UCSF
UC San Francisco Electronic Theses and Dissertations

Title
Sequence modelling and an extensible data model for genomic databases

Permalink
https://escholarship.org/uc/item/8s85k8gg

Author
Li, Peter Wei-Der

Publication Date
1991

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Copyright 1991

by

Peter Wei-Der Li

Acknowledgments

I would like to thank my advisor, Dr. Arie Shoshani, whose wisdom and guidance have made this work possible. The many hours of discussion with him were the most rewarding part of this undertaking. I would also like to thank my thesis committee members, Dr. Arie Segev, Dr. John Starkweather, and Dr. Robert Langridge, for supporting my efforts in this work.

I would like to thank the late Dr. Marsden Blois, who gave me the opportunity to pursue this goal back in 1984. I would also like to thank Dr. John Stewart and the late Dr. Jack Sadler and for advising me and guiding my interests while I was a student in their laboratories at the University of Colorado. It was at that time when the seeds of this work was formed.

I would like to especially thank my wife, Jin Jen, for her help on the molecular biology discussed in this work. In addition, her total support and love helped to make this undertaking a pleasant and rewarding experience. And of course, many thanks to my son, Jason, for providing the often needed and refreshing distractions while I was working on this thesis.

Sequence Modelling and
an Extensible Object Data Model
for Genomic Databases

by

Peter Wei-Der Li

ABSTRACT

The Human Genome Project (HGP) plans to sequence the human genome by the beginning of the next century. It will generate DNA sequences of more than 10 billion bases and complex marker sequences (maps) of more than 100 million markers. All of these information will be stored in database management systems (DBMS's). However, existing data models do not have the abstraction mechanism for modelling sequences and existing DBMS's do not have operations for complex sequences. This work addresses the problem of sequence modelling in the context of the HGP and the more general problem of an extensible object data model that can incorporate the sequence model as well as existing and future data constructs and operators.

First, we proposed a general sequence model that is application and implementation independent. This model is used to capture the sequence information found in the HGP at the conceptual level. In addition, abstract and biological sequence operators are defined for manipulating the modelled sequences. Second, we combined many features of semantic and object oriented data models into an extensible framework, which we called the "Extensible Object Model", to address the need of a modelling framework for incorporating the sequence data model with other types of data constructs and operators. This framework is based on the conceptual separation between constructors and constraints. We then used this modelling framework to integrate the constructs for the conceptual sequence model. The Extensible Object Model is also defined with a graphical representation, which is useful as a tool for database designers. Finally, we defined a query language to support this model and implement the query processor to demonstrate the feasibility of the extensible framework and the usefulness of the conceptual sequence model.

Table of Contents

Title i
Copyright ii
Acknowledgments iii
Abstract iv
Table of Contents v
Overview 1
PART I. SEQUENCE ABSTRACTION 3
CHAPTER 1. BACKGROUND 3
1.1. Human Genome Project 3
1.2. Nucleotides 4
1.3. Single Stranded DNA 5
1.4. Double Stranded DNA 6
1.5. DNA Fragments 7
1.6. Biological Organization 11
CHAPTER 2. ABSTRACT SEQUENCES 13
2.1. Current Sequence Models 13
2.2. Fundamental Sequences 17
2.3. Initial Sequence Model Framework 20
2.4. Resolving Complications 22
2.5. Improved Sequence Model Framework 27
2.6. Operations on Abstract Sequences 31
CHAPTER 3. MODELS OF BIOLOGICAL SEQUENCES 38
3.1. Nucleotides 38
3.2. Single Stranded Sequences 39
3.3. Double Stranded Sequences 41
3.4. Fragment Markers 47
3.5. Fragment As Sequence 49
CHAPTER 4. SEQUENCE SUMMMARY 54
4.1. Conceptual Basis 54
4.2. HGP Sequence Models 55
4.3. Model Expressiveness 56

4.4. Open Topics 57
PART II. EXTENSIBLE OBJECT DATA MODEL 58
CHAPTER 5. BACKGROUND 58
5.1. Framework for Data Models 60
5.2. Common Characteristics 63
5.3. Unique Characteristics 67
5.4. Model Characteristics Summary 72
CHAPTER 6. EXTENSIBLE OBJECT DATA MODEL 73
6.1. Definition of Object-Types 73
6.2. Definition of Instances 74
6.3. Graphical Notation 76
6.4. Object-Types in a Schema 76
6.5. Other Characteristics 78
CHAPTER 7. CONSTRUCTORS 79
7.1. Composition 80
7.2. Set 84
7.3. Sequence 88
7.4. IS-A Construction 93
7.5. Inheritance 94
7.6. Subset 99
7.7. Union 100
7.8. General Characteristics 103
CHAPTER 8. CONTEXT DEPENDENCY 107
8.1. Instance Association Diagrams 109
8.2. Target Dependency 112
8.3. Source Dependency 117
8.4. Peer Dependency 119
8.5. Issues of Context Dependency Usage 124
CHAPTER 9. OBJECT MODEL SUMMARY 126
9.1. Conceptual Basis 126
9.2. Goals of the Extensible Object Model 129

9.3. Comparison with Other Data Models 130
9.4. Differences From Other Models 135
9.5. Open Topics 140
PART III. EXTENSIBLE OBJECT QUERY LANGUAGE 144
CHAPTER 10. DESIGN OF A QUERY PROCESSOR 144
10.1. Data Definition Criteria 144
10.2. Data Manipulation Criteria 146
10.3. Implementation Criteria 148
10.4. Design Limitations 149
10.5. Backend Limitations 150
CHAPTER 11. QUERY LANGUAGE REFERENCE 152
11.1. Notations 152
11.2. Expressions 153
11.3. Administration 155
11.4. Type Definition 157
11.5. Instance Manipulation 160
11.6. Selection and Control Flow 163
11.7. Operators 165
Conclusions 167
Bibliography 171
Glossary 176
Appendix A. Context Interactions 180
Appendix B. Ternary Relationships 188
Appendix C. Examples of EOM Schema 193
Appendix D. Query Language Syntax 205
Appendix E. Query Processor Modules 211

Overview

In the Human Genome Project (HGP), several types of sequences are encountered. These range from simple sequences of bases to complex sequences of markers. However, current database technology does not provide a conceptual model for sequences. Therefore, when creating a conceptual schema for a genomic database, the simple sequence information is often forced into an implementation-specific form, e.g. character string or binary image, and the complex sequence information is re-modelled into available constructs, such as sets or arrays. These limitations remove the basic notion of a sequence from the conceptual schema of genomic databases and force the users either to give up sequence operations or to develop additional code to recapture the inherent sequence information.

Part I of this work develops a conceptual model for sequences. This model is application and implementation independent so that a user can focus on modelling domain information and not to be bound by implementation restrictions. This is achieved by separating the sequence information into two components: position and content. This is extended into a framework of several sequence models based on the following characteristics of position: order, metric, granularity, atomicity, and density. Although the focus of this conceptual sequence model is on the types of sequences encountered in the HGP, it was developed with sufficient generality for other applications.

Unfortunately, a sequence model, by itself, is insufficient to handle other modelling requirements associated with a complex enterprise such as the HGP. Therefore, a general purpose conceptual data model is also required. Furthermore, this general model must be able to subsume the concepts from the sequence model in a consistent and coherent fashion. Current conceptual data models are closed in their characterization, i.e. it is very difficult to extend these models with our sequence concepts in the prescribed fashion. In addition, each model has its own drawbacks in semantic richness, uniformity, and implementation independence.

Part II of this work develops an extensible conceptual data model, which we call the "Extensible Object Model", that addresses the above concerns. The result is an organized framework that integrates concepts from semantic and object oriented data models. The reason for a framework, instead of only one model, is because we cannot anticipate all the specific constructs and relationships found in different application domains. Therefore, this approach permits the organized extension for future modelling requirements. The basic components of the framework are constructors and contexts. The constructors define the structure of the information modelled: composition, set, sequence, inheritance, and union. The contexts define the constraints of values placed on other values: target, source, and peer dependency. Under this framework, it was straightforward to integrate the sequence model from Part I.

Since the Extensible Object Model is a framework, not all features are needed in order to demonstrate its capability. In particular, we implemented a query language interface for three constructors in the model: composition, set, and sequence, with some of the contexts. This combination is sufficiently powerful to subsume the relational model, the basic Entity-Relationship model, most Non-First Normal Form models, and most temporal sequence models. It is also sufficient for most applications in the domain of the HGP. In Part III, we describe this query language and its implementation design. The main design goals are its independence of the underlying system and the robustness of the framework for future extensions. For this demonstration, a relational database management system with large binary object support serves as the underlying system.

In Appendix A, we describe the interactions among the context dependencies defined in Chapter 8. In Appendix B, we present the details of a generalized constraint introduced in Part II (Section 8.4.1). In Appendix C, we provide several examples of the Extensible Object Model schema. In Appendix D, we list the full language syntax of the query language introduced in Part III. Finally, in Appendix E, we describe the major modules of the query processor.

PART I. SEQUENCE ABSTRACTION

CHAPTER 1. BACKGROUND

The main purpose of Part I is to formulate a well-defined conceptual sequence model and define a set of necessary operations. One of the weaknesses of current database technology is the lack of support for modelling and manipulating sequences [25]. As a result, the available genomic sequence databases [21] implement simple sequences as text strings, binary images, or references to an external file store. This use of implementation specific models reduces the portability and the evolvability of any given database schema. In these databases, complex sequences, such as maps (described in Section 1.5), are coerced into sets or arrays. Consequently, sequence information, such as order and distance, are lost because these characteristics are not present in non-sequence constructs. The development of a conceptual sequence model would address these problems by capturing domain sequence information at the conceptual level, i.e. independent of physical implementation. It also provides a set of operations which takes in account of the sequence specific characteristics. This permits operations to be defined independent of implementation strategies.

Part I is divided into four chapters. Chapter 1 describes the scope of the Human Genome Project (HGP) and a brief background of the biology associated with the project. Chapter 2 reviews the current sequence models and their limitations and proposes an abstract sequence model in terms of sequence structures and operations. Chapter 3 uses the abstract model and operations to capture the real-world HGP sequence information. Finally, Chapter 4 presents a summary of sequence modelling.

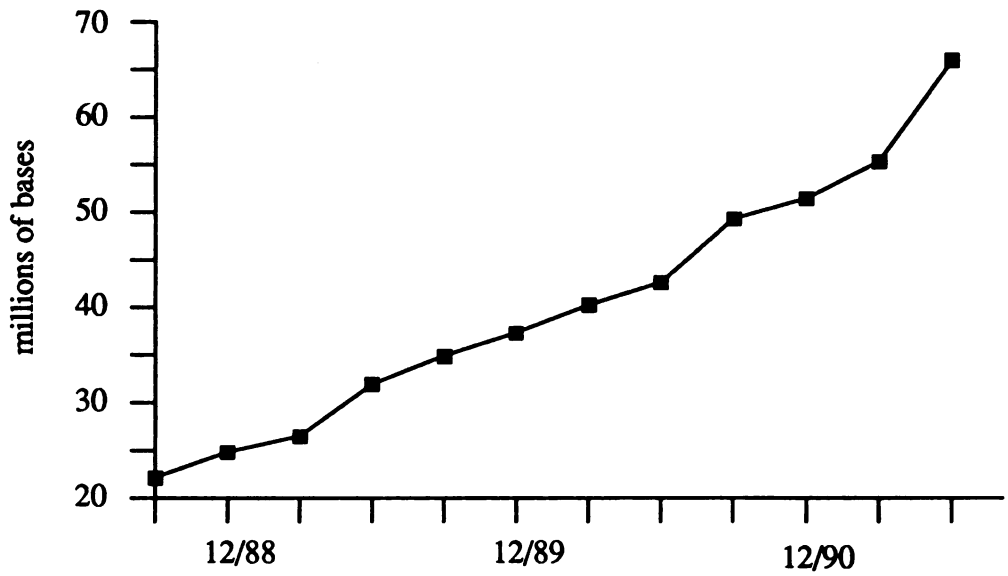
1.1. Human Genome Project

The Human Genome Projects are based on multinational research teams with the objective to sequence the human genome by 21st century [47,67]. In addition to the human genome, genomes from *Escherichia coli* (bacteria), *Mycoplasma Capricolum* (mycoplasma), *Saccharomyces cerevisiae* (yeast), *Caenorhabditis elegans* (nematode), *Drosophila melanogaster* (fruit fly), *Mus musculus* (mouse), and *Arabidopsis thaliana* (a mustard plant) will also be sequenced. The concerted effort will be performed in several major genome centers and laboratories around the world. In the US, the HGP's are collectively termed "Human Genome Initiative" (HGI) for Congressional funding reasons and major funding will be provided by the Department of Energy, the National Institute of Health, the National Science Foundation, and the Howard Hughes Medical Institute.

The Initiative is broken down into three five-year phases. The first phase will emphasize development of biochemical mapping technology and exploratory efforts in large scale mapping. The second phase focuses on the completion of maps, sequencing technology, and exploratory efforts in large scale sequencing. The third phase completes the genomic sequencing. This approach is essen-

tially a top-down successive refinement methodology. Currently, we are in the middle of the first phase [68]. Analysis and application of information gathered will be conducted throughout all the phases.

There are several sequence databases that act as central clearing houses for discovered deoxyribonucleic acid (DNA) sequences. In addition to DNA and ribonucleic acid (RNA) sequences, there are also protein sequences and structure databases which are commonly used by the scientists in the HGP. Although this work focuses on genomic sequence information, the results also have application in these other sequence-related databases. In the US, Genbank is the major database for genomic sequences [14]. The following graph shows the amount of raw sequence information stored in Genbank over the last three years:



The current growth rate is approximately 15 million bases per year and it does not include the annotations that contain important biological information. As the HGP reaches the second phase, the expected rate can be as high as 500 million bases per year. It is this data generation that motivates this work in developing a sequence-based data model.

In the next sections, we describe briefly the biological structures that we wish to support. This is necessary in order to develop the functionality required of a HGP sequence data model. For more detailed information, see [4,70].

1.2. Nucleotides

The molecular biology of the HGP starts nucleotides, which form the basis for DNA. Four types of nucleotides can be found in DNA, namely: adenine (A), cytosine (C), guanine (G), and thymine (T). Each nucleotide is oriented by molecular structures labelled as 5′ and 3′ ends. In the context of this work, nucleotides and bases are synonymous, although, a nucleotide is a base with the deox-

yribose and phosphate backbone. In some situations, there is incomplete information about a sequence because of the experimental nature of science. In these cases, a base value is not fully specified: a pyrimidine (Py) can represent either C or T, a purine (Pu) for A or G, and nucleotide (N) for any nucleotides.

1.3. Single Stranded DNA

Single-stranded DNA (ssDNA) is the next level of organizational complexity in DNA, which is formed by chemical bonding the 5′ end of a base to the 3′ end of the previous base. Although, the bulk of the information used in the HGP will be double-stranded sequences (dsDNA), ssDNA does exist in nature and can serve as a fundamental building block for modelling dsDNA. Because of the orientation of the bonding, a ssDNA has a direction based on the molecular 5′ and 3′ ends, which correspond to the beginning and the end of the molecule, respectively. A ssDNA can be cut into two sequences while keeping the same orientation. For example:

```
5'- ACGTTAGTC -3'
```

cut ↓ ↑ concatenate

```
5'- ACGT -3',5'- TAGTC -3'
```

Two sequences can be joined, i.e., concatenated, with the 5′-end of one sequence linked to the 3′-end of the other. In the example above, this will result in either the original molecule or the following sequence:

`5'- TAGTCACGT -3'`, by placing the second sequence first.

Nucleotides can form complementary pairs by hydrogen bonding: A with T and C with G, but in the opposite orientation. A dsDNA is formed by building the complementary strand from a given ssDNA and "anneal" them by hydrogen bonding:

```
Seq A:   5'- ACGTTAGTC -3'   (given strand)
   ↓
Seq B:   5'- ACGTTAGTC -3'   (given strand)
         3'- TGCAATCAG -5'   (complimentary strand)
```

▮▮ - represent hydrogen bonds, by default, this will not be shown

Therefore, a complement strand is not only a base-by-base complement, but also in reverse order relative to the given strand. Each paired nucleotides is called a base-pair.

Given two ssDNA, we can check for equality, overlap, and complementation, for example:

```
Seq A:    5'- ACGTTAGTC -3'     equality
Seq B:    5'- ACGTTAGTC -3'
-------------------------------------------
Seq A:    5'- ACGTTAGTC     -3'  overlap
Seq B:    5'-     TAGTCATG -3'
-------------------------------------------
Seq A:    5'- ACGTTAGTC -3'     complementation
Seq B:    3'- TGCAATCAG -5'
```

There are two types of complementation. First is full complementation, as seen in the above example. Another type is partial, where single stranded ends protrude from the area of complementation:

```
Seq A:    5'- ACGTTAGTC--- -3'  partial complementation
Seq B:    3'- ----ATCAGATG -5'
```

We use "–" to indicate the missing complementary base.

1.4. Double Stranded DNA

Double-stranded DNA (dsDNA) is the basic structure of nuclear DNA, i.e. inside the nucleus of a cell. It is organizationally more complex than ssDNA and is the basis for all the information in the HGP. While ssDNA has specific orientation (5′ and 3′), dsDNA is "symmetric" because of the complementary orientation, for example:

```
dsDNA A:    5'- ACGTTAGTC--- -3'
            3'- ----ATCAGCTG -5'
-------------------------------------------
dsDNA A':   5'- GTCGACTA---- -3'
            3'- ---CTGATTGCA -5'
```

Therefore, in a solution where molecules freely rotate, dsDNA A and A' are copies of the same molecule. The typical representation of a dsDNA is a sequence with the top strand in 5′-to-3′ orientation and is assumed to be rotationally free or symmetric. Under certain situations, a dsDNA is bound to a specific context (part of another dsDNA) or physical orientation (another molecule), then explicit 5′ and 3′ ends must be denoted.

When looking for a subsequence on a ssDNA, only one direction is read, namely 5′-to-3′. On the other hand, looking for a subsequence on a dsDNA requires both directions. A dsDNA can be cut and concatenated, but also denatured (split into component single strands), annealed (forming dsDNA from ssDNA), and filled (complements added to single stranded ends).

These are shown in the following:

```
DS Seq A:  ACTAATGCCATGC   cut   ACTAATGC  and  --CATGC
           TGATTACGGTACG   ---->  TGATTA--       CGGTACG
```

```
DS Seq A:  ACTAATGC
  and      TGATTA--   concat   ACTAATGCCATGC
DS Seq B:  --CATGC    ------>   TGATTACGGTACG
           CGGTACG
```

```
DS Seq A:  ACTAATGCCA---   denature   SS Seq X:  ACTAATGCCA
           --ATTACGGTACG   ------>    and
                                      SS Seq Y:  GCATGGCATTA
```

SS Seq Y is in the correct orientation for ssDNA, which is the reverse of the DS direction with respect to SS Seq X.

```
SS Seq X:  TAATGCCA   anneal   DS Seq A:  --TAATGCCA---
  and                ------>              TGATTACGGTACG
SS Seq Y:  GCATGGCATTAGT
```

The rotational symmetry of dsDNA creates certain problems in modelling biological operations. For example, several results are possible from a single concatenate:

```
DS Seq A:  ATAGGTCT--   and  DS Seq B:  CGATACGT--
           --TCCAGAGC                   --TATGCATA
```

can form both:

```
(I)  ATAGGTCTCGATACGT--   and  (II)  CGATACGTATAGGTCT--
     --TCCAGAGCTATGCATA              --TATGCATATCCAGAGC
```

Depending on whether the single stranded ends are complementary, many results are possible. In the extreme, up to the number of bases in the single stranded ends are possible:

```
DS Seq A:  GTACAAAAAAA   and  DS Seq B:  -------GTAC
           CATG-------                   TTTTTTTCATG
```

can form the following:

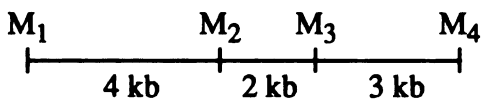
```
GTACAAAAAAAGTAC , GTACAAAAAAA-GTAC , GTACAAAAAAA--GTAC ,
CATGTTTTTTTCATG   CATG-TTTTTTTCATG   CATG--TTTTTTTCATG
```

and so on, down to a minimum overlap determined by biophysical stability.

1.5. DNA Fragments

In the first phase of the HGP, fragments of genomic DNA are created. These fragments of dsDNA have known lengths, but unknown base sequence. These DNA fragments are labelled with "markers" (described in the next two sections) and become "maps", which form a crucial portion of orga-

nized biological information for the HGP. The name "map" derives from its use as providing reference landmarks for navigation. In biology, DNA maps provide biological landmarks for exploration. Unlike the 2-D cartographic maps, DNA maps are in one dimension and only contain information on marker names and lengths of the intervening sequences. For example, in the following representation, the distance between marker M_1 and marker M_2 is 4 kilobases (kb):



The process of placing markers on fragments to generate "maps" of human DNA is one of the fundamental tasks in the HGP. There are two directions in map constructions: a) constructing higher resolution maps from low resolution maps (over the same coverage) and b) constructing larger maps from smaller maps (increasing the coverage).

In DNA maps, length values of DNA fragments are not exact until the whole fragment has been sequenced. The reason is that some amount of uncertainty is always associated with the measurement methodology. The primary means of fragment length measurement is based on gel electrophoresis, and to improve the resolution, one can change the composition of the gel. However, the resolution is, at best, two-orders of magnitude below the length of the fragment on a given gel. For example, if a gel is made to resolve a 100 kb (kilobase) fragment, its resolution is usually 1 kb or larger. Consequently, all lengths measured from this gel have an uncertainty of > 1 kb.

1.5.1. Restriction Sites as Markers

Two types of DNA map markers are commonly used: restriction sites and probes. Restriction sites are created by enzymes (endonucleases) that cut DNA at specific sequences. The length of the recognition sequence varies from 4 to 15 base-pairs, depending on the enzyme. The newly formed ends of a cutting site can be blunt or sticky (SS tails) and the recognition sequence can also include "wildcard" bases. For example:

Restriction Enzyme "*Eco* RI" recognizes: GAATTC
CTTAAG

The line denotes the cutting site and the separation of strands.

Therefore, the following dsDNA will be cut by *Eco* RI into two pieces with sticky ends:

```
ATAGAATTCATG   --Eco RI-->   ATAG----   and   AATTCATG
TATCTTAAGTAC      cut        TATCTTAA         ----GTAC
```

Other enzyme recognition sites have other characteristics, such as:

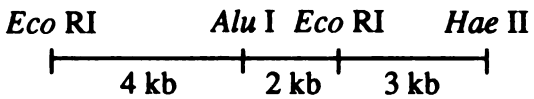
Alu I	AG|CT TC|GA	creates blunt ends
Bgl I	GCCNNNN|NGGC CGGN|NNNNCCG	N matches any base value
Hae II	PuGCGC|Py Py|CGCGPu	Pu stands for Purines (A or C) Py stands for Pyrimidines (T or G)

Some restriction enzymes have more than one specific recognition sequence. For example, *Hae* II has the following four possible recognition sequences:

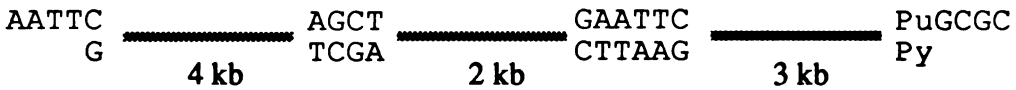
```
AGCGCT , AGCGCC , GGCGCT , or GGCGCC
TCGCGA   TCGCGG   CCGCGA     CCGCGG
```

In general, restriction enzyme recognition sequences, with wildcard bases, are palindromes, i.e. the sequence is the same reading forward or backward (on the complementary strand). Therefore, it is common to represent these sequences using only a ssDNA and a separation bar ("|").

Due to the statistical nature of short sequences, a fragment sufficiently long will be invariably cut by some enzyme. Once cut, the lengths of the smaller fragments can be measured and marked, creating a sequence of restriction markers for a fragment:



Although single values for lengths are used, it is understood that uncertainty exists for almost every value except for those DNAs whose composition has been completely sequenced. The diagram below represents the current state of knowledge of the previous fragment:

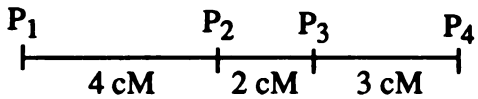


where ▬▬▬ represents unknown sequence.

1.5.2. Probes as Markers

Probes are short DNA fragments that have been positively identified, but not necessarily sequenced. Similar to restriction enzymes which recognize specific sequences in DNA, probes recognize specific sequences via hybridization, a process where tagged single stranded copies of the probe is hydrogen bonded to another DNA by complementation. Therefore, we can use the probe as a marker and then find the locations where the probe hybridizes to in a large DNA fragment.

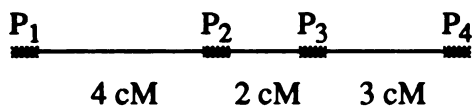
A fragment with mapped probes is shown as follows:



P_1 to P_4 are probes, cM is centiMorgan

A distance of "1 cM" represents a genetic distance of 1% recombination during meiotic segregation between two markers. It is approximately 1 Mb in length.

If the probes are 1 kb in length, the above map becomes:

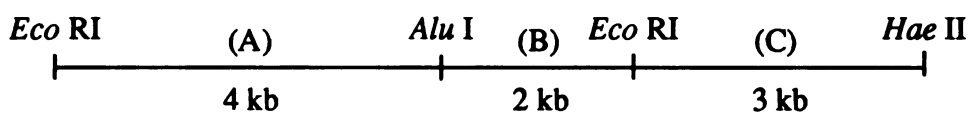


where ▬ represents a stretch of DNA 1kb in length

While the latter diagram is more correct, it is not necessarily useful to the biologist, because 1 kb fragments at cM resolution can be viewed as a point. This is similar to the representation of restriction sites as points.

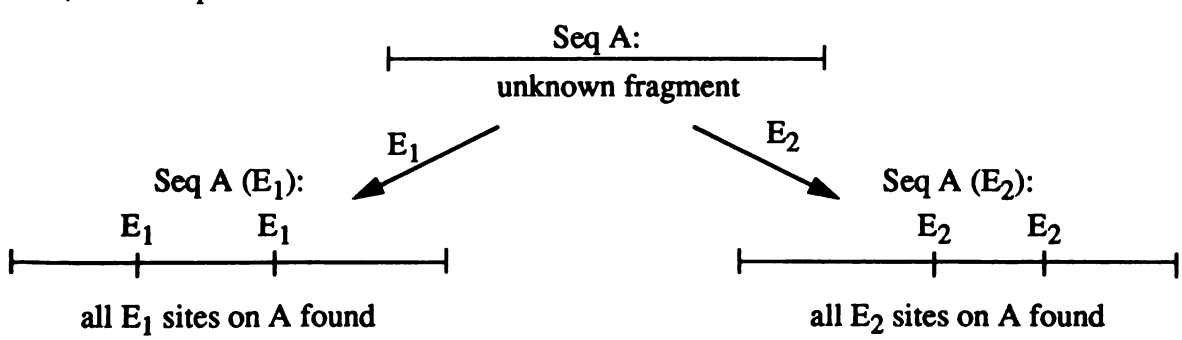
1.5.3. Marker Information

Restriction sites and probe locations can be related experimentally. Typically, if both types of markers are used on the same fragment, the restriction site marks the ends of fragment intervals and the probe becomes an experimental "handle" for the interval itself. For example,

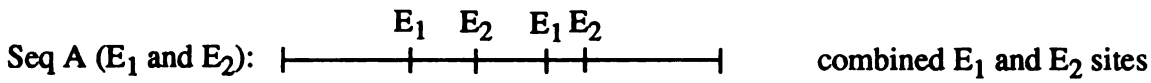


If fragment interval (B) hybridizes to probe P, for instance, we can use P to detect the presence of (B) in a pool of DNA cut with *Eco* RI and *Alu*I. However, hybridization does not provide the exact location of P in (B). In fact, if P extends past either ends of (B), then (A) or (C) could also be "picked up" by P. On the other hand, P can be used to bridge maps of different resolution. If we are given the cM map of the previous fragment and if $P = P_2$, then we have additional information around the region where P_2 is marked. This becomes useful in restriction fragment length polymorphism (RFLP), a technique that links individual uniqueness with the inheritance of genes.

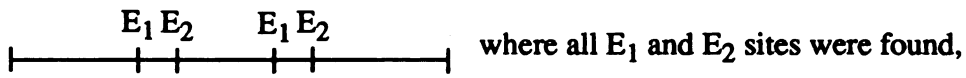
When a fragment has been cut by an enzyme, all sites that correspond to that enzyme are usually found, for example:



Superimposing the two maps, we get the following map:



If we have another map, Seq B:



then we can tell that Seq A and Seq B are different by the fact that the first E_2 site occur at different position (assuming the difference is greater than the uncertainty). However, we would not have made the distinction on the basis of the E_1 information alone.

Therefore, as each new set of markers is placed on a fragment, all new information is grouped with our current knowledge about that fragment. Negative facts about a fragment, such as a lack of cutting sites, are also useful for determining whether two fragments are equal when the available information is limited.

1.5.4. Ordered Maps

In addition to sequences of markers, DNA fragments could also exist in the form of ordered clone maps. These are collections of fragments where the ordering among them is known by determining whether overlaps occurred, but distances among them are unknown. For example, if we are given the fragments A, B, and C and we know A overlaps B, B overlaps C, but A does not overlap C, then an order of A-B-C can be inferred, but the actual distances and lengths are undetermined. In some situations, the overlap information is incomplete or insufficient to determine complete ordering, which results in a partially ordered clone maps.

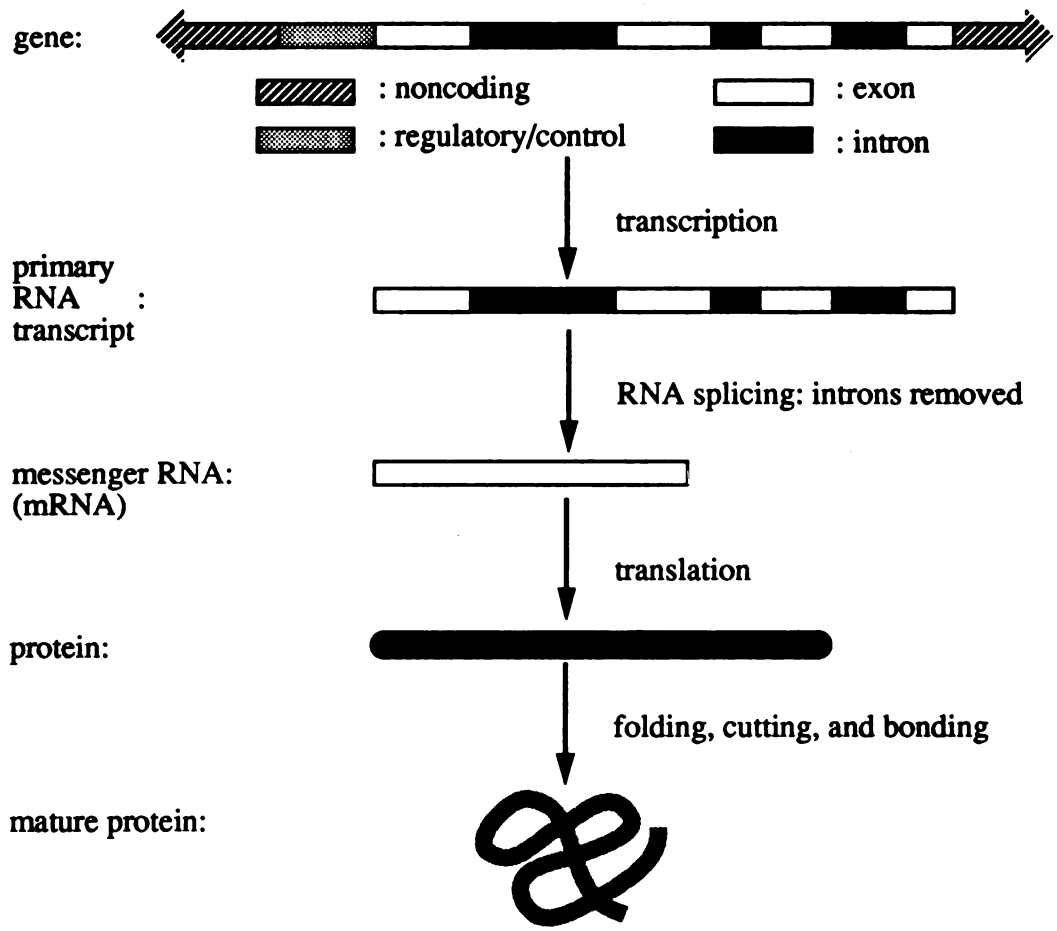
1.6. Biological Organization

The organization of biologically significant objects in human genome is usually given in a top-down fashion. Briefly, the human genome is composed of 22 pairs of autosomal chromosomes and one pair of sex chromosomes, XX for female and XY for male. Each chromosome is molecularly one dsDNA, between 50 to 250 million base-pairs long. Biologically, each chromosome holds between 1 to 5 thousand genes. Molecularly, a gene spans between one thousand to several million base-pairs. A gene usually contains regulatory sections and structural sections for transcription into ribonucleic acids (RNA). RNA differs from DNA in the type of sugar backbone (ribose vs. deoxyribose) and the substitution of the base uracil (U) for thymine (T). The transcript RNA may undergo splicing and translation if the final product is a protein, which is a sequence of amino acids. Although the splicing occurs at the RNA stage, the information which governs splicing is encoded in the DNA sequence.

The types of sequences that are of interest in the HGP is not limited to DNA sequences nor maps.

RNA and protein sequences will become important in the analytical phases of the HGP, because it is in these forms that biological effects are observed. However, we will not focus on RNA and protein sequence models because their characteristics are very similar to and can be subsumed by DNA sequence models.

The following diagram is a schematic view of the biological events:



CHAPTER 2. ABSTRACT SEQUENCES

Given the three types of sequences (ssDNA, dsDNA, and maps) used in the Human Genome Project, it is helpful to have a common model for all of them. This reduces the number of specific sequence model characteristics and simplifies the complexity of database implementation. In addition, if the abstract sequence model is sufficiently general, sequences from other application domains can also be modelled effectively. An important example is the modeling of temporal sequences used in business and scientific applications.

