Chapter 3

Evaluation of Applicability of a Deductive Database to Practical Problems

3.1 Background

Recently, many computer-aided design (CAD) systems have been developed in various engineering areas. As they have been applied to many practical problems, there has been increasing need for high level query support for CAD databases. To provide such support, database systems for CAD can handle the relationship between components as well as individual components; the latter is easily handled by the technologies of object oriented databases. To handle the relationship between components, the database systems should have the inference facilities necessary for efficiently computing the conditional connectivity between components.

The analysis of the human genome is a significant topic in both biology and medical science. The Human Genome Project is now accumulating an enormous amount of human nucleotide sequence data; hence, the importance of database management systems is growing in molecular biology and medical science. For example, GenBank [BCF+92] is now maintained internally as a relational database management system (RDBMS) and contains 138,904,393 nucleotides in 120,134 loci (Release 77.0). However, as we will see, it is necessary to develop more well-designed database system than the relational

ones.

GenBank provides two kinds of search programs, a database entry retrieval program and a sequence similarity search program. Within the retrieval program, entries can be located by searching for a keyword, such as a locus name, or a combination of keywords appearing in any of the fields of the entry's annotations. The similarity search program employs FASTA [PL88] or BLAST [AGM+90] algorithms and returns entries with nucleotide sequences similar to the query sequence. Search programs like these are also available for the EMBL data library [HFSC92], the PIR protein sequence databank [BGMT92] and the SWISSPROT protein sequence databank [BB92].

With the search programs, users can give queries such as "Find entries related to a given gene," or "Get nucleotide sequence data carrying a similarity to the sequence of interest." However, it is difficult to order such queries as, "Find entries containing a promoter region," or "Get nucleotide sequence data containing a special sequence alignment or a secondary structure, such as a stem-and-loop," because promoter regions or secondary structures cannot be simply searched for by either keywords or primary nucleotide sequences. Query expression based on inference rules are necessary for such searches.

In order to provide a framework for making such queries, we developed a prototype of an inference engine based on the query processing techniques of deductive database systems (DDBSs). This was then applied to the problem of extracting a complex configuration with conditional connectivity from the CAD databases of petrochemical plants. A second area that it was applied to was the problem of searching for promoter regions or secondary structures in nucleotide sequences. We then investigated the applicability of DDBS to practical problems by evaluating the expressive power and efficiency of the prototype. The results showed that the techniques of DDBS can be put to practical use.

A DDBS is a logic programming system that is required for handling large amounts of data. The most common logic programming system, Prolog, works as a powerful query language for RDBs by regarding tuples in a RDB as Prolog's facts. However, Prolog's one-tuple-at-a-time execution method tends to be inefficient when large numbers of facts exist.

Therefore, there have been many efforts to develop query processing techniques that can manipulate large amounts of data efficiently. For example,

the Magic Set method [BR88, BR87] is a well known technique for recursive query processing that reduces the number of irrelevant facts. There have also been other efforts to design and implement DDBSs, based on the various query processing techniques. For example, NAIL! system [MUvG86], DedGin [Vie88], LDL [BNR+87].

These DDBSs, however, are still undergoing extensive research and development, and have not yet been applied to practical problems. It has not been clear whether or not the technologies of DDBS are applicable and useful to the practical problems.

The remainder of this chapter is organized as follows. In section 3.2 we give an overview of our system and describe the query processing techniques adopted in our system. In section 3.3 we introduce the problem of extracting a complex configuration from the CAD databases of petrochemical plants and point out the functions necessary for solving this problem. Some examples are given. Section 3.4 explains the search for promoter regions and secondary structures in nucleotide sequences, and shows the results of the search using this system. Through the above applications, an inefficiency with the query processing method is found. Thus, we develop a subgoal ordering strategy for efficient query processing with this method. Section 3.5 introduces this strategy. In section 3.6, we discuss the applicability and usefulness of DDBSs.

3.2 Overview of the System

In order to evaluate the applicability and usefulness of DDBSs, we implemented a prototype of a deductive database system **DEE** (Deductive Engine for Engineering databases) on a SUN-3/80 workstation with 12 MB memory. Fig. 3.1 shows the configuration of DEE.

The engine of DEE consists of two components: the rule transformer and the bottom-up evaluator. The rules received by the rule transformer are written in the forms of Horn clauses allowing negation in their bodies; a query is a Horn clause without a head. The transformer rewrites the rules according to the binding pattern in the argument of the query. The transformed rules (which are also Horn clauses) can be evaluated efficiently by the bottom-up evaluator.

Attributes of components and direct connectivity between components are managed in a conventional DBMS (CAD database and genome database).

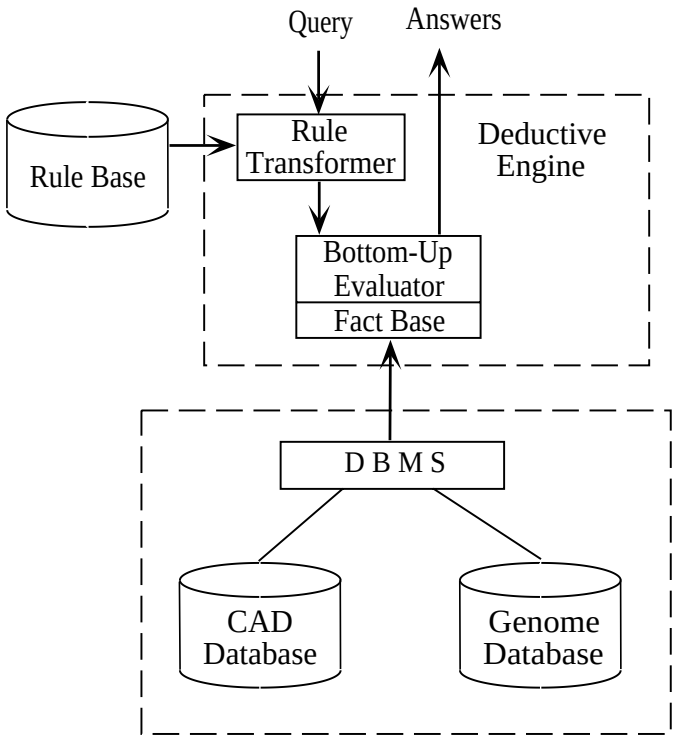


Figure 3.1: Configuration of DEE

They are converted to ground atoms (i.e. facts) and passed to the bottom-up evaluator before performing deductive procedure.

The rule transformer is written in 3,000 lines of Prolog, and the bottom-up evaluator is written in 6,000 lines of C. This prototype is a general DDBS that can be used for many purposes, in combination with other databases.

3.2.1 Recursive Query Processing

One of the important features of DEE is its efficiency in recursive query processing. There are many techniques for efficient recursive query processing. The Magic Set method [BR88, BR87] is a well-known technique based on rule transformation. The transformed rules can be handled efficiently with a bottom-up evaluation. There are other rule transformation methods, such

as the Counting method [SZ86]; transformation for left-, right-, and multi-linear rules by Naughton et al. [NRSU89b] (we will refer to this as the NRSU method); and its extension by Kemp et al. [KRS90] (the KRS method or Context method); argument reduction by factoring [NRSU89a]; etc.

There are also other techniques not based on rule transformation. For example, Naughton defines several classes of recursions and proposes efficient evaluation procedures for classes as one-sided recursions [Nau87] or separable recursions [Nau88].

DEE adopts three rule transformation methods: the Magic Set method, the NRSU method, and the KRS method. The Magic Set method can be applied to any recursive (or non-recursive) rules and is compatible with other rule transformation methods. We use the Magic Set method as a common platform for various methods based on rule transformation.

The NRSU method is applicable to the rules in a class of left-, right-, and multi-linear rules. This method reduces the number of intermediate facts generated in the bottom-up evaluation by reducing the number of arguments in the transformed rules. Naughton et al. [NRSU89b] shows that the evaluation by using the NRSU method generates $O(n)$ facts while the Magic Set method generates $O(n^2)$ facts (n is the size of data), under some conditions. We will show that, in our examples, the NRSU method is certainly efficient.

The NRSU method can only be applied to recursive predicates having ground arguments. Thus, Kemp et al. proposed the KRS method [KRS90], which is an extension of the NRSU method. The KRS method can be applied to recursive predicates in the body of other rules as well as the predicate in the query.

DEE has three rule transformations explained here for recursive query evaluation. We assume they are sufficient for estimating the applicability of DDBS. The reasons are as follows:

(1-a) Argument reduction by factoring is a generalization of the NRSU method. Although it is applicable to a more general class of rules, in many practical cases the NRSU method can also be applied where factoring can be applied.

(1-b) The decision procedure for applicability of factoring is much more complex than that of the NRSU method (in general, it is NP-complete [NRSU89a]).

(2) Evaluation procedures for one-sided recursions or separable recursions are not compatible with the Magic Set method. However, useful subclasses of

one-sided or separable recursions can be handled efficiently with rule transformation methods [NRSU89a].

(3) Even if these three methods are unable for evaluating a given query efficiently, we can add other methods based on rule transformation (such as the Counting method) to DEE without loosing upward compatibility with the current version.

3.2.2 Negation and Closed Queries

When a negative literal exists in the bodies of rules, the Magic Set method cannot be applied naively. Therefore several extensions of the method are proposed so that negative literals can be handled [BMPR87, KP88, STU91].

In DEE, we adopt a technique by Kerisit et al. [KP88] as an evaluation method for negative literals. We can regard each negative literal as a set of closed queries (a closed query is a query where all arguments are ground). In addition, the VIMS method [STU91] is adopted, which efficiently evaluates a set of closed queries.

Kerisit's method divides Magic Set transformed rules into layers according to an ordering defined by a similar condition being used to define the stratifiability of databases. Like the bottom-up evaluation of a stratified database, the evaluation starts from the lowest layer and proceeds to higher layers. The distinctive feature of Kerisit's method can be described in the following way. When new facts are generated through evaluation of layers, repeat the evaluation step, beginning with the lowest layer again.

The VIMS method, which can evaluate closed queries efficiently, is an extension of the Magic Set method. It is compatible with other transformation methods adopted in DEE and Kerisit's negation-handling method. With this method, an extra argument is added to each "magic" predicate [STU91]. The system can detect and avoid irrelevant evaluations for each closed query by using information carried with the argument. Users of DEE must carefully consider use of the VIMS method, for the VIMS transformation does not always guarantee better performance. In DEE, before the transformation, rules must be stratifiable [ABW88]. We will discuss the appropriateness of these conditions in section 3.6.

3.2.3 Strategy for Rule Transformation

For given rules and query, DEE uses the above rule transformation methods according to the following strategy:

(1) If there are right-linear recursive rules, then apply the NRSU method or the KRS method.

(2) For other rules, use the Magic Set method.

(3) If there is a negation in the rules, then divide rules into layers according to Kerisit's ordering.

(4) Use the VIMS method for closed queries when the user designates this method.

3.2.4 Bottom-Up Evaluation

The bottom-up evaluator computes a fixpoint from the database and the transformed rules. All answers of the query are in the fixpoint, and the evaluator retrieves the facts matching a query.

The bottom-up evaluator is based on the semi-naive evaluation [Ull89]. For negations, the evaluator uses Kerisit's method.

To guarantee fast and constant time access to each fact in the database, DEE manages facts with hash tables. A hash table manager is still under development. In particular, efficient hash functions for complex objects such as sets or lists are important objectives being pursued.

3.3 Application to a CAD Database

3.3.1 Retrieval of Complex Configurations

In petrochemical plants, many instruments and pipes are connected in a complicated manner. This complexity may cause problems not foreseen when the plants were designed. Every time a problem occurs in a plant, it becomes necessary to search the CAD database that stores the attributes and direct connections to the components of other plants in order to detect the configuration similar to that causing the problems. One petrochemical plant generally consists of 100 pipe-and-instrument diagrams (we call this diagram

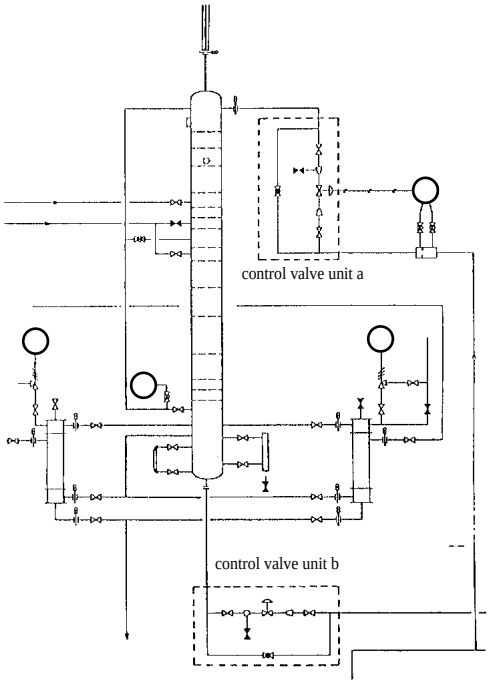


Figure 3.2: A sample P&I-D sheet

P&I-D). One P&I-D contains about 1,000 components (Fig. 3.2 shows part of a P&I-D). Therefore, it is difficult to detect all hazardous configurations by hand; thus, there is a strong need for detecting them automatically, by computer.

Simple retrieval of the attribute of a component can be performed by a traditional database management system. The problem is to compute whether or not an instrument A connects indirectly to another instrument B via some pipes. It is not practical to store all indirect connectivities between all components in a database since such relations may be very large. Therefore, it is necessary to represent the indirect connectivity by recursive rules.

We described various queries for retrieving complex configurations with

conditional connectivity to detect a hazardous configuration; we found that the following functions are necessary:

(a) Recursion: a function representing and computing indirect connectivity between components. Database systems for CAD should provide a recursive query processing facility.

(b) Negation: a function representing and computing that a component A does not connect to a component B. Database systems should provide a negative query processing facility.

(c) Set: a set-handling function which corresponds to the grouping operator in LDL [BNR+87].

P&I-D can be represented as a graph by regarding instruments and pipes as nodes, and connectivities between them as edges. Then, the above functions may be implemented by combining existing graph algorithms. However, it is difficult to describe an efficient and general graph algorithm that computes indirect connectivities to complex conditions in terms of the attributes of the instruments and pipes. DDBSs have the ability to easily represent the indirect connectivities to complex conditions by using logical formulae.

3.3.2 Examples and Performance

We have evaluated the performance of our system by solving the problems of computing indirect and conditional connectivities between components in petrochemical plants. In this section, we show three examples of queries and their corresponding execution times. These examples were executed on a P&I-D that contains 6,497 facts. Fig. 3.2 shows a portion of the P&I-D we used.

Example 1: Control valve units

The first example is a program for retrieving control valve units. A *control valve unit* is a piping configuration around a control valve. Fig. 3.2 contains two control valve units.

The relations *node* and *code* in the following rules represent attributes of a component. The relation *jointx* represents the direct connectivity between

two components. These three relations are explicitly stored in the CAD database.

The meaning of each argument in the query $:-cvunit(Cv, Node, Type)$ is as follows. The first argument Cv is the ID code for a control valve. $Node$ is the ID code of each component in the same unit as Cv. $Type$ is a type code for each component corresponding to $Node$. The rules listed below do not have recursions. They do have nested definitions, however. The result of the evaluation using the example of DEE is shown in Table 3.1.

% Rules

$cvunit(Cv, Node, Type) :- cvunitsub(Cv, Node),$
$\quad node(Node, Code),$
$\quad code(Code, Type).$

$cvunitsub(CvNode, Node) :- nextcv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next1cv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next2cv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next3cv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next4cv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next5cv(CvNode, Node).$
$cvunitsub(CvNode, Node) :- next6cv(CvNode, Node).$

$nextcv(Cv, N) :- cvvalve(Cv),$
$\quad jointx(Cv, N), pipe(N).$
$next1cv(Cv, N) :- nextcv(Cv, N1),$
$\quad jointx(N1, N), reducer(N).$
$next2cv(Cv, N) :- next1cv(Cv, N1),$
$\quad jointx(N1, N), pipe(N).$
$next3cv(Cv, N) :- next2cv(Cv, N1),$
$\quad jointx(N1, N), valve(N).$
$next4cv(Cv, N) :- next3cv(Cv, N1),$
$\quad jointx(N1, N), pipe(N),$
$\quad jointx(N, N2), kubunten(N2).$
$next5cv(Cv, N) :- next4cv(Cv, N1),$
$\quad jointx(N1, N2), kubunten(N2),$
$\quad jointx(N2, N), pipe(N),$
$\quad jointx(N, N3), valve(N3).$
$next6cv(Cv, N) :- next5cv(Cv, N1),$
$\quad jointx(N1, N), valve(N).$

$pipe(X)$:–$node(X, c3000)$.
$valve(X)$:–$node(X, c1000)$.
$reducer(X)$:–$node(X, c1001)$.
$kubunten(X)$:–$node(X, c9003)$.
$cvvalve(X)$:–$node(X, c1700)$.