2.1. Current Sequence Models

For the remainder of the thesis, the definition of a sequence will be as follows: a sequence consists of a set of ordered pairs. One component of the ordered pair is named "position" and the other is named "content". Other terminology for position and content include referent and variant, domain and range, independent and dependent, or invariant and variant. We use the terms "position" and "content", because we do not consider causal relationships, on which the other terminologies are based. There are additional restrictions on the values of position and content, these will be described later.

2.1.1. Character String Model

One of the simplest sequence type is a character string. This is considered as a "primitive" type supported by most DBMS's, i.e. it is directly implemented with prespecified storage and operations. The position values for character strings are sequential whole numbers starting from "1" (or "0" for computer scientists) and the content value are encoded by a character, e.g. 7 bits in ASCII. For example:

```
"The quick brown fox..."    is represented as:

position:  1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,...
character: T, h, e,  , q, u, i, c, k,  , b, r, o, w,...
```

In most DBMS's, the length of a character string is limited to a predetermined maximum, e.g. 256. While some DBMS's now offer unlimited string length [30,63], which could be used to model DNA sequences, the storage and access of such strings are implemented external to the query processor. Consequently, the DBMS does not support operations on these strings. One of the major operations used in HGP is pattern or substring search, which is not currently supported by DBMS's offering unlimited size string type. Furthermore, a character string cannot directly model map information, as we shall show later.

2.1.2. List or Array Model

The next construct is a "list" or "array" of an arbitrary content type. These are found in so called

NonFirst Normal Form (NFNF) [1,52] or extended relational implementations [53] and object-oriented DBMS [45]. This abstraction still maintains integer position values, but does permit more complex values to be stored in place of characters. For example:

`struct content x[5]` is represented as:

position:	1	2	3	4	5
content:	x_1	x_2	x_3	x_4	x_5

where each x_i is a value of "struct content".

We can implement certain HGP sequences in this fashion, although they lack the appropriate position semantics, e.g. map sequences with fractional kilo-base (kb) position values cannot be modelled by sequential whole numbers. Therefore, we need to encapsulate the kb position and the marker label into the "list" content value and change the "list" position into "marker order":

Eco RI *Alu* I *Eco* RI *Hae* II

4 kb 2 kb 3 kb

is represented as:

marker order:	1	2	3	4	→ "list" position
kb position:	0 kb	4 kb	6 kb	9 kb	} "list" content
marker label:	*Eco* RI	*Alu* I	*Eco* RI	*Hae* II	

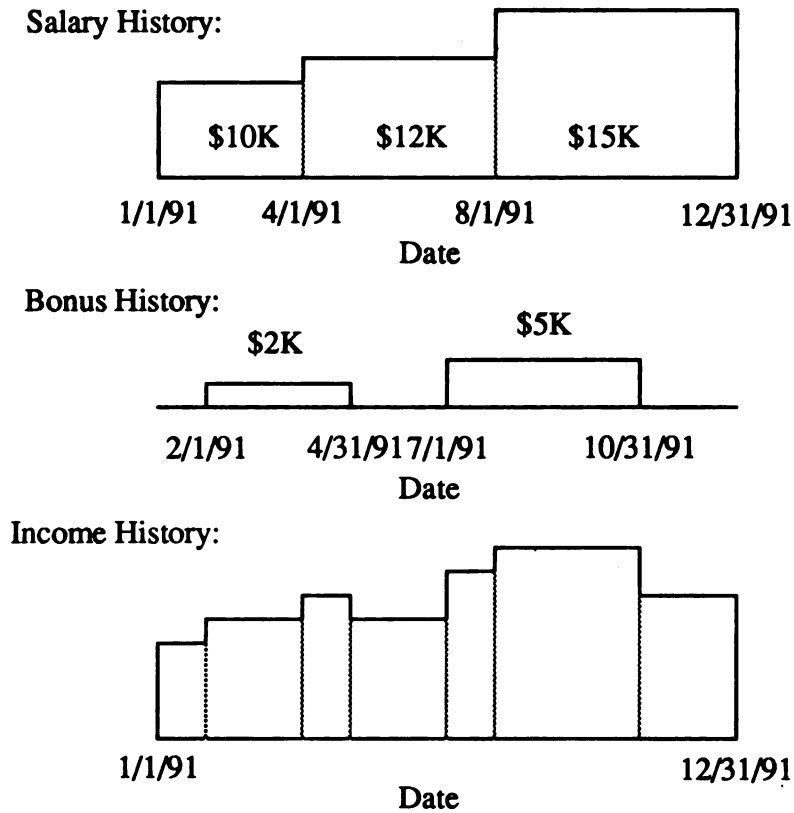
Unfortunately, such sequences cannot be correctly merged by looking at "list" positions alone. For example, overlap can only be done by analyzing the internal data (kb position) of the content type for a "list".

2.1.3. Temporal Models

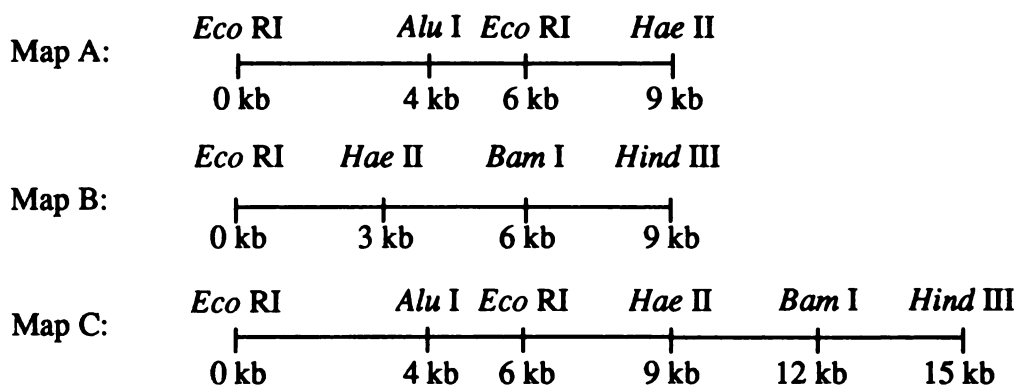
Genomic sequences have a close relative in the domain of temporal data [56]. The temporal model, described in [56], is based on time and value components, which correspond to the position and content components of a sequence model. However, each ordered pair has a specific interpretation based on one of several variants of the temporal model. For example, in the discrete-event model, each ordered pair is an independent event. In the stepwise or continuous model, each ordered pair is a transition to a new content value. From these models, a more abstract temporal model can be formed by removing all time interval interpretations. The resultant model would be a sequence model, devoid of temporal characteristics. However, because the core temporal models already have well-defined semantics and application domains, current research in temporal models are predominantly directed at implementation strategies for collections of temporal sequences [57].

Specifically, if temporal models are directly applied to genomic sequences, we encounter conceptual differences on how the two domains use sequence information. Temporal model operations over multiple sequences assume an absolute time reference frame, i.e. the time values for the

sequences are fixed while the content values are modified. For example, to compute the income history from a salary and a bonus history, the time values are matched, but the income value (new content) is the sum of salary and bonus values (old contents):



On the other hand, operations on genomic sequences have relative reference frames which change position values but not content values. For example, to combine overlapping maps A and B, the contents and relative positions are not changed, but the absolute positions in the resultant map C are new:



If we start with the temporal models as a basis for genomic sequence models, we will need to specify additional operations because of the conceptual differences on reference frames. However,

other aspects of genomic sequences will create additional problems. For example, position type in genomic sequences could have a partial order if only incomplete information is available, or have a circular order if information was derived from a circular DNA. Since temporal models presume a complete and linear order for position type, this extension will take the temporal models to beyond the original intent. Currently, there is no implementations, i.e. DBMS's, that support non-traditional algebraic systems beyond boolean algebra and standard number theories.

2.1.4. Combination Model

One approach to genomic sequence modelling is to use two or more current models. A character string model can be used for DNA sequences while a temporal model can be used for maps. However, this approach fails to unify the basic concepts of sequences. For example, the abstraction behind the operations "overlap" and "concatenate" is applicable to both models. Therefore, these operations should be the same independent of the implementation model used, but in most data models, these are considered distinct operations due to the different implementations of the data types. The objective of this Part is to unify and organize the basic concepts of sequences so that one set of operations can be applied to all sequence types.

2.2. Fundamental Sequences

A formal abstract sequence model should bridge the database domain and the application domain. It should include definitions for the basic components, operations, and semantic relationships. We start by defining a very simple and abstract model of a sequence that has few components, operations, and relationships. We then elaborate it in detail, adding characteristics to the starting model until a semantically rich sequence model is achieved. Eventually, a framework of sequences with varying complexity is created. Finally, this framework serves as the fundamental basis on which to model application-specific sequence information.

2.2.1. Components of a Sequence

As described earlier, a sequence is built from position and content values. The position and content values are collectively termed "types"; therefore, a sequence type is constructed from a position type and a content type. Every position value of a sequence is associated with a value from the content type, i.e. a word is a sequence of letters where each position is associated with a letter. Another example is a sequence of markers where every position value of this sequence has an associated marker (could be null if no markers were found). The content type minimally requires the definition of the "equality" operator. If the content type is structurally or operationally more complex, then its complexity can be made visible as new sequence structures or operations. For example, the complement operation of bases can be extended to a sequence of bases. But for now, we consider content type as a black-box structure that has some accessible operations, minimally "equality". The position-content pair can be considered as a unit, subsequently, a sequence is a set of these position-content elements. However, the nature of the "order" associated with position values makes a sequence different from a set. For the following discussion, we will use the notation below:

{...} denotes a set of "..."

[a, b] is an ordered pair of a and b.

The general model for sequences based on sets is defined as follows:

A sequence type, S, is constructed from the types:

a) position type, P, whose instances are ordered, and

b) content type, C.

An instance, I, of S, satisfies the following:

I = { [p , c] }, i.e. a sequence instance is a set of ordered pairs, where

$p \in P$, i.e. p is an instance of position type and

$c \in C$, i.e. c is an instance of content type.

If I is empty, we have a null sequence.

If $[p_1 , c_1]$ and $[p_2 , c_2] \in I$ and $p_1 = p_2$, then $c_1 = c_2$,
i.e. for each position only one content value is allowed.

This is equivalent of a well defined mathematical function, i.e. for every value x within the domain, there is one and only one value y in the range. We will use $< [p , c] >$ as a notation for I, instead of the set notation, $\{ [p , c] \}$, because of the additional requirement of unique and ordered p values.

2.2.2. Characteristics of Position

There are five qualitative characteristics for position types: order, metric, granularity, atomicity, and density, which are described as follows:

A. Order

The most basic sequence type is an "order sequence", whose position type is a completely ordered set of values, i.e. one can determine the order between any two position values. In addition, there is a minimal position value. This makes a sequence behave similarly to a "ray" in plane geometry or a "directional vector" in physics where we start at a "zero" and move in the "one" direction. When speaking of "sequences", this is the most commonly visualized type. By removing the minimal value, one creates "bidirectional" sequences, where position values can extend in both directions *ad infinitum*. While these do not have biological equivalents, they are useful for modelling relationships between local sequences. The next abstraction is reducing complete order into partial order and circular order. These sequences are more difficult to visualize because they are non-linear and less intuitive. Nevertheless, they play an important role as models of partial maps and circular plasmids in biology.

B. Metric

A metric assigns semantic significance to the differences in position values. Differences can exist between position values in a sequence, however, they remain abstract and are devoid of semantics until a metric is used to interpret the meaning. Once a metric is assigned, a difference in values is called a "distance". For example, the difference between position 1 and position 7 of a DNA sequence is 6 bases, only after a physical unit of measurement, e.g. base, is attached to its value. Therefore, a metric is defined with a unit, e.g. inch, Kelvin, or kilobases, and the measured value implies a complete order, i.e. 3 inches is greater than 2 inches. A sequence, whose position has metric characteristics, is called a "metric sequence". Since comparable physical units can be interconverted, e.g. inches to meters and pounds to kilograms, a sequence with one position metric unit can be converted into another sequence with some other comparable position metric. Although the interconversion of metric sequences are dependent on the specific application domain, the process of conversion can be generalized to become a part of the abstract sequence model.

C. Granularity

Granularity implies that multiple resolution levels can be used. For example, a geographic position can have multiple values based on the context resolution. The granularity is expressed as a fraction (or multiple) of the physical metric, e.g. 0.1, 10, 1/16, etc. We name a sequence whose position type can take on multiple granularities a "fractional sequence". The concept of a "fraction" properly embeds the concept of granularity: the "denominator" of a fraction is the granularity of the measure.

D. Atomicity

Atomicity is a consequence of metrics which are of counting type, e.g. nth letter in a word. Therefore, atomicity limits the finest level of resolution a metric can have. For example, in the context of letter positions in words and sentences, it is not meaningful to talk about half a letter or a quarter of a letter position. On the other hand, if we are concerned with physical positioning of letters in a word, such as kerning, then "half a letter" is meaningful. However, this type of "letter" is really a representation of a physical distance, not a representation of the "letter" concept in a word. To use atomicity correctly, we must know the exact nature of the position in the application context.

E. Density

Density distinguishes between actual and potential position values. A sequence is dense if no position value can exist between two consecutive position values of a given sequence, i.e., all potential values are actual. Examples are DNA sequences or continuous functions where the position values are over the whole numbers (or the reals). A sequence is sparse if there exists potential position value between two consecutive actual position values, i.e., not all potential values are filled. Examples are a sequence of physical measurements made in a temporal context or a sequence of markers in the HGP. In general, density is not completely independent of the semantic characteristics of granularity and atomicity, as a dense sequence at one granularity may become sparse in another, or vice versa.

2.3. Initial Sequence Model Framework

Three basic sequence models can be developed based on the characterization of sequences in the last section.

2.3.1. Order Sequence

The defining characteristics for order sequences are:

[< [p , c] > , D] where p ∈ Position, c ∈ Content, and D is density.

The position is of pure order only, therefore, characteristics of metric, granularity, and atomicity are not applicable. For example, we have an ordered classification set of "very short", "short", "average", "tall", and "very tall" people in a classroom. The position type is the ordered enumeration of height: {very short, short, average, tall, very tall}. The content type is the number of people in each class. Then every classroom will be a sequence of 5 numbers without metric, granularity, and atomicity properties

An example in the HGP for the dense order sequence is the ordered clone maps. In this situation, the position is an ordering value that is devoid of metric or granularity information (see Section 1.5.4).

2.3.2. Metric/Fractional Sequence

A position type that has multiple granularities invariably also has a metric, since a "fraction" has to be based on some "unit". Therefore, a "fractional sequence" also has a defined metric attribute. Similarly, a metric type also has a granularity, due to our limitation to precisely measure differences in position. For metric/fractional sequences, the defining characteristics are:

[< [p , c] > , D , [M , G]] where M is metric unit, and G is granularity.

Note that M and G are bound together, since they must co-exist together. This is typical of sequences we encounter in the physical world of measurements. For example, a temporal sequence has time as position, with multiple metrics and granularities, but does not have atomicity.

2.3.3. Atomic Sequence

The defining characteristic of an atomic sequence is:

[< [p , c] > , D , [M , G , A]] where A is atomic granularity.

Note that M, G and A are bound first, because atomicity exists only when both metric and granularities are present. A simple example is letter sequences: M is the letter position, and G and A are both 1. In DNA, we use kilobases as M, then a given fragment can have G of 0.1 and A of 0.001. Determining the atomicity is application dependent. For example, should time metrics such as month or day be considered to be atomic? The determining factor is whether the units of position are counted or scaled. Real world measurement tools often convert a scaling factor to a counting

number, therefore, making the distinction a matter of choice. In financial applications, a day is considered to be atomic, while in physics, the same period of time is not atomic.

2.3.4. Examples of sequences

The initial sequence models fall into the following framework:

density	M/G	M/G/A	examples
dense	–	–	ordered classes
dense	+	–	continuous time and distance
dense	+	+	DNA sequences, words
sparse	–	–	incomplete ordered classes
sparse	+	–	event sequences
sparse	+	+	incomplete DNA sequences

The description of models is independent of the type of order. That is, sequences whose position type are circular or partial order can still have metric, granularity, and atomicity characteristics. While it is easier to visualize and discuss sequences of complete order, it is not a necessary requirement. In the context of the HGP, the sequences discussed in Chapter 1 can be classified as follows:

HGP	density	M/G	M/G/A	Special Characteristics
ssDNA	dense	+	+	
dsDNA	dense	+	+	symmetry
plasmid dsDNA	dense	+	+	circular order
ordered clone map	dense	-	-	order only
genetic map	sparse	+	+	
plasmid map	sparse	+	+	circular order
incomplete map	sparse	+	+	partial order

Note that only three types are found in the HGP: Dense, Dense/M/G/A, and Sparse/M/G/A.

2.4. Resolving Complications

The three initial models provides a good coverage of all the sequences types encountered in the Human Genome Project. However, several complications based on the interactions of metric, granularity, atomicity, and density are observed. In order to construct a more coherent and uniform framework, these complications must be resolved. These resolutions take on the form of transformations that convert sequences of one type into another.

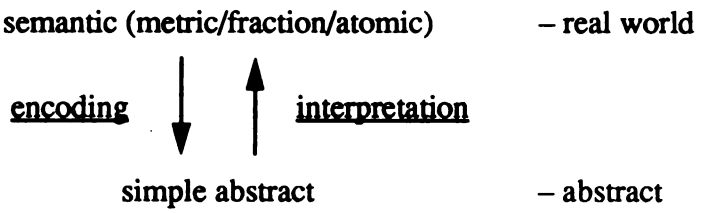
2.4.1. Semantic Interpretation and Abstraction

If a position type is measured by counting, then it is intrinsically atomic. In measurement theory, this is termed "absolute" scale. Examples of these are letter positions in a word or base position in a DNA sequence. Examples of non-atomic position types are scaled measurements such as seconds or inches. In measurement theory, this is "ratio" or "interval" scale [54]. Another measurement type that we encounter is the "ordinal" scale for pure order, as exemplified in order maps and in the enumerated position sequence discussed in Section 2.3.1. A final measurement type is "nominal" scale for arbitrary categorization, which as no corresponding sequence because there is no ordering associated with this type.

In the real world, scalable (ratio or interval) physical measurements are often converted to some form of counting prior to recording. For example, time delays are scaled, but a measurement device, such as a stop watch or its digital equivalent, measures time by counting exact increments at its granularity level. In addition, time periods, such as hours and days in the context of financial domain, are often considered countable not scalable. In all cases, at the level of the granularity, scaled measurements can only take on discrete values, thereby, become countable. If the position measurements, regardless of their countable or scalable nature, are equivalent to counting at the level of its granularity, then position values of metric/fractional/atomic (semantic) sequences can be mapped in 1-to-1 correspondence to the whole numbers. Consequently, the whole numbers can be used for the position type of the "simple abstract" sequence.

There are two directions for the 1-to-1 mapping. In this discussion, we use the term "interpretation" for the direction from abstract sequences to "real-world" sequences, because we attach semantics to abstract position values (whole numbers) by interpreting them in a specific metric/fractional/atomic context. We use the term "encoding" for the direction from "real-world" sequences to abstract sequences, because we remove the semantics from real-world position values by abstracting out the metric/fractional/atomic context. Appropriately, these mappings are domain and application specific, limiting any commonality that we shall need to account for in the abstract sequence

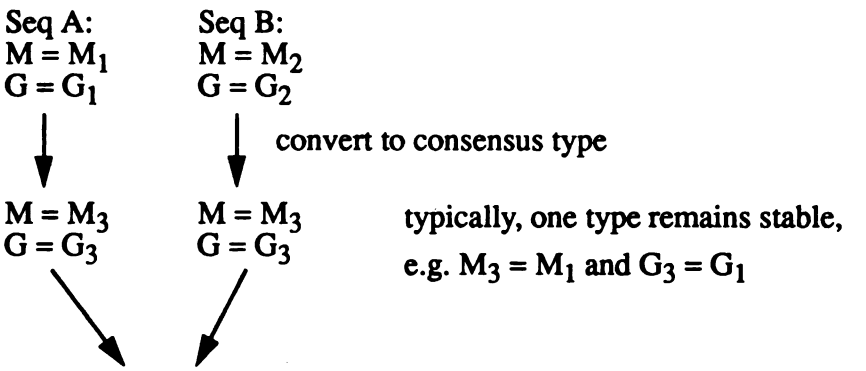
model.



2.4.2. Metric and Granularity Conversions

In the real world, all operations over metric/fractional sequences can be viewed in two stages. First, we convert the sequences into the same metric and granularity. Then, we can perform the operation, independent of the metric and granularity.

For example, in order to test for "equality":

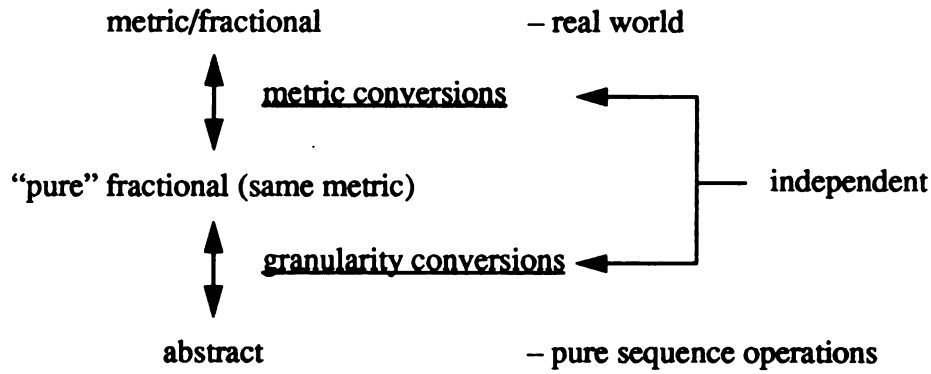


test for equality, this is operationally identical for all M_3 and G_3, i.e. metric and granularity independent.

The above analysis provides the first major insight to a uniform model of sequences: the separation of metric/granularity information from operational definitions. With this separation, operations can assume that operand sequences are derived from the same position type regardless of different metrics or granularities.

The conversion to a consensus type also has two stages: one to convert metric and the other to convert granularity. The two conversions can be categorized into different classes. Under metric conversions, there are multiplicative (e.g. inches/meters), linear (e.g. Fahrenheit/Centigrade) and nonlinear (e.g. exponential and Lorenz), which also correspond to "interval", "linear", and "nonlinear" in measurement theory. Under granularity conversions, there are averaging, minimum/maximum, and random sampling, which correspond to various statistical methodologies. The separation of metric and granularity conversions (more appropriately called transformations) simplifies our abstract model by eliminating the need to be concerned about the granularity during metric

transformations and vise versa. Such separation results in the "unbinding" of the two attributes:



2.4.3. Atomicity Constraints on Granularity

Atomicity, in addition to its countable nature, is an attribute that ensures the proper maximum resolution is not surpassed. For example, in the HGP, a common way to measure DNA lengths is by gel electrophoresis. When a given DNA fragment is measured at 1.4 kb with granularity of 0.1 kb, one can improve the measurement to 1.46 kb with granularity of 0.01 kb by using higher resolution gels and by comparison against markers of known sizes. However, DNA fragments are made of integral number of bases, thus the absolute finest granularity is 0.001 kb or 1 base and it is not possible to have a measurement of 1.4623 kb with granularity of 0.0001 kb. In this discussion, granularity is given as a fractional number, thus increasing its value is equivalent to make the granularity coarser and decreasing its value is the same as making the granularity finer.

The granularity associated with a sequence can only be made coarser, never finer, because a sequence is either *de novo* constructed or a result of an operation. A *de novo* sequence is ensured by real world semantics to have appropriate granularity, barring errors of transcription. All real world operations do not "inject" information into a sequence. Therefore, they will not decrease the granularity of the sequence and all resultant sequences are guaranteed to never have a granularity less than its real world atomicity. In the unlikely event where an operation is found to "inject" information, it is usually based on the information that is previously constructed *de novo* (at a finer granularity) or it is a violation of information theoretic foundations, i.e. getting more information out of a sequence than what it contains.

In addition to this "entropic" property of granularity, the separation of metric and granularity information from abstract sequence operations, discussed in Section 2.4.2, would also separate atomicity from the "simple" abstraction. However, the atomicity constraint only restricts which metric and granularity conversions are permissible, but does not affect the operations of the "simple abstract" sequence.

2.4.4. Changes in Density: Event Projection, Filtering, and Interpolation

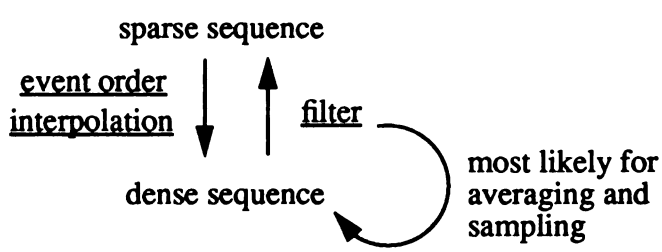
In an abstract sequence, position values are based on whole numbers. Whole number are dense, i.e. there are no whole numbers between any two consecutive whole numbers. In the real world, we do not always have sequences whose position values are dense. Thus, when we relate and operate on two sparse sequences, the gaps between the whole numbers in the "simple abstract" sequence need to be filled in order for the abstract sequence operations to continue. There are two transformations for the filling in the gaps.

First transformation is an interpolation function which converts a sparse sequence to a dense sequence. The missing position-content pairs are filled by this function, resulting in a dense sequence. This maintains synchronicity between position values of two sequences and keeps the operations on abstract sequences independent of density. In this discussion, interpolation functions are considered general, i.e. the default interpolation is filling the new content values with nulls. Specific interpolation functions are dependent on the content types, which are domain specific, and will not be elaborated here. In the real world, it is often useful to know which content values were in existence prior to interpolation and which are derived from the interpolation. This can be handled in the abstract sequence model by adding to the content type a flag for this distinction.

A second transformation is a mapping, named "event order", which only retains the ordinal value of the actual position values. This projection of sequence positions maintains the countable number of the actual position values, i.e. "events". The result is a dense sequence that can be viewed as a simple abstract sequence with the position type metric being "event number".

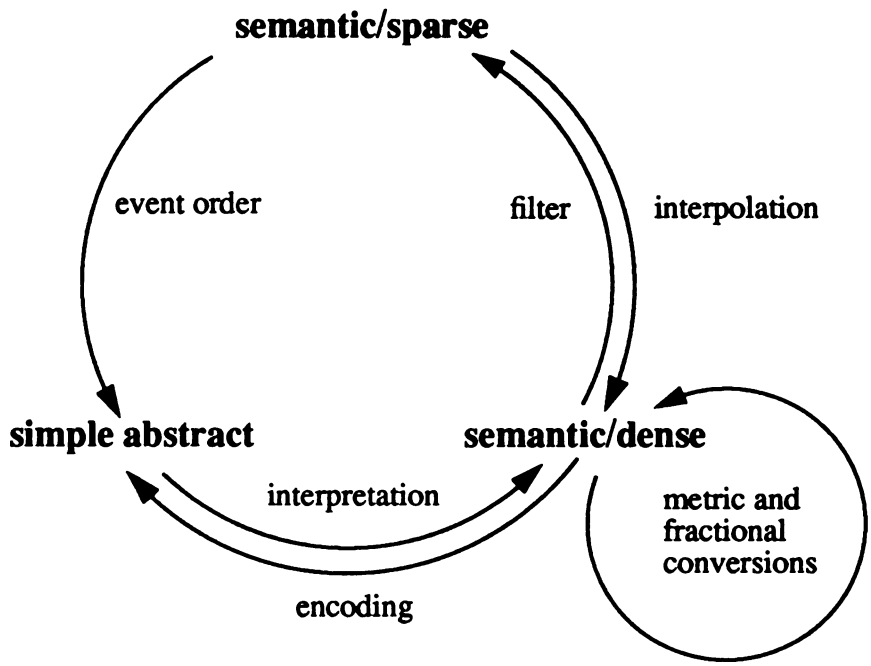
There are also operations in the direction from a dense to a sparse sequence. They are collectively termed "filter" transforms in this discussion. Examples are selection, averaging or sampling of content values to generate a representative value. If the granularity has not been changed, then the sequence becomes sparse. However, in most cases of averaging and sampling, these transforms also change the granularity, so the resulting sequence remains dense. Only the selection filter leaves the granularity unchanged.

These transforms are summarized as follows:



2.4.5. Summary of Transforms

When all the transforms are placed together, they creates the following framework:



2.5. Improved Sequence Model Framework

Consolidating the changes discussed in the previous section, we have the following new framework.

2.5.1. Order Sequence

An order sequence is defined as:

$< [p , c] >$, where $p \in$ Position and $c \in$ Content.

It is simpler than the initial framework model, since it does not participate in density interactions. Note that the position type is order only which distinguishes it from "simple sequences" (see below) whose type is metric. Since the only operations defined are equality and order, the position type is best represented by literals, i.e. "1", "2", etc. The danger of using whole numbers, e.g. 1, 2, 3, etc., as the supporting type is that arithmetic operations are permitted on whole numbers. This is not appropriate for position values of order sequences, because differences are not meaningful and users would erroneously "over-interpret" these differences as distances.

2.5.2. Simple Sequence

A simple sequence is defined as:

$< [p , c] >$, where $p \in$ Position and $c \in$ Content.

The position type is whole number based. This is the abstraction typically used whenever "sequences" are referred. The majority of the operations we define here fall into this category. A difference between any two position values has a meaning in the simple sequence while it does not in the order sequence. We use the LITERAL function to convert a simple sequence to an order sequence:

<u>LITERAL</u>: $p \rightarrow$ "p"

where p is a whole number, and "p" is the literal for that whole number.

Functions will be referenced in the discussion by a postfix "()" notation. Thus, LITERAL() represents the function defined above. Symbols listed inside the parenthesis are the arguments to the function.

A simple sequence can be converted to an order sequence by applying the LITERAL() transform to all its position values. The only characteristics of whole numbers that is preserved by the LITERAL() transform is order and equality. On the other hand, there is no direct conversion of an order sequence into a simple sequence.

All order and simple sequences are intrinsically dense, i.e. they have a content value for every position value in the sequence. A sparse sequence would contain missing content values, these are often denoted as "non-existent null's". However, there are other possible interpretations, e.g. "exist but unknown" and "not applicable". Therefore, the concept of sparsity is a semantic one, based on the

real world domain.

2.5.3. Semantic Mapping

A semantic sequence is defined as:

[< [p , c] > , [M , G , A]],

where p ∈ Position and c ∈ Content.

and [M , G , A] represents Metric, Granularity, and Atomicity, respectively.

In order to map the simple sequence into semantic sequences, we provide domain independent interpretations of position values. This way, the burden of defining sequence operations for each metric, fractional, atomic, and density type is removed. The mappings are then defined as:

INTERPRET: p ; [Metric , Granularity] → real world position.
p is a whole number from a simple sequence,
Metric is the name of real world metric, and
Granularity is the fractional number.

Similarly,

ENCODE: real world position ; [Metric , Granularity] → p

To complete the framework, we add as many conversion rules as needed. Thus,

Given: SeqA is sequence with [M_1 , G_1 , A] and
SeqB is sequence with [M_2 , G_1 , A], then
$MCONV_{1\to2}$: SeqA → SeqB, i.e. the transformation from M_1 to M_2.
Given: SeqC is sequence with [M_1 , G_2 , A], then
$GCONV_{1\to2}$: SeqA → SeqC, i.e. the transformation from G_1 to G_2.

For sparse sequences, we add one event order function and any number of interpolation and filtering functions. Thus

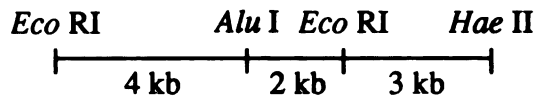
Given: SeqA is sparse and SeqB is dense with the same
[M , G , A] attribute, then
INTERPOLATE: SeqA → SeqB, i.e. transform a sparse sequence to dense.
FILTER: SeqB → SeqA, i.e. transform a dense sequence to sparse.
EVENT: SeqA → Event-SeqA, i.e. build a dense sequence of events.

Note that INTERPOLATE() should be idempotent, i.e. for any dense sequence, SeqC, INTERPOLATE(SeqC) = SeqC. The reason is that since SeqC is already dense, i.e. all content values exist, an interpolate function does not have any work to be done.

Event-SeqA can be an order sequence, simple sequence, or a dense semantic sequence whose [M , G , A] is defined as [event unit , 1 , 1]. However, a dense semantic sequence is most appropriate, since the concept of an "event" in a sequence is the same as a "letter" in a word. Since we also have

ENCODE() and LITERAL() transforms, we can transform Event-SeqA to a simple or an order sequence as dictated by domain or application. If the actual position values are needed for calculations, then one can include the original position value in the content value as a composite, but future content type operations on this transformed sequence will be different from the original content type operations. Consequently, the user will have to provide a set of necessary and correct operations. An example is shown below:

DNA map fragment:



model

Sparse Semantic Sequence:
Position = whole number
Content = enzyme name

< [0, *Eco* RI], [4, *Alu* I], [6, *Eco* RI], [9, *Hae* II] >
Metric = kb, Granularity = 1, Atomicity = 1

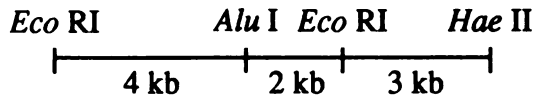
EVENT()

Event Sequence:
Position = whole number
Content = enzyme name and map distance

< [1, *Eco* RI-0kb], [2, *Alu* I-4kb],
[3, *Eco* RI-6kb], [4, *Hae* II-9kb] >
Metric = event, Granularity = 1, Atomicity = 1

A type of FILTER() is selection, which transforms a sequence to another by selecting for qualifying content values. Using the previous DNA fragment as an example, we can apply the following FILTER() to obtain another sequence:

DNA map fragment:



model

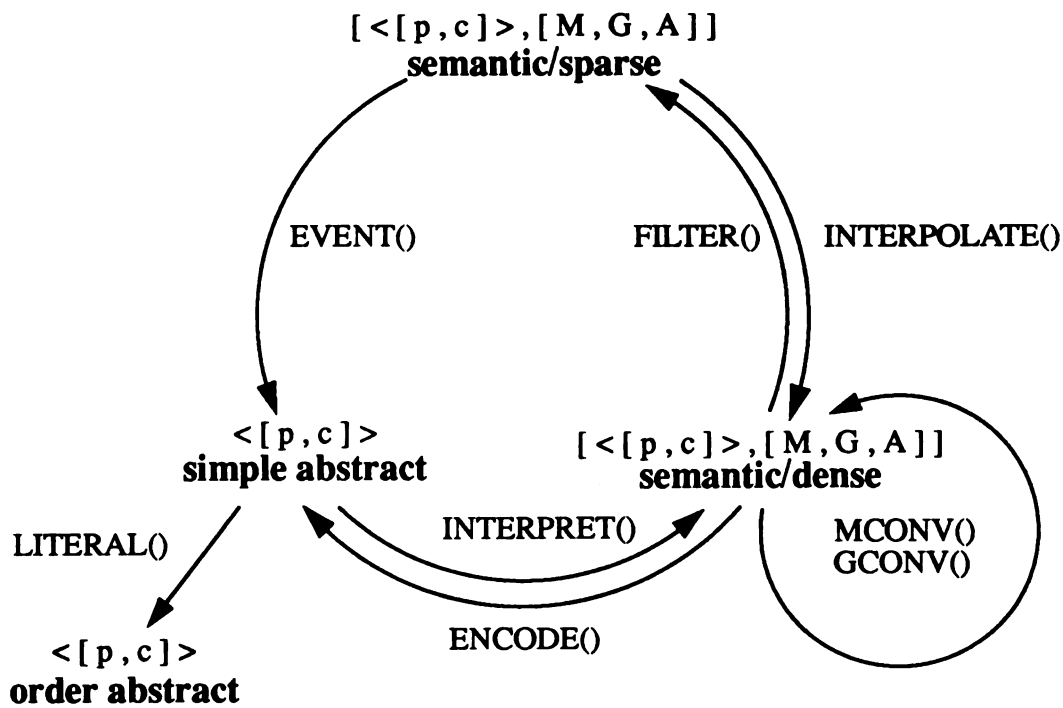
filter for *Eco* RI markers

FILTER()

< [0, *Eco* RI], [6, *Eco* RI], [9, null] >
Metric = kb, Granularity = 1, Atomicity = 1
(*Alu* I and *Hae* II removed from the sequence)

2.5.4. New Framework

The previous framework is now filled as follows:



2.6. Operations on Abstract Sequences

The following table describes all the semantic operations of sequences we have discussed:

sequence types	semantic operations
simple ↔ order	LITERAL()
semantic ↔ simple	INTERPRET(), ENCODE(), MCONV(), GCONV(), INTERPOLATE(), FILTER(), EVENT()

These operations map our abstract order and simple sequences to real world semantic sequences. They are domain and application specific, therefore, must be supplied by the users of the model. In contrast, there is a suite of sequence operations which are independent of semantics:

BASIC OPERATIONS		
content ↔ sequence	sequence ↔ sequence	multiple sequences
BUILD(), LENGTH(), VALUE(),	CUT(), CAT(), REVERSE()	APPLY(), RECURSE()

There are also operations that can be constructed from basic operations. The ones of interest to the HGP is listed below:

CONSTRUCTED OPERATIONS	
sequence ↔ sequence	multiple sequences
SUBSEQ()	SEQ_EQUAL(), MATCH(), OVERLAP()

2.6.1. Basic Notation

For discussion on operations, we will use the following notation:

<...> to denote a sequence,

{...} to denote a set,

[...] to denote an ordered n-tuple, and

a[b] to denote "subscripts", or the content value of the ordered pair [b , a] (where it is in the form [position , content]).

Given a position type, P, whose instances are ordered, and an arbitrary type, T, then the type of T-sequences, SEQ_T, is defined as follows:

P – position type,

T – content type,

x's – instances of content type T,

n – whole number or integer > 0,

then an instance of SEQ_T, $z \in$ SEQ_T, can be explicitly represented as:

$$z = < x [p_1] , x [p_2] , x [p_3] , \ldots , x [p_n] >$$

where p_i's are instances of P under the implicit order of $p_1 < p_2 < p_3 < \ldots < p_n$, and every occurrence of x associated with a "[p_i]" is an instance of T.

Operations are defined as:

op_name: list of domain variables → range type

and to be used as:

op_name (list of domain variables),

which returns an instance from the range type.

We will also assume the following:

(1) position values in a sequence is ordered and dense. In our framework the simple sequence position types can be represented by the whole numbers: 1, 2, 3, etc. The order sequence positions by whole number literals: "1", "2", "3", etc., i.e. they can be compared, but differences do not have any meaning.

(2) the existence of the some minimal position value. For simple sequence, the minimal element is denoted by 1. For the order sequence, the minimal element is also denoted by "1". Computationally, it is better to use 0 (or "0"), but it would be less intuitive to the non-computer science person.

(3) the existence of a null sequence, "null_seq", whose set of position-content pairs is the empty set.

(4) "equality" is defined for instances in T as T_EQUAL().

All the operations will be described in the context of simple sequences, whose position values are whole numbers. All the operations also apply to order sequences, however, one must use the literal form of the position values (see Section 2.5.2).

2.6.2. Content ↔ Sequence

First group of operations deal with the mapping between sequences and content types, usually through a position parameter.

BUILD: x ∈ T → SEQ_T

converts a single element of T into a T-sequence of length 1. The position value associated with x is the minimal value in the context of SEQ_T.

LENGTH: x ∈ SEQ_T → N (whole number)

returns the length of x. This is equivalent to the cardinality of the sequence. For order sequences, it is important that this number is derived by counting, because position differences do not have any meaning. For null_seq, LENGTH() returns 0.

VALUE: x ∈ SEQ_T ; p ∈ P → T

retrieves the value of x at position p. If x [p] is not defined, whether it is out of range or x is the null_seq, then VALUE(x , p) should return the null value for T.

2.6.3. Sequence ↔ Sequence

This group of operations deals with the construction of new sequence(s) from given sequence(s). The values of the content type of a sequence is not changed, but the associated position values will be changed.

CUT_FRONT: x ∈ SEQ_T ; p ∈ P → SEQ_T
CUT_BACK: x ∈ SEQ_T ; p ∈ P → SEQ_T

returns the beginning, or end, sequence of a cut placed at position p on sequence x. For simple sequences:

CUT_FRONT (x , 0) → null_seq
CUT_BACK (x , 0) → x

CUT_FRONT (x , 1) → BUILD(VALUE(x , 1))
CUT_BACK (x , 1) → z,
where z = < x [2] , x [3] , ... , x [n] >,
and n is the largest position value of x.

CUT_FRONT (x , LENGTH(x)) → x
CUT_BACK (x , LENGTH(x)) → null_seq

If x is null_seq, then CUT_FRONT and CUT_BACK returns null_seq.
If p > LENGTH(x), then the result is the same as if p is LENGTH(x) .

A question remains on whether z, the result of CUT_BACK(x , 1), starts at 1 or 2? If our assumption of minimal element is maintained, then z should be

$z = < z[1], z[2], z[3], \ldots, z[n-1] >$, and not

$z = < z[2], z[3], z[4], \ldots, z[n] >$.

So CUT_BACK(x , 1) should be:

$z[1] = x[2]$,

$z[2] = x[3]$, etc., which can be made robust by using induction.

To be completely robust, in order sequences, "n - 1", i.e. LITERAL(n - 1), has to be defined as:

SUCC(SUCC(... SUCC("1") ...))

with n - 1 composition of SUCC(), the successor function.

<u>CAT</u>: $x, y \in$ SEQ_T $\rightarrow$ SEQ_T

concatenates y to x. If we use explicit representation:

$x = < x[1], x[2], x[3], \ldots, x[n] >$ and

$y = < y[1], y[2], y[3], \ldots, y[m] >$,

then

$CAT(x, y) = < x[1], x[2], \ldots, x[n], y[1], y[2], \ldots, y[m] >$,

more correctly:

$CAT(x, y) = < z[1], z[2], \ldots, z[n+m] >$ where

for $1 <= i <= n$ do $z[i] = x[i]$ and

for $n < i <= n+m$ do $z[i] = y[i-n]$

As expected:

CAT(x , null_seq) = x

CAT(null_seq , x) = x

<u>REVERSE</u>: $x \in$ SEQ_T $\rightarrow$ SEQ_T

this reverses a sequence. It is the only function that does not conserve order information. For example:

$x = < x[1], x[2], \ldots, x[n] >$

then

$REVERSE(x) = < r[1], r[2], \ldots, r[n] >$, where

for $1 <= i <= n$ do $r[i] = x[n+1-i]$

If x is null_seq, REVERSE() returns a null_seq.

2.6.4. Multiple Sequences

Multiple sequence operations map sequences to other types. This is the most diverse group and the most difficult to generalize. Instead, we only present three basic operations in the fashion of LISP

operators. These operations permit the change of content values to construct new sequence types in arbitrary fashion.

<u>APPLY</u>: $x \in SEQ_T ; f \in (T \rightarrow T') \rightarrow SEQ_T'$

this is the iterative apply function, it takes a function f, which maps T to T′, and apply it to all the elements of an instance of T-sequence and returns an instance of T′-sequence. Position values and order are conserved under this operation, only the content values are altered.

APPLY(x , f) = < a [1] , a [2] , ... , a [n] >, where
for 1 <= i <= n do a [i] = f(x [i])

This operator permits an operation on content type to become an operation on the sequence type. This "super-operator" is independent of the content operation, therefore it eliminates the complexity of verifying each "extended" sequence operation.

<u>APPLY_TWO</u>: $x , y \in SEQ_T ; f \in (T , T \rightarrow T') \rightarrow SEQ_T'$

this is the apply "pair-wise" function over two sequences: for each position, p, in the range of x and y, we generate a new sequence by applying f:

APPLY_TWO(x , y , f) = < z [1] , z [2] , ... , z [n] >, where
for 1 <= i <= max(LENGTH(x) , LENGTH(y)) do
z [i] = f(x [i] , y [i])

The event-joins of the temporal models can be constructed by using APPLY_TWO():

1. convert the JOIN operation as f() over the content value.
2. perform APPLY_TWO(x , y , f).

This is more general than event-joins, since it is independent of sequence type or content value and it is not limited to work with only corresponding attributes between the two sequences.

<u>RECURSE</u>: $x \in SEQ_T ; f \in (T , T' \rightarrow T') ; d \in T' \rightarrow T'$

this is simple recursive apply function.

RRECURSE (x , f) = f(x [1] , f(x [2] , ... f(x [n] , d) ...))

The value d is used as default value for the initial f(). The previous example is right associative, similarly, there is a left associative RECURSE:

LRECURSE (x , f) = f(... f(f(d , x [1]) , x [2]) ... , x [n])

The last three functions are based on a new class of operations that take a function as one (or more) of their arguments. The return types are no longer restricted by content type T, position type P, nor sequence type SEQ_T. In the HGP, only APPLY() and APPLY_TWO() are of interest.

2.6.5. Constructed Operations

Many other common operations can be built from these operations. The construction is of theoretical interest only, because in the real world, most of them are implemented directly. Notable examples are:

<u>SUBSEQ</u>: $x \in$ SEQ_T; p1 , p2 $\in$ P $\rightarrow$ SEQ_T

returns a subsequence of x. This can be constructed from CUT_FRONT() and CUT_BACK():

SUBSEQ(x , p_1 , p_2) = CUT_FRONT(CUT_BACK(x , p_1) , p_2)

<u>SEQ_EQUAL</u>: x , y $\in$ SEQ_T $\rightarrow$ Boolean

checks to see whether x and y are equal. This can be constructed from the left associative RECURSE() (LRECURSE):

Let	T′	= [Boolean, SEQ_T],
	X and Y	$\in$ SEQ_T,
	x and y	$\in$ T,
	t	$\in$ Boolean, and
	left([A , B])	returns A.

Define f : T′, T $\rightarrow$ T′ as f ([t , Y] , x) returns [t′ , Y′], where:

t′ = AND(t , T_EQUAL(x , VALUE(Y , 1)))
Y′ = CUT_BACK(Y , 1)

then SEQ_EQUAL(X , Y) = left(LRECURSE (X , f , [True , Y]))

<u>MATCH</u>: x , y $\in$ SEQ_T ; p $\in$ P $\rightarrow$ Boolean

checks to see whether y is a subsequence of x at position p:

SEQ_EQUAL(SUBSEQ(x , p , p + LENGTH(y)) , y)

<u>OVERLAP</u>: x , y $\in$ SEQ_T ; p $\in$ P $\rightarrow$ Boolean

checks to see whether y overlaps x at position p:

SEQ_EQUAL(CUT_BACK(x , p) , y)

Overlap definition could be extended to check all possibilities, including embedding, i.e. y is within x.

<u>MAX_OVERLAP</u>: x , y $\in$ SEQ_T $\rightarrow$ p $\in$ P

determines the position of maximum overlap. A simple $O(n^2)$ algorithm is:

```
for p from 1 to LENGTH( x ) do
        if ( OVERLAP( x , y , p ) ) then return( p )
endfor
```

CHAPTER 3. MODELS OF BIOLOGICAL SEQUENCES

With the definition of the abstract sequence model complete, we can now develop the model for biological objects. We will start with the nucleotides, then single stranded (SS) sequences, double stranded (DS) sequences, and finally, markers and fragments.

3.1. Nucleotides

DNA bases can be modelled as an enumerated type, BASE: {A, C, G, T}. As required, we have base_equal(). In addition, we have base_complement(), which returns the complement of a base. We could also model Pu, Py, and N as a part of the enumerated type. Then the functions base_equal() and base_complement() are defined as:

base equality	A	C	G	T	Pu	Py	N
A	T	F	F	F	T	F	T
C	F	T	F	F	F	T	T
G	F	F	T	F	T	F	T
T	F	F	F	T	F	T	T
Pu	T	F	T	F	T	F	T
Py	F	T	F	T	F	T	T
N	T	T	T	T	T	T	T

base	complement
A	T
C	G
G	C
T	A
Pu	Py
Py	Pu
N	N

"Pu" stands for Purines, which could be either A or G.

"Py" stands for Pyrimidines, which could be either C or T.

"N" stands for any Nucleotide.

DNA sequences would not normally require the use of Pu, Py, or N, because if the sequence is known, then the exact base is known, or if the sequence is unknown then it would not be a "valid" sequence of bases. The use of Pu, Py, or N is to model intermediate uncertainty, i.e. under certain situations, we only have partial information about the sequence. Instead of losing this partial information by not storing anything, we could use these special enumerations. A situation where this is most useful is the modelling of restriction enzyme recognition sequences. However, there are other groupings that lack widely used names, such as W for A or T and S for C or G [48].

3.2. Single Stranded Sequences

To model real world sequences in our framework, we only need to specify the [M , G , A] triplet and the semantic operators. For single stranded (SS) DNA sequences, [M , G , A] is [base unit , 1 , 1]. Let:

SEQ_BASE = sequence of bases,

P = SEQ_BASE position values, which are in "base units", and

W = whole numbers of the simple sequence position values.

We will use ssDNA to represent the biological single stranded DNA and SS_DNA to represent the sequence model type for single stranded DNA. For abstract operations, SS_DNA is synonymous with SEQ_BASE, however, for operations that are significant only in biology, we will use SS_DNA.

For brevity, we will use the character string representation for SEQ_BASE: "$b_1b_2b_3$ ...", instead of the verbose form: < [1 , b_1], [2 , b_2], [3 , b_3], ... >.

3.2.1. Semantic Operators

The semantic operators, which map semantic sequences to and from abstract sequences, are defined in Section 2.5.3. They are described in the following table, under the context of biology:

Semantic Operators	Mapping	Definition
INTERPRET()	$n \in W \rightarrow p \in P$	p = n base_unit
ENCODE()	$p \in P \rightarrow n \in W$	n = numerical value of p
INTERPOLATE()	X ∈ SEQ_BASE → Y ∈ SEQ_BASE	y[p] = N for all p not in X and y[p] = x[p] for all p in X.
EVENT()	X ∈ SEQ_BASE → Y ∈ SEQ_EVENT	y[n] = x[p], where n is the ordinal number of p in X

Comments:

1. MCONV() and GCONV() are not defined in the biological domain of ssDNA because there is only one metric ("base unit") and one granularity ("1 base unit").
2. The INTERPOLATE() function can be the null function where N is "not applicable" or the total function where N is "any base".

3.2.2. Sequence Operators

With these definitions, all ssDNA operations can be derived from the abstract simple sequence

operations (define in Section 2.6). The ones of interest for the HGP are:

Sequence Operators	Comments
LENGTH()	length in base units
VALUE()	returns the base value
BUILD()	length is 1 base unit for building SS_DNA
SUBSEQ()	returns a subsequence
CAT()	concatenate two ssDNA
SEQ_EQUAL()	defined by using base_equal()
APPLY()	useful for building biological valid functions
REVERSE()	since ssDNA is directional, this is used only for building COMPLEMENT() (see below)

3.2.3. Biological Operators

From the set of abstract sequence operators, we can construct operators specific to biology.

Biological Operators	Comments
COMPLEMENT()	complements a ssDNA
IS_COMPLEMENT()	checks to see if two ssDNA are complementary
MATCH_CUT()	cut ssDNA where sequences match

1. <u>COMPLEMENT</u>: X ∈ SS_DNA → SS_DNA

   A biological complementary sequence is both base-by-base complement and in reverse order. Therefore, the definition is:

   REVERSE(APPLY(X , base_complement()))

2. <u>IS_COMPLEMENT</u>: X , Y ∈ SS_DNA → Boolean, is defined by:

   SEQ_EQUAL(X , COMPLEMENT(Y))

3. <u>MATCH_CUT</u>: X , Y ∈ SS_DNA → set of SS_DNA

   cuts at where Y matches X, returns a set of SS_DNA. We will use a SS_DNA "match cut" as a building block function for the enzyme digest of dsDNA. For example, if we are given:

   X = "ACTAGAAAAAGTC" and

   Y = "AAA",

then assuming that the cut is placed before the match, a straight forward computational result will be: {"ACTAG", "A", "AAAGTC"}. However, biology does not deal with a single instance of SS_DNA. Typically, in an experiment, there is a pool of identical instances (value-wise) undergoing the same processing. The result of MATCH_CUT(), in the biological domain, is:

{ "ACTAG", "AA",
"ACTAGA", "A",
"ACTAGAA", "AAAGTC"}.

This is very different from the simple model of MATCH_CUT().

3.3. Double Stranded Sequences

3.3.1. DS_DNA Models

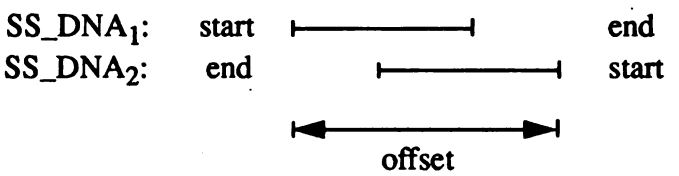
There is no direct sequence model for dsDNA because a dsDNA has rotational symmetry and because it can have arbitrary SS regions. Therefore, dsDNA has to be modelled as a complex type. We shall discuss three complex models for dsDNA: symmetric, asymmetric, and polymorphic. To preserve rotational symmetry, a "rotate" operator is needed to obtain a rotational equivalent DS_DNA from a given DS_DNA. This operator, when used in conjunction with simple equality, provides the basis for DS_DNA equality.

Semantic Operators	Mapping	Definition
ROTATE()	X ∈ DS_DNA → DS_DNA	returns the value of X after rotation

A. Symmetric Model

The first approach is to build a model that is "symmetric" to rotation.

$DS_DNA_1 = [\ SS_DNA_1\ ,\ SS_DNA_2\ ,\ offset\ ]$, where:

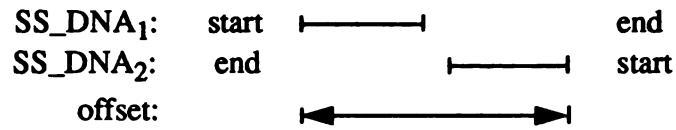


offset is the distance between the starts of the component SS_DNA's, more appropriately, the position of SS_DNA_2's start relative to SS_DNA_1's start.

The rotational symmetry of DS_DNA is preserved as a swap of the sequence elements within the ordered triple, since:

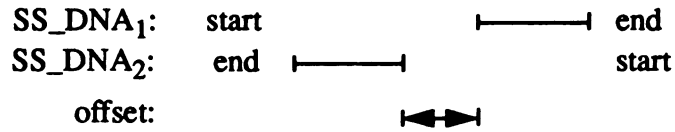
$\underline{ROTATE}(\ [\ s_1\ ,\ s_2\ ,\ offset\ ]\ ) \rightarrow [\ s_2\ ,\ s_1\ ,\ offset\ ]$.

If offset ≥ (length of SS_DNA_1 + SS_DNA_2 – 1) or offset < 0, we have a null DS_DNA_1:



offset ≥ summed length –1, thus invalid DS_DNA

or

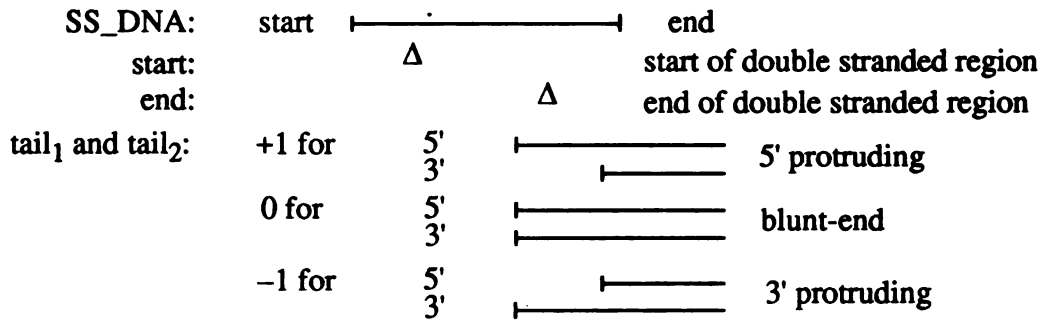


offset is negative, thus invalid DS_DNA

B. Asymmetric Model

The second model is a brute force model, that specifies the components as we encounter them. Here DS_DNA is broken down into three sections. It lacks the symmetry of the first model.

DS_DNA_2 = [SS_DNA , start , end , $tail_1$, $tail_2$], where:



Rotational symmetry is defined as the following operation:

<u>ROTATE</u>(DS_DNA_2) → [COMPLEMENT(SS_DNA) , LENGTH(SS_DNA) – end , LENGTH(SS_DNA) – start , – $tail_2$, – $tail_1$]

C. Polymorphic Model

The third approach to modelling DS_DNA is a polymorphic sequence, made of SS sequences and filled DS sequences. The intermediate sequence models needed to create the overall model are:

1. SIMPLE_DS is a filled, blunt-end DS_DNA.

   Since it has no SS tails, we can use a simple sequence as its model. But rotational symmetry requires that an orientation be specified when a simple sequence is used to model a DS_DNA.

   SIMPLE_DS = [ss ∈ SS_DNA , orientation], where

   orientation = 1 if ss is in the same orientation with respect to the larger context,

–1 if ss is in the opposite orientation, and
0 if ss is not in a larger context.

ROTATE(SIMPLE_DS) → [ss , – orientation] or [COMPLEMENT(ss) , orientation]

Either results is valid.

2. SIMPLE_SS is the same as SS_DNA.

However, to maintain the correct biological orientation, it must also have an attribute to determine its orientation with respect to the larger sequence.

SIMPLE_SS = [ss ∈ SS_DNA , orientation], where

orientation = 1 if ss is in the same orientation as the larger context,
i.e. 5' to 3' is in the same direction.
-1 if ss is in the opposite orientation, 5' to 3' is the other direction, and
0 if ss is not in a larger context.

ROTATE(SIMPLE_SS) → [ss , – orientation]

Since a SS sequence is always read from 5' to 3', we do not complement ss, but change its orientation.

3. POLY_SEQ is a sequence made of SIMPLE_SS, SIMPLE_DS, and POLY_SEQ.

The rotation operation on a POLY_SEQ is similar to COMPLEMENT() on SS_DNA, it is defined by using the APPLY() operation:

<u>ROTATE</u>(ps) = REVERSE(APPLY(ps , ROTATE))

Since POLY_SEQ is recursive, we can recursively apply the ROTATE() operation until SIMPLE_DS or SIMPLE_SS has been reached.

A piece of DS_DNA, with two SS end-tails, can be modelled under the polymorphic model as:

$ps = < ss_1 , ds , ss_2 >$, where $ss_1 , ss_2 \in$ SIMPLE_SS and $ds \in$ SIMPLE_DS

This is very similar to the asymmetric model. However, it has the advantage of constructing DS sequences with SS gaps. In general, this advantage is not required, because internal SS regions are usually filled by biological processes.

If we restrict the polymorphic model to a simple SS-DS-SS sequence, then the three models are equivalent because they capture the same amount of information. The interconversion between them is straight forward, therefore, we can use whichever model is convenient for the operation under study. This reduces the overhead that we make to implement a particular operation for a particular model. We will predominantly use DS_DNA_1 and DS_DNA_2 for the following discussion.

3.3.1. Semantic and Sequence Operators

Another consequence of rotational symmetry is its effect on position values in a DS_DNA. The

meaning of a position can be viewed as a physical point, not just a distance value from a reference. Since DS_DNA has rotational symmetry, a position should be orientation independent, i.e. it marks the same physical point regardless of the orientation of the DS_DNA. This implies that a position is always attached to the sequence from where it was obtained. For example, a DS_Position is [p , ds], where p is a whole number and ds is the originating DS_DNA with the implicit understanding that p is based on the orientation of ds. We minimize the complexity of this problem by only using position values as inputs to operators. In this case, position values are whole numbers and are to be associated with the orientation of the input DS_DNA.

Because DS_DNA is no longer a direct sequence model, semantic operators are no longer appropriate. However, an equivalent set of them can be constructed to maintain the "sequence" nature of DS_DNA:

Semantic Operators	Mapping	Definition
INTERPRET()	$n \in W \rightarrow p \in DS_P$	p = [n base_units , ds]
ENCODE()	$p \in P \rightarrow n \in W$	n = numerical value of p
INTERPOLATE()	$X \in DS_DNA \rightarrow$ $Y \in DS_DNA$	y[p] = N for all p not in X and y[p] = x[p] for all p in X.
EVENT()	$X \in DS_DNA \rightarrow$ $Y \in DS_DNA$	y[n] = x[p], where n is the ordinal number of p in X
ROTATE()	$X \in DS_DNA \rightarrow$ DS_DNA	returns the value of X after rotation

Although the simple sequence operators (defined in Section 2.6) are no longer directly applicable to DS_DNA, an equivalent set of them can be created. The ones of interest are:

Sequence Operators	Comments
DS_LENGTH()	total length in base units
DS_VALUE()	returns the base value under the default orientation
DS_SUBSEQ()	returns a subsequence with the same orientation

3.3.2. Biological Operators on DS_DNA

The biological operators are:

Biological Operators		
SS_DNA ↔ DS_DNA	DS_DNA ↔ DS_DNA	mixed
ANNEAL(), DENATURE()	FILL(), TRIM(), DS_CAT(), DS_EQUAL()	DS_MATCH(), DS_MATCH_CUT()

1. ANNEAL: s_1 , $s_2 \in$ SS_DNA → DS_DNA_1

   takes two SS_DNA and construct a DS_DNA based on complementary overlap. Conceivably, there can be a set of DS_DNA, since there may be more than one overlap. We will be interested in only the maximum overlap.

2. DENATURE: ds $\in$ DS_DNA_1 → { s_1 , $s_2 \in$ SS_DNA }

   takes a DS_DNA and returns the two SS_DNA components. This returns a set, not an ordered pair, because biologically, the two SS_DNA components can not be ordered.

3. FILL: ds $\in$ DS_DNA_2 → DS_DNA_2

   fills the SS tails of a DS_DNA. If we let ds $\in$ DS_DNA_2 and:

   ds = [ss , start , end , $tail_1$, $tail_2$] ,

   then, the resultant DS_DNA_2 would be:

   filled_ds = [ss , 1 , LENGTH(ss) , 0 , 0].

4. TRIM: ds $\in$ DS_DNA_2 → DS_DNA_2

   removes the SS tails. The resultant DS_DNA_2, from previous ds, would be:

   trimmed_ds = [SUBSEQ(ss , start , end) , 1 , end – start + 1 , 0 , 0].

5. DS_EQUAL: ds_1 , $ds_2 \in$ DS_DNA_1 → Boolean

   determines whether two DS_DNA are equal. Since rotational symmetry is possible, we have:

   ($ds_1.s_1 = ds_2.s_1$ AND $ds_1.s_2 = ds_2.s_2$ AND ds_1.offset = ds_2.offset)

   OR

   ($ds_1.s_1 = ds_2.s_2$ AND $ds_1.s_2 = ds_2.s_1$ AND ds_1.offset = ds_2.offset)

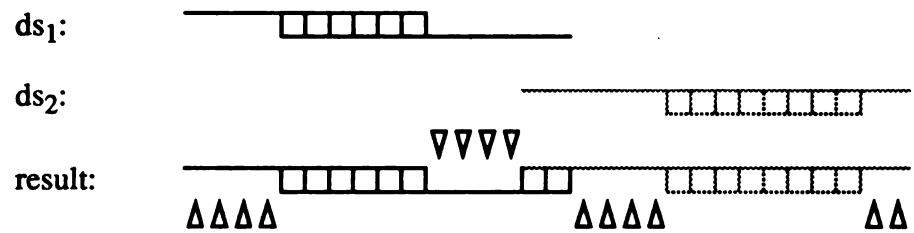
   where $ds_i.s_j$ denotes the s_j component of ds_i and "=" is SS_EQUAL for SS_DNA components and integer equality for offsets.

   Another implementation of DS_EQUAL is based on a SIMPLE_EQUAL() that operates on the components of the specific DS_DNA model:

SIMPLE_EQUAL(ds_1 , ds_2) OR SIMPLE_EQUAL(ds_1 , ROTATE(ds_2))

6. <u>DS_CAT</u>: ds_1 , $ds_2 \in$ DS_DNA$_2 \rightarrow$ DS_DNA$_3$

concatenates two DS_DNA together. Conceivably, ds_1 and ds_2 can have four possible orientations for concatenation, and since more than one overlap is possible, this should return a set of all possible results. Furthermore, such a concatenation may have internal SS gaps:



Δ and ∇ mark the single stranded gaps

The full model would require the use of polymorphic sequence. For simplicity, we will assume the resultant DS_DNA will have all the gaps filled and the overlap is maximal.

7. <u>DS_MATCH</u>: ds $\in$ DS_DNA$_1$; ss $\in$ SS_DNA $\rightarrow$ set of DS positions

returns all the DS positions where ss matches ds. We can use SS MATCH() and iterate through the length of $ds.s_1$ to generate matching positions on the given orientation, i.e. associated with ds. Then apply MATCH() against $ds.s_2$ for the complementary strand, which would generate matching positions associated with ROTATE(ds). This is an example of an operator where returned position values must be associated with the source DS_DNA.

8. <u>DS_MATCH_CUT</u>: ds , $ds_e \in$ DS_DNA$_2 \rightarrow$ set of DS_DNA

This is the definition of the restriction enzyme cutting. The result is a set of DS_DNA. We use ds_e, a DS_DNA$_2$, to model restriction enzyme site instead of a SS_DNA. The reason is that SS_DNA is insufficient as a specification for enzyme cutting site, because the cut site is often offset from the start of the recognition sequence on both component strands of the DS_DNA. Therefore, ds_e is used as a template, matched against ds in both orientation. We will discuss the use of DS_DNA for restriction recognition site in the next section. The complete operation specification is nontrivial, but is computable.

3.4. Fragment Markers

There are two types of markers: restriction enzyme sites and probes. Both can be modelled as DS_DNA.

3.4.1. Restriction Enzyme Sequences

Restriction enzyme sites are not physical DS_DNA, but rather recognition sequences. Since some enzymes can recognize more than one specific sequence, therefore, a marker can correspond to a set of exact sequences. Since a recognition sequence is usually thought as a simple sequence, why do we modelled it as a DS_DNA? The reason is that restriction enzyme recognize DS_DNA, not SS_DNA, and the cut site can be modelled by the SS tails. Furthermore, the rotational symmetry is preserved by enzyme, i.e. it matches a sequence independent of orientation:

Eco RI recognizes:

```
G|AATT C
C TTAA|G
```

when given this DS_DNA sequence:

```
ATGAATTCGTATGCTTAAGATG
TACTTAAGCATACGAATTCTAC
```

Eco RI will cut at:

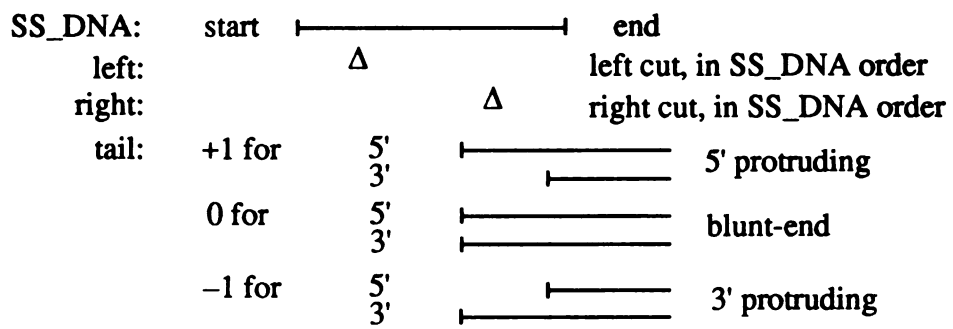
```
ATG|AATT CGTATGC TTAA|GATG
TAC TTAA|GCATACG|AATT CTAC
```

creating three DS_DNA sequences:

```
ATG----      AATTCGTATGCTTAA        ----GATG
TACTTAA  ,   ----GCATACG----  , and AATTCTAC
```

Therefore, if the recognition DS_DNA sequence is considered as a template overlay, we will use it in both orientations, as appropriate for all DS_DNA operations. Using DS_DNA_2 as a base model, we modify it slightly to conform to recognition semantics.

RE_SEQ = [SS_DNA , left , right , tail], where:



For example:

Eco RI sequence, $\begin{matrix} \mathtt{G} & \mathtt{AATT} & \mathtt{C} \\ \mathtt{C} & \mathtt{TTAA} & \mathtt{G} \end{matrix}$, becomes ["GATTC" , 1 , 5 , +1].

Alu I sequence, $\begin{matrix} \mathtt{AG} & \mathtt{CT} \\ \mathtt{CT} & \mathtt{GA} \end{matrix}$, becomes ["AGCT", 2 , 2 , 0].