% Query

:–$cvunit(Cv, Node, Type)$.

Table 3.1: Result of the *cvunit* Query

Method	Time[1]	Facts[2]
Magic	5.92sec	838

The result of the evaluation is given in a set of facts. One can use this result for various purposes. For example, by extracting ID codes bound to variables Cv and $Node$, and passing these ID codes to a P&I-D display program, one can easily check the positions and configurations of the control valve units on a CRT display.

Example 2-1: All components connecting to a certain instrument without interception by valves

The second example is a program for computing all components connecting to a certain instrument (ID $n100004$). For the rules containing right-linear recursion, DEE can adopt the NRSU method, which is theoretically more efficient than the Magic Set method. To compute negation (represented by the symbol $\neg$) in the body of rules, DEE uses Kerisit's method.

% Rules

$ctovalve(X, Y)$:–$jointx(X, Y), \neg valve(X)$.
$ctovalve(X, Y)$:–$jointx(X, X1)$,
$\neg valve(X), ctovalve(X1, Y)$.
$valve(X)$:–$node(X, c1000)$.

[1]'Time' means the CPU time required by the bottom-up evaluator.
[2]'Facts' is the number of facts generated during the bottom-up evaluation.

% Query

$:\!-ctovalve(n100004, X).$

Example 2-2: Retrieval by name of instrument

The query $:\!-ctovalve(n100004, X)$ in the above example contains the ID code $n100004$. In the P&I-D, this is the ID code for the drum D-801. It will be convenient for the user to specify the name 'D-801' directly in a query. This can be done by adding the following rule to the program.

$$ctovbyname(Name1, N2):\!-key(N1, Name1),\\ ctovalve(N1, N2).$$

The predicate *key* is used as a translation table between the ID code of each instrument and its name.

By using the new predicate *ctovbyname*, the query in example 2-1 can be replaced with the following query, which, as in example 2-1, becomes

$$:\!-ctovbyname(`D\text{-}801', Node).$$

In this case, one cannot apply the NRSU method because the first argument of the $ctovalve(N1, N2)$ in the above rule is non-ground[3]. However, the KRS method can be applied even in this case. The result of the evaluation is in Table 3.2.

Example 3: Instruments with drain or purge (safety) valves

The third example computes all *instrument*s connected to drain valves or purge valves.

% Rules

$$instwithsafetyvalve(X):\!-instrument(X),\\ connecttosafety(X).\\ connecttosafety(X):\!-jointx(X,Y), safetyvalve(Y).\\ connecttosafety(X):\!-jointx(X,Y),$$

[3]For this particular example, there are mutual functional dependencies between the two arguments of $key(N1, Name1)$. Hence, taking semantics of the database into account, one can use the NRSU method. However, the KRS method can be applied to more general cases (irrespective of the existence of a functional dependency).

Table 3.2: Result of *ctovalve* and *ctovbyname* Queries

Method	Time	Facts
ctovalve query		
Magic	48.78sec	4120
NRSU	1.90sec	362
ctovbyname query		
Magic	65.96sec	4181
KRS	5.66sec	365

$$\begin{array}{l}
\qquad\qquad \neg blockdevice(Y), \\
\qquad\qquad connecttosafety(Y). \\
safetyvalve(X) \text{:-} valve(X), \\
\qquad\qquad jointx(X,Y), pipe(Y), \\
\qquad\qquad onlyjointto(X,Y). \\
onlyjointto(X,Y) \text{:-} \neg jointtoother(X,Y). \\
jointtoother(X,Y) \text{:-} jointx(X,Y1), \\
\qquad\qquad \neg(Y = Y1). \\
blockdevice(X) \text{:-} instrument(X). \\
instrument(X) \text{:-} drum(X). \\
\qquad\qquad \vdots \\
\% \text{ Query} \\
\qquad \text{:-} instwithsafetyvalve(X).
\end{array}$$

The rule defining $safetyvalve(X)$:– ... identifies drain or purge valves. It uses the definition of $onlyjointto(X,Y)$ to confirm that an end of the $valve(X)$ is free. There is double negation in the definition of $onlyjointto(X)$.

Connection to the $safetyvalve(X)$ is defined recursively by the rules with the head predicate $connecttosafety$.

In this example, one can apply the VIMS transformation to the rules for $connecttosafety(X)$. Intuitively, this means that one does not need to search whole drain valves and purge valves connecting to a certain instrument in order to prove that the instrument is connected to $safety$ valves.

Table 3.3 shows the results of the evaluation.

In this case, the VIMS method outperforms the Magic Set method. The reason for the superior performance of the VIMS method is that the following

Table 3.3: Result of *instwithsafetyvalve* Query

Method	Time	Facts
Magic	252sec	16,765
VIMS	37sec	5,666

two conditions [STU91] are satisfied:

(a) $instrument(X)$ is a relatively small set compared to the size of all the components in the database (there are many pipes and valves but not so many drums and tanks).

(b) For each id that $instrument(id)$ holds, it is probable that $connecttosafety(id)$ also holds (instruments are commonly connected to drain valves or purge valves).

General forms of the conditions are in [STU91].

3.4 Application to a Genome Database

3.4.1 Retrieval of Biological Features

The importance of database management systems is growing in molecular biology and medical science, as mentioned in section 3.1. In the field of protein studies, logic-based systems have been applied to analyze structural topology [MT86b, RTN+86]. However, although logic-based theories and techniques are now well known, a practical logic-based system for analyzing nucleotide sequence data has not been reported. Nucleotide sequence data are much more numerous than data for proteins.

Since the features of DDBSs are suitable for biological databases, we applied DEE for retrieving biological characteristics of nucleotide sequence data. Ordinary keyword searches and simple similarity searches can also be carried out.

Database construction

All the primates' nucleotide sequence data of GenBank (Release 77.0) were transformed and stored into four relational tables; *entry*, *keyword*, *feature* and *sequence*. Relational tables *entry*, *keyword*, and *feature* correspond to "LOCUS," "KEYWORDS" and "FEATURES" records of GenBank, respectively. (An entry of GenBank is shown in Fig. 4.1.) The relational table sequence stores nucleotide sequences. The latter three relational tables have an additional column *locus* that is used as the foreign key for connecting the four tables. Other GenBank records, such as "DEFINITION" and "REFERENCE," are not stored due to limitations of disk space. Relational tables *entry*, *keyword* and *feature* contain 22,934, 48,525 and 67,413 tuples, respectively. The relational table *sequence* contains 22,934 tuples and 24,242,846 nucleotides. They occupy about 220 MB on a disk, while the corresponding GenBank flat file occupies about 70 MB. The relational tables are maintained on the commercially available RDBMS SYBASE (Emeryville, CA, USA). The tuples in relational tables are retrieved, collected and converted into base facts. Fig. 3.3 shows an example of base facts converted from the four relational tables.

3.4.2 Results

Keyword Search

The following query retrieves entries that contain *'TFII'* as the keyword.

% Query 1

$:-keyword(Locus,'TFII').$

% Answer 1

$keyword('HUMCOUPII','TFII').$

$keyword('HUMTFIIB','TFII').$

$\vdots$

Database searches by a combination of two or more keywords are often performed, and the following rule is an example of their description.

% Rules 2

$tf2d(Locus):-$

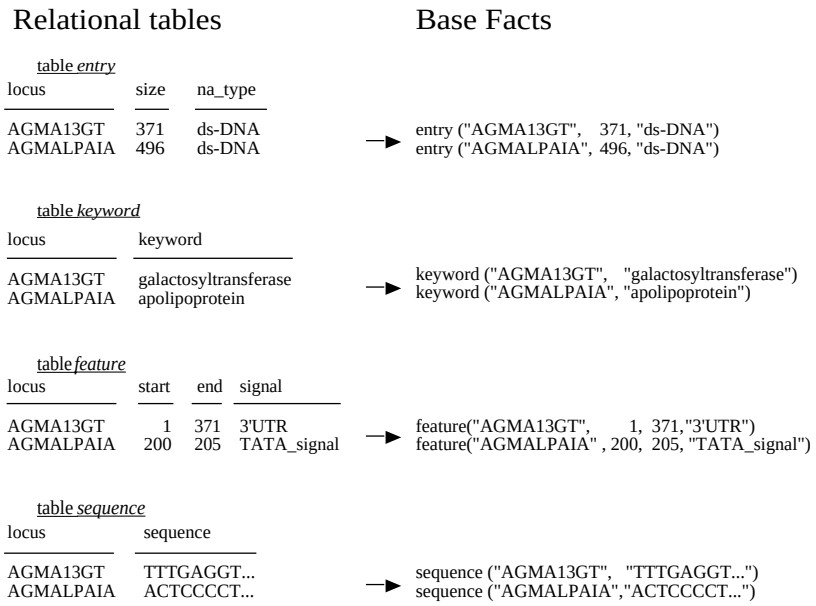


Figure 3.3: Example of Facts Converted from the Four Relational Tables

$$keyword(Locus,'TFIID'),$$
$$keyword(Locus,'RNA\ Polymerase\ II').$$

This rule is used to search for entries that contain both "TFIID" and "RNA Polymerase II" as keywords.

Search for TATA boxes

A promoter is a very essential region regulating a gene expression. It is known to contain some elements such as a TATA box, a CAAT box and a GC box. Among these, the TATA box is characterized as the best eukaryotic promoter element. It is usually located $25 \sim 30$ base pairs upstream of the transcription initiation site in the majority of eukaryotic promoters [MT86a]. Hence, searches for TATA boxes are important for analyses of promoters. However, TATA boxes are not always denoted by "`TATA_signal`" as a feature key in GenBank "`FEATURES`" records. Some TATA boxes are denoted by "`misc_signal`" and others by "`promoter`". Since in the latter two cases de-

tailed information on the biological features is described in feature qualifiers with a free text format, automatic handling is difficult. Thus, it is necessary to investigate nucleotide sequences corresponding to "misc_signal" and "promoter" in order to retrieve all the TATA boxes.

The following set of rules defines TATA boxes.

% Rules 3

$tata\_box(Locus, Start, End)$:–
$feature(Locus, Start, End, 'TATA\_signal').$
$tata\_box(Locus, Start, End)$:–
$feature(Locus, Start, End, 'misc\_signal'),$
$contain(Locus, Start, End, '[A/T]ATA').$
$tata_box(Locus, Start, End)$:–
$feature(Locus, Start, End, 'promoter'),$
$contain(Locus, Start, End, '[A/T]ATA').$
$contain(Locus, Start, End, Conseq)$:–
$subsequence(Locus, Start, End - Start + 1, Subseq),$
$consensus(Subseq, Conseq).$

"[A/T]ATA" is a consensus sequence of TATA boxes [BT86]. The predicates

$subsequence(Locus, Start, Len, Subseq)$ and $consensus(Subseq, Conseq)$

are built-in, specifying that the *Len* base pairs nucleotide sequence from *Start* of *Locus* entry is *Subseq* and that the consensus sequence *Conseq* matches *Subseq*, respectively. The following query retrieves all the TATA boxes based on the above rules.

% Query 3

$:-tata\_box(Locus, Start, End).$

% Answer 3

$tata\_box('AGMALPAIA', 200, 205).$
$tata\_box('BABADHCI', 190, 198).$
$tata\_box('CEBGLOBIN', 1680, 1685).$
$tata\_box('CEBGLOBIN', 8539, 8544).$
$\vdots$

The CPU time required by the bottom-up evaluation was about 15 seconds. In addition to the "TATA_signal", "misc_signal" and "promoter", which contain the consensus sequence of TATA boxes, were collected. Thus, this query

revealed 652 possible TATA boxes while 416 "TATA_signal"s were reported in the GenBank primates' entries.

Sequence Comparison

Sequence comparison is one of the most significant methods for analyzing nucleotide sequence data by computers because similar nucleotide sequences are considered to have almost the same biological function. The following rules are written to analyze the similarity of nucleotide sequences between *Locus1* and *Locus2* entries. They employ the most elementary form of the dot matrix method [GM70].

% Rule 4

$base(Locus, Pos, Nt)$:–
$subsequence(Locus, Pos, 1, Nt)$.
$neardown(Pos1, Pos2, Dist)$:–
$near(Len)$,
$Dist = Pos2 - Pos1$,
$Dist > 0$,
$Dist <= Len$.
$match(Locus1, Start1, Locus2, Start2, Len)$:–
$ktuple(Len)$,
$subsequence(Locus1, Start1, Len, Subseq)$,
$subsequence(Locus2, Start2, Len, Subseq)$.
$match(Locus1, Start1, Locus2, Start2, Len)$:–
$match(Locus1, Start1, Locus2, Start2, Len - 1)$,
$base(Locus1, Start1 + Len - 1, Nt)$,
$base(Locus2, Start2 + Len - 1, Nt)$.
$max\_match(Locus1, Start1, Locus2, Start2, Len)$:–
$match(Locus1, Start1, Locus2, Start2, Len)$,
$\neg match(Locus1, Start1, Locus2, Start2, Len + 1)$.
$\neg match(Locus1, Start1 - 1, Locus2, Start2 - 1, Len + 1)$.
$join(Locus1, Start1, Locus2, Start2, Len, 0, 0)$:–
$max\_match(Locus1, Start1, Locus2, Start2, Len)$.
$join(Locus1, Start1, Locus2, Start2, Len, Unmatch1, Unmatch2)$:–
$join(Locus1, Start1, Locus2, Start2, Lenup, Unmatch1up, Unmatch2up)$,
$max\_match(Locus1, Start1down, Locus2, Start2down, Lendown)$,
$neardown(Start1 + Lenup + Unmatch1up - 1, Start1down, Dist1)$,
$neardown(Start2 + Lenup + Unmatch2up - 1, Start1down, Dist2)$,
$Len = Lenup + Lendown$,

$Unmatch1 = Unmatch1up + Dist1 - 1,$
$Unmatch2 = Unmatch2up + Dist2 - 1.$
$similar(Locus1, Start1, Locus2, Start2, Len, Unmatch1, Unmatch2)$:–
$ktuple(6)$,
$near(4)$,
$join(Seq1, Start1, Seq2, Start2, Len, Unmatch1, Unmatch2)$.

The predicate *ktuple* represents a window size. The predicate *base* designates that the *Pos*-th residue of the nucleotide sequence of *Locus* entry is *Nt* and the predicate *neardown* designates that *Pos*2 is within *Dist* base pairs downstream of *Pos*1. The predicate *match* is recursively defined. When the nucleotide sequence of some region of the *Locus*1 entry is identical to the corresponding *Locus*2 entry and the adjacent nucleotide sequence of *Locus*1 is identical to that of *Locus*2, then the two continuous sequences are considered to be identical. Thus, the match of nucleotide sequences can be recursively defined. Furthermore, when there are unmatched residues between the two matched regions, the entire regions are considered to be similar and joined regions of the adjacent similar regions would also be similar, as defined in the rule of *join*.

Search for special sequence alignments

Repeated sequences that are oriented in opposite directions are called inverted repeats. While these have no known functions in protein sequences, they do indicate regions that are self-complementary for nucleotide sequences. Within inverted repeats along a single DNA or an RNA strand, an inverted repeat could form an intramolecular double helical segment if there are several residues separating the repeats. If the distance between the nucleotide sequence and its inversion is long, the entire segment is called a stem-and-loop. A stem-and-loop structure is often found in a region that controls a gene expression. Thus, in analyses of nucleotide sequences, searches for the special sequence alignments are significant. For example, the nucleotide sequence of the human transferrin receptor (TfR)'s iron responsive element (IRE) has the potential to form five similar stem-and-loop structures [MKR84, SOBW84]. Through these structures IRE is involved in the gene expression, although IRE is located in TfR's 3' untranslated region (3' UTR) rather than in a promoter region. Iron regulates the degradation rate of TfR mRNA through the TfR mRNA's IRE.