Hae II, sequence, $\begin{matrix} \mathtt{Pu} & \mathtt{GCGC} & \mathtt{Py} \\ \mathtt{Py} & \mathtt{CGCG} & \mathtt{Pu} \end{matrix}$, becomes ["PuGCGCPy" , 1 , 5 , –1].

Pu stands for Purines (A or C)
Py stands for Pyrimidines (T or G)

Although, these three examples are symmetric, there are restriction sequences where the cut site is asymmetric, thus both "left" and "right" is needed. However, the "tail" information only needs one value. The use "Pu" and "Py" in *Hae* II recognition sequence is a form of sequence shorthand for a set of exact sequences (see Section 1.5.1). However, this may not be sufficient to describe all possible recognition sequences. If the original DS_DNA$_2$ model is used, then tail$_1$ = tail and tail$_2$ = – tail. This would retain the consistency between restriction sequences and probes.

3.4.2. Probe Sequences

A probe, on the other hand, is physically based on a DS_DNA, however, it is used as a SS recognition sequence via the IS_COMPLEMENT() operator. Although the exact sequence of a probe may be unknown, it is sufficiently unique to be identifiable. In actual biological experiments, one or both of the component SS_DNA can used as the recognition sequence, but operationally, it is the same as using a DS_DNA template in both orientations.

3.4.3. Markers in Fragments

Markers, when they are placed on a fragment, are considered as point-like, i.e. the intrinsic lengths of markers are considered too small to be visible in the context of a fragment. Appropriately, they are considered as "events" on a sequence whose granularity and atomicity is greater than 1 base. Subsequently, both restriction sequences and probes can be abstracted as a single quantum, i.e. a "marker".

3.5. Fragment As Sequence

A fragment is a sequence of markers. The sequence content type is a marker, i.e. a DS_DNA as described in the previous section. The only operation required of markers is equality. The sequence position type is dependent on the measuring device for the fragment. When a fragment is measured by a gel, its metric is usually in units of bases or kilobases and the granularity is dependent on the composition of the gel and varies between 1 base to 100+ kb. However, when a fragment is measured by recombination, the metric unit is centiMorgan and its granularity ranges from 0.001 to 0.1. If a chromosomal fragment is visualized by cytological staining, the metric unit can be "percent of arm-length" and its granularity ranges between 0.001 to 0.1. In all cases, the atomicity never goes below 1 base. The table below describes some of the values encountered in the HGP.

Methodology	Metrics	Granularity Range	Atomicity
Physical	base	1 and up	1 base
	kilobase	0.001 and up	0.001 kilobase
	megabase	0.001 and up	10^{-6} megabase
Biological	centiRad	0.001 to 0.1	not exact
	centiMorgan	0.001 to 0.1	not exact, $\sim 10^{-6}$
Histological	band position	p1 to p111 and q1 to q111	not exact
	arm length	0.001 to 0.1	not exact

The relationships between biological and histological metrics to bases cannot be described by a simple ratio. Under current technology, the finest granularity achieved by these methodologies do not approach the resolution of a single base. However, the mapping between these metrics and bases are monotonic and, in theory, an experimentally derived function can be found.

We will assume our fragment sequence to be dense, with a "null" (unknown) marker value for the positions that we do not have actual marker values.

3.5.1. Semantic Operators

The semantic operators for fragments are:

Semantic Operators	Mapping	Definition
INTERPRET()	$n \in W \rightarrow p \in P$	p = n metric units
ENCODE()	$p \in P \rightarrow n \in W$	n = numerical value of p

Semantic Operators	Mapping	Definition
MCONV()	$X \in SEQ_MARK_1$; $m \in M_2 \rightarrow SEQ_MARK_2$	convert metrics
GCONV()	$X \in SEQ_MARK_1 \rightarrow SEQ_MARK_2$	convert granularity
INTERPOLATE()	$X \in SEQ_MARK \rightarrow Y \in SEQ_MARK$	y[p] = N for all p not in X and y[p] = x[p] for all p in X.
EVENT()	$X \in SEQ_MARK \rightarrow Y \in SEQ_MARK$	y[n] = x[p], where n is the ordinal number of p in X
ROTATE()	$X \in SEQ_MARK \rightarrow SEQ_MARK$	returns the value of X after rotation

Comments:

1. SEQ_MARK represents a sequence of markers with an appropriate [M , G , A].
2. coarse-to-fine GCONV

   These are rare, and if needed, are usually defined by insertion of null marker values for the newly formed position values.
3. fine-to-coarse GCONV

   These can be defined by rounding off position values. However, multiple markers may collect into a single resultant position. For example, if two or more markers now occupy the new position value, then a composite value, e.g. a set or a bag, is needed as a marker value. Another method is to directly use Set of DS_DNA as a marker value. Then no matter how many markers are collected per position, we still have a sequence of markers.
4. The rotational symmetry of dsDNA is conserved for operations that consider fragments as large DS_DNA sequences. When applicable, rotational transform is defined as:

   <u>ROTATE</u>(fs) = REVERSE(APPLY(fs , ROTATE)), where

   fs is a sequence of markers, and

   ROTATE() is DS_DNA ROTATE().

   This symmetry is broken for operations at the level of fragments derived from histological methodologies, because the marker positions now have an absolute reference frame based on the chromosome structure. Although one can argue that all sequences are located somewhere on a chromosome and therefore, they must have an absolute reference. However, it is the operations that determine whether rotational symmetry is meaningful or not.

3.5.2. Sequence Operators

Sequence operators (based in Section 2.6) for fragments are:

Sequence Operators	Comments
LENGTH()	length in appropriate metric units
VALUE()	returns a marker
FRAG_BUILD()	requires a beginning Marker, an end Marker, and length.
SUBSEQ()	returns a subsequence
CAT()	concatenate two fragments
SEQ_EQUAL()	defined by using marker_equal()
APPLY()	useful for building biological valid functions
REVERSE()	this is used only for building a biologically valid reverse

Comments:

1. The original BUILD() only constructs a sequence of 1 unit long. The FRAG_BUILD() is a shorthand for building a longer sequence with implicit "null" values.
2. A check should be made in CAT() to make sure the end marker of fs_1 is biologically compatible with the beginning marker of fs_2.

3.5.3. Biologically Operators

For biologically significant sequences of markers, we will use the synonym FRAGMENT, instead of SEQ_MARK. Biologically pertinent operators for fragments are:

Biological Operators		
FRAGMENT ↔ others	FRAGMENT ↔ FRAGMENT	mixed
FRAG_OVERLAP()	FRAG_EQUAL(), FRAG_CONTAINS(), INSERT_MARKER()	CUT_AT_MARKER(), CUT_AT_POS()

1. <u>FRAG_EQUAL</u> : fs_1 , $fs_2 \in$ FRAGMENT → Boolean

   checks for exact equality of two fragments

2. FRAG_OVERLAP : fs_1 , $fs_2 \in$ FRAGMENT → Position

   returns the position in fs_1 where fs_2 overlap.

3. FRAG_CONTAINS : fs_1 , $fs_2 \in$ FRAGMENT → Boolean

   checks for containment of fs_2 in fs_1.

4. INSERT_MARKER : fs ∈ FRAGMENT ; m ∈ MARKER ; p ∈ Position → FRAGMENT

   inserts Marker m at Position p in fs.

5. CUT_AT_POS : fs ∈ FRAGMENT ; p ∈ Position → Set of FRAGMENT

   cuts fs at Position p. The result is a set of two fragments. The newly created ends do not have any known marker value, i.e. they have a "null" value.

6. CUT_AT_MARKER : fs ∈ FRAGMENT ; m ∈ MARKER → Set of FRAGMENT

   cuts fs wherever Marker m occurs. The result is a set of fragments.

3.5.4. Circular Fragments

A biologically significant form of DNA fragment is the circular DNA. To model this under the abstract sequence model requires an additional attribute of Topology. If the value of Topology is "linear", then no changes are required for the operators. If the value is "circular", then the "zero" point, i.e. the beginning of the equivalent linear fragment, is used as a reference point. The following operators are defined:

Circular Operators		
circularization	FRAGMENT ↔ FRAGMENT	mixed
CIRCULARIZE()	FRAG_EQUAL(), FRAG_CONTAINS(), INSERT_MARKER()	CUT_AT_MARKER(), CUT_AT_POS()

1. CIRCULARIZE: fs ∈ LINEAR_FRAGMENT → CIRCULAR_FRAGMENT

   This takes a linear fragment and circularize it if the ends of the fragment are compatible.

2. FRAG_EQUAL() and INSERT_MARKER() remains the same.

3. CUT_AT_MARKER: fs ∈ FRAGMENT ; m ∈ MARKER → Set of FRAGMENT

   This is identical to the linear form of CUT_AT_MARKER() defined above, except that the ends of the original fs are sealed between two Markers. Same applies to CUT_AT_POS().

4. FRAG_CONTAINS() is defined if fs_1 is circular and fs_2 is linear.

3.5.5. Ordered Maps

A special type of sequence information found in the HGP is from the ordered clone maps. They are sequences of DNA fragments whose order is known, but not their distances. Therefore, the position type is of pure order. In Section 2.3.4, we used the dense order sequence to model them and in the current framework (Section 2.5.4), they are considered as order abstract, which is still dense. However, we now propose a different model.

The justification for the new model is based on the following problem. If we are given an ordered clone map of fragments with preassigned order values and if we introduce a new fragment into this map, then we need to assign the new fragment an order value. If the sequence model is dense, then all the fragments that follow the new one will have to be reassigned with new order values. If the sequence model is sparse, then the other order values can remain the same. This is achievable only if the position type is based on the numbers used in the Dewey decimal system, i.e. the one used cataloging of books in a library. In this numeric system (based on rational numbers with infinite precision), one can always insert a number between any two given numbers. Therefore, one can insert a new fragment into an ordered clone map without affecting the other preassigned position values.

This approach has the advantage of minimizing changes when an ordered clone map is updated with new information. Therefore, we could model ordered clone maps as order abstract sequences, but with an intrinsically sparse position type, e.g. the Dewey decimal numbers.

CHAPTER 4. SEQUENCE SUMMMARY

4.1. Conceptual Basis

The basis for our conceptual sequence models has two major components: the abstract mathematical function and the separation of abstraction from semantics. Mathematical functions are purely characterized by a set of domain values and a set of range values. A subset of these functions has ordered domain values and, by casting these domains into positions, it formed the basis of our sequence models. In order to extend the semantics of position values, the position type was further categorized based on measurement theory. Thus, metric scales, i.e. ordinal and interval, were integrated into the position type and form a framework of models. At this point, the sequence models were still abstract. The next step was the addition of the real world semantics.

Implementation issues, such as storage efficiency or access methods, and application specific issues should be ignored in a conceptual model. Therefore, it is critical that we do not inadvertently introduce implementation or application specifications when adding the semantics. With this principle in mind, the relationships between metric, granularity, atomicity, and density were added to the framework as semantic components to be specified by the user in domain specific fashion. Other sequence models are typically concerned with implementation or application issues prior to this stage. Therefore, some of the basic interactions between sequence characteristics were never modelled and the end result was an *ad hoc* development of sequence operations.

In addition to the static description of sequence information and its semantics, we also defined a set of sequence operations. They are summarized below:

Types	Names	Comments
Semantic Operators (domain specific)	INTERPRET()	abstract to semantic
	ENCODE()	semantic to abstract
	MCONV()	convert metrics
	GCONV()	convert granularities
	INTERPOLATE()	usually the default null interpolation function
	EVENT()	generate a sequence type based on event order

Types	Names	Comments
Sequence Operators (domain independent)	LENGTH()	length of a sequence
	VALUE()	returns the content value
	BUILD()	content type to sequence type
	CUT(), SUBSEQ()	returns a subsequence
	CAT()	concatenate two sequences
	REVERSE()	reverses a sequence
	SEQ_EQUAL()	defined by using equal() from content type
	APPLY()	apply an arbitrary function to content values
	RECURSE()	recursion over elements in a sequence

4.2. HGP Sequence Models

We proposed a model for each of the sequence types found in the HGP. For ssDNA and fragments, the models were as simple as SEQ_BASE and SEQ_MARK, respectively. On the other hand, dsDNA required a more complicated model. In the context of the HGP, dsDNA is never "seen" with SS tails, because the native form of DNA in the cell nucleus is always fully complemented. Therefore, genomic database models for DNA sequences always assumed the native double stranded form. Under this assumption, dsDNA can be modelled as ssDNA and, subsequently, character strings. However, rotational symmetry of dsDNA cannot be accounted for by character strings. Consequently, programs external to the database must maintain this information and perform the necessary transformations to recapture this characteristic. More importantly, the schema for a database with the string implementation would fail to capture this aspect, which would introduce potential maintenance problems for the application developers as clients of the database.

If we made the same simplifying assumption, then our dsDNA can be modelled as SEQ_BASE. However, it would still be named DS_DNA and would still keep all the DS_DNA operators. This forces the semantics of the rotational symmetry to be associated with the dsDNA model and removed the burden of maintaining this information from the clients of the database. In software engineering terms, this improves the cohesion of information modelling.

Finally, we defined a set of biologically relevant operators:

Types	Names	Comments
ssDNA	COMPLEMENT()	complements a ssDNA
	IS_COMPLEMENT()	checks to see if two ssDNA are complementary
	MATCH_CUT()	cut ssDNA where the sequence matches
dsDNA	ROTATE()	preserving equality for dsDNA
	DS_EQUAL()	equality for DS_DNA
	ANNEAL(), DENATURE()	ssDNA ↔ dsDNA
	FILL(), TRIM()	makes blunt-end dsDNA
	MATCH_CUT()	cut dsDNA where the sequence matches
fragments	FRAG_EQUAL()	equality for FRAGMENT
	FRAG_CONTAINS()	checks to see if a fragment is contained in another
	INSERT_MARKER()	insert a marker into a fragment
	CUT_AT_MARKER()	cut a fragment at matching marker positions
	CIRCULARIZE()	circularize a linear fragment

4.3. Model Expressiveness

In addition to the sequences found in the HGP (see Section 2.3.4), the proposed framework of sequences can properly capture the sequence information from the commonly used sequence types of other domains:

Models	density	Metric	Granularity	Atomicity
text	dense	character	1	1
lists	dense	index unit	1	1
discrete events	sparse	time	variable	variable
step-wise continuous	dense	time	variable	variable
continuous	dense	time	variable	variable

Metric, Granularity, and Atomicity of temporal sequences will depend on the particular applica-

tions. Temporal models chose to implement step-wise continuous sequences as discrete event sequences because of their relational implementation. While it may be more economical to store a step-wise continuous sequence as a discrete event sequence, this choice should not be made at the conceptual level.

4.4. Open Topics

The sequence framework only addresses the abstractions of a single sequence, it does not address the issue of a collection of sequences. This is insufficient for database data modelling, because a database contains collections of sequences and data models are to reflect the gestalt nature of these collections. Therefore, the sequence framework must be integrated into a database data model (as opposed to an programming data model) in order for it to be useful for modelling databases sequence information. This integration is the goal of Part II of this work. However, there are other topics open for future discussion.

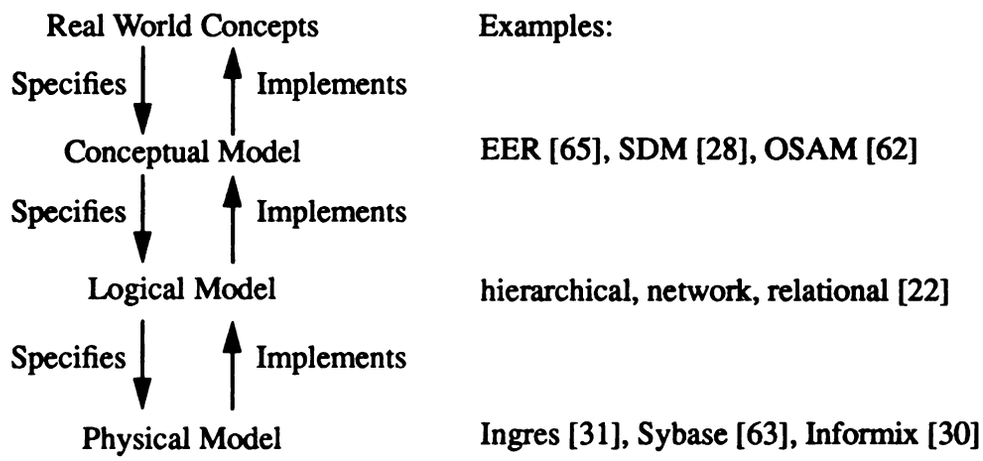
4.4.1. Position Order

Issues pertaining to partial and circular order were not addressed in the current framework. A non-standard algebraic position type will affect the arrangement of a framework based on linear, measurable position type. For example, what is the meaning of granularity of a partial order position type? In addition, operator semantics will need modification. For example, concatenate for partial order sequences could be a merge of two partial order graphs based on common position-content values.

PART II. EXTENSIBLE OBJECT DATA MODEL

CHAPTER 5. BACKGROUND

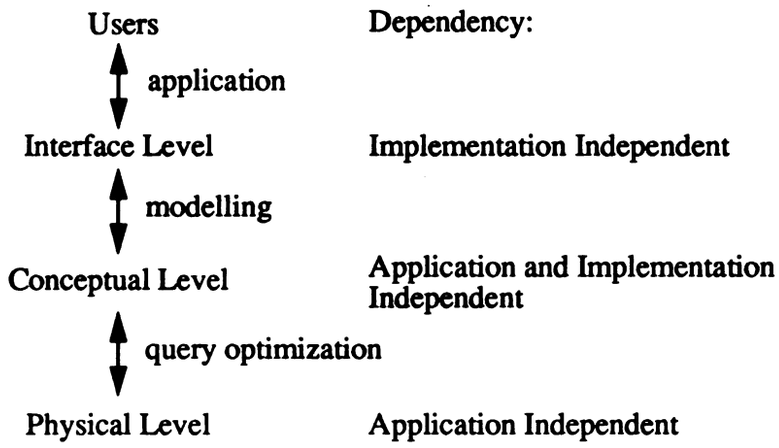
Data models serve as languages of discourse, they convey meanings from one domain to another. There are three major groups of data models: conceptual, logical, and physical [66]. A conceptual model provides a way for users to model the real world and to communicate with precision about their application domains. This is accomplished by capturing the "semantics" of the real world, such as object abstraction, integrity, and relationships. The physical model, on the other hand, contains information about the actual implementation of data structures and operations that support the conceptual model. The specific implementation constitutes the database management system (DBMS). The logical model is based on an abstraction of the physical model by removing optimization and implementation-specific features. It acts as an intermediate language between the conceptual and physical models. The following diagram demonstrate their interrelationships:



The distinction between conceptual models from logical or physical models is based on the semantic richness of the model and not on the level of implementation. Therefore, the conceptual models are also termed semantic models, as an indication of their semantic richness. It is desirable to go directly from a rich conceptual model to a physical implementation, which is the reason that certain logical model (especially the relational model) are being extended [8,16,43].

In addition to the organization of models based on semantic richness, there is an alternative organization that is based on dependency. Under this scheme, the models are grouped into three groups: interface, conceptual, and physical. The interface group deals with how the users interact with the system. In database applications, this translates into query facilities and user interface issues. In this group, it is very application dependent but implementation independent, therefore, it is sometimes called the "external level". The conceptual group is similar to the conceptual models in the previous scheme. It deals with data models and formal query languages in an application and

implementation independent fashion. Lastly, the physical group handles implementation specific information, such as data structures and access methods. However, it is application independent, therefore, it is sometimes called the "internal level". Their relationships are diagrammed as follows:



Our goal in Part II is to develop a conceptual data model, with sequence constructs, that corresponds to both organizations, i.e. it is both semantically rich and application and implementation independent.

There are two approaches to the integration of the abstract sequence models into a conceptual data model. The first approach is to choose an existing data model and graft the sequence model as an extension. For example, we can extend the relational model with sequence constructs in the style of Non First Normal Form (NFNF) [1]. A better alternative is to use a semantic model for sequence extension. Since semantic models already provide rich conceptual power, the only task is to add a sequence construct. However, such an addition has to be semantically consistent with the base model and remains orthogonal to other constructs. The verification of consistency and orthogonality would be a limited exercise, but the creation of a new construct may introduce conceptual incongruities in the model. In addition, some semantic models already have a limited version of sequences, e.g. "lists" and "arrays" [28], does one extend or replace existing constructs?

The second approach is to develop a new model from scratch with sequences as an integral component. Here, we run the risk of redeveloping constructs found in other models and resulting in redundant work. On the other hand, it allows us to build a new self-consistent model without the possibility of conflicts with the presumed intent of existing constructs. In addition, it gives us the opportunity to resolve open questions in conceptual data modelling. For this work, we choose the latter approach because we are free to address open modelling issues. The remainder of this chapter reviews the issues of conceptual data modelling. In Chapter 6, a new data model is proposed. Chapters 7 and 8 describe the model in detail. Finally, Chapter 9 summarizes the results of Part II.

5.1. Framework for Data Models

Currently, conceptual data models are based on two approaches: semantic and object-oriented. Many semantic data models have been proposed and they were reviewed in [29,51]. Briefly, semantic data models borrow constructs from knowledge representation and semantic modelling in the Artificial Intelligence field. Their objectives are to capture real world concepts in an application independent fashion. Object-oriented models are logical models based on Object-Oriented DBMS's (OODBMS) [38], but due to their semantic richness, these can also serve as conceptual models. They primarily focus on features of Object-Oriented programming languages, i.e. object identity, classes, and inheritance. For this research, we are proposing yet another conceptual model, independent of these two approaches.

Unfortunately, debating merits of data models is like debating merits of programming languages: arguments range from the arcane to the religious. Therefore, we first propose a framework where conceptual data models can be evaluated and compared. Clearly, the assumptions behind any framework are also debatable, but once accepted, the models can be evaluated in a systematic fashion. Our framework is built on two central issues of every conceptual model: what a model is and how the model is used. This framework is based on the one presented in [46].

5.1.1. Semantic Mapping

In the core of all models is the semantic mapping. The result of a modelling process is the schema. For conceptual modelling, a schema is "an implementation of the real world". In addition, it can be viewed as a specification or as a template for translation into a logical or physical schema. This is diagrammed in the previous figures. Consequently, the conceptual model (as a language to produce a schema) serves two goals: one is semantic richness to capture real world meaning, another is formalism to specify implementation. Together, these goals determine the power of the semantic mapping. The main aspects of this mapping are:

1. Semantic Richness

   A direct representation of a real world concept is better than an indirect one. A rich model will have more constructs that map directly to real world concepts. However, a balance must be struck between richness and simplicity.

2. Representation Uniqueness

   For a given real world concept, it is less confusing if there is only one unique representation in the model. If there are many constructs for the same concept, then additional rules are needed to decide which construct is to be used.

3. Implementation Independence

   A conceptual model should be independent of its implementation, i.e. the underlying

model. However, the model is also a bridge between the real world information and the actual database (or logical model), therefore, the concepts it captures must be implementable on any sufficiently capable database (model), not just one specific database.

4. Mathematical Completeness

   In a mathematical framework, a data model can be considered as a logical theory with a denotation for each construct and a set of definitions and axioms. Completeness is a property that states "model" transformations correspond to "real world" transformations [23]. It stipulates that all constructs in the data model are well-defined and all operations are closed.

In addition, there are certain features of semantic mapping that a model should address:

5. Model Extensibility

   A model should be extensible to capture application specific concepts in a general fashion. This is the ability to add new constructs to the model in a systematic fashion when new concepts appear in an application domain. This is equivalent to extending a language, not just expressing statements from a given language.

6. Domain Invariance

   A conceptual schema should only describe the invariant properties of the application domain. If this is not observed, then after each operation that changes the database state, the schema would also be changed. This would force operations to be of *ad hoc* nature because one cannot ensure a stable schema prior to the actual operation. Therefore, a conceptual model should only model operation-invariant properties.

5.1.2. Usability

The usability of a model determine how it is used and, in turn, its acceptance. Although semantic richness is important, human factors dominate in this determination. Two factors of usability are mental comprehensibility and user interface. Comprehensibility is, in turn, based on a mixture of factors and it is also a subjective criterion: each application domain and each user will weigh each factor differently. The main components to comprehensibility is richness and uniqueness from the semantic mapping part plus the following:

7. Model Simplicity

   A simple model is easier to learn than a complex model. A model should have a small number of simple constructs and a small number of rules governing the interaction between constructs. The interaction rules are usually the most complex part of a model, because there can be $O(n^2 \times m)$ cases where n is the number of base constructs and m is the number of association constructs.

8. Schema Modularity

A conceptual model should allow the user to divide the "real world" into smaller, manageable chunks of related information. There are two mechanisms: encapsulation and references. Although encapsulation is based on semantic mapping, i.e. how groupings of real world concepts are mapped to model constructs, its execution as a usability factor is more important because it determines how we use the model to group concepts. Referencing is the ability to separate definition from reference. It is a feature that helps the users to handle the complexity of real world information by breaking apart a large schema into smaller pieces.

User interface issues deal with the mechanical process of modelling by a user:

9. Graphical Form

A graphical form is often more visually comprehensible than a textual form. A model should support a graphical representation of its constructs. Presumably, each construct in the model would have an easily identifiable graphical representation.

10. User Support

A set of tools for schema generation, editing, and viewing would be useful. In addition, if the conceptual schema is independent of the underlying system or model, then a translation tool to different systems and models would be useful.

This is a large and comprehensible framework. Therefore, our proposed model will only emphasize some of the features. Of those addressed, some will overlap with other conceptual models.

5.2. Common Characteristics

We will start with the features that most semantic models fulfill.

5.2.1. Semantic Richness

The semantic "richness" of a model is a measure of how well it can model real world information. All conceptual models have a set of basic constructs. The constructs have been reviewed in [29,51]. Briefly, they are:

1. Types and Instances

These are also termed Classes and Entities, Objects and Instances, Classes and Objects. The exact terminology isn't important as long as they are used consistently within a model. A model instance corresponds to a real-world object or abstraction. A collection of instances with common information and behavior (with respect to other collections) is called a Type. For example, a Person Type would include information such as ID, Name, and Age, and "contains" zero or more instances of Dependent Type. You and I are instances of Persons, each with different values for ID, Name, Age, and different dependents.

2. Aggregation and Grouping

Aggregation is also termed Tuple or Cross-Product. Grouping is termed Sets. The terminology is reversed in some data models. These concepts define two methods of collecting instances together to form a new Type. Tuple construction builds a new Type from a fixed number of instances, each of a specific Type. When given an instance of the Tuple Type, specific member instances are usually accessed by their Type name. For example, if x is an instance of Department Type, x.Manager would retrieve the manager instance. Set construction builds a new Type from an indeterminate number of instances, usually from one specified Type. The instances within a set cannot be individually addressed, instead, they are accessed through selection or iteration on the set.

3. Specialization and Generalization

These are aspects of IS-A hierarchies. Sometimes, they are also used to extend the concepts of Subsets and Union. Specialization constructs a new Type by adding information or behavior to previously defined Types. Generalization is the reverse of specialization, it constructs a new Type from other Types, but only keeps the common information and behavior. For example, a Student is a specialization of Person and a Publication is a generalization of Article and Book.

4. Relationships and Roles

They define how one Type relates to another. Relationships can be represented as a mathematical mapping between two sets of instances, therefore, we find concepts of 1-to-1, 1-to-many, many-to-many, unto, and into expressed by relationships. Relationships in a data model are named after the real world relationship. For example, some of the relationships between two Person Types are "Par-

ent-of", "Managed-by", and "Married-to". Roles are aliases the Type assumes at the end-point of a relationship. For example, a Person assumes the roles of Parent and Child at the ends of a "Parent-of" relationship.

5. Class Construction and Meta-Typing

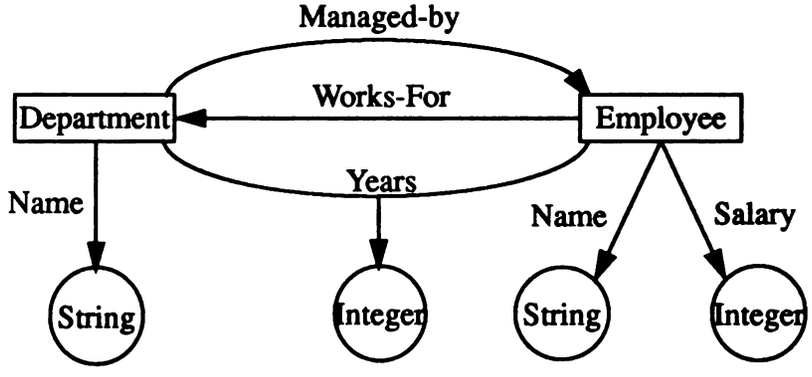
Some models permit Class construction, where arbitrary instances are explicitly grouped into new Types. Other models support Meta-Typing, where arbitrary Types can be explicitly grouped into new (meta-)Types. The two mechanisms allow Type formation based on explicit instances. These are supported by data models which are more instance-oriented than type-oriented.

The more real world concepts for which there is a direct representation, the richer the model. Given this set of basic constructs, there are many ways to design a consistent model, as demonstrated by the current set of conceptual models. Each has its own extensions, but the problem is not semantic expressiveness because all of them cover the basics. In fact, some users would be satisfied with the richness from a subset of these five constructs, because for their particular application domains, a subset is sufficient.

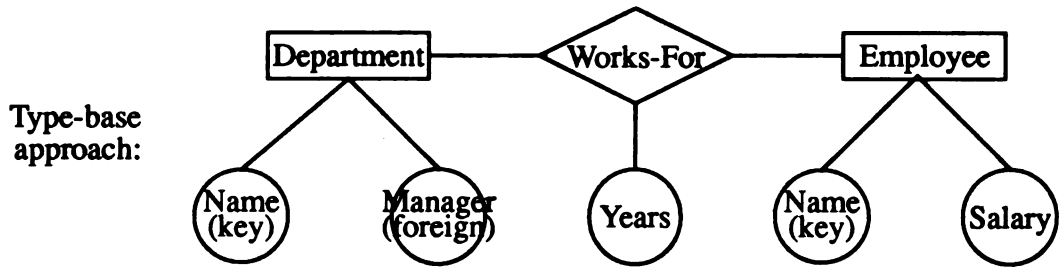
5.2.2. Representation Uniqueness

For any particular real world concept, most models provide only one unique representation. However, this fails when relationships are encountered. There are two basic approaches [29]: one approach is function-based where all relationships are designated by directed arcs and the other approach is type-based where new types represent relationships. The function-based approach lacks the ability to refer to a relationship and its characteristics together. The type-based approach allows ambiguous mapping of relationships. These are demonstrated in the following diagrammatic example:

Functional approach:



In the functional approach, there is no mechanism to reference Years based on Works-For or to use Works-For as a basis to another relationship because the functions have only one destination.



In the type-based approach, one cannot prevent modelling Manager as an attribute of Department. There is no resolution for this problem in "graph"-based data models, because it is a fundamental semantic gap based on our ability to abstract a relationship as entity and vise versa. We will choose the type-based approach because it allows us to scale-up relationship abstractions. Therefore, we will consider the problem of Department/Manager as an error in modelling semantics, i.e. the correct schema should use an explicit "Managed-by" relationship between Department and Manager.

5.2.3. Implementation Independence

Most conceptual data models can be implemented on top of different logical or physical models. However, the ease and efficiency is dependent on the particular constructs. For example, Entity-Relationship (ER) constructs have direct implementation into relational schema [18]. However, the integrity constraints of cardinality and foreign keys specified by an ER schema require additional processing in the relational model via triggers or supporting queries to maintain integrity. Specialized constructs, e.g. summary of Object-Oriented Semantic Association Model (OSAM) [62] and collection of IFO [2], are difficult to shoehorn into the relational form. On the other hand, set constructs from the IFO model has a simple and direct implementation on a hierarchical DBMS. Difficult implementations indicates that the underlying model or system truly lacks certain features that would make them conceptual. After all, if the relational model captures a rich set of semantics, we won't have the proliferation of relational extensions and semantic data models. There is also a principle behind the drive to capture real world semantics: if the model is rich enough to capture the domain semantics, then it is sufficient.

Object Oriented data models are logical models of OODBMS, therefore, they lack independence from physical systems. On the other hand, they are rich in semantic constructs. Recent progress has been made to provide mathematical completeness to OO models, but has not pursued logical-level independence [69]. However, OO models are rich in mechanisms to implement conceptual abstractions, albeit only on top of OODBMS.

5.2.4. Mathematical Completeness

When discussed in the context of mathematical logic, completeness depends on a balance between the number of distinct constructs and the number of axioms governing their behavior. In a mathematical theory where constructs out-number axioms, completeness is unlikely to be achieved. On the other hand, if axioms out-number constructs, contradictions may be present. A theory that is complete implies all statements are provably correct or incorrect, i.e. no statement exists that cannot be proved. For example, Pressburger arithmetic is complete, but with the addition of "multiplication" to form Number Theory, it becomes incomplete [23]. From another perspective, completeness is a measure of "semantic complexity". The more complex a theory, the more difficult for it to be complete.

With the advent of relational model, mathematical formalism has been pursued by all subsequent models. Given the fact that completeness is related to "semantic complexity", it is unlikely to be achieved by semantic data models. On the other hand, certain amount of formalisms can be obtained, e.g. all terms are well-defined and all operations are closed and computable. For this work, we will not investigate completeness in our proposed model. This is reserved for future research.

5.2.5. User Support

Support is often determined by the maturity of available tools for the model. These tools provide the ability to view, edit, print, and store model schemata. They also provide semantic translators to other data models for implementation or graphical translators to document processors. These tools enhance the usability of a model, but the lack of such tools should not detract from its usability as a conceptual model. The development of these tools is in the domain of user interface research and not in data modelling research. Furthermore, the development of tools is expensive, therefore, new or uncommon data models often lack the wide-spread acceptance for such investment. The only support tool for our proposed model is the query processor described in Part III. Other support tools are reserved for future research.