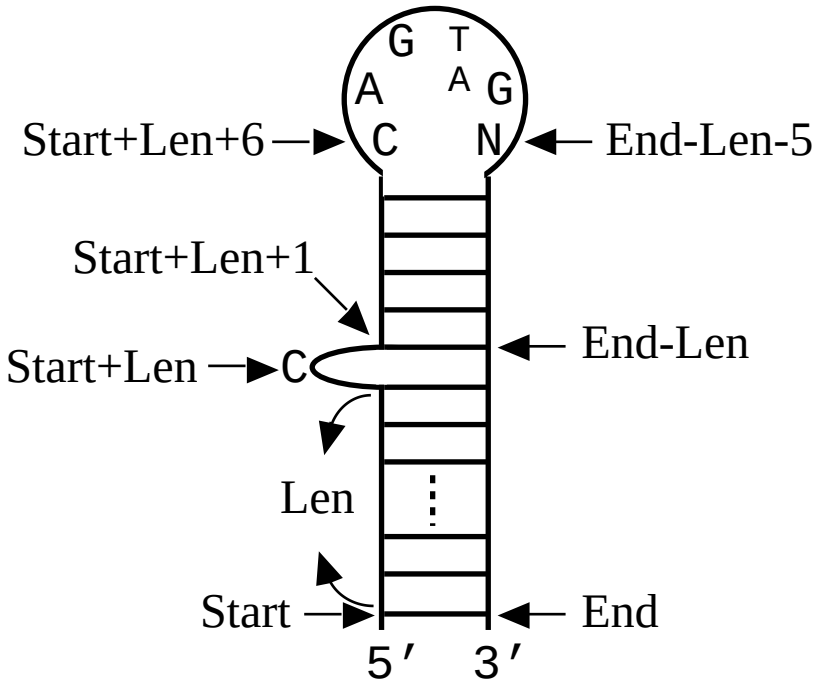


Figure 3.4: Schematic Stem-and-Loop Structure of TfR's IRE

Fig. 3.4 shows the schematic stem-and-loop structure of TfR's IRE. The IRE's stem-and-loop structures satisfy the following common features:

1. Consensus sequence of the loop is "CAG[A/T]GN."
2. The upper stem consists of five base pairs, and the lower stem varies in length.
3. There is an unpaired "C" separating the upper and lower stems.

It is difficult to detect these structures by only using existing similarity searches, although self-comparison of a nucleotide sequence can be the method of choice for detecting and characterizing inverted repeats in nucleic acids. On the other hand, logical rules are suited to expressing these complicated structures. Among the following rules, the rules defining *pair*,

stem and *max_stem* are generally used for stem structures or inverted repeats and are defined in a way similar to the rules for sequence comparison. The rules of *loop_IRE*, *stem_upper* and *IRE* are simple but sufficient for describing the features of the IRE's stem-and-loop. Variables correspond to those in Fig. 3.4. The fact $pair('G','T')$ is added to permit a "G-T" base pair in addition to the usual Watson-Crick's pairs.

% Fact 5

$pair('A','T').$
$pair('C','G').$
$pair('G','T').$

% Rule 5

$pair(Nt1, Nt2){:-}$
$pair(Nt2, Nt1).$
$stem(Locus, Start, End, Len){:-}$
$ktuple(Len),$
$subsequence(Locus, Start, Len, Subseq),$
$rev\_com(Subseq, RCseq),$
$subsequence(Locus, End - Len + 1, Len, RCseq).$
$stem(Locus, Start, End, Len){:-}$
$stem(Locus, Start, End, Len - 1),$
$base(Locus, Start + Len - 1, Nt1),$
$base(Locus, End - Len + 1, Nt2),$
$pair(Nt1, Nt2).$
$max\_stem(Locus, Start, End, Len){:-}$
$stem(Locus, Start, End, Len),$
$\neg stem(Locus, Start, End, Len + 1).$
$\neg stem(Locus, Start - 1, End + 1, Len + 1).$
$loop\_IRE(Locus, Start, End){:-}$
$End = Start + 5,$
$contain(Locus, Start, End, 'CAG[A/T]GN').$
$stem\_upper(Locus, Start, End){:-}$
$max\_stem(Locus, Start, End, 5).$
$IRE(Locus, Start, End){:-}$
$keyword(Locus, 'Transferrin'),$
$keyword(Locus, 'Receptor'),$
$ktuple(4),$
$max\_stem(Locus, Start, End, Len),$
$base(Locus, Start + Len, 'C'),$
$stem\_upper(Locus, Start + Len + 1, End - Len + 1),$

$$loop\_IRE(Locus, Start + Len + 6, End - Len - 5).$$

The predicate $rev\_com(Seq, RCseq)$ is built-in, designating that nucleotide sequences Seq and $RCseq$ can form base pairs.

% Query 5

$:-IRE(LOCUS, Start, End).$

% Answer 5

$IRE('HSTRR', 3429, 3461).$
$IRE('HSTRR', 3478, 3512).$
$IRE('HSTRR', 3883, 3913).$
$IRE('HSTRR', 3950, 3976).$
$IRE('HSTRR', 3996, 4024).$

In the RNA secondary structure prediction program of DNASIS (Hitachi Software Engineering Co., Ltd., Tokyo, Japan), the first and fourth regions of the IRE were not illustrated as stem-and-loop structures, and the lower stem of the second region was predicted to be one base pair shorter (data not shown). It took about 6 hours to predict the secondary structure on Mac Quadra 700. However, the deductive database system returned the answer in less than 2 minutes and in the answer all of the IRE's stem-and-loop structures are retrieved correctly since the DDBS computes all answers without use of any control constructs. The returned answers are potential stem-and-loop structures, identified by the simple dot matrix method. Detailed analyses, such as the minimum free energy calculation method [ZJT91], are required to confirm the answers.

3.5 Improvement of Query Evaluation

Through the application of DEE to a CAD database and a genome database, we noticed that the KRS method is sometimes less efficient than the Magic Set method. The reason was a poor approach to giving a SIPS (Sideways Information Passing Strategy). In this section, we propose a SIPS for efficient evaluation and the rule transformation method based on the SIPS.

3.5.1 SIPS in Deductive Databases

A deductive database system (DDBS) is a logic programming system that is required for handling large amounts of data. The logic programming system

usually does not provide an automatic control for evaluating orders of body literals (subgoals). Users must decide which is the best. On the other hand, the DDBS must provide such control because one of the advantages of the DDBS is the declarativeness of the queries.

The bottom-up evaluation is used, based on the fixpoint operation, and has the following advantages:

- It can be implemented easily.
- It always terminates and computes the answer set correctly for definite Datalog databases composed of range restricted rules.
- Some optimization techniques for relational databases are applicable; thus, large numbers of facts can be handled efficiently.

However, even by using semi-naive evaluation, the fixpoint operation is not sufficient for evaluating a given query, because there are many facts in the least model that are not relevant to the query. Therefore, there are many evaluation techniques for reducing irrelevant facts in the bottom-up evaluation by using the SIPS, which represents how bindings are passed through subgoals [BR87]. The Magic Set method [BR87], the NRSU method [NRSU89b] and the KRS method [KRS90] are typical ones using the SIPS. They transform given rules into more efficient rules based on a given SIPS.

The control in the DDBS can be divided into two steps. The first step is to give a SIPS. The second step is to implement the SIPS, for example, by using a rule transformation. The second step was actively researched and various methods such as the Magic Set method were proposed. The first step, however, has been researched very little. The problem is how the DDBS selects an efficient SIPS when the query evaluation methods are applied. Users can give the SIPS if they are very familiar with query processing. However, this leads to insufficient declarativeness for end users. We propose a method of automatically producing an efficient SIPS. It is indispensable to the implementation of DDBS and application to practical problems.

The rule transformation methods first produce a set of adorned rules P^{ad} according to a given SIPS, and then transform the adorned rules into more efficient ones. Each predicate in P^{ad} is attached to information called a binding pattern that represents the boundness of its arguments. A binding pattern is a string of b(representing a bound argument) and f(representing a free argument).

The Magic Set method can be applied to any recursive (or non-recursive) rules. The NRSU method is applicable to the rules in a class of left-, right-, and multi-linear rules. Although it is smaller than the class to which the Magic Set method can be applied, the NRSU method is more efficient than the Magic Set method. The KRS method is an extension of the NRSU method. The applicability of the NRSU method and the KRS method is decided by checking recursive rules in P^{ad}. The efficiency of transformed rules varies dramatically with SIPS's. Therefore, it is important for efficient query evaluation that the appropriate application of SIPS be determined.

The NRSU method and the KRS method, however, only provide methods of transforming rules in P^{ad}. The determination of SIPS's was not discussed. However, some heuristics have been proposed [Mor88]. They include:

- a SIPS that produces adorned recursive predicates with as many bound arguments as possible, and
- a SIPS that produces adorned recursive predicates with as few types of binding patterns as possible.

Intuitively, for the more bound arguments in recursive predicates, methods produce more efficient transformed rules. However, we show that, for some programs, contrary to this intuition, the KRS method produces less efficient programs by using the SIPS for the more bound arguments. We also propose a method for giving a SIPS in such a case. The method first determines a SIPS for recursive rules and then transforms the rules.

3.5.2 SIPS and the KRS Method

We assume that:

- the program is Datalog, which is a set of Horn clauses without function symbols,
- rules in the program are range restricted, and
- all recursive rules are linear. (A recursive rule is linear if the head predicate appears only once in the body).

Rules are adorned to distinguish bound arguments from free arguments of IDB predicates in the rules. An argument of a body literal is bound if the variable in the argument position is bound to a constant when the literal is evaluated. An argument of a body literal is free if the variable in the argument position is not bound to any constants when the literal is evaluated.

SIPS(Sideways Information Passing Strategy) [BR87] is a strategy for deciding how binding information should be passed through the body when a rule is evaluated. It is represented as a labeled directed graph whose edges are labeled with bound variables and nodes represent predicates (or sets of predicates). The argument position of the bound variables are adorned with b and the positions of the free variables are adorned with f.

Example 3.5.1 Consider the following program:

$r_1 : query(C_1, C_2) \text{:}-t(C_1, C_2), anc(C_1, C_2).$
$r_2 : anc(X, Y) \text{:}-par(X, Z), anc(Z, Y).$
$r_3 : anc(X, Y) \text{:}-par(X, Y).$

When the query $:-query(C_1, C_2).$ is given, the following represents the SIPS evaluating the rule r_1 from left to right.

$$\{query_h, t\} \rightarrow_{C_1, C_2} anc$$

The subscript h means that $query$ is the head predicate. The SIPS evaluating the rule r_2 from left to right is:

$$\{anc_h\} \rightarrow_X par; \quad \{anc_h, par\} \rightarrow_{Z, Y} anc$$

The adorned program based on the SIPS is as follows.

$r_1' : query^{\mathrm{ff}}(C_1, C_2) \text{:}-t(C_1, C_2), anc^{\mathrm{bb}}(C_1, C_2).$
$r_2' : anc^{\mathrm{bb}}(X, Y) \text{:}-par(X, Z), anc^{\mathrm{bb}}(Z, Y).$
$r_3' : anc^{\mathrm{bb}}(X, Y) \text{:}-par(X, Y).$ □

A binding pattern is a string of b and f that indicate a bound argument and a free argument, respectively.

The KRS method can be applied to right-, left- and multi-linear recursive rules and produces more efficient rules than those produced by the Magic Set

method. The KRS method is applied to an adorned program P^{ad} by using the following procedures. We describe the transformation for right-linear recursive rules. Suppose that the right-linear recursive rule in P^{ad} is r^{α} and the head predicate is p^{α}.

The KRS method

Step 1 Apply the Magic Set method to the rules without a p^{α} head predicate.

Step 2 Produce the following rule:

$$mc\_p^{\alpha}(\bar{C}, \bar{C}) :- T_1, \ldots, T_k.$$

from the rule that has p^{α} in its body, except r^{α}:

$$q(\bar{X}) :- T_1, \ldots, T_k, p^{\alpha}(\bar{C}, \bar{Y}), T_{k+1}, \ldots, T_{k+\ell}.$$

$\bar{C}$ is a row of bound variables designated by α and $\bar{Y}$ is a row of free variables by α. This rule is used to generate contexts.

Step 3 Produce the following rule:

$$mc\_p^{\alpha}(\bar{C}, \bar{Z}) :- mc\_p^{\alpha}(\bar{C}, \bar{X}), G_1, \ldots, G_i.$$

from the rule:

$$r^{\alpha} : p^{\alpha}(\bar{X}, \bar{Y}) :- G_1, \ldots, G_i, p^{\alpha}(\bar{Z}, \bar{Y}).$$

$\bar{X}$ and $\bar{Z}$ are rows of bound variables designated by α, and $\bar{Y}$ is a row of free variables by α.

Step 4 Produce the following rule:

$$ac\_p^{\alpha}(\bar{C}, \bar{Y}) :- mc\_p^{\alpha}(\bar{C}, \bar{X}), H_1, \ldots, H_j.$$

from the rule that has the p^{α} head predicate, except r^{α}:

$$p^{\alpha}(\bar{X}, \bar{Y}) :- H_1, \ldots, H_j.$$

Step 5 Replace the predicate p^{α} that appears in the program transformed by Step 1 $\sim$ Step 4 by ac_p^{α}. □

Example 3.5.2 Application of the KRS method to the adorned program in Example 3.5.1 is as follows:

Step 1 Apply the Magic Set method to the rule r_1'.

Step 2 $\sim$ 4 Produce the rules r_4 (Step 2), r_5 (Step 3) and r_6 (Step 4) from the rules r_1', r_2' and r_3', respectively.

$$r_4 : mc\_anc^{bb}(C_1, C_2, C_1, C_2) :- t(C_1, C_2).$$
$$r_5 : mc\_anc^{bb}(C_1, C_2, Z, Y) :- mc\_anc^{bb}(C_1, C_2, X, Y), par(X, Z).$$
$$r_6 : ac\_anc^{bb}(C_1, C_2) :- mc\_anc^{bb}(C_1, C_2, X, Y), par(X, Y).$$

Step 5 Replace anc^{bb} in r_1' by ac_anc^{bb} □

3.5.3 SIPS and Rule Transformation for Efficient Query Processing

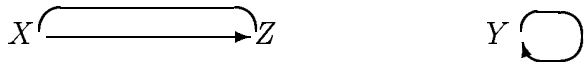
We propose a method of using binding information for the efficient query processing of a subclass of the linear recursive class. We also propose a rule transformation method based on this information.

Dependency relationships between variables in recursive rules

We define a graph, which represents relationships between variables in a recursive rule, to determine an efficient binding pattern by judging from the form of the recursive rules. The graph is a variation of I graph [YHH88] by eliminating labels and weights.

Definition 3.5.3 We call the graph $G_r = (V, E_d, E_u)$, a V-graph for each recursive rule r. V is a set of nodes, and they are the variables that appear in the recursive rule. E_d is a set of directed edges, from the variables that appear in the head predicate to the variables that appears in the same argument position in the same predicate. E_u is a set of undirected edges, and its elements connect the variables in each EDB predicate. □

Example 3.5.4 The V-graph of the rule r_2 in the example 3.5.1 is as follows.



□

Definition 3.5.5 A strongly connected component in a V-graph is isolated, if all the following conditions are satisfied:

- The strongly connected component has only one node.
- The input degree and the output degree of the node are both 1.
- The only edge in the component is directed. □

Example 3.5.6 The strongly connected components with the node Y in the V-graph in Example 3.5.4 are isolated. □

Definition 3.5.7 A node is included in a pseudo-isolated strongly connected component, if the component is isolated when all the undirected edges originating from the node are removed. □

Let us consider the recursive rules whose V-graphs have (pseudo-)isolated strongly connected components. These rules include right- and left-linear recursive rules.

Rule transformation for right-linear recursive rules

The outline of the rule transformation for right-linear recursive rules is as follows:

1. Produce a V-graph G_r for each recursive rule r in the given program. Using G_r, determine the binding pattern α for the head predicate p of the rule r. α is the most general binding pattern among the patterns for r to be right-linear. A binding pattern α is more general than α', if the bound argument positions in α are also bound in α'.
2. Generate an adorned program P^{ad} based on the SIPS that makes the binding pattern for p in the program be α.

3. Transform P^{ad} based on a variation of the KRS method.

A more precise procedure of the transformation is shown below.

First, we determine the binding pattern α for a recursive rule r using its V-graph G_r. The binding for an argument position is f in α, if the node corresponding to the position is included in an isolated strongly connected component in G_r. Otherwise, the binding is b. The binding pattern α is the most general among the patterns for r to be right-linear.

Suppose that the recursive rule r is the following rule,

$$p(X_1, \ldots, X_m, Y_1, \ldots, Y_n) :- G_1, \ldots, G_k, p(Z_1, \ldots, Z_m, Y_1, \ldots, Y_n).$$

and the binding pattern for the predicate p obtained by using the above procedure is:

$$\alpha = \overbrace{\mathsf{b} \cdots \mathsf{b}}^{m} \overbrace{\mathsf{f} \cdots \mathsf{f}}^{n}$$

We give the SIPS:

$$\{q_h, T_h, \ldots, T_i\} \rightarrow_{\bar{c}} p$$

to each rule containing the predicate p in its body:

$$q(\bar{X}) :- T_1, \ldots, T_k, p(\bar{C}, \bar{Y}), T_{k+1}, \ldots, T_{k+\ell}.$$

where $\{T_h, \ldots, T_i\} \subseteq \{T_1, \ldots, T_k, T_{k+1}, \ldots, T_{k+\ell}\}$. $\bar{X}, \bar{C}$ and $\bar{Y}$ represent rows of variables. $\bar{C}$ is the row of m variables, and $\bar{Y}$ is the row of n variables. Each variable in $\bar{C}$ should appear in $T_h, \ldots, T_i$, or should be a bound variable in q_h. If we produce the adorned program P^{ad} based on the SIPS, then the binding pattern for the predicate p in P^{ad} is α.

The feature of the SIPS is that even if a variable in $\bar{Y}$ appears in $T_h, \ldots, T_i$, it is not considered as a bound variable. However, the binding pattern, in which such a variable is considered to be bound, should be remembered. We call the binding pattern α'. The binding pattern α is more general than α'. The binding pattern α' is used in the Steps T1-2 and T2-4 in the following Transformation (1), to reduce the number of irrelevant facts.

If the SIPS cannot be given to make the binding pattern of the predicate p to be α, the recursive rule r cannot be right-linear. We do not consider such a rule in the following transformation, but will address it later.

The following is the transformation for the program P^{ad}. If $\alpha = \alpha'$, then the KRS method should be applied. We assume that the binding patterns are:

$$\alpha = \overbrace{\mathsf{b}\cdots\mathsf{b}}^{m}\overbrace{\mathsf{f}\cdots\mathsf{f}}^{n}, \quad \alpha' = \overbrace{\mathsf{b}\cdots\mathsf{b}}^{m+j}\underbrace{\mathsf{f}\cdots\mathsf{f}}_{n-j}$$

where $1 \leq j \leq n$.

Transformation (1)

Step T1-1 Apply the Magic Set method to the rules that do not have the predicate p^α in its head.

Step T1-2 Assume that the rule:

$$q(\bar{X}){:-}T_1, \ldots, T_k, p^\alpha(\bar{C}, \bar{Y}), T_{k+1}, \ldots, T_{k+\ell}.$$

has p^α in its body and is not the rule r^α. From this rule, produce the following rules:

$$mc\_p^\alpha(\bar{C}, \bar{C}){:-}T_h, \ldots, T_i.$$
$$check(\bar{C}, Y_1, \ldots, Y_j){:-}T_h, \ldots, T_i.$$

where $\bar{C}$ is the row of bound variables according to α, and $\bar{C}, Y_1, \ldots, Y_j$ is the row of bound variables according to α'.

Step T1-3 Produce the rule:

$$mc\_p^\alpha(\bar{C}, \bar{Z}){:-}mc\_p^\alpha(\bar{C}, \bar{X}), G_1, \ldots, G_s.$$

from the recursive rule for the predicate p^α

$$p^\alpha(\bar{X}, \bar{Y}){:-}G_1, \ldots, G_s, p^\alpha(\bar{Z}, \bar{Y}).$$

where $\bar{X}$ and $\bar{Z}$ are the rows of bound variables according to α and $\bar{Y}$ is the row of free variables according to α.

Step T1-4 Produce the rule:

$$ac\_p^\alpha(\bar{C}, \bar{Y}){:-}mc\_p^\alpha(\bar{C}, \bar{X}), H_1, \ldots, H_u, check(\bar{C}, Y_1, \ldots, Y_j).$$

from the non-recursive rule with p^α as its head predicate

$$p^\alpha(\bar{X}, \bar{Y}){:-}H_1, \ldots, H_u.$$

Step T1-5 Replace the predicate p^{α} in the transformed program with the predicate ac_p^{α}. □

The difference between the Transformation (1) and the KRS method described in the section 3.5.2 is the Step T1-2 and T1-4. Step T1-2 produces a rule for generating *check* facts. Step T1-4 produces a rule for reducing irrelevant facts by appending the *check* literal to its body.

Theorem 3.5.8 The program transformed by the Transformation (1) for right-linear recursive rules produces the same answer as the original right-linear program. □

Example 3.5.9 The following program is obtained by applying the Transformation (1) to the program ($r_1 \sim r_3$) in Example 3.5.1.

$r_4' : mc\_anc^{\mathrm{bf}}(C_1, C_1) \text{:-} t(C_1, C_2).$
$r_4'' : check(C_1, C_2) \text{:-} t(C_1, C_2).$
$r_5' : mc\_anc^{\mathrm{bf}}(C_1, Z) \text{:-} mc\_anc^{\mathrm{bf}}(C_1, X), par(X, Z).$
$r_6' : ac\_anc^{\mathrm{bf}}(C_1, Y) \text{:-} mc\_anc^{\mathrm{bf}}(C_1, X), par(X, Y), check(C_1, Y).$
$r_7' : query(C_1, C_2) \text{:-} t(C_1, C_2), ac\_anc^{\mathrm{bf}}(C_1, C_2).$

Suppose that the database for the relation (predicate) t is given as follows:

$$\{(x_i, y_j) \mid 1 \leq i \leq m, 1 \leq j \leq n\}$$

The number of tuples in t is $m \times n$. C_1 and C_2 are given m and n types of binding, respectively.

We shall now consider the comparison between the numbers of facts produced by the above program and the program in Example 3.5.2. The number of facts produced by rule r_4 is $m \times n$. On the other hand, rule r_4' produces m facts. When the variable Z in rule r_5 is substituted with a binding, it produces n facts for the binding. Rule r_5 produces new facts only if a new binding is given to the variable Z (The value for the variable Y is always the same as the value of C_2). Rule r_5' produces only one fact when a binding is given to the variable Z. Therefore, if the number of facts produced by the rules r_4 and r_5 is k and rules r_4' and r_5' produce k' facts, then the equation $k = n \times k'$ holds.

The number of facts produced by rule r_6 is at most $m \times n$. Assuming that the number is ℓ, we can observe that rule r_6' produces at least ℓ facts,

because r'_6 produces facts without the binding given to variable C_2. The same number of facts are produced by rules r_7 and r'_7. Rule r''_4 produces the same number of facts as that produced by rule r_4. However, if the relation t is small, the overhead necessary for producing *check* facts is negligible. □

The following are the conditions under which the Transformation (1) generates an efficient program, judging by the example above.

- Rule r'_5 produces more facts than r'_4.
- For a binding to the variable X, several bindings to the variable Y exist, i.e. the relation t is not one-to-one. This condition holds, for example, when the bindings to X and Y are given by different relations.

If rules r'_4 and r'_5 produce the same number of facts as r_4 and r_5, the Transformation (1) generate a less efficient program than the program generated by the KRS method. However, even in that case, since the arity of mc_anc^{bf} is less than that of mc_anc^{bb}, it may be still efficient.

The rule transformation for left-linear recursive rules

First, we determine the binding pattern for a recursive rule r to be left-linear using the V-graph G_r.

The binding for an argument position is b in α, if the node corresponding to the position forms an isolate strongly connected component in G_r. Otherwise, the binding is f. The binding pattern α is the one for r to be left-linear. An arbitrary binding pattern (except α) for r to be left-linear is more general than α.

Suppose that the rule r is a linear recursive rule in the form of:

$$p(X_1, \ldots X_m, Y_1, \ldots, Y_n) \text{:-} p(X_1, \ldots X_m, Z_1, \ldots, Z_n), G_1, \ldots, G_k.$$

and the binding pattern for the predicate p based on the above procedure is:

$$\alpha = \overbrace{\mathsf{b}\cdots\mathsf{b}}^{m}\overbrace{\mathsf{f}\cdots\mathsf{f}}^{n}$$

Next, we give the SIPS to make the binding pattern for p to be α, as in the case of right-linear rules, and generate the adorned program P^{ad}. The KRS

method should be applied. Therefore, our method only give the SIPS in the left-linear case.

Considering rules $r_1 \sim r_3$ in Example 3.5.1, change rule r_2 into the following rule.

$$r_2 : p(\bar{X}, \bar{Y}) :- p(\bar{X}, \bar{Z}), G_1, \ldots, G_{k'}.$$

The following program is obtained by applying the KRS method, based on the above SIPS.

$$r'_1 : mc\_anc^{\mathrm{bf}}(C, C) :- t(C, Y).$$
$$r'_2 : ac\_anc^{\mathrm{bf}}(C, Y) :- ac\_anc^{\mathrm{bf}}(C, Z), par(Z, Y).$$
$$r'_3 : ac\_anc^{\mathrm{bf}}(C, Y) :- mc\_anc^{\mathrm{bf}}(C, X), par(X, Y).$$

If we make the binding pattern for *anc* to be bb and transform the program, the following program is obtained.

$$r''_1 : m\_anc^{\mathrm{bb}}(X, Y) :- t(X, Y).$$
$$r''_2 : anc^{\mathrm{bb}}(X, Y) :- m\_anc^{\mathrm{bb}}(X, Y), ac\_anc^{\mathrm{bf}}(X, Z), par(Z, Y).$$
$$r''_3 : anc^{\mathrm{bb}}(X, Y) :- m\_anc^{\mathrm{bb}}(X, Y), par(X, Y).$$
$$r''_4 : mc\_anc^{\mathrm{bf}}(C, C) :- m\_p^{\mathrm{bb}}(C, Y).$$
$$r''_5 : ac\_anc^{\mathrm{bf}}(C, Y) :- ac\_anc^{\mathrm{bf}}(C, Z), par(X, Y).$$
$$r''_6 : ac\_anc^{\mathrm{bf}}(C, Y) :- mc\_anc^{\mathrm{bf}}(C, X), par(X, Y).$$

The rules $r'_1 \sim r'_3$ produce the same number of facts as $r''_4 \sim r''_6$. Therefore, our method can produce an efficient program by the number of facts produced by the rules $r''_1 \sim r''_3$.

3.5.4 Performance Evaluation on DEE

We measured the CPU time and the number of produced facts of the transformation method in section 3.5.3 on the query evaluator in DEE. We used Sun SparcStation2 (64 MB memory).

Performance evaluation of right-linear recursive rules

We measured the CPU time and the number of produced facts of the method for the following program and query. We used a binary tree for the relation *par* (depth 7 and 8); the numbers of the tuple in the relation t are 1 ($\sharp t = 1$) and 12 or 25. When several tuples are in the relation t, we consider two

cases: one case is where there is a functional dependency between the first argument and the second argument of t (FD); the other case is where there is no functional dependency between the arguments of t.

Program:

$query(X,Y)$:$-t(X,Y),anc(X,Y)$.
$anc(X,Y)$:$-par(X,Y)$.
$anc(X,Y)$:$-par(X,Z),anc(Z,Y)$.

Query:

$:-query(X,Y)$.

Below is the result of the evaluation. Row α represents the result of the evaluation of our method (anc^{bf}), and row α' represents the evaluation of the KRS method applied to the adornment anc^{bb}. For each column of $\sharp t$, the left column shows the CPU time (sec) and the right column shows the number of the produced faces.

	Depth 7					
	$\sharp t = 1$		$\sharp t = 12$		$\sharp t = 12$(FD)	
α	0.47	258	0.79	291	0.97	934
α'	0.31	257	21.61	3084	1.06	922

	Depth 8					
	$\sharp t = 1$		$\sharp t = 25$		$\sharp t = 25$(FD)	
α	1.17	514	2.13	586	2.75	2376
α'	0.88	513	371.41	12825	3.11	2351

Our method is more efficient than the KRS method when there is no functional dependency between the arguments of t, although the two methods indicate almost the same efficiency when the number of tuples in t is 1 or when there is a functional dependency between the first and second argument in t.

Performance evaluation of left-linear recursive rules

We show the result of the two SIPSs; one that makes the recursive rule be left-linear, and the other that gives as many bound arguments as possible to the recursive predicate. The program is as follows:

Program:

$query(X,Y)$:$-t(X,Y),anc(X,Y)$.
$anc(X,Y)$:$-par(X,Y)$.
$anc(X,Y)$:$-par(X,Z),anc(Z,Y)$.

Query:

:$-query(X,Y)$.

The data are the same as those in the application to the right-linear rule.

	Depth 7					
	$\sharp t = 1$		$\sharp t = 12$		$\sharp t = 12$(FD)	
α	0.19	256	0.22	267	0.49	919
α'	0.49	258	3.73	291	1.00	934

	Depth 8					
	$\sharp t = 1$		$\sharp t = 25$		$\sharp t = 25$(FD)	
α	0.37	512	0.44	536	1.46	2347
α'	1.25	514	58.14	586	2.80	2376

When we can make the recursive rule left-linear, our method is more efficient than the SIPS, which gives as many bound arguments as possible to the recursive predicate.

3.5.5 Related Works

In this section, we have assumed that the KRS method is applied. Yet, we must consider whether our method is also useful for other methods. When we apply the NRSU method, for instance, the transformation corresponding to the $\sharp t = 1$ case results in the improvement of the KRS method. Therefore, our method does not affect the efficiency of the NRSU method. When the Magic Set method is applied, we should consider the following three cases.

- If a recursive rule is right-linear, we cannot introduce the *check* predicate. Therefore, our method is not necessarily efficient. However, since the KRS method generates more efficient programs than the Magic Set method in general, we do not have to consider the Magic Set method in the right-linear case.

- If a recursive rule is left-linear, our method generates a more efficient program than the Magic Set method, because the Magic Set method generates the same irrelevant rules as those in the KRS method.

- If the KRS method cannot be applied to a recursive rule, although our method reduces the number of irrelevant Magic facts, the number of other facts can increase. Therefore, our method is not necessarily efficient. Thus, the determination of whether our method can achieve efficiency must be addressed through further study.

3.6 Discussion

We developed and evaluated DEE on the CAD system for petrochemical plants and the genome database based on GenBank. Our experience in this development and evaluation has made the following facts clear.

(1) Most queries for retrieving complex configurations in petrochemical plants can be written in the form of logical rules (recursive and negative) for DDB. It is difficult to describe complex configurations without recursion or negation. Recursive and negative rules are also useful for expressing secondary structure searches in nucleotide sequences. We believe, furthermore, that DDB techniques are useful for queries based on conditional connectivities between components of CAD databases, not only for petrochemical plans but also for other engineering diagrams, as well as for secondary or tertiary structure searches in nucleotide and amino acid sequences.

(2) The users of logic programming, such as Prolog, must write a control of inferences to terminate the inference and improve efficiency. For end users, it is difficult to write programs containing an appropriate control strategy. In DDB, the users do not need to describe the control. Thus, this feature of DDB is important for end users.

(3) We proposed a method for determining the SIPS from the set of rules, when linear recursive rules are evaluated in DDBSs. We also proposed a method for transforming the adorned program based on the SIPS. Using our method, an efficient evaluation can be achieved, even though the KRS method produces less efficient programs if the binding pattern of the recursive predicate has as many b's as possible. Most of the current rule transformation methods, such as the KRS method and the NRSU method, are applied to adorned programs and the generation of adorned rules depends on systems.

Our method determines the SIPS to generate the adorned program, which can be transformed into a more efficient one, using these methods.

(4) As mentioned in section 3.2.2, our technique for handling negations is based on the assumption that the set of rules is stratified and allowed. In the complex configuration retrieval of petrochemical plants, as well as signal sequences and secondary structure searches in nucleotide sequences, all rules satisfy these conditions. These assumptions can be considered as a natural restriction.

(5) The performance of the DEE system shows that the techniques of DDB are applicable and useful for the maintenance of P&I-Ds produced by the CAD system of actual petrochemical plants, and for the various searches in genome databases. In the maintenance phase of CAD and searches of genome databases[4], a short response time is not required. This fact means that a DDBS may be put to practical use.