5.3. Unique Characteristics

There are some aspects of conceptual data modelling that have been improved in our proposed model as compared to others. These will be explored in more detail:

5.3.1. Domain Invariance

The concept of invariance is very important in all modelling endeavors. It allows the model constructs to only reflect the domain information that is "generally" true. In most semantic data models, there does not exist a clear definition of invariance, i.e. what domain information is considered to be "generally" true. Most models typically start with some statements concerning the definition of "objects" and their representations, relationships, or implementations. In addition, most models focus on the abstraction of object characteristics, i.e. type definition, and not on the specific instances of objects. This is fortuitous because type definition is a form of invariance principle. This approach works for many applications, because during the lifetime of a database schema, the type definitions do not change, only the instances of the types. But in certain domains, e.g. statistical databases, it is the instances that do not change, only the types or categorizations of them change. Clearly, in these domains, the standard type definition may not properly describe the domain information that is "generally" true and a fundamental semantic gap forms.

In our model, we define the invariance at the level of collection of instances. Domain concepts related to all the instances of the same collections are then mapped onto the schema. Concepts which are specific to individual instances are not modelled. In addition, a collection-level construct based on "context dependency" will define the relationship between instances. This will maintain a constant level of abstraction and invariance for the model.

5.3.2. Model Extensibility

Most semantic data models do not provide the mechanism to add new constructs in a systematic fashion. After all, the goal of a semantic data model is to be semantically rich, therefore, no new constructs should be needed. This is inadequate for specialized domains, since in the real world, there are many semantic concepts that cannot be mapped onto a strict set of constructs and rules. Therefore, our proposed model is constructed so that extensibility is a core feature. This reflects the "open-ended" nature of all mathematical models, i.e. new axioms and definitions can always be added to an existing model to create a new model.

Under what situation would there be a need for new constructs? Typically, it is the emergence of new abstractions or encapsulations. For example, a Set is an encapsulation of a collection of instances and it is usually included in a semantic model. However, a sequence or list is an encapsulation of a collection of instances that is different from sets, i.e. the information that goes into the construction of a sequence is different from a set. If an existing semantic data model is used, one would have to shoehorn tuple and set constructions and integrity constraints to emulate a sequence.

On the other hand, extensibility would allow the introduction of a new construct to represent a sequence, without a complex mapping based on existing constructs.

The ability to extend an existing model depends on the uniformity and orthogonality of the underlying model. In most models, no attempt has been made to fully categorize its modelling constructs. Therefore, there is no *a priori* mechanism to determine if all the possible semantics have been captured. For example, the addition of cardinality constraints to ER relationships is considered *ad hoc*. There is no precedence of why it should be there, except for the fact that, semantically, it is meaningful. In order to make cardinality less *ad hoc*, one has to have a problem statement that is consistent (with cardinality) and, potentially, a solution that is complete (cardinality as a part of it). The use of "consistency" and "completeness" stems from their meaning in relating proof-theoretic and model-theoretic mathematical models.

In our proposed model, we have focused on three orthogonal features: structural constructors, identity vs. property, and context dependency. The core model provides a rudimentary set of structural constructors and context constraints. Therefore, extensions to the core model could include new structural constructors and new context constraints.

5.3.3. Model Simplicity

Most semantic models have at least five constructs, usually two to three node types and two to three arc types. Although the number of basic constructs are few (less than seven), the interaction between them (node × arc × node) is much more complex. This can be quantified by a table of node-to-node interactions. For example, in the basic ER model [18], there are four node types: attributes, entities, weak entities, and relationships; and two arc types: between attributes and other nodes and between relationships and entities. The interaction table for this ER model is given as:

N-to-N	Entity	W-Entity	Attribute	Relationship
Entity	-	attr	attr	rel*
W-Entity	attr*	attr	attr	-
Attribute	attr*	attr*	-	attr*
Relationship	rel	-	attr	-

where: "attr" link entities to attributes,
"rel" link entities to relationships,
"-" indicates an invalid link, and
'*' indicates dependence on directionality.

The table format gives us a handle on the complexity of a data model. In addition, it is a form of algebraic semantics, providing a "closure" of derivable model concepts, i.e. each (node × arc ×

node) entry has a defined meaning. For example, between Entity and Weak-Entity, the "attribute" link defines how the Weak-Entity relates to the Entity, and the meaning of this link is constant for all occurrences of Entity/Weak-Entity in a schema. This entry forms one distinct interaction and it is the number of distinct interactions that determines the complexity of the model.

In more complex models, there is directionality to the arcs. If we add implicit directionality to the arcs, as defined in the standard ER model, the entries with '*' become invalid and the table becomes nonsymmetric. Furthermore, there can be multiple arc types in each node × node cell when we include IS-A for the Extended ER (EER) model [26]. The resultant table should be a 3-D table, with arc type being the third dimension.

The IFO model [2] has five node types with 4 arc types and has this interaction table:

N-to-N	Printable	Abstract	Free	Collection	Product
Printable	-	-	-	-	-
Abstract	F	F	F/G	F	F
Free	F	F/S	F/S/G	F/S	F/S
Collection	C	C	G/C	C	C
Product	C	C	G/C	C	C

where F for functional mapping,
S for specialization,
G for generalization,
C for composition,
"-" indicates an invalid link.

In the cell Abstract × Free, the entry Abstract × function × Free is semantically different from Abstract × generalization × Free. However, the common occurrences of function and composition links imply generalized features, indicating a similar semantic can be applied to all ×-function-× and ×-composition-× entries. Localized occurrences, such as specialization and generalization links, imply more specific features. These may indicate cell-by-cell differences. The directionality of arcs is noticeable from the asymmetry of the table.

The simplicity (or complexity) of a model is not determined by the number of basic constructs, but by the way the interaction table is filled. In many models, the node-to-node interaction is often given by a series of rules or formal definitions using mathematical terminology. This makes direct comparison of complexity difficult. In addition, there are also rules and definitions governing the interaction between non-adjacent nodes. For example, cardinality and participation constraints in the ER model govern the association between entities across a relationship. Another example is IS-

A paths for specializations and generalizations which forbids an entity to occur twice along an IS-A path.

After understanding how easy it is for a model to achieve complexity, what would achieve simplicity? We again focus on the features of uniformity and orthogonality. IFO functions and compositions links are uniform, but are insufficiently uniform, i.e. certain node combinations are not permitted. ER cardinality and participation constraints are orthogonal to nodes and links, but they can't be applied to attribute links. If the constructs and their interactions are more uniform and orthogonal, then the model becomes simpler to comprehend and to use.

These two features are not separable. Orthogonality enhance uniformity and vise versa. Therefore, applying this doctrine, one would attempt to create as many orthogonal or uniform concepts for a data model as possible. However, if the final result becomes too "atomic", the semantic nature of the model is lost and the end result will lack richness. For example, the relational model is where concepts are orthogonal and uniform, but semantically impoverished. A balance must be found between the semantic richness of model constructs and the pursuit of orthogonality and uniformity.

We approach this problem in three steps. First step is a structural model based on information content. A database holds information, therefore, the logical mechanism for abstraction is the information content. Second step is explicit definitions of identity and property. Every real world object has an identity and some associated property. This separation is crucial to the integrity of the domain information and should be retained in the model. Finally, context information is captured. Every real world object also exists in some context and this affects how it relates to other objects. Therefore, our model would systematically categorize and abstract context information. Each of these characteristics will remain orthogonal, so that associations between them can remain uniform.

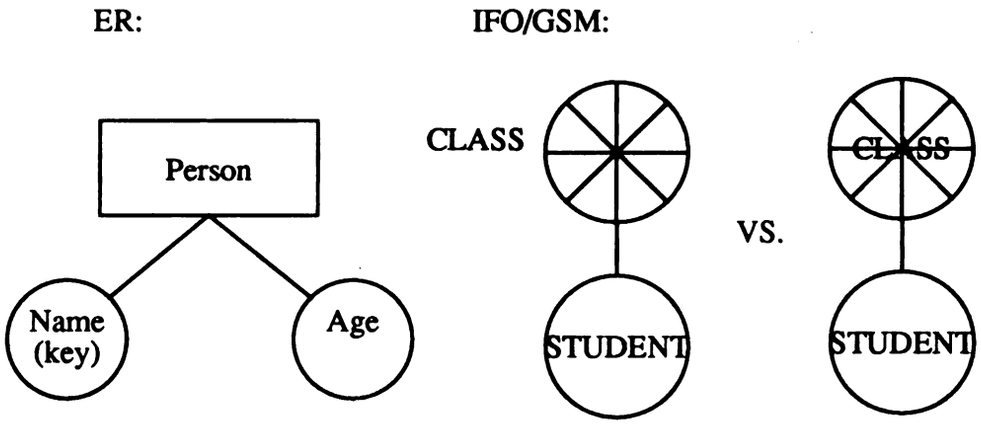
5.3.4. Graphical Form

Most semantic models have a graphical representation. However, the choice of symbols and their interconnections will affect the usability of the model. This is an issue separate from the semantics of the model. There are three major problems with graphical design:

1. Labelling of Objects

This is a problem of where to place the label associated with an object. For example, the ER model allows the labelling of most objects within the graphical form, but the IFO model almost forces

external labelling because of internal patterns:



On a large schema, such external labelling creates pairing conflicts, when one cannot readily determine which object is associated with which label.

2. Modularity

Modularity determines whether references can have a graphical form. Certain models, e.g. the ER and the IFO models, do not have a graphical distinction between object definition and reference. Therefore, a schema would only have one node per abstraction and references are based on arcs converging on that node. For a large schema, this would create a graph that has tangled topology. In practice, references and definitions are separated, but this requires an additional notation on a graphical object to determine its actual nature.

3. Directionality

Another problem is the lack of directionality. For example, in both ER and IFO models, the arcs can fan out in any directions. This reduces the comprehensibility of a schema diagram because one tend to lose track of visited nodes and be overwhelmed by the snaking of arcs.

These are problems in graphical design, not in data modelling. There are many factors governing graphical design, but the focus of this research is not about these factors. However, data model designers should be aware of these issues because they affect the comprehensibility of the graphical diagrams. We will apply the three described principles to the graphical representation of our proposed model. First, all symbols are uniform and internally labelled. This reduces the label-to-symbol pairing problem. Furthermore, a uniform symbol is also a hallmark of semantic uniformity in the model. Second, a separation of definition from reference in the graphical form. The "spaghetti" look in most semantic models is due to the lack of this separation. This simple strategy will permit a schema to be separated into manageable chunks. Finally, an implicit directionality of information flow is provided, which will give a consistent direction of reading and drawing a schema to prevent the maze situation.

5.4. Model Characteristics Summary

Our proposed model will address features common to other conceptual models, but in addition, it will also address features not present in the others. The main features are:

1. Semantic Richness. The constructs supported by our proposed model provide type definition, aggregation and grouping, specialization and generalizations, and relationships and roles.
2. Model Extensibility. The proposed model is actually a framework of models. Therefore, extensions to the core model can be seamlessly integrated. The abstract sequences, as defined in Part I, will be one such extension. Other extensions are metatyping, classification, and generalized constraints, which are reserved for future research.
3. Model Simplicity. The model achieves simplicity by orthogonality and uniformity. We have divided the conceptual modelling into three categories: structural, identification, and contexts. Variants of each category can be applied to any variants of other categories without loss of generality. This also permits extensions organized along these separate categories.
4. Graphical Form. The graphical representation of this model has a natural organization. It trades off some artistic freedom for comprehensibility and reinforcement of modular construction.

The minor features of the proposed model are:

5. Domain Invariance. We have decided to explicitly relate domain invariant concepts to our model constructs. This is a choice which has advantages in understanding of how domain concepts map to and from our model.
6. Representation Uniqueness. We have chosen the type-based approach, which follows our paradigm of extensibility by upward scaling of domain abstractions.
7. Implementation Independence. We will define our basic constructs with abstractions which are implementation independent.

Some features are not discussed in this work and are reserved for future research: Completeness and Support.

CHAPTER 6. EXTENSIBLE OBJECT DATA MODEL

Our approach to data modelling starts with the concept of information content, which is further divided into structural and location components. The structural component determines what information is stored. This is defined by an abstraction in our model called the Object-Type. The location component determines where the information is stored and is defined by an abstraction called the Instance. The coupling between Object-Type and Instance is mediated by the abstraction called Context. A schema with Object-Type and Context definitions can unambiguously determine the Instances. The remainder of this chapter will describe the basic definitions of the Extensible Object Model (EOM). Chapter 7 will discuss Object-Type creation by using constructors. Chapter 8 will discuss context definitions.

6.1. Definition of Object-Types

(1) A database is a depository of information.

It is divided into individual blocks of information. Each block of information holds some information content. This stored content is an abstraction or an array of binary digits, i.e. bits, when considered from an information theoretic point of view. Object-Types define the sizes of these blocks and how they relate to other blocks, defined by other Object-Types.

(2) Each Object-Type defines two mapping functions:

(a) the interpretation of bit patterns (symbolic values) in a storage block to the semantic values associated with the real world concept of which the Object-Type represents

(b) the encoding of semantic values of a real world concept into a pattern of bits (symbolic value).

These two functions provide the link between the real world and the symbolic universe and is identical to the interpretation and encoding transforms discussed in Section 2.5.3.

(3) Each Object-Type is named.

The name corresponds to a concept in the real world. For example, numbers, such as 1, 2, and 3, can be encoded in binary as "1", "10", and "11". Conversely, binary digits can be interpreted as numbers. However, letters, such as "A", "B", and "C", can also be encoded in binary as "1000001", "1000010", and "1000011", and in turn, be interpreted as ASCII characters. Thus, the concepts of "Integer" and "ASCII character" each have their own mapping functions.

The same symbolic value can be interpreted, by different Object-Types, into different semantic values and vice versa. Therefore, the name of an Object-Type is very important to the maintenance of the integrity between the real world and the symbolic universe. From

now on, unless we specifically state a value is semantic, it is assumed to be symbolic, i.e. a value is its bit pattern.

(4) Each Object-Type defines a set (collection) of potential values.

Since a type defines the mapping between semantics and symbolics, we can map all the possible semantic values, thereby obtaining all the possible symbolic values (bit patterns). Therefore, an Object-Type defines a set of potential or permissible symbolic values.

(5) Each Object-Type has a set of symbolic operators or functions.

The operators correspond to the transforms of the real world that affect instances of an Object-Type. A semantic value will change under real world processes, and because each semantic value has a symbolic value, we can construct a symbolic operator which performs the equivalent transform in the symbolic domain. For example, the semantic concept of "addition" has a symbolic equivalent in "2's-complement addition operation", which operates on ordered binary digits, without regard of the semantic interpretation of these digits.

6.2. Definition of Instances

(1) An Instance of an Object-Type is a particular value.

The collection of instances forms the actual data in the database, i.e., they have "physical reality" and correspond to actual blocks of bits. Although the Object-Type can define a very large set of potential values, we are only interested in the actual values in the database. Therefore, we can use an Object-Type in a schema to represent the collection of actual instances found in the database, not the potential instances. For now, we will use "collection" to represent the "relaxed" form of a set, i.e. a bag, since there can be a multiplicity of instances having the same value, within the database, represented by one Object-Type.

With this specification, we deviate from the counting principle of constructing Object-Types. For example, the traditional generation of a cross product term is considered to be "larger" in collection size and number of instances. However, in a database, not all possible combinations are present. In fact, the cardinality of the cross product instances may actually be less than the cardinality of the sources. On the other hand, the number of bits needed to store the cross product information is always at least the sum of the bits of the sources, i.e. "larger" than the sources.

The definition of instances simply states the existence of instances, it does not dictate the physical location of the information contained by the instances, e.g. volatile memory, disk sector, cluster, or file. Although the "source" component of the information content defines <u>where</u> the information is located, but it is an abstract form of location, i.e. an Instance is a conceptual location.

(2) Identity of instance is value-based.

If two instances, defined by an Object-Type X, have equal values, then the instances are identical. In other words, if using all the information available as Object-Type X and we cannot distinguish two instances apart, then they are the same instance. If this rule is absolute, then collections (bags) are not permitted. However, as we shall see later, this value-based equivalence is amended with their context information for greater expressiveness. The corrected rule is "when given two instances, from the same context, defined by an Object-Type X,...".

(3) An instance value has two components: ID and Property.

The ID component must always be present and interpretable, thus it has both semantic and symbolic values. Equality based on this value contributes to the "identity" of the instance. The Property component contains all the information other than the ID component, thus it may or may not be present. These components can be defined as operators of instances, e.g. ID() would return the value that form the ID portion of a particular given instance.

Technically, only instances have these components because only instances have an actual value that can be partitioned into ID and Property values, Object-Types do not have actual values and cannot provide these component values. However, the derivation of component values within an instance is identical for all instances of the same Object-Type. Thus we can represent these components at the Object-Type level, i.e. an Object-Type "has" ID and Property components.

(4) Fundamental relationship of ID and Property: ID determines Property.

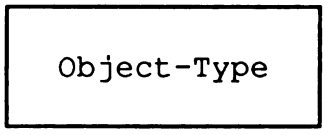
Given two instances, x_1 and x_2: If $ID(x_1) = ID(x_2)$, then $Property(x_1) = Property(x_2)$.

ID(instance) returns the instance ID value
Property(instance) returns the instance property value
= is value-equality or bit-wise equality.

The advantage of using explicit ID specification is better control over the scope and extent of object instances. In OO data models, a system-wide implicit (surrogate) object ID is assumed to be present and unique. Instance identity is then based on the equality of this surrogate ID value. In the relational models, identity is based on the key-attribute equality, therefore it does not have a system-wide uniqueness. In conceptual modelling, there are situations where the objects should have system-wide uniqueness, but there are also situations where they should not. In this model, we enforce an explicit ID specification, which allows both situations to be modelled correctly.

6.3. Graphical Notation

We will also use a graphical form for Object-Types. When drawn in a schema, Object-Types are shown as:



The rationale for a rectangular shape is uniformity. Using the principle of encapsulation, instances of an Object-Type can be referenced uniformly without knowledge of its underlying definition, thus all Object-Types have the same shape on the top. On the other hand, once we open up an instance, then its internal structure, defined by the Object-Type, becomes visible. If the structure is defined elsewhere, we find an alias or an implementation and if the structure is constructed, we find the type of construction and its sources (see below). This forms a very simple but effective graphical notation for the Extensible Object Model schema.

6.4. Object-Types in a Schema

For the remainder of the discussion, upper case variables, such as A, B, and X, represent Object-Types and lower case variables, such as a, b, and x, represent object instances. Lower case variables can be extended with subscripts to label individual instances, for example: x_1 and x_2. However, this labelling does not necessarily imply the instances are different. In fact, a single instance may be given different labels, and only later, we may learn that they refer to the same instance. We will use "=" to denote value equality between two labelled instances, and "≡" if they are identical, i.e. the same instance.

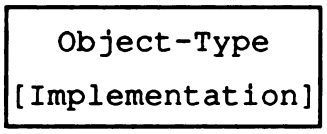
We resolve multiple occurrences of an Object-Type by separating Object-Type definitions from declarations (or references). A schema can contain multiple declarations of an Object-Type, these represent where the Object-Type information is used. However, a schema has only one definition per Object-Type, which defines the information structure of the Object-Type. All Object-Types with the same name in a schema will denote the same Object-Type information structure. This separation enhances the model's ability to construct a schema in readable portions and promote proper abstractions and encapsulations of Object-Types.

An Object-Type is defined under these three situations: implementation, alias, and construction.

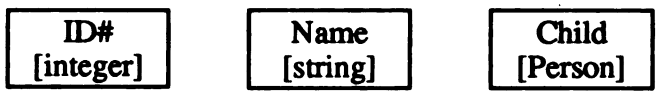
A. Implementation

Abstract Objects are not very meaningful until we implement it with a well-known semantic. For example, Object-Type Name can be implemented as a character string and Object-Type ID# as an

integer. Implemented Object-Types are shown as:



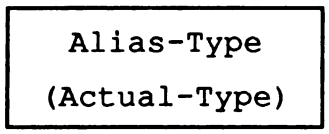
Examples Usage:



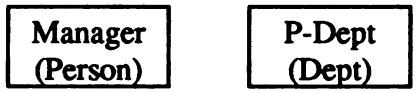
The implementations of ID# and Name are shown inside the square brackets. A natural extension to well-known implemented types is any other Object-Types defined in the schema. For example, the Object-Type Child is an implementation of the Object-Type Person, which means a Child instance is structurally the same as a Person instance. If an implemented type is supported by the underlying DBMS, then it is considered as "primitive". Examples of primitive types are integer, string, date, and time.

B. Alias

Object-Types can be aliases, i.e. renamed from another Object Type:



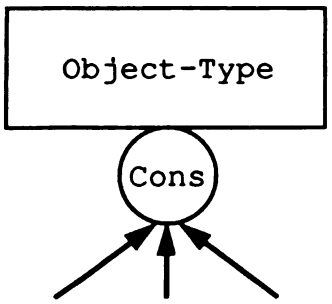
Examples Usage:



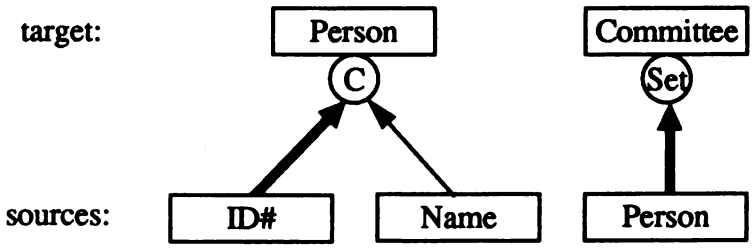
This allows more natural and semantically meaningful names. In this example, Manager is an alias for an Object-Type Person, which is shown in parenthesis. This means a Manager instance is an instance of Person, and if Person is defined elsewhere, then Manager is also defined. In addition to the "structural" definition, alias also defines the collection where the instances are to be obtained, i.e. the collection represented by the actual Object-Type.

C. Construction

The last type of definition is by construction. Here the Object-Type is on top of a small labelled circle denoting the type of construction, with one or more incoming arrows.



Examples Usage:



The "C" is an abbreviation for "Composition".

The Composition, Set, and three more will be discussed in Chapter 7. The direction of the arrow

denotes "information flow", i.e. the information content of the source contributes to the information content of the target. Therefore, a target Object-Type is defined if its source Object-Types are defined. If we combine the schema from the implementation example above, then Person would be defined. Furthermore, Committee would also be defined. Otherwise, we have partial definition and the current schema is considered incomplete.

6.5. Other Characteristics

The process of defining an Object-Type by implementation, alias, or constructors, fixes the semantic functions of interpretation and encoding. For the case of implementation and alias, ID and Property components are defined by the referenced Object-Type. If an implementation type is "primitive", then ID is set to the implementation value and Property is set to null. For constructors, we will define ID and Property components individually.

The EOM forces the user to define a set of operators when an Object-Type is defined. By default, a minimal, generic set of operators is provided with each constructor, however, the user is free to define additional operators. Since we are dealing with information abstraction, the results of these operations can be viewed as Property values of the instance. As long as the results behave as Property, it does not matter <u>how</u> the value is provided. Therefore, they can be explicitly stated in the EOM schema by defining Property components that are "implemented" as procedures or queries.

CHAPTER 7. CONSTRUCTORS

Constructors determine the structure of the information content in an Object-type, i.e. what information is included in an Object-Type and how it is organized. Under the symbolic viewpoint, a constructor determines how to partition the block of bits into smaller blocks based on the defined source Object-Types. We now describe five basic constructors to build complex Object-Types:

(1) Composition

(2) Set

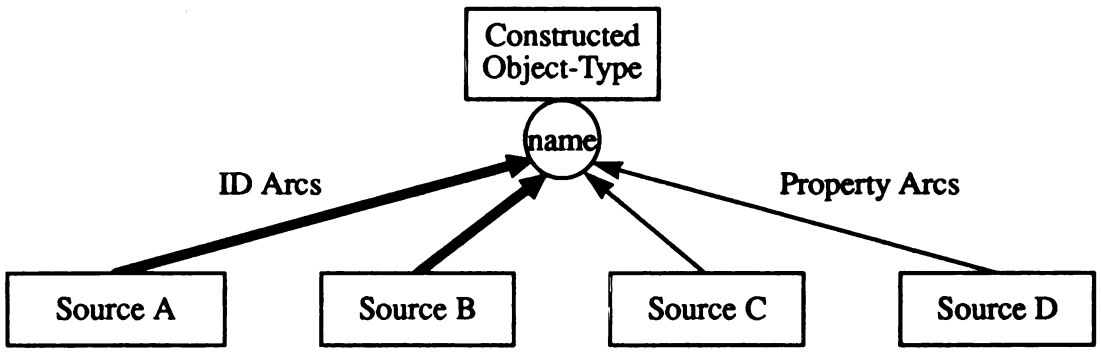
(3) Sequence

(4) Inheritance

(5) Union

The constructors are similar to the generics of Object-Oriented programming. They are independent of the source Object-Types, yet all Object-Types, derived from the same constructor, organize information in a similar manner. These constructors have been explored in previous semantic data models. The most traditional constructor is Composition, which is identical to cross product and tuple construction. Set and Sequence constructors are new to relational and ER modelling, but well developed in Object-Oriented models and NFNF relational extensions. Inheritance and Union have been explored in many semantic models. However, we are going to bring them together in a consistent manner.

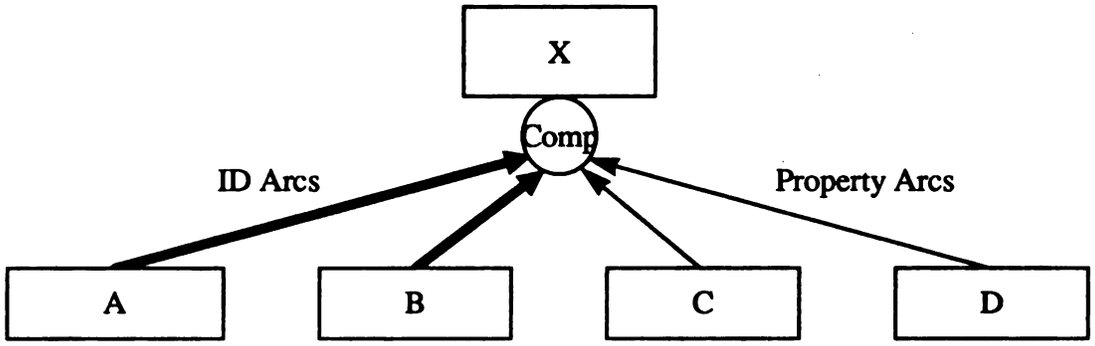
The general form of a constructor is diagrammed as follows:



The name of the constructor is defined in the circle under the constructed Object-Type. It receives one or more incoming ID arcs (thick) and zero or more incoming Property arcs (thin). Each of the arcs originates from another Object-Type. The constructed Object-Type is named the "target" and the sources for the constructor is named the "component" or "source". We will use the "." (dot) notation to reference a source of a construction, e.g. "X.A", and "[...]" (vector or tuple) notation to reference a set of sources, e.g. "[A , B]".

7.1. Composition

The paradigm of abstract information storage allows us to construct an Object-Type from a composition of other Object-Types. This is similar to a C "struct" or Pascal "record" implemented as a contiguous block of memory storage. This constructor is similar to the tuple structure in the relational model. We denote a Composition as follows:

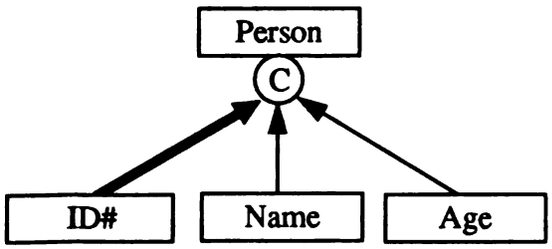


The links will determine explicitly which source Object-Types will make up the ID component and which will make the Property component. The fundamental relationship of ID determining Property would then be enforced. In ER and relational terminology, ID defines the key attributes and Property defines the non-key attributes. Since we are not forced into a two-level construct ("tuple and field" or "entity and attribute"), the "key concept" is applied to the component Object-Types, not specialized "attributes".

The default operators furnished by a Composition constructor is source reference and equality. This can be applied to both types and instances. In the diagrammed example, X.A is the Object-Type A in X, x.a is the instance value of A in the instance x, and [X.C , X.D] is the Property portion of X, while [x.c , x.d] is the property value of instance x. ID(X) is defined as [ID(X.A) , ID(X.B)]. It is understood that when we apply ID() and Property() to Object-Type, we imply instance application. Therefore, two instances of X, x_1 and x_2, are equal iff ID(x_1.a) = ID(x_2.a) and ID(x_1.b) = ID(x_2.b).

7.1.1. Examples of Composition

Here is an example to the Employee/Department problem. We define Person as:



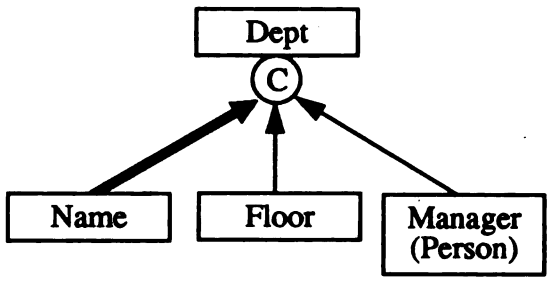
Comments:

(1) A Person is defined as a Composition of ID#, Name, and Age.

(2) ID(Person) = [ID(ID#)]

(3) Property(Person) = [Name , Age]

Dept as:



Comments:

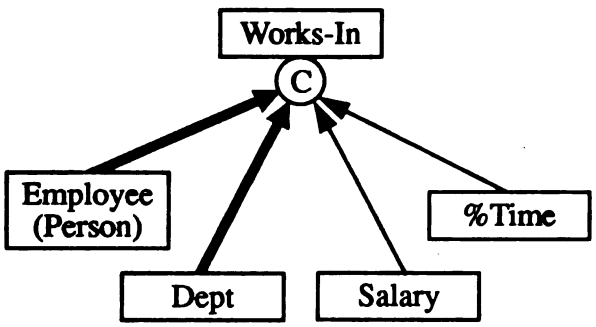
(1) A Dept is defined as a Composition of Name, Floor, and Manager.

(2) ID(Dept) = [ID(Name)]

(3) Property(Dept) = [Floor , Manager]

(4) Manager is an alias for a Person.

and Works-In as:



Comments:

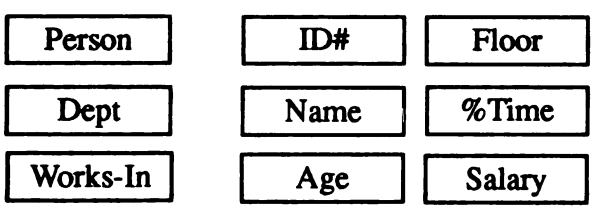
(1) Works-In is defined as a Relationship (Composition) of Employee, Dept, Salary and %Time.

(2) ID(Works-In) = [ID(Employee) , ID(Dept)]

(3) Property(Works-In) = [Salary ,

In Section 7.8.2, we will show that ID equality is the same as instance equality. Therefore, ID(Works-In) can be defined as [Employee , Dept]. For now either form is sufficient.

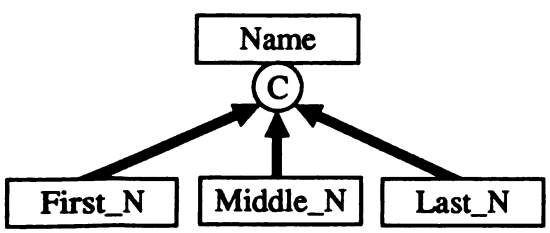
The current state of the EOM schema is:



Comments:

(1) The abstract Object-Types can be implemented as primitive types or be defined as a construction.

As the schema evolve, we may wish to define Person.Name. We can give Name a new definition, so the previous schema is not affected:



Comments:

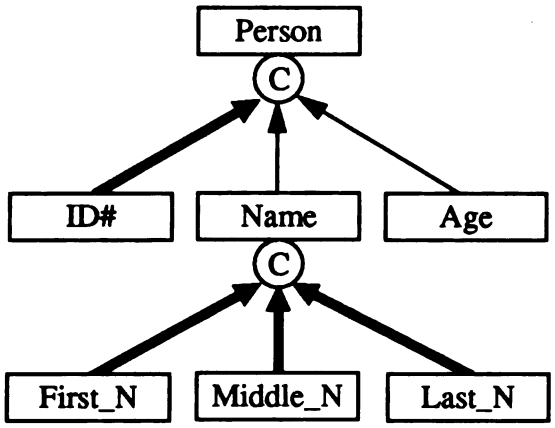
(1) Name is defined as a Composition of First_N, Middle_N, and Last_N.

(2) ID(Name) = [First_N , Middle_N , Last_N]

(3) Property(Name) = [], i.e. none

However, this would change Dept.Name as an unexpected side-effect. The solution is to rename the new definition "P_Name" and implement Person.Name as P_Name. Then Dept.Name is left

unaffected. An alternative is to attach the new definition directly below an existing one:



Comments:

(1) Person.Name is now defined as a Composition of First_N, Middle_N, and Last_N.

(2) Since access to Person.Name is still through Person, this ensures Person.Name behaves properly and Dept.Name is unaffected.

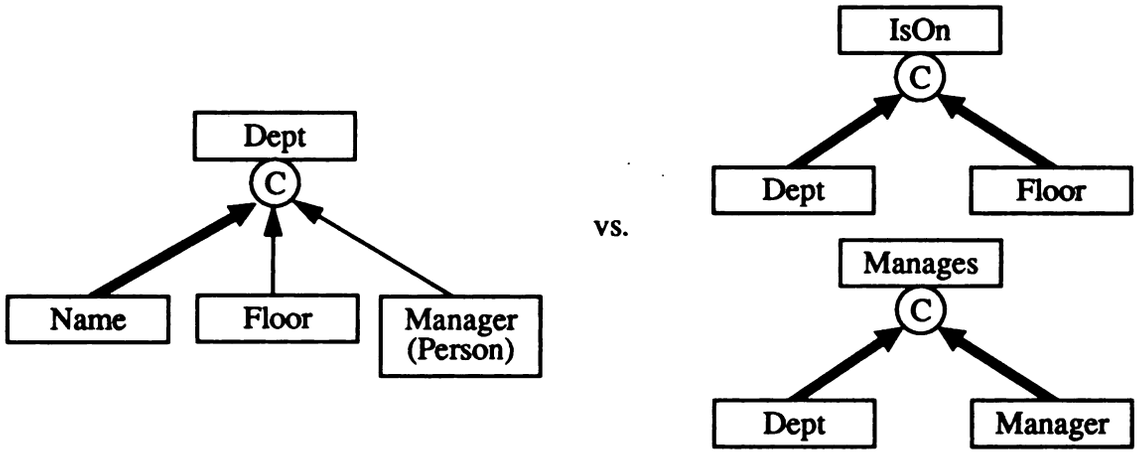
This schema requires local Object-Type definition. If permitted, then future uses of Object-Type "Name" will be ambiguous, i.e. it could be Person.Name or Dept.Name. Thus, for reason of simplicity, we disallow "context-dependent" Object-Type definition for the core EOM, see Section 7.5.2 for more discussion and Section 9.5 for extension.

7.1.2. Composition and Relationships

The Composition constructor can also be used to model relationships. Some would argue that a different constructor should be used, e.g. "R" for relationship, so that the real world relationships, such as Works-In, can be distinguished from the real world compositions. Since the operational definition of the new constructor, Relationship, is the same as a Composition, we choose to use one constructor to model two real world semantics. On the other hand, there is an advantage of using the Relationship constructor to clarify the interpretation of schemata. Therefore, we shall allow the use of the Relationship constructor as an alias for Composition.

Since we unified both Composition and Relationship, it opens the problem of the semantic meaning of the ID and Property arcs. Our formal definition of the relationship between the target and the source is information containment. The most obvious interpretation for this definition is "has" or "part-of". However, other interpretations are permitted provided the containment semantics are not violated. For example, in a typical E-R schema, a Person "has" a ID#, Name and Age. On the other hand, a Dept "has" a Name and Manager, but "is on" a Floor. The alternative is to construct a "IsOn" Object-Type as a relationship that has Dept and Floor as components and a "Manages"

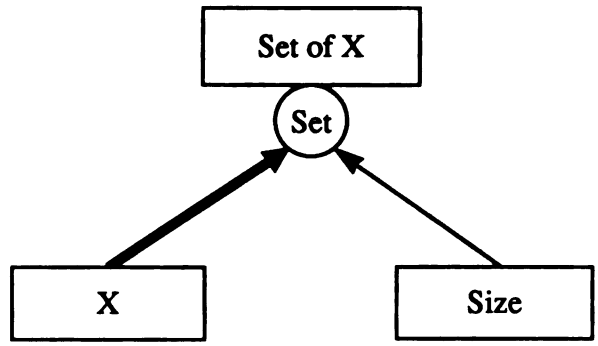
Object-Types as a relationship that has Dept and Person components:



The choice depends on the user's perception of attributes vs. relationships. If expressed as a relationship, then we expect the Objects "IsOn" or "Manages" to have additional Properties. If expressed as a component of Dept, then there should not be additional components to the real world relationship between Dept and Floor or Dept and Manager. Therefore, as long as the source information is properly contained within the target information, the arc between them can take on other meanings without conflicts.

7.2. Set

A SET constructor builds an Object-Type of instances that are "sets of instances". We diagram a Set constructor as follows:



We do not support heterogeneous sets, so only one ID arc is permitted for the Set constructor. The source Object-Type supplies the elements to the instances of the Set-constructed Object-Type. If Set_X (Set of X) is such a construct, then an instance in Set_X is a particular set of X's. For example, let the set of source instances $X = \{x_1, x_2, x_3\}$. Let Set of X, named "Set_X", be the Set-constructed Object-Type from Object-Type X, then instances of Set_X are sets of x's. And the set of instances of Set_X can be as large as the power set of X:

$$\text{PowerSet_X} = \{ \{ \} , \{ x_1 \} , \{ x_2 \} , \{ x_3 \} , \{ x_1 , x_2 \} , \{ x_1 , x_3 \} , \{ x_2 , x_3 \} , \{ x_1 , x_2 , x_3 \} \}$$

Since we are only interested in the actual elements in a database, Set_X would be a subset of PowerSet_X. If we want to retain the ID() and Property() operators, then we need to define ID() over the instances of Set_X such that it distinguishes instances of Set_X from each other. The simplest approach is to define ID(s), where s is an element of Set_X, as the set of ID components over x's in s:

$$\text{ID}(s) = \{ \text{ ID}(x) \mid \text{where } x \in s \}$$

Then equality of ID() is simply set equality. When coupled with the fact that ID() of an instance of Set is value dependent on the source instances, the source arc for the Set constructor is like an ID arc and is given the same form.

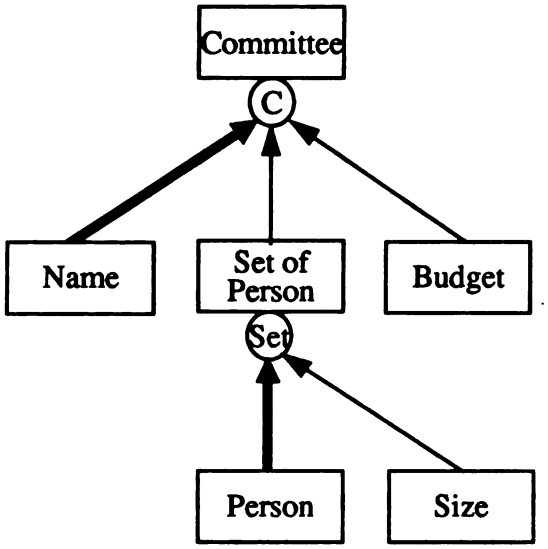
As an Object-Type, a Set constructor provides a set of operators. The standard ones we associate with sets are "member-of", "subset", and "union", "intersection", and "equal".

7.2.1. Usage

The Set constructor provides us a way to resolve the relational multi-value dependency (MVD) [22,64]. When a MVD appears in a relational database, it is usually the result of modelling sets in strict relational form. However, the concept of the "set" is far more intuitive than MVD, the form described using relational algebra. The solution from the relational model would be a second rela-

tion to carry the set information. However, at the conceptual level, we only need to know that a set is operated upon as a unit, not as individual instances. This is equivalent to the NFNF models where the basic relational semantics have been extended with a set (or more correctly, relational) attribute.

Here is an example usage of the Set constructor:



Comments:

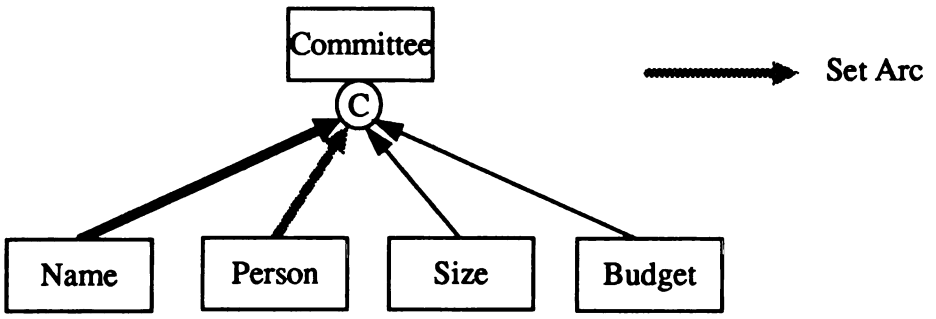
(1) Committee is composed of a Name, a Set_of_Persons, and a Budget.

(2) Size is a property of the Object-Type Set_of_Persons.

(3) Budge is a property of Committee.

7.2.2. Set Instance Naming

The pragmatist would argue that identifying a set instance by its contents is not very efficient nor intuitive. He would ask for a "name" to be associated with each set instance. If we do this for the previous example, then the Committee Object-Type becomes a Composition and the Set constructor becomes a Set arc:



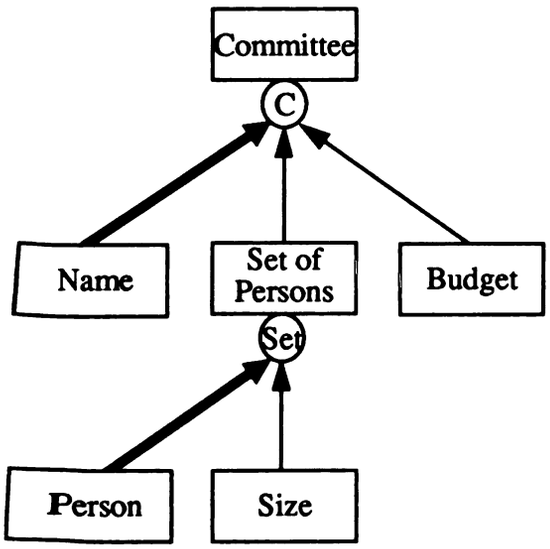
Consequently, this removes the construction of a distinct Set Object-Type. Unfortunately, this is not true for all situations, i.e. there are occasions where "anonymous" sets are used or where properties of the Set Object-Type are distinct from the named component.

7.2.3. Type Naming and Object Property

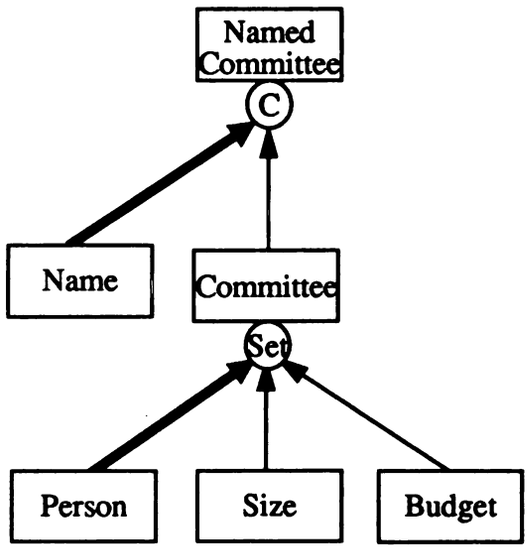
What is the appropriate name for the Set constructed Object-Type? If we named it "Committee", then we expect it to have some properties associated with its function or characteristics, e.g. Budget. If we named it "Set of Persons", then it would only have properties associated with generic

sets, e.g. size. In the process of modelling, the user attaches a name that carries real-world meaning to a type, i.e. the type name is always associated with the two semantic mapping functions. The problem now becomes a question of whether a set instance has distinct Property components. Naturally, size is a property of a Set instance. However, a more interesting question is whether there exist non-computable property as a result of real world semantics? For example, Budget in the previous example could be a candidate for non-computable Set property. If such property information exists, does it belong to the context of the set or does it belong to the set itself? To resolve this question, one would need to determine the "epiphenomenon" of a construct, i.e. what is the difference between the whole and the sum of parts? Using our previous example, Budget is a property of the Committee, but we can also view Budget as an epiphenomenon of the Set of Persons. There are two answers to the question of where Budget belongs.

To a "constructivist", simple constructs, e.g. sets and sequences, has little or no epiphenomenon. Therefore, he would argue that we would assign little or no non-computable property values: a Set of Persons has the characteristics of generic sets and no more. However, when an "intuitionist" sees us using sets to model the real world, he would say we are coercing complex real world relationships into such simple constructs that we will lose "epiphenomenon" information unless we allow explicit capture. He would also caution us that even if the property component of a Set Object-Type is computable, it may not be feasible nor accurate given our current state of knowledge. The two approaches are diagrammed as follows:



Constructivist's Answer



Intuitionist's Answer

Which answer is correct? The solution is based on the name given to the Set-constructed Object-Type. As discussed earlier, a name given to an Object-Type conveys a semantic concept and provides the interpretation and encoding functions. Therefore, the intuitionist's model has to rename

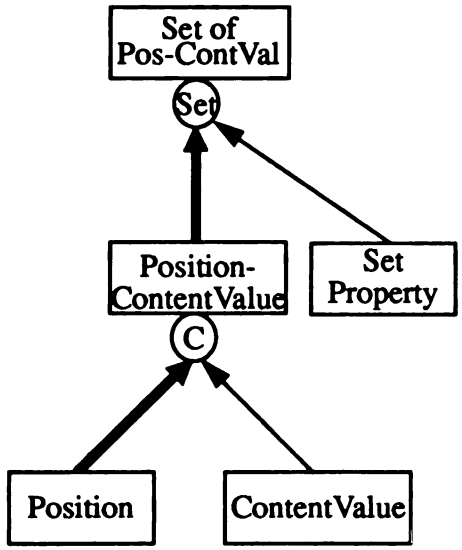
Committee to Named_Committee and Set of Persons to Committee. As it turns out, these two schemata model different real world situations. In the constructivist's case, the budget is not a property of the Set of Persons. Instead, it is the context of the Set of Persons which determines the budget. Conceptually, this budget is not a property of a Set of Person as much as being "associated" with a Set of Person, similar to the way two entities relate through a relationship in the ER model. In the intuitionist's case, the budget is determined by the number of persons in the Committee instance. The result is a budget which is an epiphenomenon of a Set of Persons, i.e. it is dependent on the number of instance members. This situation exists if Budget is not determined by the context of the committee. The correct model, therefore, depends on the real world and each schema has its own semantic interpretation.

7.3. Sequence

As described in Section 4.1, our framework for sequences is based on mathematical functions, i.e. for each position in a sequence, there is one and only one content value. There are several approaches to building sequences of this type.

7.3.1. Set Approach

The simplest approach is to compose the position and content Object-Types together and build a set of them:



Comments:

(1) Position-ContentValue is a composition. For a given Position, there is only one ContentValue. Thus ContentValue can be viewed as Property.

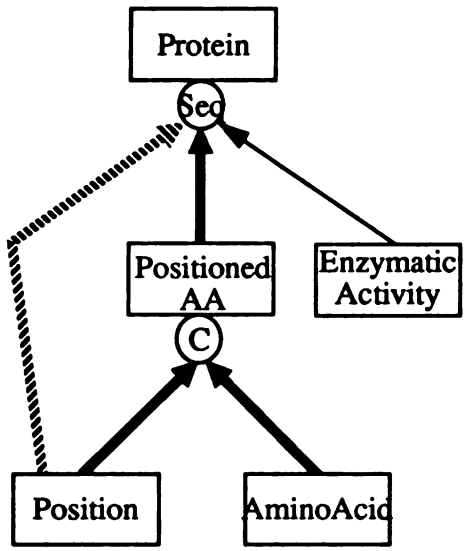
(2) Set Property is dependent on the Set constructor, e.g. size.

Set of Position-ContentValue is ordered, based on the value of Position. Unfortunately, this schema only provides the functions and operators associated with a Set and the semantics associated with a sequence would be lost. Therefore, we would not be able to perform sequence operations on this construct.

7.3.2. Sequence Constructor

Another approach is to develop a position or "ordering" arc that can emanate from any component of the ID component of the sequence. This allows sequences ordered from any component of the sequence-source Object-Type.

The following example demonstrate this approach:



Comments:

(1) Enzymatic Activity properly belongs as a property of Protein. The intrinsic enzymatic activity is always dependent on the sequence of amino acids. Thus for any given sequence, the activity will be the same for any other matching sequence.

(2) AminoAcid is our sequence base type and is promoted to an ID component.

(3) A special "ordering" arc goes from the Position Object-Type to the Sequence constructor.

As a new Object-Type, the Sequence constructor provides new operators. These include "length", "concat", "cut", etc., as described in Section 2.6. Since all Object-Types support ID(), we need to construct a sufficiently distinct value for ID function. In the protein example, a given protein sequence, pr, has

ID(pr)= < ID(pr.1st positionedAA) , ID(pr.2nd positionedAA) , ... >.

Reminder: <...> denotes a sequence,
{...} for sets, and
[...] for tuples or compositions.

Since the ID of Positioned AA is [ID(Position), ID(AminoAcid)], we have

$= < ID(pr.pa_1) , ID(pr.pa_2) , ... >$

$= < [ID(pr.pa_1.p) , ID(pr.pa_1.aa)] , [ID(pr.pa_2.p) , ID(pr.pa_2.aa)] , ... >$

where pa_n is nth Position-AA instance,
p stands for Position instance, and
aa stands for AminoAcid instance.

This is sufficient to distinguished instances of protein based on the amino acid sequence. If we did not promote AminoAcid to an ID, then:

$ID(pr)= < [ID(pr.pa_1)] , [ID(pr.pa_2)] , ... >$

$= < [ID(pr.pa_1.p)] , [ID(pr.pa_2.p)] , ... >$

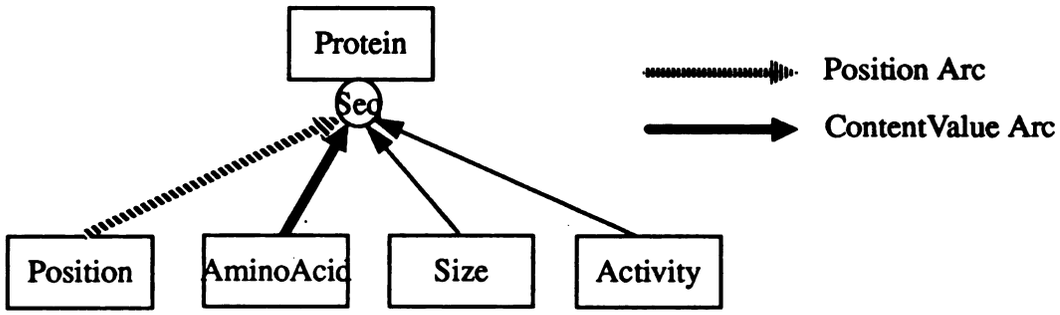
$= < 1 , 2 , 3 , ... >$

which would not be unique enough to distinguish protein instances apart. The ordering arc permits the user to order a sequence based on an arbitrary, but well-defined, component. In order to support

the sequence model developed in Part I, the necessary conditions for Position Object-Type are uniqueness and complete order when viewed as a set.

7.3.3. Alternative Sequence Constructor

The use of a Position-ContentValue composition Object-Type as the source of a Sequence constructor seems to be redundant. We could include its assumed existence in the Sequence constructor directly:



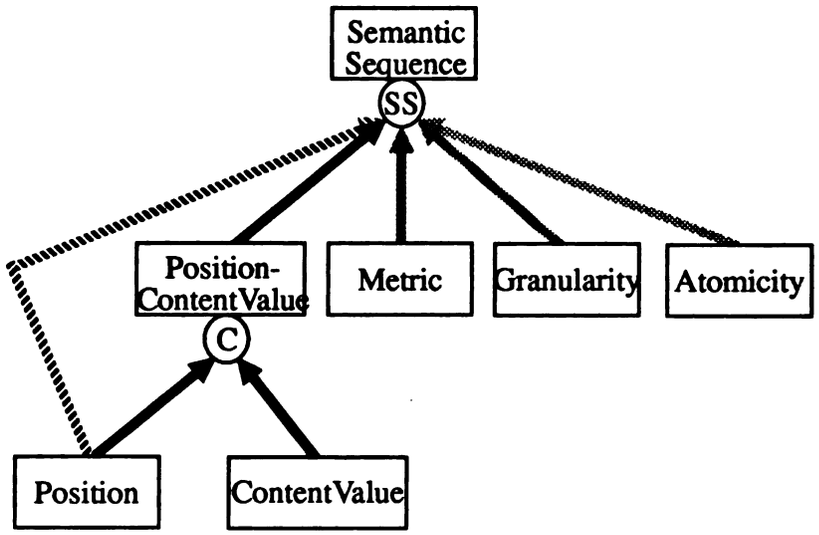
This raises the problem of attaching Property component to the Position-ContentValue Object-Type as a result of Composition. The Property components at the level of the Sequence constructor does not apply to the intermediate level Composition. In certain domains, the intermediate Property information is important. For example, annotations are usually attached to this intermediate composition. Therefore, our explicit representation is a necessary construct.

7.3.4. Semantic Sequences

The Sequence constructor, as it stands, can model abstract order and simple sequences. In order to model abstract semantic sequences, the positional characteristics of Metric, Granularity and Atomicity (see Section 2.2) need to be added as generic components. There are three approaches.

In the first approach, we add a "semantic sequence" constructor. Typically, the positional characteristics of Metric, Granularity, and Atomicity are invariant within a sequence, i.e. all the position values of a given sequence have the same Metric, Granularity, and Atomicity value. Therefore, despite the fact that these characteristics are positional, they should be a component of the Semantic Sequence Object-Type, not a component of the Position Object-Type. Since these characteristics are semantically distinct, they should be distinguishable from each other by arc type, not by Object-Type name.

The following is an example of the graphical representation:



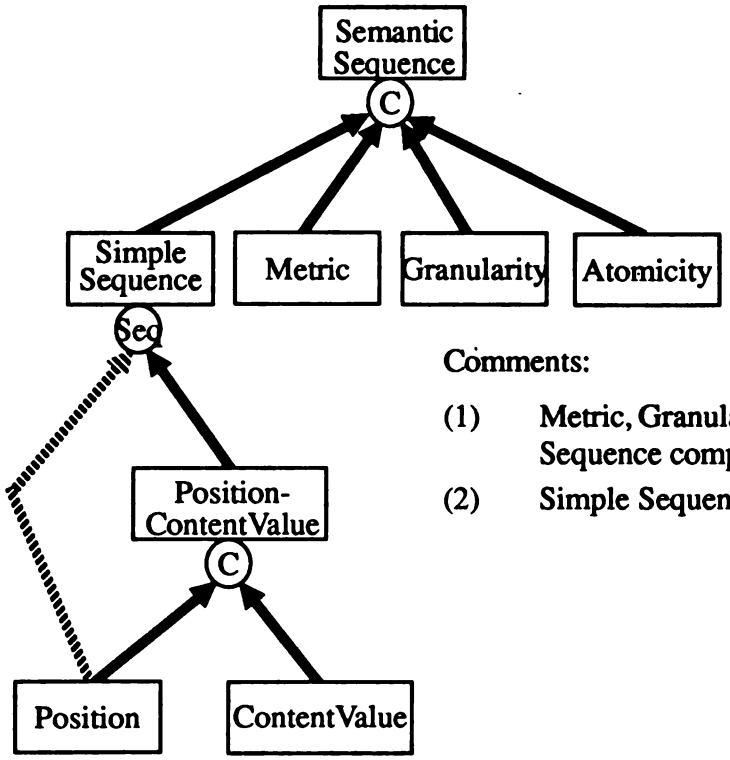
Consequently, there are now five arc types: Order, Sequence-Source, Metric, Granularity, and Atomicity. All the arcs are like ID arcs in their thickness because each component participates as an ID component. For example, two Semantic Sequence instances, which only differs in Metric value, should be considered different. Similar arguments hold for Granularity and Atomicity. However, when we have two Semantic Sequence instances that can be changed to matching values through Metric or Granularity conversions, then the sequences are considered to be "equal". This implies "equality" for Semantic Sequence constructor is no longer a simple value-by-value comparator.

The drawback to this approach is the number of arc types. Each new arc type needs to be evaluated with respect to the arc source Object-Type. For example, the Order arc requires an Object-Type whose values are completely ordered, Metric requires a semantic label, Granularity and Atomicity requires rational values. Therefore, the Position Object-Type cannot be arbitrary as seen in Composition or Set-Source. This limits the generality of the arcs.

The second approach is the other extreme where the positional characteristics are encoded into the Sequence Object-Type name, e.g. "SequenceName_M_G_A" where M, G, and A stand for Metric, Granularity, and Atomicity domain names. In this case, positional characteristics are not modelled as independent Object-Types, but as domain-specific semantic information. For example, for a sequence of markers with kilobase Metric, 0.1 Granularity, and 0.001 Atomicity, the name would be "MarkerSequence_kb_10th_1000th". In this case, all instances of a Sequence Object-Type would have the same positional characteristics. Under some situations this is desirable, but it would be limiting for the management of interconvertable sequences with different positional characteristics.

The third approach is based on the first, but uses a Composition constructor to "implement" a

semantic sequence. The positional characteristics are implemented as a set of ID components for the semantic sequence. This retains the simplicity of the Sequence constructor and satisfies the requirements of the abstract semantic sequence model. However, like the first approach, it requires a new definition for "equality" for the semantic sequence. The following is a representation of Semantic Sequence using the Composition and Sequence constructors:



Comments:

(1) Metric, Granularity, and Atomicity are Semantic Sequence components.

(2) Simple Sequence remains the same as before.

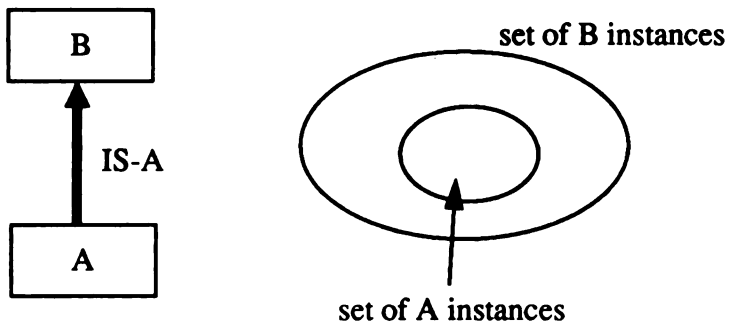
The association of Object-Type name to positional characteristics becomes domain specific and user defined, not mediated by distinct arcs. This is compatible with the EOM definition for Object-Type naming, i.e. it is the user to defines the semantic mapping (see Section 6.1). We will choose the third approach has the basis for semantic sequences.

7.4. IS-A Construction

So far we have only described how instances from one Object-Type associate with instances from another Object-Type. For example, a target instance is associated with source instances in the Composition constructor, or with a set of instances in the Set constructor. Object-Types define the collections and an instance can only come from a single collection. However, in the real world we also can have an instance which came from two collections, i.e. it is from the intersection of two. This is the concept behind "IS-A hierarchy". The cardinal statement of IS-A characterization is:

An instance of A is an instance of B.

Let the following diagram denote a IS-A relationship between A and B:



Comments:

(1) The direction of arrow defines the "IS-A".

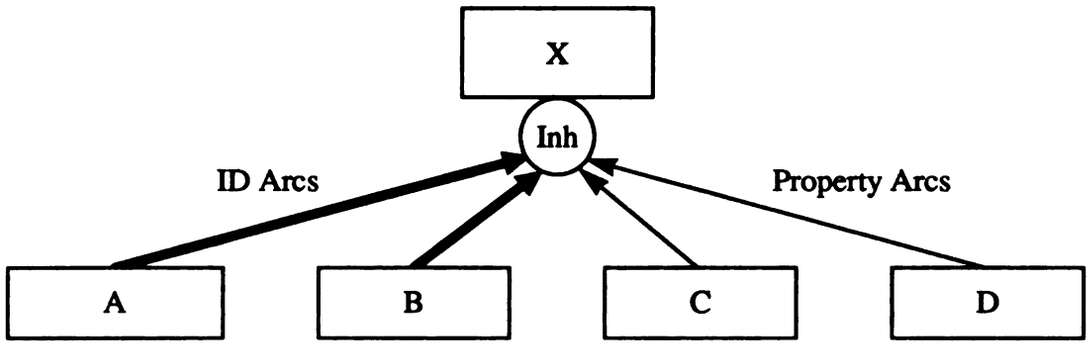
(2) Set A is a "subset" of B, or

(3) An instance of set A "belongs" to both sets A and B.

There are several possible mechanisms that satisfy the definition of a natural language IS-A: inheritance, subset, and union. We will discuss these in order.

7.5. Inheritance

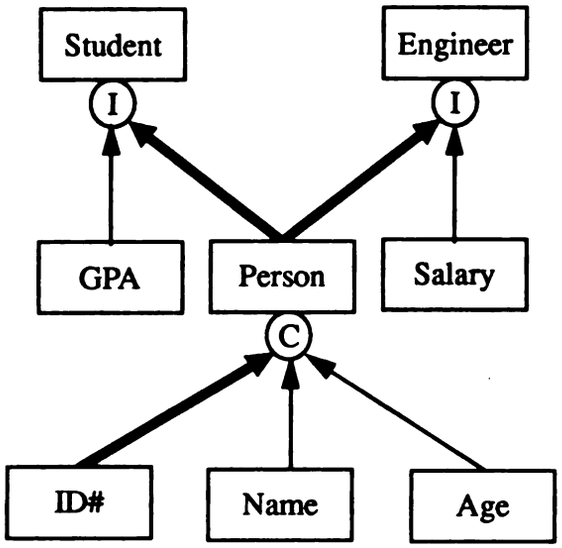
Inheritance is a form of "IS-A" relationship. This constructor is diagrammed as follows:



In the EOM, we define it as strict subtyping hierarchy, i.e. every instance of X has all the information of some instance A or some instance B (or both). In the Object-Oriented terminology, A is the supertype and X is the subtype. Since operations are based on the information content, all operations defined on A can be applied to instances of X. When given an instance of X and if we "reduce" it into an instance of A, then only the information and operations as an instance of A should be available. Likewise for B. Therefore, the "navigational" step we take from X to A would limit the amount of information we can work with. An analogy in the relational model is the "projection" of X onto A. In other semantic models, this is considered as "specialization": the subtype is a specialization of the supertype.

We reverse the direction of IS-A arc: the supertype is the source and the subtype is the target. The reason for this reversal is information content. Our constructors follows the general principle that the constructed type has more "information" than its sources. Therefore, a subtype, under inheritance, has more information than the supertype. As a result, the arrow is directed toward the subtype.

An example using the inheritance constructor is diagrammed as follows:



Comments:

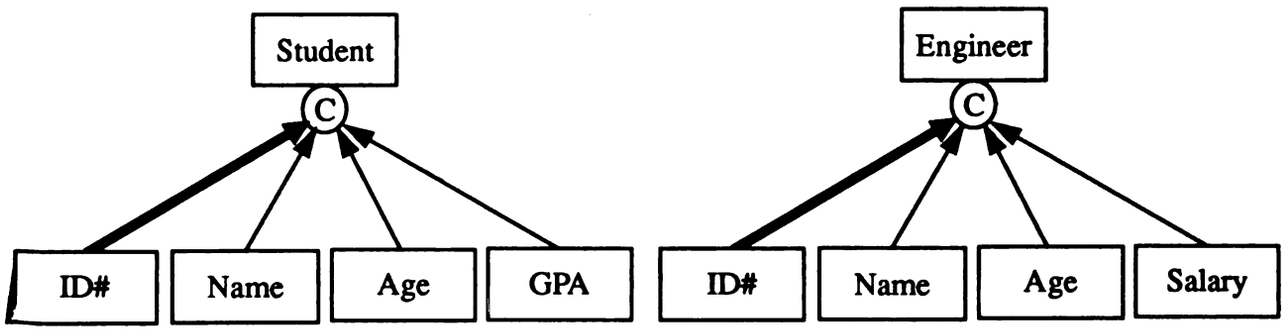
(1) An instance of Student or Engineer "is an" instance of Person. The Person Object-Type is the "supertype". Student and Engineer Object-Types are "subtypes".

(2) Property of Student and Engineer is different.

(3) Property of Person is common to Student and Engineer.

(4) In this example, we permit the situation where an instance of Person is common to both Student and Engineer. For example, persons in a Co-op student program.

The arc from the supertype to the Inheritance constructor is thick because the identity of the subtype instance is determined by the supertype instance, i.e. ID is dependent on that arc. The inheritance constructor can be viewed as a short hand notation for an otherwise flat schema. The transformation between the schemata is a simple reattachment of supertype components to the subtype Object-Type. The above schema is equivalent to:



Although the transformation of schema with inheritance constructors to one without is simple and straightforward, inheritance is a very useful abstraction tool to be retained. For example, it permits the existence of persons who have not been assigned to students nor engineers. In addition, the concept of a Person Object-Type and the operations on a Person Object-Type is now collected and independent of both subtypes. In programming languages, inheritance improves cohesion [59]. Similar improvements between data objects can be obtained in database modelling. Therefore, despite the technical simplicity of "de-inheritance" transform, it is a useful tool of conceptual modelling. However, several issues remain to be resolved.

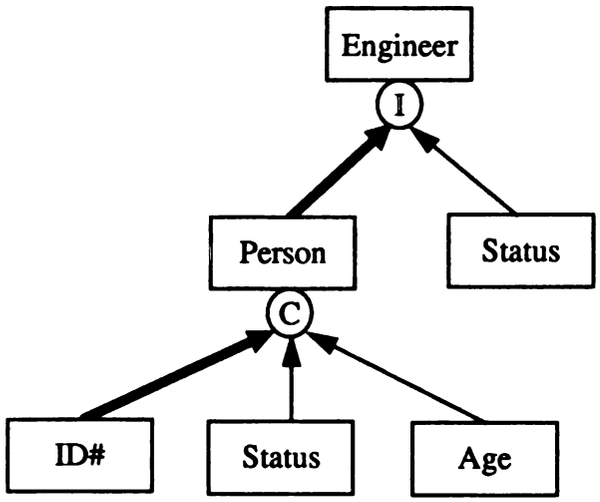
7.5.1. Composition vs. Inheritance

What is the operational difference between inheritance and composition? Inheritance "percolate" both ID and Property components, as defined by the transformation of the schema, so the components of the supertype is fully visible as components of the subtype. If the inheritance constructor is replaced by a Composition constructor, then Property() of the subtype would not retrieve the Property() of the supertype. If we used a Composition in our example, Property(Engineer) would return Salary and ID(Engineer) would return ID(Person), or ID#. Name and Age would have to be retrieved by Property(Engineer.Person). Using Inheritance, Property(Engineer) would return Name, Age, and Salary, as expected from the flat schema and the semantics of the real world.

There are other semantic differences between composition and inheritance. An instance from a Composition is "associated" with instances from its sources. On the other hand, an instance from an Inheritance "is" the same instance from the ID sources of inheritance. "Association" imply different instances, therefore, one instance can change without affecting the other one. "IS-A" implies the same instance, which means changes to one will affect the other. For example, changing the Age of an Engineer instance would affect the Age value of the source Person instance and will also affect the Age value of any inherited Student instance of that person.

7.5.2. Overloading

The perceived strength of inheritance is derived from the concept of overloading. This is a result of the natural language deficiency for real world semantics, i.e. multiple meanings are mapped to the same linguistic token. In the process of modelling, there is a tendency to maintain this overloaded mapping and to use context information (*a la* natural language) to resolve the ambiguity. In general, there are two types of overloading. Consider the following schema:



The first type is definition overloading. If we permit context-dependent Object-Type definition, then Engineer.Status can contain different information content than Person.Status. However, in the

core model, an Object-Type's name determines the semantic mapping between real world concepts and symbolic values. Therefore, Object-Types with the same name must have the same mapping. If Person.Status was meant to be marital and Engineer.Status was meant to be professional, then the two Status Object-Types should be named differently. Therefore, definition overloading is not permitted in the core model.

The second type of overloading is value overloading. Here, Engineer.Status and Person.Status has the same Object-Type definition, but different values. The "de-inherited" schema remains the same, i.e. Engineer is still a composition of ID#, Status, and Age. However, the retrieval of component instances is changed to reflect overloading. For example, we can use direct source priority to determine which overloaded value to use. In the lookup for Engineer.Status, if Status is declared as a source directly under Engineer, it will be the one used. Otherwise, a search is made for Status in the supertype Person. In the lookup for Person.Status, only the sources of Person is scanned. Thus it would not return the Status value under subtype Engineer.