(6) By improving the efficiency of the deductive engine of our system and performing deduction on high performance workstations, DDBS may be useful in the design phase of CAD systems.

(7) The Magic Set method is effective for most queries. However, there are some queries that cannot be solved within a practical response time. In these cases, other rule transformation techniques should be combined with the Magic Set method. Therefore, the framework for DEE, which consists of a bottom-up evaluation and a rule transformation based on the Magic Set method, is effective because various query processing techniques can be easily implemented.

(8) The NRSU and KRS method mentioned in section 3.2.1 are applicable to many rules describing complex configuration retrieval problems. For the rules in the class that the NRSU and KRS are applicable to, these methods together with our method of determining the SIPS are certainly effective.

[4]Actually, the homology search program FASTA or BLAST takes a few to a dozen minutes to search a genome databank. Of course, this is not sufficient and there is currently active research being conducted on this search.

We evaluated the applicability of a deductive database system to CAD systems and genome databases by applying it to the maintenance of petrochemical plants and various searches in nucleotide sequences, respectively. We also clarified the functions necessary for high level query support of CAD databases. Our system is still under development and is essentially a prototype. Much research and development still remains to be done. We are investigating the following problems.

- In section 3.3.1, we mentioned that a set-handling function is necessary for high level retrieval on CAD systems. Our system does not fully provide support for that function. We are investigating how to add this function to DEE.

- Our system uses an existing CAD database, which is a relational database. The bottom-up evaluator of DEE translates tuples in the database into a set of facts whose data structure is flat. In CAD systems, components and diagrams have complex structures. Genome data also have complex structures; therefore some GenBank data are not used in our application. To represent these structures, more flexible data structures will be necessary. For example, object-oriented database systems are suitable for handling such structures. We intend to reconstruct the database by using an object-oriented database and unifying it with our deductive engine. We will discuss this approach in the following chapters.

- The current version of DEE has three recursive query processing techniques. We intend to implement other query processing methods and develop knowledge bases for supporting selection of the optimal method for a given query from various transformation methods.

- Method of determining the SIPS:

    - If the binding pattern of a recursive predicate consists of only b, our method can produce both right- and left-linear recursive rules. We must develop a procedure for determining which is more efficient.

    - We did not consider the mixed-linear recursive case, where both a right-linear rule and a left-linear rule exist. Development of

a method for producing an efficient program in such a case is necessary.

- It is important in determining a SIPS to consider information about the facts, such as the number of facts of each relation (predicate) and the existence of functional dependencies among their arguments. If we can estimate the size of the produced facts by using this information, we can generate more efficient rules using the method.

Chapter 4

Design and Implementation of a Deductive Object-Oriented Database

4.1 Background

4.1.1 Necessity of Flexible Data Management for Genome Databases

In chapter 3 we showed the applicability of a deductive database system to practical problems. Its deductive function, using logical rules, is powerful enough to build and test hypotheses. However, it is difficult to manage various types of information in the form of tuples. Two types of problems pertaining to the management of data arose out of the application to genome analyses. First, because genome data are often complex and nested, some of them cannot be stored in the form of tuples. Second, several databanks have been developed independently. Although some databanks have cross references, users must write programs to search over them by using the cross references. Frequent join operations would be necessary, even if we could manage them using a DDB or RDB system.

Here we address the first problem in detail, taking a genome databank as an example. Fig. 4.1 shows an entry of GenBank [BCF+92], which is one of the more widely used nucleotide sequence databanks. It includes not

```
LOCUS       HSA1MBG1     7656 bp    DNA             PRI       24-APR-1991
ACCESSION   X54816
KEYWORDS    alpha-1-microglobulin; bikunin; glycoprotein; proteinase inhibitor.
FEATURES             Location/Qualifiers
     exon            1554..1802
                     /number=1
                     /product="alpha1-microglobulin"
     exon            3152..3294
                     /number=2
                     /product="alpha1-microglobulin"
                               ⋮
     intron          5856..6859
                     /number=4
     intron          6962..>7656
                     /number=5
     CAAT_signal     1273..1282
     GC_signal       1389..1395
     CAAT_signal     1432..1438
     TATA_signal     1524..1530
     mRNA            join(1554..1802,3152..3294,4856..4932,5739..5855,6860..6956,
                     X45817:302..348,X54818:172..253,X54818:1339..1506,X54818:
                     1832..2005,X54818:5658..2801)
                     /gene="alpha1-microglobulin-bikunin"
                     /label=A1MB_mRNA_a
                     /note="alternate"
     mRNA            join(1595..1802,3152..3294,4856..4932,5739..5855,6860..6956,
                     X45817:302..348,X54818:172..253,X54818:1339..1506,X54818:
                     1832..2005,X54818:5658..2801)
                     /gene="alpha1-microglobulin-bikunin"
                     /label=A1MB_mRNA_b
                     /note="alternate"
     CDS             join(1686..1802,3152..3294,4856..4932,5739..5855,
                     6860..6961,X54817:302..348,X54818:172..253,
                     X54818:1339..1506,X54818:1832..2005,X54818:2658..2689)
                     /gene="alpha1-microglobulin-bikunin"
                     /label=A1MB_cds
                     /product="bikunin"
                     /codon_start=1
                     /translation="MRSLGALLLLLSACLAVSAGPVPTPPDNIQVQENFNISRIYGKW
                                                  ⋮
                     EYCGVPGDGDEELLRFSN"
BASE COUNT     1791 a   2015 c   1953 g   1893 t      4 others
ORIGIN
        1 gatcacctga ggtcgggagt tagagaccag tctggccaat atggtgaaac gctgtctcta
       61 ctaaaaatac aaaaattagc caggcatggt ggtgggcacc tgtaatctca gctactcggg
      121 aggctgaggc acaagaatca cttcagccca gaatgtggag gttgcagtga gccgagatcg
                                                  ⋮
     7501 tgcagtcaaa catactggct gctgccatgg ggtgggcagc tccaggtgct ggcatggtgc
     7561 cagcactctg taagggtatg gctggaccag gtgcaccaca agcagatttc ccagctggcg
     7621 ccagggaatg gagtggtgct tggagctcgg agatgc
//
```

Figure 4.1: A Part of a GenBank Entry

only a nucleotide sequence (ORIGIN), but also its name (LOCUS), identifier (ACCESSION), keywords (KEYWORDS), reference data (not shown), and biological features of the sequence (FEATURES). In the FEATURES field, we can see much information about the sequence, for instance, feature keys (exon, intron,...), correspondence of the key to the sequence (Location), a translated amino acid sequence, and a produced gene. We can not store all the information in a table.

Fig. 4.2 shows tables of GenBank data that were converted into the form of tuples and used in the application of DEE to genome analyses. They are linked by the key attribute *Locus*. We stored FEATURES data in the Table *Features*. Some biological features were not stored in the table, because Qualifiers such as /number, /product, /gene, and /note vary with the feature keys. In this example, the key exon includes /number and /product, while the key CDS includes /gene, /label, /product, /codon_start and /translation. Furthermore, they sometimes include natural language description. It is difficult to store them in a table.

Another reason why we could not store all biological features in the table is because of joined regions of Location for the keys mRNA and CDS (coding regions of a protein). The joined regions often include references to other entries. In this case, while the accession number of the entry is X54816, the entries with accession numbers X54817 and X54818 are referred.

The variety of sequence lengths is another problem in managing genome data. The human nucleotide sequences kept in GenBank vary from less than 10 base pairs (bp) to more than 50,000 bp in length. If nucleotide sequences are stored as text type in RDB systems, the retrieving methods using a database query language such as SQL are limited. Therefore, it is necessary to combine the query language and the application program for comparing nucleotide sequences (e.g. FASTA [PL88], BLAST [AGM$^+$90]). This leads to the problem of the impedance mismatch between the query language and the programming language. We dealt with this problem by dividing nucleotide sequences into 8 nucleotide long overlapping oligonucleotides (Fig. 4.2: Table *Sequences*).

In the research on genome data, nucleotide sequences and their biological features are frequently referred to. For saving disk space, it is necessary to store these data in separate tables; therefore, these tables must be frequently joined.

Some entries are related to each other, because the relationship between

Table *Keyword*

Locus	Keyword
HSA1MBG1	alpha-1-microglobulin
HSA1MBG1	bikunin
⋮	⋮

Table *Features*

Locus	Start	End	Signal
HSA1MBG1	1554	1802	exon
HSA1MBG1	3152	3294	exon
⋮	⋮	⋮	⋮
HSA1MBG1	5856	6859	intron
HSA1MBG1	6962	7656	intron
HSA1MBG1	1273	1282	CAAT_signal
⋮	⋮	⋮	⋮

Table *Sequences*

Locus	Start	Sequence
HSA1MBG1	1	GATCACCT
HSA1MBG1	2	ATCACCTG
⋮	⋮	⋮

Figure 4.2: Examples of Genome Data Used in the Application of DEE

GenBank entries and genes is not one-to-one. Several entries may refer to the same gene and some DNA entries have mRNA entries and protein data that originate from them. Hence, the data structure that can store an attribute for another set of entries is necessary for enabling the databases to manipulate the genome data.

Object-oriented database (OODB) systems are suitable for managing such information because of their ability to handle various types of data, such as nested relations and long text data, such as nucleotide sequences. Some applications of OODB systems to multimedia databases show this ability.

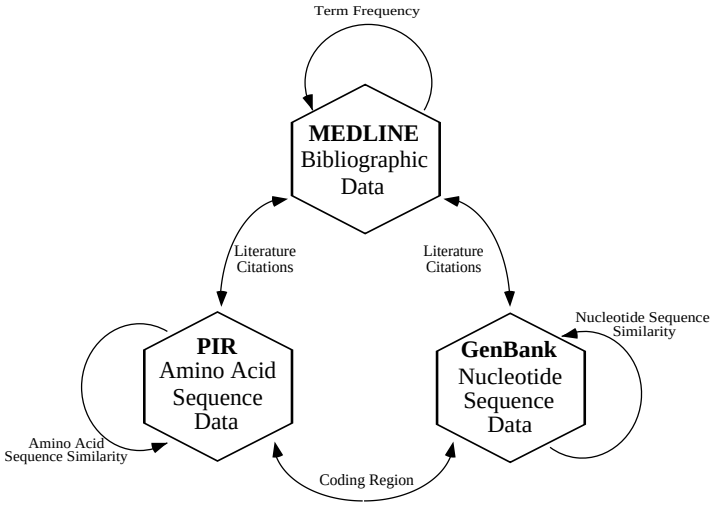


Figure 4.3: Multidatabank Structure of Entrez

Recent database systems developed for genome data emphasize the use of OODBs for managing the data and integrating the databanks. The Entrez system [Nat93] is one of those systems and was developed to integrate GenBank, PIR (an amino acid sequence databank) and MEDLINE (a bibliographic databank). This multidatabank structure with neighbors within databanks and hard links between them is shown in Fig. 4.3. The data in the system are described in ASN.1 [Nat92], which is a data description language with an object-oriented concept.

4.1.2 Necessity of Flexible Analyses in Genome Databases

The systems using OODB are powerful, and can manage complex data and integrate several databanks. However, they lack a common query language interface and do not provide a flexible query expression for performing the analyses described in chapter 3.

For example, the following typical genome analysis shows the necessity of the flexible query expression4.4. Assume that we determined sequences

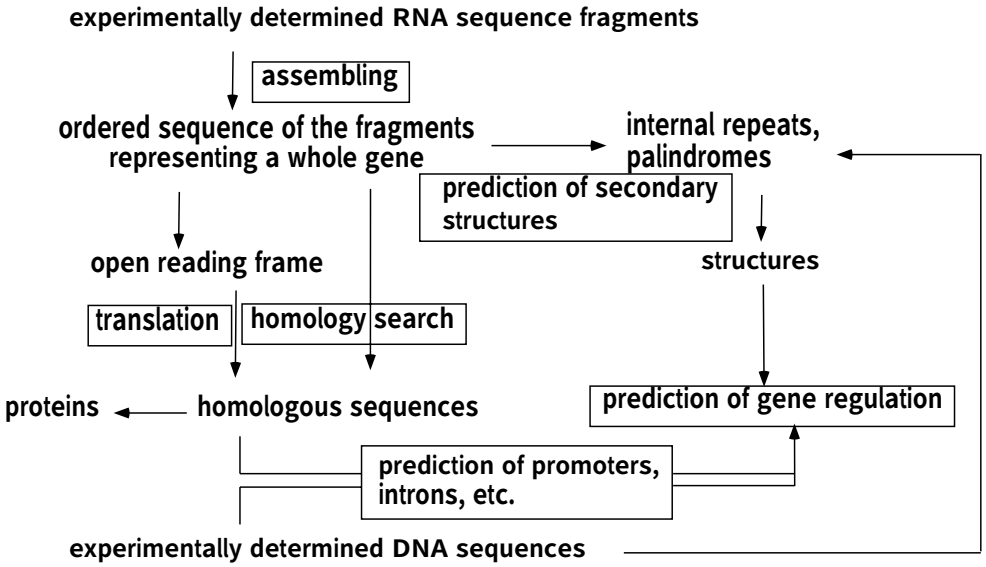


Figure 4.4: A Typical Genome Analysis

experimentally (containing 300∼500 nucleotides). First of all, we must assemble the sequences so that they may construct a contiguous sequence that represents a whole gene (usually containing several thousands of nucleotides). Then we will search a nucleotide sequence databank for homologous sequences in order to know what kind of biological function it possesses. In case we are interested in proteins, we must search an amino acid sequence databank by translating the nucleotide sequence into the amino acid sequence. When we want to know the regulation of a gene (the sequence), we must check whether the retrieved homologous sequence has biological features related to the regulation, and must predict the secondary structure of the original sequence in order to check whether it has a secondary structure related to the regulation. Although this flow of analyses is routine work, it consists of many types of programs or algorithms, and is therefore very complex. Hence, it is much more complicated to repeat the analysis with changing parameters or algorithms and perform further analyses, for instance, than to deduce knowledge on evolution.

Flexible analyses should be easily performed in genome databases. A de-

ductive language can provide an interface for performing such analyses. Recently, there has been much research in supporting OODB systems through logical languages [KW89, KL89, TNF91, BNR+87, CKW89, LO91]. However, it is mainly focused on theoretical aspects and there are few systems and applications.

4.2 Design of a Deductive Object-Oriented Database for Genome Analyses

We designed a deductive object-oriented database for genome analyses. It consists of an object-oriented database and a deductive language for the database.

4.2.1 Approach to Integrating an OODB and a Deductive Language

The requirements for genome databases discussed in section 4.1 are summarized as follows:

- management of complex data such as `FEATURES`,
- integration of several databanks currently developed independently,
- interface for flexible genome analyses.

The OODBs are sufficient for the first two requirements, while they are not sufficient for the third one.

Fig. 4.5 shows some of the classes in an OODB that we designed for genome data. (The details are described in chapter 5.) A GenBank entry is stored in some classes by using an attribute for complex objects. Integration between databanks is implemented by using cross references.

The data stored in the OODB were from GenBank, PIR and GDB original flat files. Even though they should be updated frequently, users usually do not (or should not) update them. Only the users who have access to the data from existing databanks and system administrators are responsible for the update of the data. Therefore, an interface for flexible analyses that

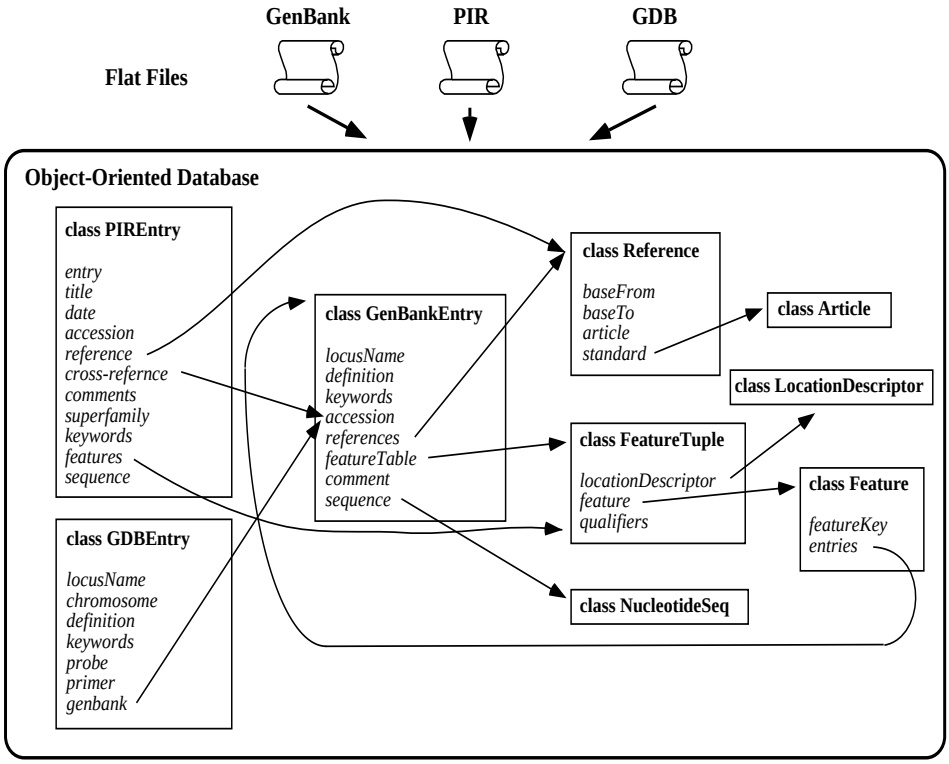


Figure 4.5: An Example of Classes for Genome Data
The words in *italic type* show attributes of the classes

enables users to easily access and manipulate the data in the OODB should be developed.