7.5.3. Overloading and Consistency

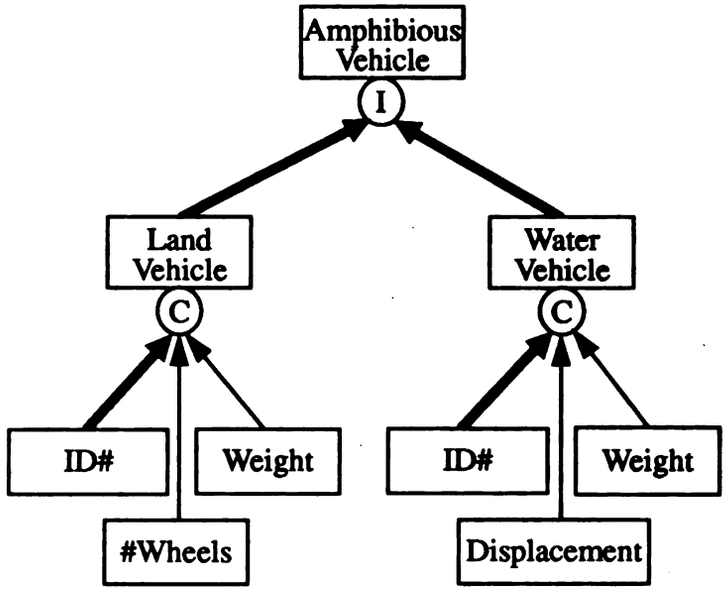
Value overloading creates another problem of its own: consistency. If we adopt the value resolution algorithm as described above, then there exists the potential that an Engineer's instance status is different from his status as a Person. This would be a violation of the consistency in the real world. If an instance of Person is also an instance of Engineer, then the two instances are "identical". Consequently, their Status values should also be the same. Therefore, if Status was meant to be marital, then a Person instance and the corresponding Engineer instance should have the same value.

If we consider the fact that overloading is a linguistic limitation and not a modelling limitation, it places doubts on the worth of value overloading. After all, if a value is to be overloaded, then that value could be easily made a part the supertype instance value in the first place. Under this analysis, value overloading is superfluous and dangerous, therefore it is not supported in the core model.

7.5.4. Multiple Inheritance

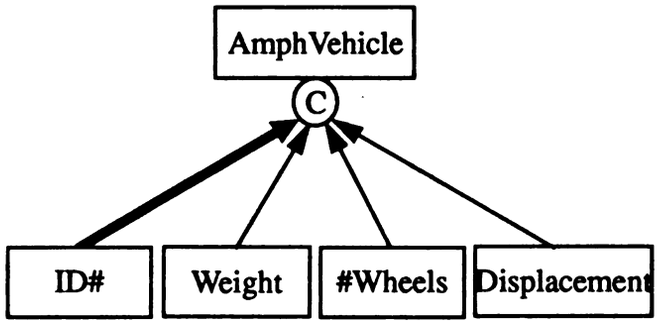
Multiple inheritance is an extension to permit 1-to-many "IS-A" relationship between a subtype instance and supertype instances. However, a "competition" overloading presents an additional problem for the redeclared Object-Types in the supertypes.

This problem can be demonstrated in the following example:



AmphibiousVehicle inherits from LandVehicle and WaterVehicle and the Property of AmphVehicle includes #Wheels and Displacement. However, from which supertype shall it inherit Weight? If the Weight instance value is the same, then there is no problem. On the other hand, if they are different, then there is no *a priori* mechanism to resolve the ambiguity. A similar, but more serious problem also exist for ID#.

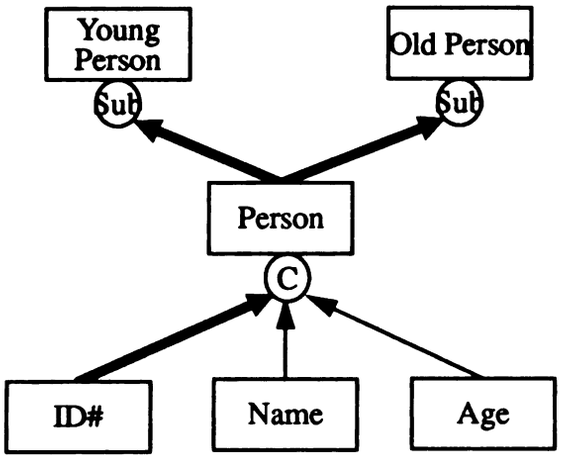
Since we excluded context dependent Object-Type definition, i.e. no definitional overloading, then the resolution is simple, since all same-name Object-Types refer to the same content. Then LandVehicle.ID# and WaterVehicle.ID# are the same Object-Type definition. Similarly, both Weights are the same. The resultant "de-inherited" schema for AmphibiousVehicle would look like:



Since we disallowed value overloading, i.e. values must be consistent, then it does not matter which ID# and Weight values will be associated with the instances of AmphibiousVehicle. Presumably, in a consistent database, LandVehicle.ID# and WaterVehicle.ID# (and Weight) will have the same value for the instances which are also Amphibious. This becomes an integrity constraint on the components of Inheritance constructors to maintain value consistency.

7.6. Subset

Subset is the separation of instances based on some predicate over the value of the instances. For example, Person is the superset to Young_Person and Old_Person subsets:



Comments:

(1) An instance of YoungPerson or OldPerson "is an" instance of Person. Person is the "superset". YoungPerson and OldPerson are "subsets".

(2) Properties of YoungPerson and OldPerson are the same as Person.

(3) In this example, a Person cannot be both Young and Old. However, in other examples, the subsets could be overlapping.

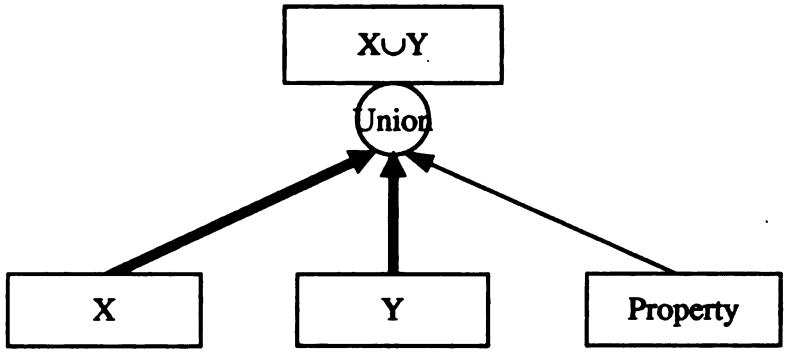
Strict Subset cannot have additional components. So the Subset constructor has only one incoming ID arc and no Property arcs. The direction of the arc in a Subset constructor goes from the superset to the subset, because a subset instance contains the information content of its superset instance plus the information encoded by the subset predicate. Therefore, a subset instance "holds" more information than the superset instance.

7.6.1. Subset and Inheritance

Subset can be viewed as Inheritance, although the distinction between inheritance and subset is the presence of the predicate. If the model permits arbitrary predicates to be supplied, then the result is the indeterminacy of a Turing machine. This would argue for moving the predicate out of the definition of the EOM and, instead, use a "hook" for an outside agent in the application domain to perform the semantic decision. Similarly, the real world decision that determines the correspondence between a supertype instance and a subtype instance in an inheritance schema is also outside the definition of the conceptual model. For example, the process that decides whether a person is a student or an engineer is not modelled, but rather, the decision is accepted and reflected in the state of the database. If so, then inheritance and subset can be represented by the same constructor in our model. Furthermore, Subset can be viewed as a special case of Inheritance where no new source components are added. This "overloaded" use of a constructor for both Inheritance and Subset is similar to the use of Composition constructor for both ER relationships and attribute characterization.

7.7. Union

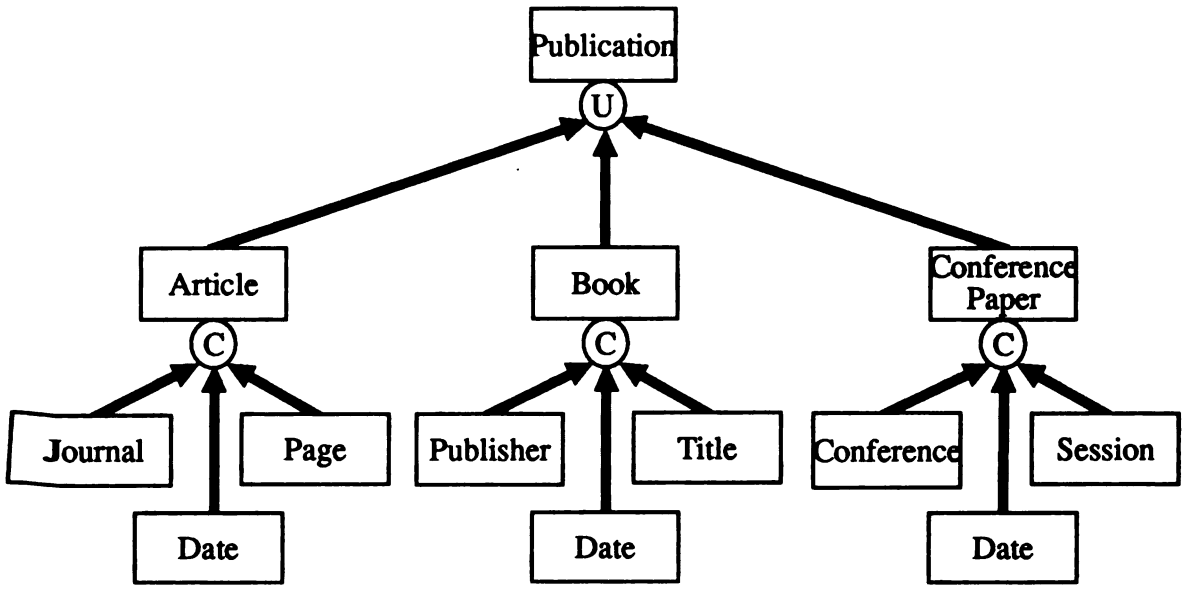
The third type of "IS-A" is the Union constructor:



This is equivalent to the "generalization" concept in other semantic data models: an instance of X∪Y is an instance of X or an instance of Y. A Union instance must also contains the information that determines which source Object-Type it is derived from. As a result, it contains "more" information than the sum of its sources, and the arc points toward the Union Object-Type. Naturally, the ID arc determines the possible sources for the Union. Property arc is for any new information available as a Union Object-Type. Our usage of "Union" is based on the "union"-type of programming languages. The Union constructor provides dynamic binding, which allows an instance reference to be defined by the dynamic state of the database system and not by the static definition of a schema.

7.7.1. Example

An example of the Union constructor is:



Comments:

(1) An instance of Publication "is an" instance of Article, Book, or (Conference) Paper.

(2) Not shown are the Property components of Article, Book, and Paper.

(3) There is no a priori common information expected of Article, Book, or Paper. In this particular schema, Date happens to be common.

7.7.2. Union and Inheritance

Some would argue that a Union schema can be viewed as an Inheritance schema upside down. In our example, Publication would be the supertype, providing the source Object-Type to subtypes Article, Book, and Paper. If this is the only difference between Union and Inheritance, then Union, as a distinct constructor, should be dropped from the model and its semantics replaced with the Inheritance constructor. However, a union instance retains the full information content of the source type, while the inheritance supertype only has the "reduced" information content. This difference in information content determines what kind of operations are permitted on a Union instance.

A second argument to eliminate Union constructor is its similarity in form to multiple inheritance. However, a union instance does not inherit the information content of its source instances nor is it subject to the integrity constraint of consistent values that a multiply inherited instance in under. Therefore, the semantics of multiple inheritance is not the correct interpretation of the union constructor.

7.7.3. Union and Superset

The concept of Union is distinct from Superset because Union allows heterogeneity of information content between source and target while strict Superset does not. In fact, strict Superset is a special case of Union, where all the sources are homogeneously defined. If this is the case, then all the source Object-Types are aliases of a core Object-Type.

7.7.4. Considerations of IS-A

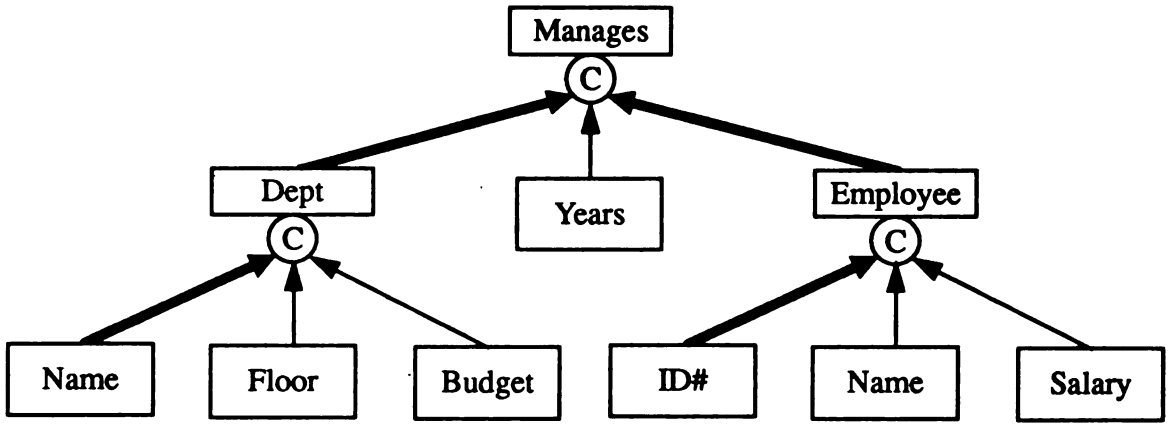
There are two uses of IS-A constructors. First objective is the simplification by removing redundant information in a schema. Common themes can be grouped together into a "supertype", and only the specifics need definition in the subtypes. Second objective is the graceful evolution of the schema: new extensions and groupings can be added without redesigning the old schema. This preserves the validity of previous queries and reduces the software maintenance of database applications. The second objective also forces the separation of Inheritance ("specialization") constructor from Union ("generalization") constructor. A data model can propose only one constructor for both, but in doing so, the ability to gracefully evolve a schema is lost. Using the Publication example, if we first define Article and Book as subtypes of Publication in the schema, then when we add

Paper, the sources for Publication could be shuffled. Thus any queries defined for Publication before the addition of Paper is now inconsistent with the new schema. However, with a Union constructor, queries for Publication can remain independent of its sources.

7.8. General Characteristics

7.8.1. Information Content of Instances

We defined ID and Property components to an Object-Type, hence, also to any instances belonging to that Object-Type, but is the total information content of an instance the same as the sum of the ID components and the Property components? As it turns out, the definitions of ID, as functions for Composition, Set and Sequence constructors, do not return all the information of the ID components. For example:



ID(Manages) = [ID(Dept) , ID(Employee)]
= [ID(Dept.Name) , ID(Employee.ID#)]
≠ [Dept , Employee]
Property(Manages) = [Years]

Therefore, ID(Manages) + Property(Manages) ≠ Manages. This difference raises the question of whether to "enlarge" the definition of ID() to include the property component of an ID source, i.e. let ID(Manages) = [Dept , Employee]. Since we can reference the information content of the ID components by tuple notation, i.e. "[...]" and the equality determination is not dependent on the Property value (see following section), we will not support this proposal.

7.8.2. Extension to the Fundamental Relationship

For any constructor, in addition to the fundamental relationship of ID and Property, we can also state:

If $ID(x_1) = ID(x_2)$, then $x_1 = x_2$, i.e. bit equality over all the bits.

Proof:

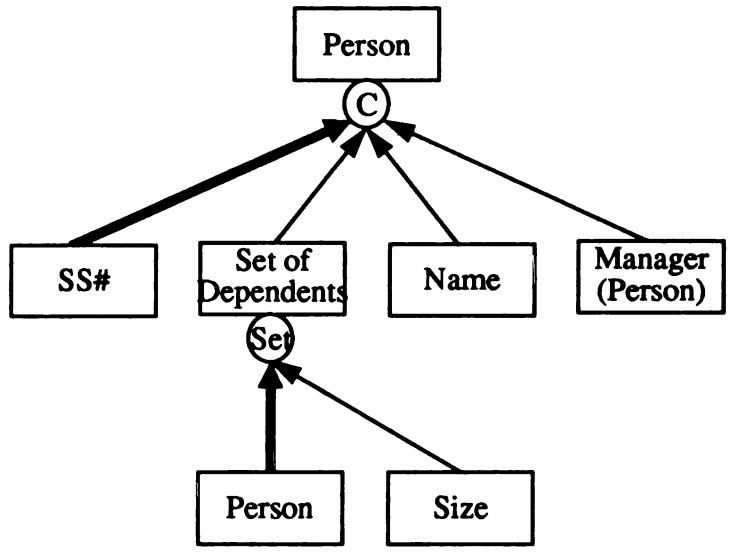
1. For every ID component A, of X:
   $ID(x_1) = ID(x_2) \rightarrow ID(x_1.a) = ID(x_2.a)$
   $\rightarrow Property(x_1.a) = Property(x_2.a)$,

2. if A is primitive, then x1.a = x2.a.
3. if A is constructed, then we recursively traverse the construction tree, until primitive Object-Types are reached. Technically, this is done via induction from the primitive to the complex.
4. For every Property component C of X:
   $ID(x_1) = ID(x_2) \rightarrow x_1.c = x_2.c$,
   by the original definition of fundamental relationship.
5. Therefore we have $x_1 = x_2$.

This is much stronger than the fundamental "ID determines Property" rule, because it asserts equality over all the component values of two instances. In addition, this removes the "enlargement" of ID() proposed in the previous section and keeps the "computation" of ID() down to a minimal.

7.8.3. Recursive Application of Constructors

Object-Types can be defined only once, but they can be declared or referenced multiply. What would happen if an Object-Type is encountered multiply within its definition:



In this definition of Person, we encounter Manager, which is another Person and a Set of Dependents, which are made of Persons. In the Manager path, there is a potential conflict if a person is his/her own manager. In the Dependent path, there is usually no conflict, since a person is never his/her own dependent (except for tax purposes). The conflict becomes apparent when recursive access is attempted. For example, to answer the question: "find all managers of X", one usually apply recursive access. If an instance is re-encountered in the access, some heuristic must be applied to prevent infinite regress. For example, if a Person instance X has a manager who is himself, then the search must be terminated. These problems have been addressed in the recursive

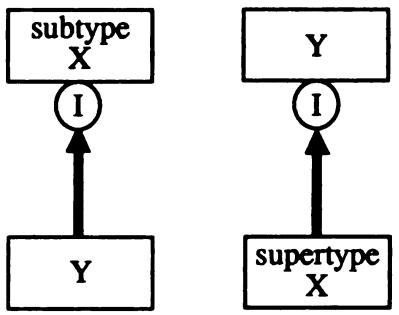
query processing research and will not be reiterated here.

A deeper problem exists for the Extensible Object Model if recursion occur along the ID path. We analyze this problem in two parts, first with IS-A constructors and second with conventional constructors.

A. IS-A Constructors Must be ID Acyclic

Any IS-A constructor cycle along the ID path is invalid. This is because an instance constructed by an IS-A constructor is the same instance of its source. An example of cyclic ID path can be demonstrated by the Inheritance constructor.

For example:



We have:

ID(subtype x) = ID(y) and ID(y) = ID(supertype x),

but since the x instances for both sub- and super-types are identical, we also have:

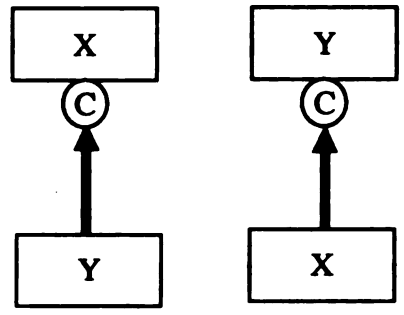
supertype x ≡ subtype x.

Therefore, it bring us back to the beginning. Consequently, the ID value of an X instance cannot be found in this cycle. Therefore, X does not have a definable ID. A similar argument can be made for a cycle of pure Union constructors and a cycle of mixed Inheritance/Union (any IS-A) constructors.

B. Conventional Constructors Must be ID Acyclic

This situation is more subtle. An Object-Type has one or more ID source components, which in turn, have their own. If one of its descendent ID Types eventually refer back to the initial Object-

Type, we would encounter a similar problem to the IS-A example:



We have:

$$ID(x_i) = ID(y_i) \text{ and } ID(y_i) = ID(x_j).$$

If $x_i \equiv x_j$, have the same situation as in IS-A.

If $x_i \not\equiv x_j$, we still need to define x_j:

$$ID(x_j) = ID(y_j) \text{ and } ID(y_j) = ID(x_k), \text{ ad infinitum.}$$

Consequently, the ID value of an X instance cannot be fully defined due to infinite regress. Therefore, X is not "ID definable". The end result is that Object-Types along ID arcs cannot form cycles, independent of constructor usage. On the other hand, cycles in Property arcs are permitted, but only if some form of termination heuristic is provided.

CHAPTER 8. CONTEXT DEPENDENCY

In Chapter 6 and Chapter 7, we have defined the following:

(1) the concept of an abstract Object-Type.

(2) the concept of an Instance.

(3) the representation of a collection of instances by an Object-Type.

(4) the concept of ID and Property values with respect to the abstract Object-Type and the fundamental rule that "ID determines Property".

(5) the concept of constructors, from which we construct new Object-Types.

(6) the use of ID and Property arcs in the Object-Type constructors for the explicit determination of ID and Property of an instance.

These definitions build a conceptual model in which we know the following:

(1) what information is stored → as defined by Object-Type constructors and definitions.

(2) how to interpret the stored information → by the semantic mapping associated with the name of the Object-Type.

(3) basic properties of the information content → two instances with equal ID values must have equal Property values.

A schema based on the above principles provides relationships between Object-Types, i.e. structural information, but this is insufficient to model instance relationships such as cardinality and dependency. Therefore, we will extend the model by defining where the information is stored, i.e. the relationships between Instances. The approach is by augmenting Object-Type definition with the concept of "context dependency" to form an abstract definition of location for the Instances. There are three types of context dependencies:

(1) target dependency, which determines the relationships between instances and their target instances. This also establishes the equivalence relationship for instances, i.e. identity, beyond the equality relationship.

(2) source dependency, which determines the relationship between instances and their source instances. This provides integrity constraints for instance existence.

(3) peer dependency, which determines the relationship between instances of the same Object-Type. This provides the basis for cardinality relationships and is also an integrity constraint for instance existence.

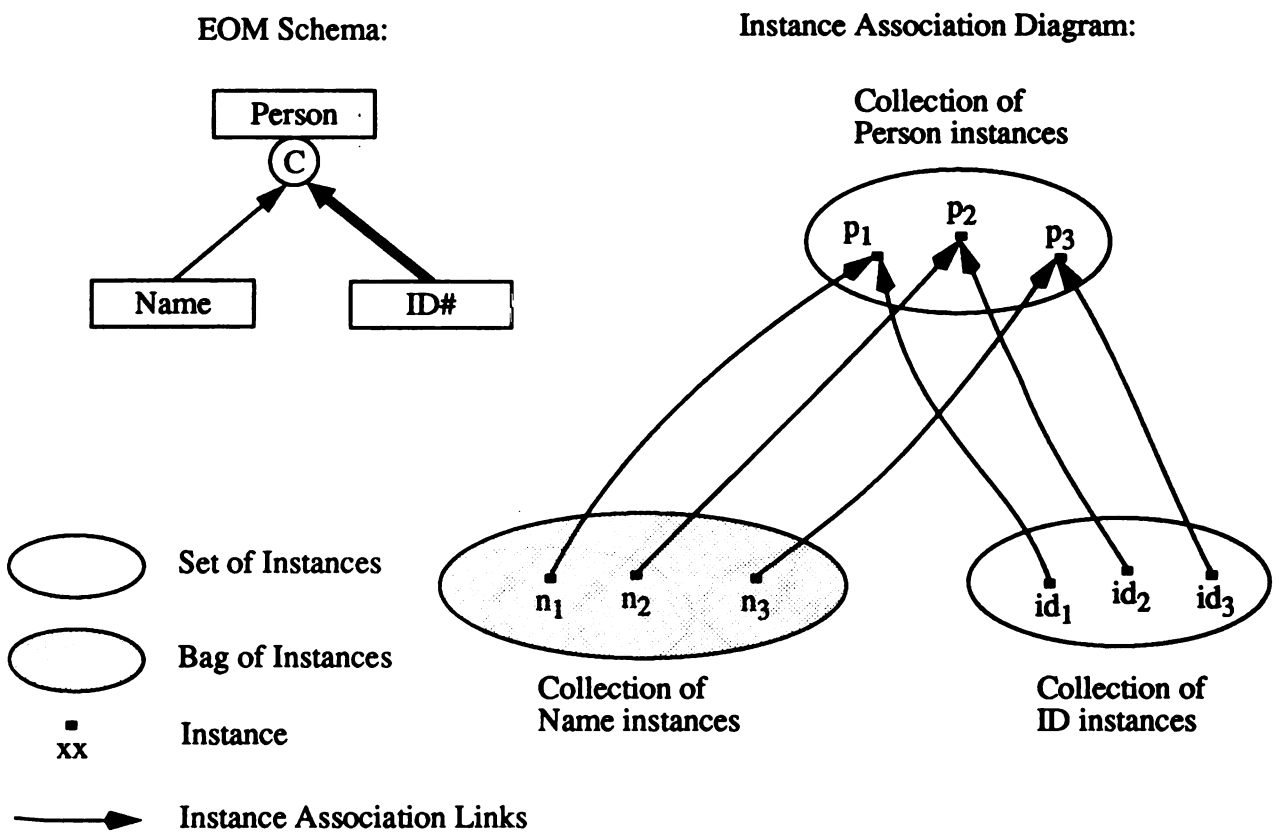
Context Dependencies create an orthogonal set of specifications that can be applied independent of the constructors. Visually, following the arcs and constructors, target dependency "goes up" the

8.1. Instance Association Diagrams

Before we describe the various context dependencies, we introduce a different graphical notation, named "instance association diagrams", to help us visualize Instance locations. Previously, a schema "graph" only shows the Object-Types and arcs between them, which defines how information from one Object-Type associates to another. However, a schema is also a representation of information content in a database, i.e. each Object-Type represents a collection of instances in the database. Therefore, we introduce a different graphical representation to denote actual instances defined by a schema. This representation is to be used as a tool to visualize the interaction between Object-Types and collections of instances under context dependency, they are not to be used in an Extensible Object schema. In addition to the visualization, this instance association diagram is also a semantic "model" for the instances defined by our conceptual Extensible Object schema. Ultimately, it will help us define and resolve the issue of context dependency.

8.1.1. Composition, Set, and Sequence

An example of the instance association diagram is show as follows:



The Person Object-Type defines a collection of Person instances. In this case, the Person collection is a set and holds three instances, p_1, p_2, and p_3. The same applies to ID#. The Name collection is a bag and holds three instances. The links from n's to p's indicate which Name instance is associ-

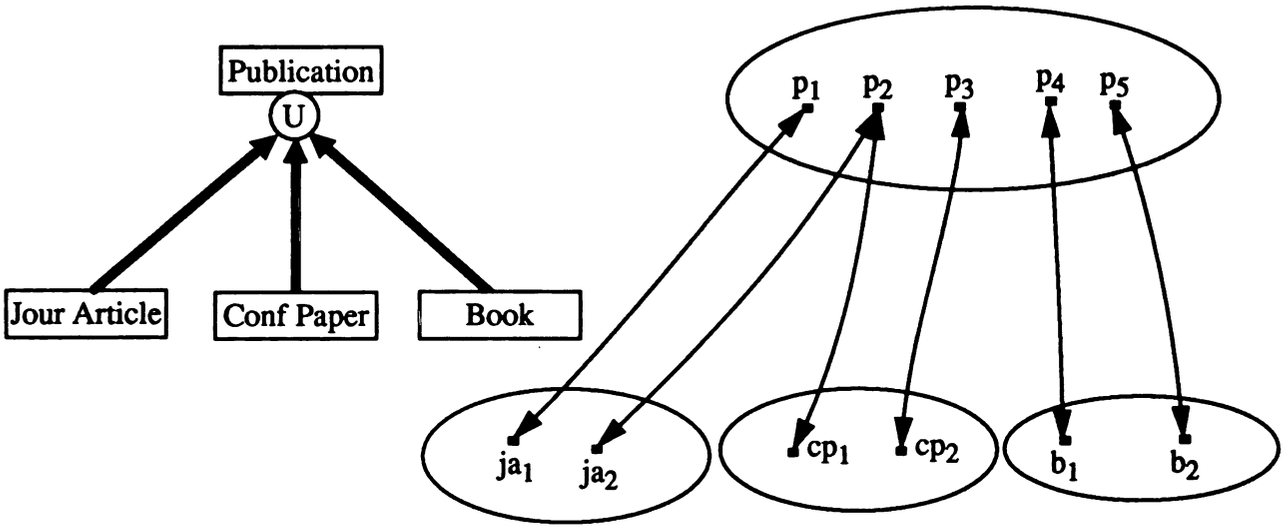
ated with which Person instance. These links are the "instance" equivalent of the schema arcs for Composition. We do not differentiate ID from Property because, on the "instance" side, they are all association links. There should be only at most one arc between any two instances, since none of our constructors support a multiplicity of instances within a strict set.

From now on, the term "set" is defined as a strict set, i.e. each element in a set is unique and has a different value from all other elements of the same set. For collections with multiple equal-valued instances, the term will be a "bag". In the case of a bag, we could have two instances whose values are equal ("="), but they are not identical or equivalent ("≡").

In the instance association diagrams, we denote distinct instances by distinct physical locations on the diagram. However, a single instance may be given different labels because of different paths were used to reach the instance. These paths include navigation from target instance or selection applied to a collection of instances. When used as references, without the benefit of an instance association diagram, we face the possibility that two differently labels may actually refer to the same instance.

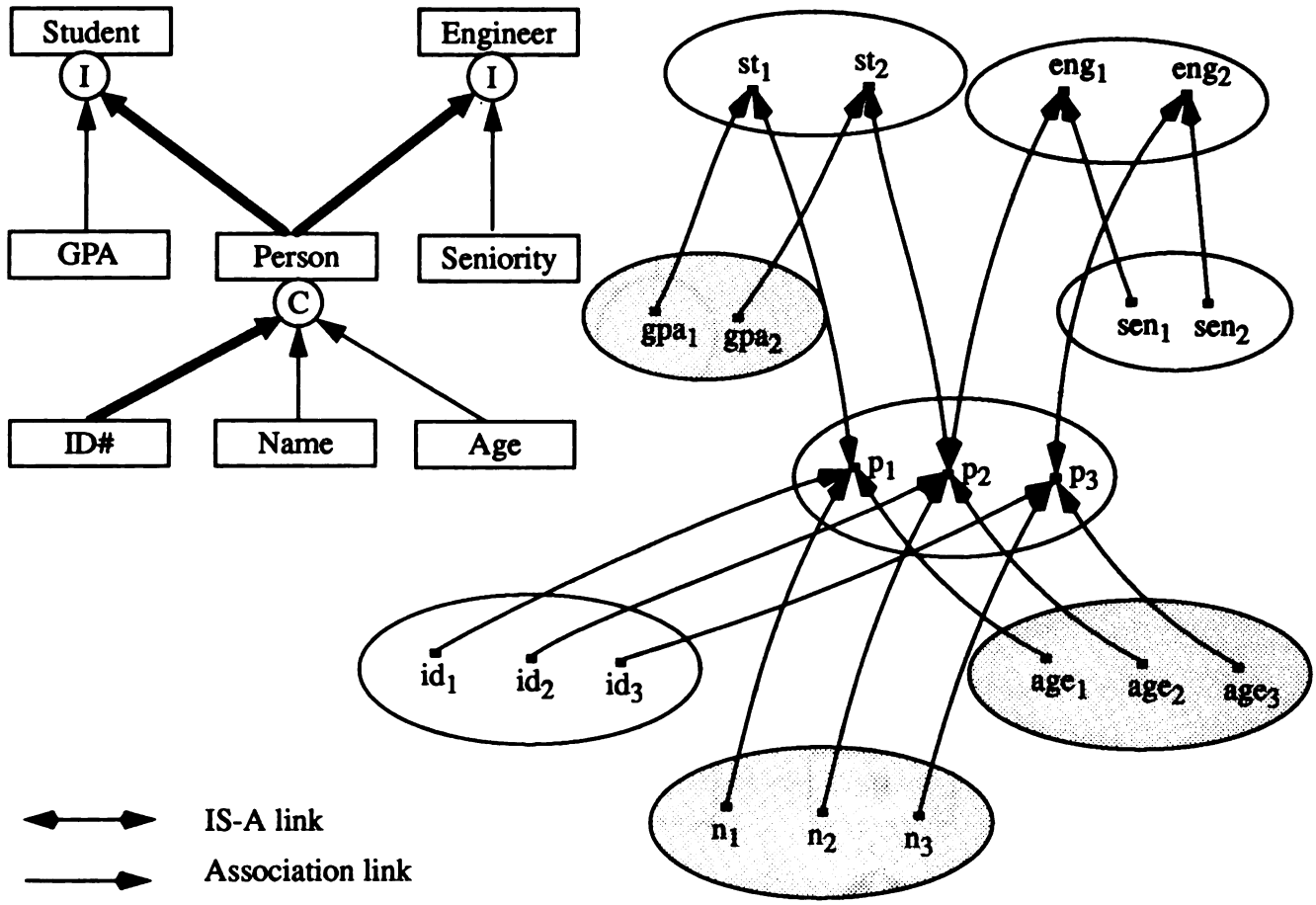
8.1.2. Inheritance and Union

Instance association diagrams can also be applied toward the Inheritance and Union constructors. A Union schema will create the following example diagram:



The differentiation between "structural" constructors and IS-A constructors is represented in the instance association diagrams by the different types of links. Therefore, the links between a Publication instance and its source instances have double-ended arrows to indicate IS-A. Note that p_2 is both a Journal Article instance and a Conference Paper instance.