Our approach to integrating an OODB and a DDB is:

1. design and implement classes in an OODB for the genome data, by using a commercial OODB management system,

2. design and implement a deductive language on the database for flexible analyses.

The classes are shown in chapter 5. In this chapter, we show the deductive language and an overview of the system.

4.2.2 Deductive Language for Object-Oriented Databases

In OODB systems, although there are some functions like SQL in the RDB system, the standard query languages are not established yet. In addition, users have to write a program with such procedural language as C++ to achieve a high level query. Especially in the research in the genome project, it is necessary that queries be easily described for building and testing hypotheses and representing higher structures of nucleic acids or proteins.

The database is considered to be a set of facts in DDBs. In RDBs, each row of the table is a tuple, and a tuple is easily transformed into a fact. On the other hand, OODBs contain varieties of data comprised of both complex types, such as arrays, lists and sets, as well as simple data types such as integers and strings. Therefore, it is difficult to manage them in the form of facts.

Another problem is the representation of methods in DDBs. Furthermore there are some methods that return a set of objects. We must implement message passing to each object in the set.

To cope with these problems, several logical languages for OODB systems have been proposed [KW89, KL89, TNF91, BNR+87, CKW89, LO91]. These languages, however, have not yet been applied to practical problems. We designed a deductive language for OODBs. The rules in the language contain class atoms for sets of objects and method terms for message passing. Although it is a subset of those languages like LLO [LO91], it is useful enough for genome analyses. In this subsection, we introduce an overview of the language. Since a genome database merely requires updates for data structure, the proposed query processing function can handle rules in the current fixed class hierarchy.

The syntax of the language

We introduce a class atom and a method term. The class atom is used to retrieve instances of the class. The method term is used to invoke a method of access to an attribute value of an object. It can be extended to invoke a method with arguments. The syntax of the language is as follows.

Terms:

- Constants and variables are terms.
- If f is a function symbol with n arguments and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
- If t is a term, m is a method, and $t_1, \ldots, t_n$ are terms, then $t.m(t_1, \ldots, t_n)$ is a term. We call this a method term.

Atoms:

- If A is a class name and X is a variable, then $A(X)$ is an atom. This means:

  $$A(X) \text{:-} X \text{ is an instance of the class } A.$$

  We call it a class atom.
- If p is a predicate symbol with n arguments and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an atom. We call it an ordinary atom.

Literals: Atoms and negations of atoms are literals, and we call them positive literals and negative literals, respectively.

Rules: A Rule is a clause of the following form:

$$H \text{:-} L_1, \ldots, L_n.$$

where H is an ordinary atom. H is called the head of the rule. $L_1, \ldots, L_n$ is a conjunction of literals and called the body of the rule.

4.3 System Description

Fig. 4.6 shows an overview of our system. We use a commercially available OODB system GemStone (version 3.1) and its Smalltalk Interface. We use the primate data file gbpri.seq (about 70MB) in GenBank's release 78. The program which transforms the GenBank file into the data for GemStone is written in ObjectWorks\Smalltalk R4.1. The size of the database is about 120MB.

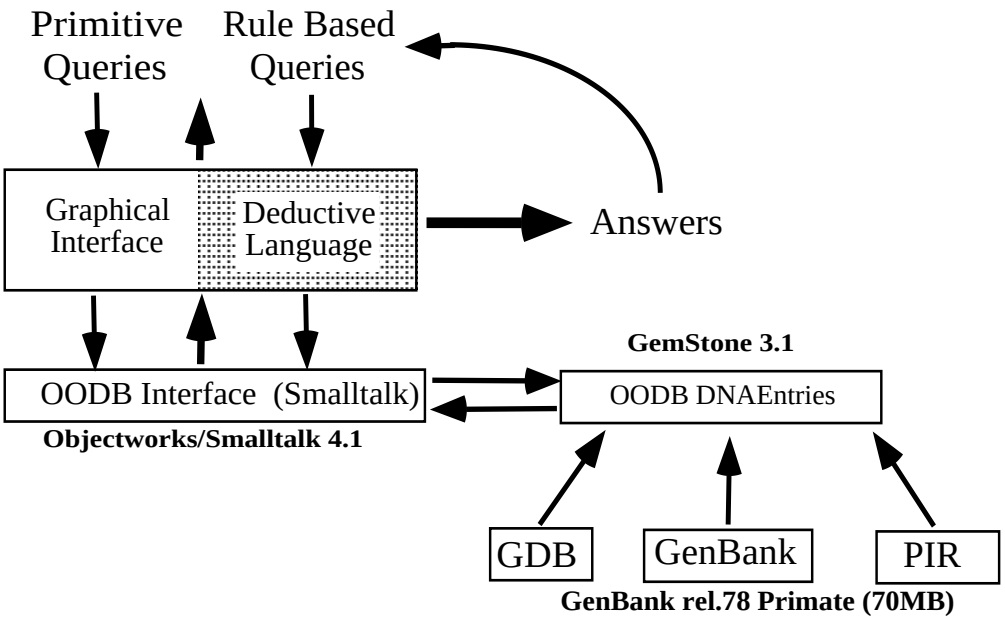


Figure 4.6: An Overview of the System

We also developed a graphic user interface to search for entries by keywords, biological features, and other simple conditions (e.g. the name of an entry).

An example of a GenBank window is shown in Fig. 4.7.

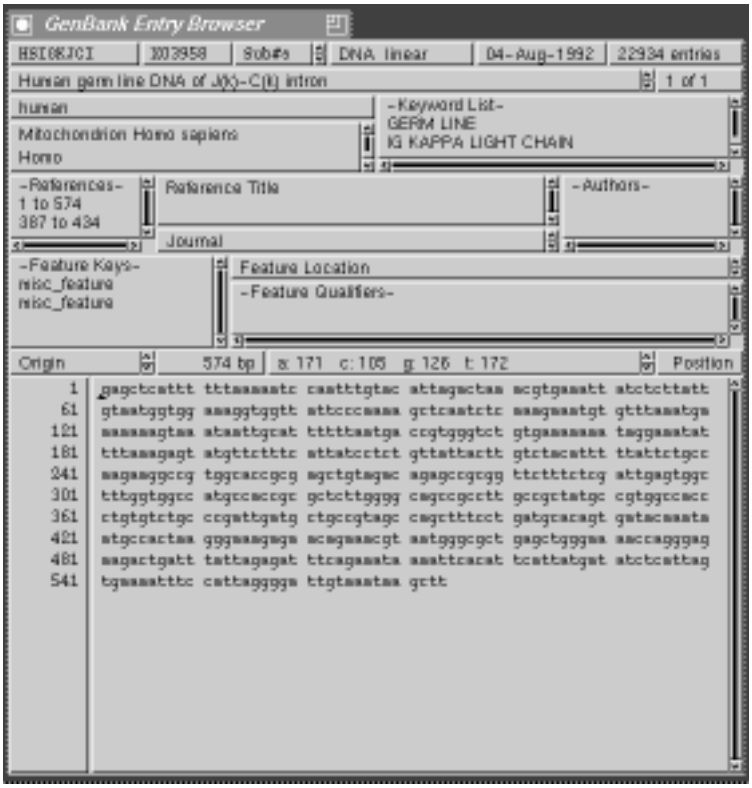


Figure 4.7: An Example of a GenBank Window.

Chapter 5

Application to Genome Analysis

5.1 Background

Recently, there have been many projects concerning genome analysis. Of these, the human genome project is one of the largest, and its goal is to discover all human genetic information.

The genome of an organism is the set of chromosomes which contains the entire genetic information. There are various kinds of genetic information, which include nucleotide sequences of DNA, amino acid sequences of proteins, tertiary structures of proteins, physical and genetic maps of genes, and their relationships.

A large amount of genome data has been collected to achieve this goal. Several databases were developed in order to organizing data experimentally obtained. The following are examples:

- databases for nucleotide sequences of DNA (GenBank [BCF+92], EMBL, DDBJ)
- databases for amino acid sequences of proteins
    - primary sequences (PIR [BGMT92], SwissProt)
    - motif: the common feature of sequences that have the same function (PROSITE [Bai92])

- a database for tertiary structures of proteins (PDB [BKW+77])
- a database for physical and genetic maps of genes (GDB [PMFR92])
- a database for bibliographic information (MEDLINE)

The databases include various kinds of data and are closely related to each other. For instance, GenBank includes features of sequences, keywords, and references to bibliographic data, as well as nucleotide sequences.

There are three problems associated with the databases. First, they are managed by relational databases or flat files, even though they contain data with complex structures. Hence, their efficient management for allowing searches for various types of information is difficult. Second, they were developed independently, and it is therefore difficult to search over several databases.

The third problem is that there is no framework for analysis of genome data, such as building and testing biological hypotheses. There is a need for discovering a common feature (sequences of nucleotides or amino acids) in the function of a protein, evolutionary distance between two proteins, the location of a gene and its relationship to the other genes that are located near it, etc. Users build hypotheses from such new knowledge and test them using the databases. Therefore, the project requires a framework for such issues, as well as efficient collection of the data.

The conventional genome databases restrict users to only two kinds of searches:

- homology search: search for sequence data similar to a given sequence.
- keyword search: search for entries that include given keywords. An entry in a database corresponds to one unit of experimentally obtained data, which does not necessarily correspond to a gene.

For further analyses of their results, the users must search another database or write programs. Therefore, a framework for easily analyzing genome data in databases is required. We adopted the idea of a deductive object-oriented database (DOOD) paradigm to meet this requirement for the following reasons.

- Object-oriented database (OODB) paradigm is useful for managing complex genome data that have various attributes and for integrating several databases.
- Deductive database (DDB) paradigm is useful for providing a framework for flexible analyses; for instance, for describing complex search conditions and for reusing the result of one query for another.

There has been much research on DOOD [CKW89, LO91], but it has mainly focused on theoretical aspects and there are few systems and applications. We have developed a deductive database system for analyzing GenBank data [STS93]. As previously reported [GST93], we managed GenBank data via an OODB and developed a rule-based query interface, which enables users to manipulate a set of objects.

5.2 Application to Genome Analyses

Users of the current databanks [Kam92] are confronted with two major problems. First, it is not possible to easily search for complex data such as biological features of a sequence, because the data are managed by using relational database systems and distributed in a flat file format. Second, it is also difficult to search for a relationship between data in several databanks, because they are developed and managed independently.

5.2.1 Classes of Genome Data in the OODB

The most important class for GenBank is the class GenBankEntry, which corresponds to the GenBank entry and contains all data fields (LOCUS, DEFINITION, etc., in the example in Fig. 4.1) as its attributes. The attributes *id* and *sequence* are necessary for the data not only in GenBank but also in the other data banks. Those attributes are inherited from the class DNAEntry. Classes for the other data banks could be easily designed as the subclasses of the class DNAEntry.

There are attributes defined as complex objects such as *references* and *featureTable* (Fig. 5.1). The attribute *featureTable* in the class GenBankEntry is defined as a set of instances of the class FeatureTuple. The class FeatureTuple is for biological features of the sequences. As described in the

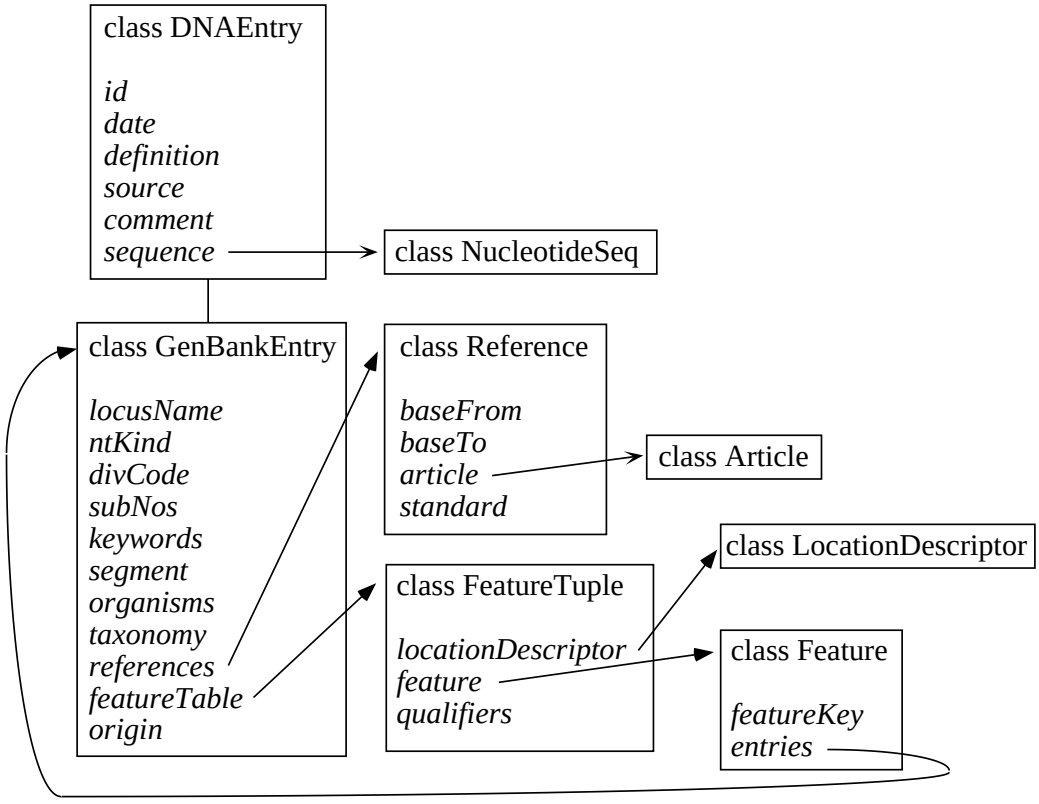


Figure 5.1: The Class GenBankEntry and Its Related Classes

previous section, they are important for researchers and are very complex. It has *locationDescriptor*, *feature* and *qualifiers* as attributes. The value of the attribute *feature* is an instance of the class Feature. The class Feature has *featureKey* for "exon," "5'UTR," etc., and a set of pointers to the GenBankEntry in which the instance of *featureKey* is included. The attribute *featureQualifiers* is a set of instances of the class Qualifier, which describes complex objects.

The value of the attribute *locationDescriptor* is the instance of the class LocationDescriptor. This is either the instance of the class SimpleLocation or the instance of the class ComplexLocations such as 'join(426..463,654..670),'

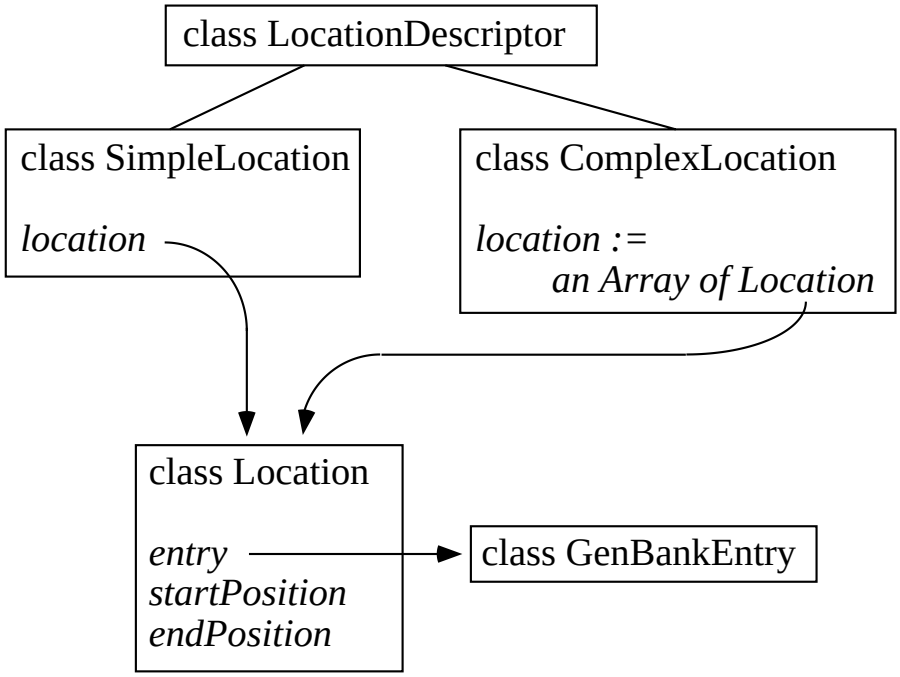


Figure 5.2: The Class LocationDescriptor and Its Related Classes

as shown in Fig. 4.1. The class Location contains the attributes *startPos* and *endPos*, representing the regions of biological features in the nucleotide sequence (Fig. 5.2). These classes have the methods length and toSequence. The method length returns the length of the features and the method toSequence returns the sequence to the corresponding feature.

The attribute *sequence* in the class GenBankEntry is an instance of the class NucleotideSeq. It is a subclass of the class Seq that contains the attribute *sequence*, which represents a sequence as a string (Fig. 5.3). The methods length and composition are defined in the class NucleotideSeq. The method length returns the length of the nucleotide sequence and the method composition returns the number of each nucleotide (the number of A, C, G, T respectively) in the sequence. Users do not have to distinguish these methods from the attributes because they can give the query for the computed value in the same form of the query for values stored in attributes.

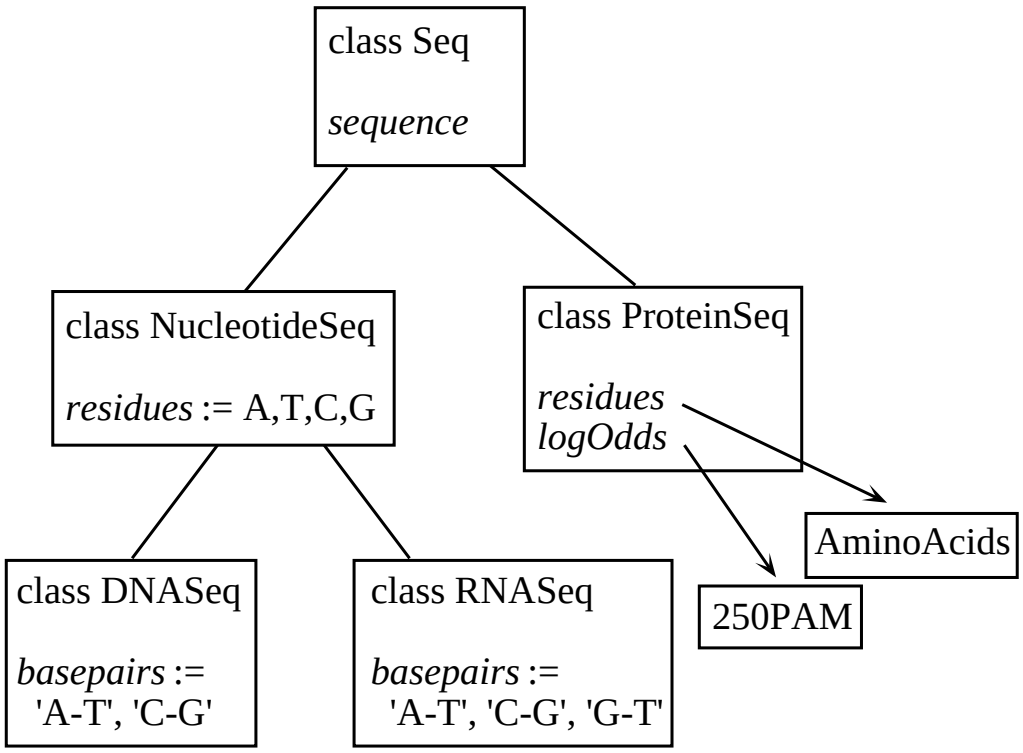


Figure 5.3: The Class Seq and Its Related Classes

In the class NucleotideSeq, the method homology is defined. It retrieves homology among nucleotide sequences. This ability enables the database to have the same function as homology retrieval programs, which are difficult to implement using the built-in SQL function in the RDB systems. This method makes the combination of retrievals easy, such as retrievals based on keywords and homology searches of nucleotide sequences. In the class ProteinSeq, an attribute is defined for storing the information of proteins.

5.2.2 Integration of Several Databanks

We can integrate other databases into it by creating pointers from them to the GenBank entries, because they have cross reference fields such as *#Cross-*

reference in PIR and *GenBank* in GDB (Fig. 4.5).

Other approaches to the integration include Entrez system and HyperGenome system. The Entrez system is an integration of GenBank, PIR and MEDLINE. Numerous predefined links between data enable the users to traverse databases. HyperGenome system is the integration of GenBank and GDB, using the *GenBank* field in GDB.

Although these systems are very useful for traversing databases, integrating databases alone is not sufficient. In these systems, the relationships between data were defined beforehand by the system developers. They do not provide a framework in which the users can define a relationship between data. However, such a framework is necessary for building and testing biological hypotheses. It is also necessary to integrate the conventional methods of searching genome databases. For example, if we have an experimentally obtained nucleotide sequence with an unknown function, the following analyses are necessary:

- search a nucleotide sequence database for homologous sequences and check their functions;
- transform it into amino acid sequence data, search an amino acid sequence database for homologous sequences and check their functions,
- compare the results of the above and predict the function of the obtained sequence;
- search for a tertiary structure using the homologous amino acid sequences and derive the relationship between the nucleotide sequence and tertiary structures of proteins; and
- search a mapping database for physical and genetic maps of the sequence and related genes.

Of course, combinations of each type of analysis, plus additional analyses are necessary for acquiring complete genetic information. A framework for deriving new knowledge from the integrated database, including an easy method of performing such analyses, is required. The framework should provide easy methods for:

- extracting detailed information regarding an answer to a query, such as attribute values of an object,

- storing the result of a query and other biological knowledge,
- using them for answering other queries and acquiring new knowledge,
- describing various complex search conditions easily, and
- searching by trial and error; that is, making similar queries after changing some parameters.

A deductive language provides the methods for these analyses. However, conventional deductive languages (the first order predicate languages) do not provide a way of handling objects in the OODB and some object-oriented features. It is thus necessary to extend the language to enable integration and management of complex objects using an OODB management system. The deductive language in our system can meet this requirement.

5.2.3 Examples of Genome Analyses Using the Language

Example 1: Suppose that there are two classes, Feature and GenBankEntry, mentioned in section 5.2.1. First, we retrieve the objects in the class Features where *featureKey* is 'CAAT_signal.' Second, we retrieve instances in the class GenBankEntry which contain the objects retrieved from the class Features. Finally, we obtain *locusNo*, *locusName* and *features* of the retrieved objects; as for *features*, we obtain *location* and *featureKey*. The rules and the query that execute above retrieval are as follows.

$$
\begin{aligned}
&caat\_signal(Entry){:-}\\
&\qquad Feature(X),\\
&\qquad X.key =' CAAT\_signal',\\
&\qquad Entry = X.entries.\\
&query(LocusNo, LocusName, Location, Key){:-}\\
&\qquad caat\_signal(Entry),\\
&\qquad LocusNo = Entry.accession,\\
&\qquad LocusName = Entry.locus,\\
&\qquad Feature = Entry.features,\\
&\qquad Location = Feature.location,\\
&\qquad Key = Feature.key.
\end{aligned}
$$

The atom $Feature(X)$ is a class atom, and X is substituted by an instance of the class Feature. The terms $X.key$ and $X.entries$ are method terms, therefore *key* and *entries* should be the methods defined in the class Feature. □

Example 2: Next we show an example that contains recursion. The following rules are an example of the combination of a keyword search and a homology search.

$$
\begin{aligned}
&cf\_features(Locus, Features) :- \\
&\qquad GenBankEntry(X), \\
&\qquad X.keyword = '\,cystic\ fibrosis', \\
&\qquad Locus = X.locus, \\
&\qquad Features = X.feature. \\
&cf\_sequence(Locus, Sequence) :- \\
&\qquad GenBankEntry(X), \\
&\qquad X.keyword = '\,cystic\ fibrosis', \\
&\qquad Locus = X.locus, \\
&\qquad Sequence = X.sequence. \\
&cf\_exon(Locus, Seq) :- \\
&\qquad cf\_features(Locus, Features) \\
&\qquad Region = Features.region, \\
&\qquad Seq = Region.sequence. \\
&same\_gene(Locus1, Locus2) :- \\
&\qquad cf\_sequence(Locus1, Sequence1), \\
&\qquad cf\_sequence(Locus2, Sequence2), \\
&\qquad homology(Sequence1, Sequence2, Similarity), \\
&\qquad Similarity > 95.0. \\
&same\_gene(Locus1, Locus2) :- \\
&\qquad cf\_exon(Locus1, Sequence1), \\
&\qquad cf\_exon(Locus2, Sequence2), \\
&\qquad homology(Sequence1, Sequence2, Similarity), \\
&\qquad Similarity > 95.0. \\
&same\_gene(Locus1, Locus2) :- \\
&\qquad cf\_sequence(Locus1, Sequence1), \\
&\qquad cf\_exon(Locus2, Sequence2), \\
&\qquad homology(Sequence1, Sequence2, Similarity), \\
&\qquad Similarity > 95.0. \\
&same\_gene(Locus1, Locus2) :-
\end{aligned}
$$

$$same\_gene(Locus2, Locus1).$$
$$same\_gene(Locus1, Locus2):-$$
$$same\_gene(Locus1, Locus),$$
$$same\_gene(Locus, Locus2).$$

The first two rules retrieve the entries that have keyword '*cystic fibrosis*' in their keyword. They retrieve the locus names and the features of the entries, and the locus names and the sequences of the entries, respectively. The rule for $cf\_exon(LocusName, Sequence)$ retrieves the exon subsequence of the entry (designated by LocusName). *same_gene*($LocusName1$, $LocusName2$) means that the entry whose LocusName is $LocusName1$ and the entry whose LocusName is $LocusName2$ can be the same gene, because the similarity of the two sequences is more than 95% . The predicate *homology* is built-in. This homology search function can be implemented as a built-in method in the OODB system by using the BLAST program. □

Example 3 Suppose that we have an experimentally obtained nucleotide sequence and we want to know its function. Fig. 5.4 shows the analyses required to determine the function of the sequence. In this case, we search over two databases, GenBank and PIR. We show the analyses using the deductive language following the arrows numbered (1) $\sim$ (10) in Fig. 5.4.

(1) Homology search in GenBank: There are tools for homology search such as FASTA [PL88] and BLAST [AGM+90]. We can define the homology search as a built-in predicate in a way that is different from the previous example. Here, we use

$$homology(Sequence, Database, Result)$$

where $Sequence$ is a given sequence, $Database$ is a database from which we search for homologous sequences (GenBank in this case, and PIR in the case of (5) and (6)), and $Result$ represents an object that has the attributes *entry*, *score*, etc. The following rule can be defined to search for an entry whose sequence is homologous with the given sequence.

$$genbankHomology(Sequence, Entry):-$$
$$homology(Sequence, genbank, Result),$$
$$Entry = Result.entry.$$

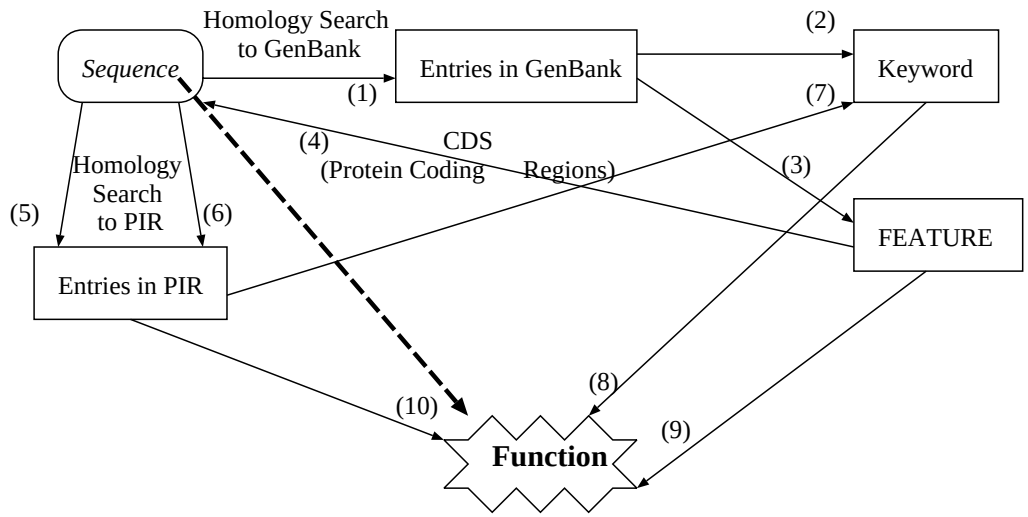


Figure 5.4: An Example of Analysis for Acquiring the Function of a Given Sequence.

(2),(3) Each entry in GenBank has the attributes $keywords$, $featureTable$, $sequence$, etc. Thus, we can get keywords from the entries obtained in (1) by defining the following rule:

$$keywordHomology(Sequence, Keyword) :-$$
$$genbankHomology(Sequence, Entry),$$
$$Keyword = Entry.keywords.$$

We can also obtain the biological features of the sequence in the entry in the same way.

(4) We can select sequences that have protein coding regions (cds) from the entries obtained in (1).

$$cds(Sequence, CDS) :-$$
$$genbankHomology(Sequence, Entry),$$
$$FeatureTuple = Entry.featureTable,$$
$$Feature = FeatureTuple.feature,$$

$Feature.featureKey =$ "cds",
$CDS = FeatureTuple.sequence($"$cds$"$)$.

where *sequence* is a method defined in the class FeatureTuple, which returns a subsequence of the sequence in the entry corresponding to the feature key specified by its argument.

(5),(6) We can define a homology search in PIR in the same way as (1).

(7) We can obtain keywords in the same way as (2).

(8)∼(10) Using the above results, we can define the function of the sequence in various ways (e.g. conjunction of keywords, features and description of PIR, or their disjunction). In the case where we want to define the function as the keywords of the homology sequences, the following rule can be defined.

$$predictedFunction(Sequence, Function) :- keywordHomology(Sequence, Function).$$

If we want to define the function as the keywords that appear in both GenBank and PIR entries, we can define *keywordHomology* as follows:

$$keywordHomology(Sequence, Keyword) :- genbankHomology(Sequence, Entry1), pirHomology(Sequence, Entry2), Keyword = Entry1.keywords, Keyword = Entry2.keywords.$$

It is easy to redefine a homology search in GenBank. Using a recursive rule, we can obtain more entries than in (1). In addition to the rule in (1), we define the following rule:

$$genbankHomology(Sequence, Entry) :- cds(Sequence, CDS), genbankHomology(CDS, Entry).$$

We can also easily define similar functions as hypotheses using the results of (2) and (3):

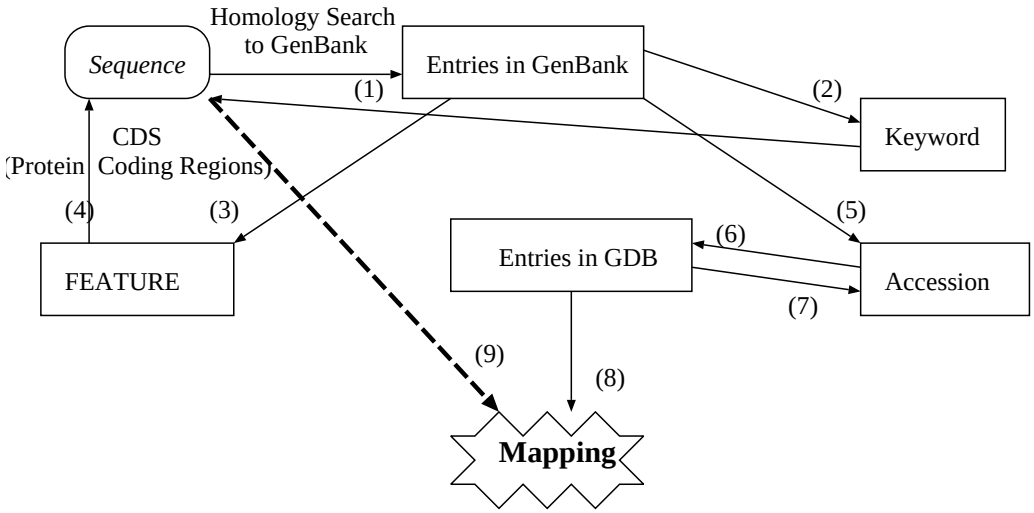


Figure 5.5: An Analysis for Acquiring the Mapping Information of a Given Sequence (Example 4).

$$similar\_function(Func1, Func2) :- \\ keywordHomology(Sequence, Func1), \\ keywordHomology(Sequence, Func2).$$

Of course, we can use the results of (5), (6) and their combinations as body literals. □

Example 4 This example represents a relationship between a given nucleotide sequence and its mapping information obtained by searching both GenBank and GDB.

(1)∼(4) Homology search in GenBank, as in Example 3: We can reuse the definition of $genbankHomology$ and $keywordHomology$ in Example 1.

(5)∼(9) We can use cross references from GDB to GenBank to search for mapping information of the homologous sequences obtained in (1). The rules defining the relationship between the homologous sequences in GenBank and GDB entries are as follows:

$gdb\_to\_genbank(GDBEntry, Accession)$:–
$GDBEntry(X)$,
$Accession = X.genbank$.
$genbank\_to\_gdb(Sequence, GDBEntry)$:–
$genbankHomology(Sequence, Entry)$,
$gdb\_to\_genbank(GDBEntry, Entry.accession)$.

$GDBEntry$ is the class name for the GDB data. We can extract the mapping information ($chromosome$) from the answers to the query
$:-genbank\_to\_gdb(Sequence, GDBEntry)$. □

Comparison to conventional analyses
If the users want to perform analyses similar to those in the above examples using genome databases that are managed by flat files or other conventional techniques, they must enter commands for FASTA or BLAST. To obtain a database entry, they must then edit the resulting file for later analyses. Otherwise, the users must write programs.

In the object-oriented database, users can define various methods for these commands, such as homology search, and obtaining the sequence in the coding region. However, they must also write programs. In our system, the Smalltalk code for the $genbankHomology$ recursive rules in Example 1 would be the same as that shown in Fig. 5.6, which is more complicated than the deductive rules.

5.3 Discussion

The genome database is required for handling data derived from biological experiments. These data are not in the preferred form for conventional relational or deductive databases. Therefore, object-oriented databases that can handle various data types and represent complex data structures are suitable for managing genome data.

We have developed a deductive object-oriented database for genome analysis. It consists of an object-oriented database for genome data and a deductive language for flexible querying.

Our deductive language does not support all object-oriented features like LLO [LO91], such as method definition and inheritance, and class definition

```
genbankHomologyInit: aSequence
    | cds results |
    entries := Set new.
    cds := Set new.
    results := self blast: aSequence db: 'genbank'.
    results do: [ :result |
        | entry |
        entry := result entry.
        entries add: entry.
        (entry features) do: [ :feature |
            (feature key) = 'CDS'
               ifTrue: [cds add: (feature sequence: 'CDS')]]].
    cds isEmpty
        ifFalse: [self genbankHomology: cds].
    ^entries

genbankHomology: aSeqSet
    aSeqSet do: [ :aSeq |
        | tmpEntries cds |
        tmpEntries := Set new.
        cds := Set new.
        results := self blast: aSeq db: 'genbank'.
        results do: [ :result |
            | entry |
            (entries includes: entry)
                ifTrue: [tmpEntry add: entry
                         entries add: entry]].
        tmpEntry isEmpty ifFalse: [
            tmpEntry do: [ :entry |
                (entry features) do: [ :feature |
                    (feature key) = 'CDS'
                       ifTrue: [cds add: (feature sequence: 'CDS')]].
            cds isEmpty ifFalse: [self genbankHomology: cds]]]].
    ^entries
```

Figure 5.6: A Smalltalk Code for the Recursive Rules in Example 3.

and inheritance. These features all depend on an OODB framework, and our language can call up a method from an OODB and search for objects. It provides a simple and powerful way of searching for objects and building biological hypotheses.

Various work is still required in the future. The establishment of more sophisticated data modeling is necessary for defining the relationship between databases is necessary. Several GenBank entries have information on the chromosomes (sometimes the reference to GDB) in the COMMENTS or FEATURES qualifiers fields. This is useful information, even though it is written in the form of natural language. Among our future goals is the use

```
Seq-entry ::= set {
  id {giim {id 340068},
      genbank {name "AGMPPINS", accession "X61092"}},

  descr {title "C.aethiops gene for preproinsulin",
         genbank {source "Cercopithecus aethiops DNA.",
                  keywords {"insulin", "preproinsulin"},
                  date "09-MAR-1992",
                  div "PRI",
                  taxonomy "Eukaryota; ... Cercopithecinae."},
         org {taxname "Cercopithecus aethiops"},
         pub {pub {gen {serial-number 2},
                   gen {cit "Title=""Sequences of ...in  monkeys"",
                             Journal=""Unpublished (1991)""" ,
                   authors {names str {"Seino,S.",..., "Li,W."}}}}},

  inst {repr raw,
        mol dna,
        length 1909,
        strand ds,
        seq-data iupacna "GGGCCATCCATGGGGGCATC...CACGCTCTCT" } ,
  annot {{data ftable {{data imp {key "5'UTR"},
                       location mix {
                          int {from 425, to 462, id genbank {accession "X61092"}},
                          int {from 653, to 669, id genbank {accession "X61092"}}}},...}
```

Figure 5.7: An Example of Genome Data in ASN.1 Format

of this information for establishing this relationship.

It is also necessary to develop a framework for storing rules and their results in an OODB so that we can utilize them to acquire new knowledge. From the viewpoint of implementation, a method of reducing the number of homology searches, one of the most time consuming processes in genome analysis, needs to be developed.

Our system is now using the data from a GenBank flat file. A distribution of the genome data in an ASN.1 [Nat92] format (Fig. 5.7) will be available. We are planning to use these data in future research.

Chapter 6

Conclusion

The findings of this research project can be summarized in the following points.

1. A prototype of a query evaluator DEE based on the rule transformation technique and the fixpoint evaluation was developed. DEE employs several existing transformation techniques such as the Magic Set method, the NRSU method and the KRS method. We applied DEE to configuration detection problems of petrochemical plants and secondary structure retrieval of RNA sequences. Performance of several query processing techniques were measured using real world examples. Applicability of a query processing technique in deductive databases to these problems is feasible.

2. Experience with DEE shows that in some cases the KRS method produces a less efficient program than the original one. We developed a method of determining the SIPS and transforming the program into an efficient one, even if the KRS method produces a less efficient program.

3. Experience with DEE also shows that the efficient retrieval of complex objects is important for managing data structures stored in CAD databases and genome databases. We designed and implemented a deductive object-oriented database and applied it to genome analyses. Flexible management of the complex objects, as well as flexible analyses can be achieved using the database.

Future Projects

We planned to develop an improved version of the deductive object-oriented database. Possible extensions are as follows:

1. Management of users' biological knowledge in the database: The current database does not have the facility to store users' knowledge. It is important for efficient and flexible analyses to use biological knowledge that is not stored in the database.

2. Developing an efficient query evaluation technique for the database: The current version of the database does not support any query processing techniques developed for efficient evaluation of the deductive databases. It is important for efficient query processing in the deductive object-oriented database to determine whether conventional DDB techniques can also be applied to DOOD.

Bibliography

[ABD+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In W Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 223–240. North-Holland, 1990.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, California, 1988.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[Bai92] A. Bairoch. PROSITE: a dictionary of sites and patterns in proteins. *Nucleic Acids Research*, 20:2013–2018, 1992.

[BB92] A. Bairoch and B. Boeckmann. The SWISS-PROT protein sequence data bank. *Nucleic Acids Research*, 20:2019–2022, 1992.

[BCF+92] C. Burks, M. J. Cinkosky, W. M. Fischer, P. Gilna, J. E.-D. Hayden, G. M. Keen, M. Kelly, D. Kristofferson, and J. Lawrence. GenBank. *Nucleic Acids Research*, 20:2065–2069, 1992.

[BGMT92] W. C. Barker, D. G. George, H. W. Mewes, and A. Tsugita. The PIR-international protein sequence database. *Nucleic Acids Research*, 20:2023–2026, 1992.

[BKW+77] F. C. Bernstein, T. F. Koetzle, G. J. Williams, E. F. Meyer, M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and

M. Tasumi. The protein data bank: A computer-based archival file for macromolecular structures. *Journal of Molecular Biology*, 112:535–542, 1977.

[BMPR87] I. Balbin, K. Meenakshi, G. S. Port, and K. Ramamohanarao. Efficient bottom-up computation for stratified databases. Technical report, Department of Computer Science, University of Melbourne, Melbourne, 1987.

[BNR+87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language LDL1. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, 1987.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM SIGMOD Conference on the Management of Data*, pages 16–52, Washington DC, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, 1987.

[BR88] F. Bancilhon and R. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 519–543. Morgan Kaufmann, Los Altos, California, 1988.

[BT86] P. Bucher and E. N. Trifonov. Compilation and analysis of eukaryotic POL II promoter sequences. *Nucleic Acid Research*, 14:10009–10026, 1986.

[CCS92] D. Casey, C. Cantor, and S. Spengler. Primer on molecular genetics. Technical report, U.S. Department of Energy, Washington, D.C., Apr. 1992.

[CKW89] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, 1989.

[Cod79] E. F. Codd. Extending the relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4), Dec. 1979.

[Eri92] D. Erickson. Hacking the genome. *Scientific American*, Apr. 1992.

[GM70] A. J. Gibbs and G. A. McIntyre. The diagram: a method for comparing sequences. Its use with amino acid and nucleotide sequences. *European Journal of Biochemistry*, 16:1–11, 1970.

[GST93] S. Goto, N. Sakamoto, and T. Takagi. Object-oriented database with rule based query interface for genomic computation. In *Proceedings of the Third International Conference on Database Systems for Advanced Applications*, pages 65–72, Apr. 1993.

[HFSC92] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[Kam92] N. N. Kamel. A profile for molecular biology databases and information resources. *Computer Applications in the Biosciences*, 8(4):311–321, 1992.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases*. The MIT Press, Cambridge, 1990.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 134–146, Portland, June 1989.

[KNKT91] M. Kanehisa, K. Nitta, A. Konagaya, and H Tanaka. Knowledge processing technologies and human genome project. *Journal of Japanese Society for Artificial Intelligence*, 6(5):630–640, Sept. 1991. (in Japanese).

[KP88] J. M. Kerisit and J. M. Pugin. Efficient query answering on stratified databases. In *Proceeding of the International Conference on Fifth Generation Computer Systems*, pages 719–726, Tokyo, 1988.

[KRS90] D. B. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformation that maintain context information. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 380–391, Brisbane, 1990.

[KW89] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-Logic revisited). In *Proceedings of the 8th ACM Symposium on Principles of Databases Systems*, pages 379–393, Philadelphia, Mar. 1989.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 2nd and extended edition, 1987.

[LO91] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and methods inheritance. In J. Clifford and R. King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 198–207, Denver, May 1991.

[Miy90] S Miyazawa. DNA DataBank of Japan: Present status and future plans. In G. I. Bell and T. Marr, editors, *Computers and DNA*, pages 47–61. Addison-Wesley, Massachusetts, 1990.

[MKR84] A. McClelland, L. C. Kuehn, and F. H. Ruddle. The human transferrin receptor gene: Genomic organization, and the complete primary structure of the receptor deduced from a cDNA sequence. *Cell*, 39:267–274, 1984.

[Mor88] K. A. Morris. An algorithm for ordering subgoals in NAIL! In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 82–88, Austin, 1988.

[MT86a] S. L. McKnight and R. Tijian. Transcriptional selectivity of viral genes in mammalian cells. *Cell*, 46:795–805, 1986.

[MT86b] A. J. Morffew and S. J. P. Todd. The use of PROLOG as a protein query language. *Computer Chemistry*, 10:9–14, 1986.

[MUvG86] K. Morris, J. D. Ullman, and A. van Gelder. Design overview of the NAIL! system. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 554–568, London, 1986.

[Nat92] National Center for Biotechnology Information. *NCBI Software Development Kit Programmer's Reference*, 1992.

[Nat93] National Center for Biotechnology Information. *Entrez User's Guide*, 1993.

[Nau87] J. F. Naughton. One sided recursions. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 340–348, San Diego, 1987.

[Nau88] J. F. Naughton. Compiling separable recursions. In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data*, pages 312–319, Chicago, 1988.

[NRSU89a] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proceedings of the 15th International Conference of Very Large Data Bases*, pages 173–182, Amsterdam, 1989.

[NRSU89b] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the 1989 ACM SIGMOD Conference on the Management of Data*, pages 235–242, Portland, 1989.

[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the Natural Academy of Sciences U.S.A.*, 85:2444–2448, 1988.

[PMFR92] P. L. Pearson, N. W. Matheson, D. C. Flescher, and R. J. Robbins. The GDB™ human genome data base Anno 1992. *Nucleic Acids Research*, 20:2201–2206, 1992.

[PP90] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 321–368. North Holland, 1990.

[Rei78] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[RTN+86] C. J. Rawlings, W. R. Taylor, J. Nyakairu, J. Fox, and M. J. E. Sternberg. Using Prolog to represent and reason about protein structure. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 235–242, Berlin, 1986. Springer-Verlag.

[SOBW84] C. Schneider, M. J. Owen, D. Banville, and J. G. Williams. Primary structure of human transferrin receptor deduced from the mRNA sequence. *Nature*, 311:675–678, 1984.

[STS93] N. Sakamoto, T. Takagi, and Y. Sakaki. Development of overlapping oligonucleotide database and application to signal sequence search of the human genome. *Computer Applications in the Biosciences*, 9(4):427–434, 1993.

[STU89] T. Suzuki, T. Takagi, and K. Ushijima. Efficient bottom-up evaluation of negative closed queries on stratified databases. In *Proceedings of Advanced Database System Symposium '89*, pages 143–150, Kyoto, Dec. 1989.

[STU91] T. Suzuki, T. Takagi, and K. Ushijima. Efficient computation of negative queries and closed queries by using Vertically Indexed Magic Set method on stratified databases. *Transactions of Information Processing Society of Japan*, 32(2):206–219, 1991. (in Japanese).

[SZ86] D. Sacca and C. Zaniolo. The generalized counting method for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, pages 31–53, Rome, 1986.

[Tak92] T. Takagi. Genome databases. *Journal of Information Processing Society of Japan*, 33(10):1126–1133, Oct. 1992. (in Japanese).

[TNF91] M. Tsukamoto, S. Nishio, and M. Fujio. DOT: A term represtaion using DOT algebra for knowledge-bases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases*, pages 391–410. Springer-Verlag, 1991.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, 1989.

[Vie87] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proceedings of the 4th International Conference on Logic Programming*, pages 74–103, Melbourne, 1987.

[Vie88] L. Vieille. Recursive Query Processing: Fundamental Algorithms and the Dedgin system. In P. Gray and R. Lucas, editors, *Prolog and Databases*, pages 135–158. Ellis Horwood, Chicester, England, 1988.

[YHH88] C. Youn, L. J. Henschen, and J. Han. Classification of recursive formulas in deductive databases. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 320–328, Illinois, 1988.

[YN89] K. Yokota and S. Nishio. Towards integration of deductive databases and object-oriented databases: A limited survey. In *Proceedings of Advanced Database System Symposium '89*, pages 253–261, Kyoto, Dec. 1989.

[Yok92] K. Yokota. Deductive object-oriented databases. *Computer Software*, 9(4):3–18, 1992. (in Japanese).

[ZJT91] M. Zukar, J. A. Jaeger, and D. H. Truner. A comparison of optimal and suboptimal RNA secondary structures predicted by free energy minimization with structures determined by phylogenetic comparison. *Nucleic Acid Research*, 19:2707–2714, 1991.