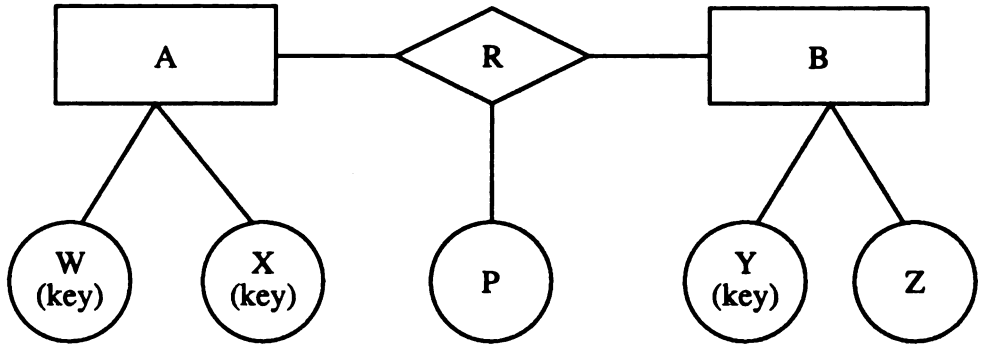
An example that includes both "structural" and IS-A constructors is diagrammed in the following:



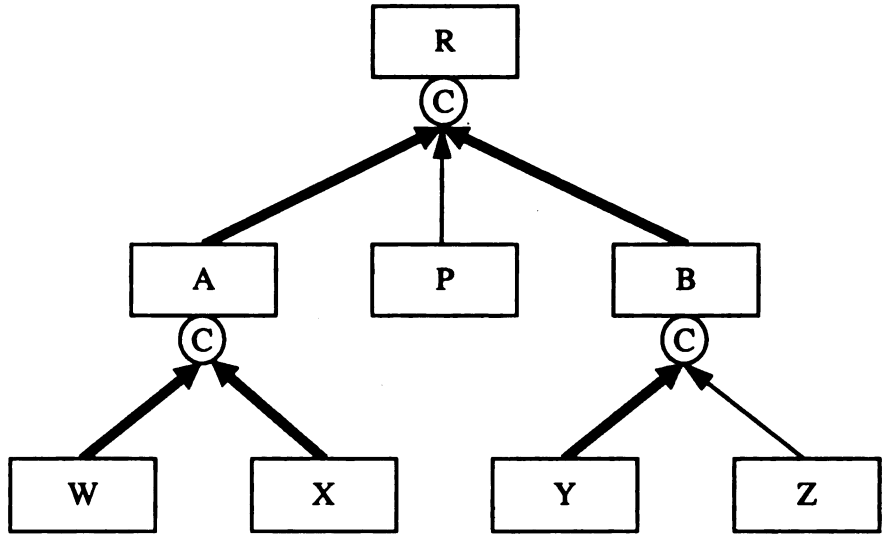
The links between Student instances and Person instances have double-ended arrows to indicate IS-A. Same for Engineer instances. Person p_2 is both a student and an engineer.

8.2. Target Dependency

To address the issue of target dependency, let's take an ER schema as a starting point:



If we convert this schema to the Extensible Object Model, we have:



In the ER model, W has a different symbol than A, therefore they are interpreted differently. However, in the Extensible Object schema, they are equivalent, i.e. they both are ID source to a Composition constructor. Therefore, if we wish to capture the ER difference, we need to examine the associations between Object-Type W and A vs. A and R. The question becomes what are the semantic differences between W/A association and A/R association in the ER model? and how do we capture these differences in our model?

One interpretation for Object-Type W is that of an attribute for an entity. That is, the attribute's value is the sole property of the entity and is not subjected to constraints from other Object-Type definitions. Consequently, the value of this attribute is modifiable without constraints. Another interpretation for W is that they are implemented values, such as integers, dates, and strings, with independent storage. Therefore, changing any of these values do not affect the values of other

instances within the same target instance nor the corresponding instance values of other instances. The equivalent in the relational model are field values: each tuple (row) has independent field values. Certain semantic models define a specific node for these type of objects, e.g. "Printable" in IFO [2].

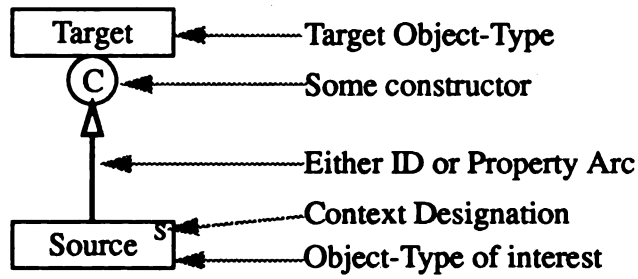
The interpretation for A is that of an abstract object, in which instances coexist under a strict set criterion, e.g. duplicates are not permitted. In the ER model, they are Entities and Relationships, and in the relational model, they are tuples and relations. The "gestalt" values of entity or tuple instances can not be changed without the consideration for constraints of set membership, cardinality (e.g. 1-M), etc..

Most data models have this separation between attribute and abstract values. However, in the Extensible Object Model, there is no distinction between attribute and abstract, i.e. all Object-Types are equivalent and behave the same. Although we provide the "implementation definition" of an Object-Type, this should not weaken the uniformity of the abstraction. Therefore, we need to define this problem in a broader scope, for which the attribute/abstract dichotomy is only a limited solution for it. Our solution for the extended problem is the concept of target dependency.

There are two equivalent interpretations for target dependency: identity and set generation. The first is based on the idea that the identity of an instance is dependent on its target. Instance identity can be phrased as the following question: when we are given two instances from a collection, how do we determine if they are different instances or the same instance? This is the core problem of equivalence vs. equality. The standard approach is "downward", i.e. object identity is determined by the value of the instance. This approach is taken by the relational model: a tuple's identity is determined by its key fields. It is also the approach taken by pure Object-Oriented models: an instance's identity is determined by the surrogate (virtual, system, hidden, intrinsic, or implicit) ID value. Target dependency introduces the concept of "upward" object identity, i.e. identity of an instance is not only based on its "lower" values, but also the "upper" values, defined by its target. This dependency is a form of context dependency, i.e. the context of the instance is needed to determine its identity.

The second interpretation of target dependency is the generation of sets. While instance identity interprets target dependency by "upward" direction, set generation interprets by "downward" direction from the instance's target. The basis of this interpretation is that every instance belongs to a strict set. However, there may be many strict sets associated with each declared Object-Type, with the multiplicity dependent on the target dependency designation. Since our concept of identity is value-based, the formation of strict sets ensures identity of instances within a strict set is always achievable. Furthermore, this explicit identity provides a useful integrity constraint that prevents the collision of instance values and, subsequently, identities.

Target dependency is a "vertical" concept, so our discussion is restricted to the Object-Type of interest and to its "target", i.e. the constructed Object-Type of which one of its sources is our given Object-Type. The following diagram describes the terminology:



8.2.1. Types of Target Dependency

We define target dependency with the following assumptions:

(1) It applies to the immediate target level, i.e. target dependency is not defined between Object-Types spanning more than one level of constructors.

(2) It applies to both ID and Property arcs without differentiation. This preserves the orthogonality between ID/Property differentiation from target dependency.

(3) The corresponding "downward" characterization, between a given Object-Type and its sources, is strictly value-based. This means the context of an Object-Type X, with respect to its target is independent of the context(s) of X's sources.

(4) Target Dependencies are not applied toward Inheritance nor Union constructors. Because the instance links of IS-A constructors are not of the "association" type.

There are three target contexts we use in the EOM, each will be defined with both identity and set generation interpretations.

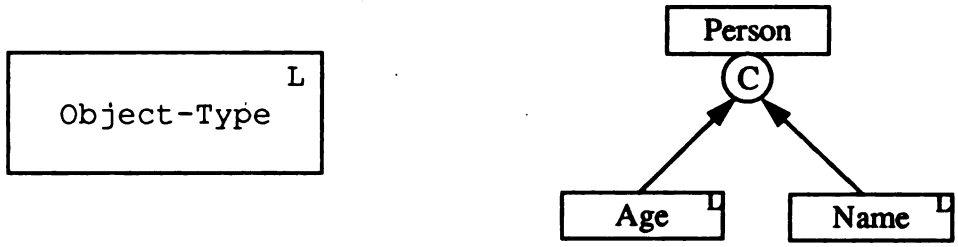
A. Local (or target-instance) context

Under local-context, two instances are identical if and only if they are value-equal and their target instances are identical, i.e. the given instances are local to their target instances. The target instance is the instance in the constructed Object-Type above the given source Object-Type in the schema. Using our Person example in Section 8.1.1 and assigning local-context to Person.Name, an instance of Name, $p_1.n_1$, is identical to another, $p_2.n_2$, iff ID($p_1.n_1$) = ID($p_2.n_2$) and $p_1 \equiv p_2$. In other words, two Name instances are identical if they have the same value and belongs to the same Person instance. The latter "equivalence" on Person instances is stronger than equality of the Person.ID#, because it forces a potential "upward" percolation of equality tests.

If the given Object-Type is of local-context, then the each instance of the target Object-Type defines an isolated set of the given Object-Type instances. This creates a set of sets such that the

instances of the given Object-Type is partitioned by the corresponding target instances. Attributes in the E-R model has this characteristic. By default, a declared Object-Type has instance context.

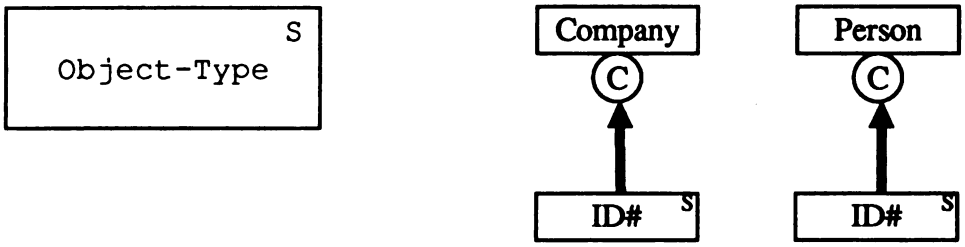
Example Usage:



B. Shared (or target-set) context

Here we overlook target instance differences, but recognize target instance sets, i.e. the given instances are shared among their target instances. In other words, instances are partitioned into sets based on the sets of their respective target instances. Therefore, if we assign shared-context to Person.ID#, then an instance, $p_1.id_1$, is identical to another, $p_2.id_2$, iff ID($p_1.id_1$) = ID($p_2.id_2$) and p_1 and p_2 are instances from the same set, i.e. Person. This is equivalent to defining a set of ID# instances for each set of target instances. If the schema defines another composition, Company, also made of ID#, then by this definition, the set of instances of Person.ID# is partitioned separately from Company.ID#. Thus, even if ID(p_1.id#) = ID(c_1.id#) for some Person instance p_1 and some Company instance c_1, the two ID# instances are not identical because p_1 belongs to a different set than c_1. This is commonly used to preserve one-to-one correspondence between a Composed Object and a singular ID component. An Object-Type with shared-context is shown as:

Example Usage:

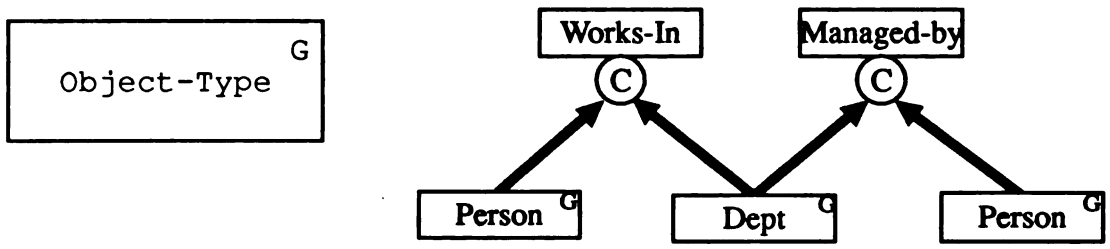


C. Global (or target-independent) context

In this case, ID value equality is object identity, i.e. no target information is required. This is equivalent of defining only one set of instances for all declarations of the Object-Type. For example, if every instance of Person came from the same set, regardless from which Person declaration it is chosen from, then the Person Object-Type is considered to have global-context identity. The E-R models uses this form of global-context identity for entities. This is also used for the EOM Object-Type definition from aliases, i.e. the alias type is considered to be global. We denote a global-con-

text Object-Type as:

Example Usage:

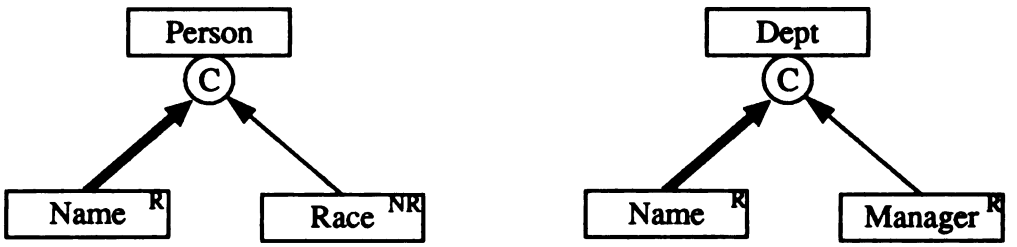


These contexts are analogous the scopes of variables in a high-level programming language. Local variables inside a subroutine is of local-context. Static variables, local to a module and common to a set of subroutines, are of shared-context. Global variables has global-context. In data models, the context of a data instance is not determined by lexical nor execution scope, but is determined by its relationships with other data instances. The instance association diagrams for the various contexts are shown in Appendix A.

8.3. Source Dependency

A converse situation to target dependency is source dependency, i.e. an instance existence is dependent on its source instances. This problem is similar to the relational model as null vs. non-null column designation. If an attribute or field in a relation is specified non-null, then every row in that relation must have a valid value for that field. On the other hand, if the field can be null, then there can exist rows without a valid value, or a "null" value, for that field. In the Extensible Object Model, a target's existence can be similarly dependent on the existence of a source, i.e. requires a source instance. We will denote them as follows:

R for Required (source-dependent), default is Non-required (source non-dependent):



In this example, a Person's Name instance is required, but Race is not a required instance. For every instance in the Dept Object-Type, both Name and Manager source instances are required. In general, Property source objects can be either required or non-required, while ID should be required. Unlike target dependency, source dependency is applicable to IS-A constructors. The instance association diagrams for the various contexts are shown in Appendix A.

8.3.1. Dependency vs. Null

Dependency and null address different issues. Null is a semantic problem while dependency is an integrity problem. The semantic problem is resolved in the Extensible Object Model by allowing semantically meaningful "null" values. If a null value is semantically meaningful, then it must have a symbolic representation, i.e. a unique bit pattern. Since equality on bits is defined, a null value is distinct from the any other values. Furthermore, there can be many distinct null values, e.g. "unknown" and "not applicable". Therefore, a null value can be used as a "normal" value, and it can also be used for ID value without losing the properties we have defined. This extension comes from the ability to add new tokens carrying new semantics to almost any model without loss of consistency. In reality, however, a null value must be operationally well-defined, because a set of values, as defined by an Object-Type, is paired with a set of operations. Although this requirement makes the construction of operations more difficult, it occurs outside the scope of the Extensible Object Model. In summary, in the Extensible Object Model, a null value is treated like any other "normal" value.

The integrity problem of source dependency is whether a target instance can exist without a source instance. It is determined by the presence or absence of the instance association (or IS-A) link between the target instance and the source instance. It is not determined by the values of instances. For Required source dependency, the link must be present and be sourced from a valid instance. For Non-required, the link can be absent.

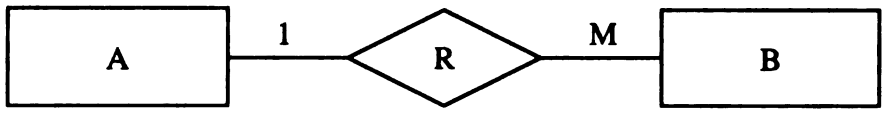
The null value problem in the relational model is a mixture of semantic and integrity problems. The reason is that the relational model is purely value-based, therefore the integrity problem of instance-to-instance relationships cannot be fully expressed. Our source dependency deals only with the integrity problem.

8.3.2. ER Participation Constraints

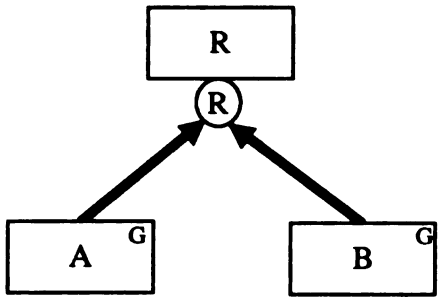
Mandatory vs. optional participation in the ER model has a similar interpretation to source dependency. If applied between an entity and an attribute, the entity is source-dependent on the mandatory attribute. If applied between a relationship and an entity, the relationship is source-dependent on the mandatory entity. However, ER has another constraint where the implication is reversed: a "total role" constraint is where the entity is "source-dependent" on the relationship [64]. Under this constraint, an entity instance can exist only if a relationship instance, associated with that entity instance, also exists. This is not target dependency, because it can be applied on global Object-Types. Nor is this source dependency, because of the reverse direction of dependency. The "total-role" constraint actually opens up the possibility for arbitrary dependency between unrelated Object-Types. Since the objective of the Extensible Object Model is to provide a core model, we will not pursue this extension.

8.4. Peer Dependency

If an Object-Type defines the actual instances of a database, we will need to determine which potential instances are permitted to be inserted into a database. Clearly, if we are to accommodate E-R relationships as a composition, we need additional characterizations for cardinality. For example, if we are given:



then it can be modelled as:



Comments:

(1) The Relationship constructor is an alias for the Composition constructor.

(2) When given an instance of R, r, ID(r) = [ID(r.a) , ID(r.b)].

In order to model the 1-M relationship between A and B, we need to restrict the instance set R by ID values of A and B. Unfortunately, the ID value of a single instance is insufficient to determine whether a given instance belongs to the set defined by the Object-Type.

This is the basis for peer dependency, i.e. the valid existence an instance is dependent on the existence of its peers. Therefore, a Composition constructor needs to include one or more peer dependency rules which operates "side-to-side".

8.4.1. Approaches to Peer Dependency

There are two basic approaches: functional dependency and involvement cardinality. To compare their differences, we use the "pigeon-hole" paradigm: with every instance added to the set, it occupies a new slot and covers a set of incompatible slots. To insert a new instance, therefore, we must find an open slot. If we don't find an open slot, then the new instance is either identical to an existing instance or in conflict with one. When given an instance x, we determine whether x has a valid existence in the instance set R through the following steps:

(1) We test x against r_1 , r_2 , r_3 , ... of the instance set. The test is determined by whether we choose functional dependency or involvement cardinality.

(2) The results of successive tests are AND'ed to obtain the final answer.

Thus if any current instance rules against insertion, then the final result is false. Although the test

is phrased as an insertion problem, it can be used as a static consistency check. For example, if we are given a set of r's, then we can check to see if this set is consistent with the peer dependency rules by inserting the r's one by one, in any order, into a new set with the same rules. If all can be inserted, then the set is consistent. If not, the set is inconsistent.

8.4.2. Functional Dependency

The first approach is functional dependency. We will use the previous schema for the binary case.

A. Binary Relationships

For 1–1 relationship between A and B, we have:

If x is an instance of R, then for all instances, r, of R in the database:

$$ID(x.a) = ID(r.a) \leftrightarrow ID(x.b) = ID(r.b)$$

For 1–M relationship:

If x is an instance of R, then for all instances, r, of R in the database:

$$ID(x.b) = ID(r.b) \rightarrow ID(x.a) = ID(r.a)$$

For M–M relationship, no restrictions are applied. However, we always have the fundamental "ID → Property" rule:

If r_1 and r_2 are instances of R:

$$ID(r_1) = ID(r_2) \rightarrow Property(r_1) = Property(r_2).$$

The logical predicates expressed here are equivalent to the functional dependency principle of 1–1 or 1–M relationships [64].

B. Ternary Relationships

We will let $x = [a_1, b_1, c_1]$ and $r = [a_2, b_2, c_2]$, where r ranges over $r_1, r_2, r_3, \ldots$. We now use a truth table to describe the results of all possible comparisons. The first three columns describe the result of the comparison of the component values between x and r. The next four columns describe the combined component result based on the functional dependency. For example, if we are testing for 1-1-1, then the result of x and r_1 (chosen from the 1-1-1 column) is AND'd with the result of x and r_2 (chosen from the 1-1-1 column) and so on.

This produces the following (comments in parentheses):

$a_1=a_2$	$b_2=b_2$	$c_1=c_2$	1–1–1	1–1–M	1–M–M	M–M–M	Comments
T	T	T	T	T	T	T	(1)
T	T	F	F	T(4)	T	T	
T	F	T	F	F	T(5)	T	
T	F	F	T(3)	T	T	T	
F	T	T	F	F	F	T(6)	
F	T	F	T(3)	T	T	T	
F	F	T	T(3)	T	T	T	
F	F	F	T	T	T	T	(2)

Comments: Let $A \equiv (a_1=a_2)$, $B \equiv (b_1=b_2)$, and $C \equiv (c_1=c_2)$.

(1) If all three components are equal, then $x = r_i$ for some r_i in R, thus insertion of x does not affect the instance set of R (by the definition of a set). However, this can be considered as invalid if Property(x) ≠ Property(r_i), then all values should be False.

(2) If x is different from any r already in the set R, then x can be inserted.

(3) Since 1–1–1 is defined by the predicate ((A AND B)→C) AND ((A AND C)→B) AND ((B AND C)→A). As long as there are two components that are different, x can be inserted into R.

(4) The functional dependency of 1–1–M is ((A AND C)→B) AND ((B AND C)→A). Therefore, this is True.

(5) The functional dependency of 1–M–M is (B AND C)→A.

(6) There is no dependency under M–M–M, therefore, any x can be inserted.

8.4.3. Involvement Cardinality

The second approach is by involvement cardinality, which is based on the principle that a source instance can originate association arcs at its given cardinality, i.e. be involved at the specified cardinality. Under this interpretation, the 1–M binary relationship would be the converse of its logical interpretation under functional dependency, i.e. ID(x.a) = ID(r.a) → ID(x.b) = ID(r.b). Since an instance of A can be associated with only one instance of R, this implies if two instances of R that have the same instance of A, they must be the same instance. Involvement cardinality and functional dependency interpretations for 1–1 and M-M relationships remain the same.

The ternary relationship under involvement cardinality produces the following table:

$a_1=a_2$	$b_2=b_2$	$c_1=c_2$	1–1–1	1–1–M	1–M–M	M–M–M	Comments
T	T	T	T	T	T	T	(1)
T	T	F	F(3)	F(4)	F(5)	T(6)	
T	F	T	F(3)	F	F	T	
T	F	F	F(3)	F	F	T	
F	T	T	F(3)	F	T	T	
F	T	F	F(3)	F	T	T	
F	F	T	F(3)	T	T	T	
F	F	F	T	T	T	T	(2)

Comments:

(1) Same as functional dependency.

(2) Same as functional dependency.

(3) Involvement cardinality of 1–1–1 is (A↔B) AND (B↔C) AND (C↔A). Therefore, if any sources are equal, all sources must also be equal. This will preserve the number of occurrences of that source instance in a target instance set at the specified cardinality of 1.

(4) 1–1–M is (A↔B) AND (A→C) AND (B→C). Therefore, this is False. The last predicate, (B→C), is redundant.

(5) 1–M–M is A→(B AND C), which is a deduction from (A→B) AND (A→C).

(6) Same as functional dependency, since there are no additional rules.

8.4.4. Other Uses of Truth Table

In any modelling process, one encounters the problem of equivalence between two schemata. When we have a schema with three or more ID source Object-Types, is there a mechanism to determine if this schema can be converted to one where the ID sources are cascaded? This is achievable because the pigeon-hole mechanism makes any peer dependency condition based on equality and boolean algebra into a truth table. Since we can enumerate all possible truth tables for a fixed number of components, we can determine whether or not a given composition is truly n-ary. It will also provide a decomposition, and possibly, a unique one. Naturally, if the peer dependency condition goes beyond equality, e.g. arithmetic counting and procedural, this method will not succeed.

In Appendix B, we examined all the cases of ternary relationships using the generalized truth table. Although half of the ternary compositions can be decomposed into cascaded binary compositions, there may not be a motivation to do so because of the semantics of Object-Type definition. A similar situation exists for choosing a particular cascade against an equivalent cascade. Ultimately, it is the user who will decide which one is semantically correct.

8.4.5. Approaches to Peer Dependency

The basic approach to peer dependency could be either functional dependency or involvement cardinality, with the arcs designated "1" or "M". The more general approach is to specify the truth table for a given composition. However, this may be too broad and may not be semantically meaningful to the user. That is, an arbitrary combination of T and F is consistent but is not as interpretable as functional dependency or involvement cardinality. On the other hand, this does not mean that other meaningful combinations do not exist. An extension of cardinality is to provide a number range for involvement cardinality. This approach goes beyond the truth table because it requires counting variables. We will not explore this issue as a part of the core Extensible Object Model.

The most general approach is an arbitrary function based on the new instance and the set of existing instances. This goes beyond the truth table because the operations are no longer based on equality and boolean algebra. This peer dependency function is interpretable only by "name", and, in general, it is not possible to determine if the function is computable or not. A "rule" or "trigger" mechanism for database updates is such an implementation.

Functional dependency does not generalize to Set or Sequence constructors, because there is only one ID source with these constructors. However, involvement cardinality still applies in these cases. Furthermore, functional dependency becomes more difficult to define when $n > 3$. Therefore, in this model, we will use involvement cardinality, and denote a singular involvement by a "1", otherwise it would be assigned "many" (M) by default. The instance association diagrams for the various contexts are shown in Appendix A.

8.5. Issues of Context Dependency Usage

8.5.1. Target Local-Context vs. Attribute Value

One may argue that local-context instances are not instances, but "attributes", as in the ER model. Thus they should not be considered as Object-Types and should be modelled as attribute values directly than some "instance" that has a given value. For example, a Person's Age should be a simple value rather than an "instance". The end result is similar to a primitive implementation definition, but it reduces the uniformity of the Extensible Object Model. It may be true that value storage is more compact and direct, but such optimization decision should be left to the schema compiler, just as current language compilers and query processors perform optimizations to generate "code" for the underlying system or model.

Target local-context also generalizes to Set and Sequence constructors. For the Composition constructor, a local-context component implies only one instance will exist under a target instance. This is why an attribute approach is applicable, but misguided. However, under the Set constructor, multiple instances can exist under a local-context for the ID component. In this situation, the attribute approach would lose the ability to reference the elements of a Set instance as instances.

8.5.2. Target Shared-Context and Labelling

Shared-context Object Types are usually an artifact of "labelling" (surrogate), i.e. the real world has constructed an identifying label for objects which have the same characteristics (values). This is usually achieved by analyzing context information, but because of the complexity of real world context information, it is simpler to reference the object through its label. For example, when two chairs have the same features (Property values), such as shape, # of legs, color, etc., and require two identities, we could use their locations or owners for identification, but these may change without affecting the chairs themselves. The solution is to use a sticker on the underside of each chair and write unique numbers to the stickers. The value on the sticker becomes our shared-context ID value. As long as the values are unique for chairs, they are adequate for identification, and the one-to-one relationship to the real world is maintained.

8.5.3. Definition vs. Declaration Context Dependency

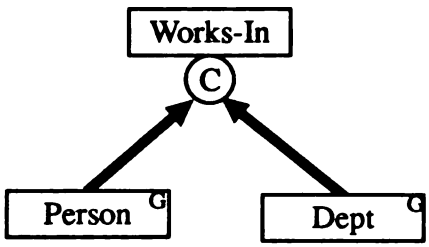
Can different context dependencies be assigned to different declarations of the same Object-Type in a single schema? Before adding context dependency, the definition of an Object-Type only dealt with the information content from a structural point of view. No assumptions were made about <u>where</u> the information is located. However, a database is dynamic, and we need a specification of information locality so that changes remain semantically consistent with the real-world and can be described by a static schema. Context dependency provides this specification in a compact and static form. If an Object-Type can have different dependency among its declarations, then the dependencies must be orthogonal to Object-Type definitions. Since there is only one definition per

Object-Type, the operations defined by an Object-Type must be independent of context dependency to maintain this orthogonality. If this requirement is satisfied, then one should be able to assign different dependencies to different declarations. This is the approach we took with the EOM.

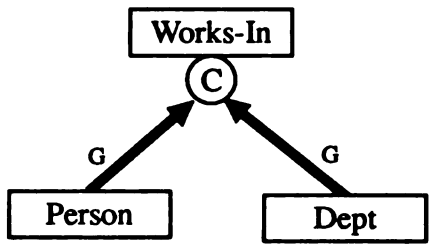
8.5.4. Placement of Context Dependency Designation

There are two places where context designations can be placed:

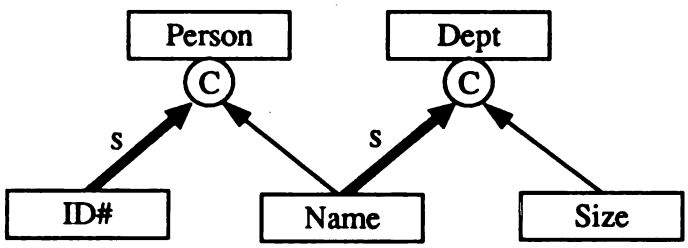
In the Objec-Type:



On the Arc:



Since the Extensible Object Model uses different arcs for ID and Property, so one would assume that context dependency relationships between target and source should also be placed on the arcs. However, if an Object-Type declaration sources two or more arcs, the interpretation becomes ambiguous. For example, if the designation is placed on the arc:



The declared Name Object-Type is used as local-context for Person, but shared-context for Dept. We can interpret this schema as a shorthand for a schema with two separate Name declarations, which then resolves the ambiguity by distinct arc annotations. But this schema gives the impression that Person and Dept are semantically associated by one collection of Name instances. If the context designation is placed in the Object-Type, then we avoid this false impression because we would be forced to use two Name declarations. This may look inconsistent with the ID/Property arc differentiation, but in fact it is not. The reason is that the ID/Property characteristic is orthogonal to context dependency. Furthermore, each Object-Type declaration can potentially represent different collections, in which case, the internal context designation is closer to the expectation. For example, a global Object-Type is not global from the "arc" designation, but from some intrinsic semantic properties of the Object-Type.

CHAPTER 9. OBJECT MODEL SUMMARY

9.1. Conceptual Basis

9.1.1. Basis for Object-Types

The Object-Type in the Extensible Object Model (EOM) was a distillation of concepts presented in semantic data models and Object-Oriented programming (OOP). The core concept was the information theoretic mapping between semantic concepts and physical implementations [58]. In many models, this was left undefined or implicitly stated. With an explicit definition, our model formed a clean separation between real world concepts and symbolic values (bits) stored in a database. This also determined which real world concepts properly belonged in a schema and which didn't.

The concept of an "instance" has taken shape in the database world independently of the similar concept in OOP. The development of OODBMS has since then merged the two efforts and strengthened its "right to existence" [36]. However, the problem of identity and its operations were addressed at the implementation or physical level, not at the semantic or conceptual level. This model brought identity and property into the conceptual level so that it can be related to other aspects of the conceptual level, such as structural abstraction and integrity. The explicit identity of EOM was a departure from the use of surrogate or virtual ID's espoused by many database implementations. However, they are not incompatible, because the use of surrogates is an implementation technique that should be hidden at the conceptual level. On the other hand, the notion of "identifying characteristics" is a "conceptual" abstraction.

9.1.2. Basis for Constructors

A schema defined how Object-Types are associated with each other through the semantics of constructors and ID/Property arcs. The constructors were based on several concepts. First was the use of data abstraction in programming languages [42]. This ability to encapsulate a collection of information units into a new unit was also crucial for semantic modelling. The explicit semantic mapping provided the basis for data abstraction: a semantic grouping of real world concepts is implemented by an equivalent grouping of database values. Second was the separation of type and instances. In the EOM, only types are shown in the schema. This maintained a constant "conceptual level" in which a user can model real world semantics. Finally, the graphic design was based on ER and other semantic data models, but the biggest influence was Jackson Design Methodology and its data structure diagrams [32].

The specific constructors were based on well-known data abstractions. Composition and set constructors were well characterized in previous efforts of relational and semantic data models. The sequence constructor was developed to support the biological applications, but it was based on the

results of the abstract sequence model (Section 2.5) and previously developed concepts of "text", "list", "vector", and "time". Our IS-A constructors, inheritance and union, were developed from specialization and generalization. In this model, the semantics of IS-A was defined by the use of instance association diagrams, which removed many ambiguities of multiple inheritance and overloaded attributes.

9.1.3. Basis for Context Dependency

Context dependency was introduced to extend constructor-defined Object-Type associations to the instance level. Context dependency was classified into three types: target, source, and peer. The objective of this classification was to construct a systematic framework for integrity constraints. At the same time, this framework removed the need of "references", "foreign keys", and "surrogates", all of which are implementation solutions to context questions. Therefore, they should be hidden by the proper use of context dependency. In addition, context dependency retained explicit value-equality as a mechanism for resolving instance identity. This preserved the ideal that instance identity is a semantic (real-world) concept that is accessible and can be directly modelled.

The instance association diagrams, which were used to illustrate context dependencies and abstract instance locations, also formed a semantic interpretation mechanism for integrity constraints. This interpretation model was an intermediate between the EOM and the actual database. Since it was essentially based on instances and sets of instances, it can be readily implemented *de novo* or on top of existing DBMS's.

(1) Target Dependency

The inspiration for local context was based on two concepts. First was the hierarchical model where each instance has its own set of children instances. Second was the concepts of value replication and independent modification. Global context was based on the global references of programming languages and the use of foreign keys in the relational model. Shared context was based on the use of module-local variables in programming languages, e.g. "static" variables of an ".c" file in C. It was further refined by the examples of real world 1-to-1 labelling of objects.

(2) Source Dependency

Source dependency was based on the problem of null vs. non-null in the relational model and the mandatory vs. optional participation in the ER model. We did not include a "total role" constraint because it would have permitted dependencies not specified in the local schema and between arbitrary Object-Types. On the other hand, this provided a new direction for future research.

(3) Peer Dependency

Peer dependency was based on a generalization of the cardinality constraints found in the ER models. Although this research has extended the original idea, we chose involvement cardinality

9.2. Goals of the Extensible Object Model

Part II started by reviewing some of the problems of conceptual data modelling and set forth a list of goals for a new data model: semantic richness, model extensibility, model simplicity, organized graphical representation, domain invariance, representation uniqueness, and implementation independence (see Section 5.4). We then presented the EOM with the following features:

(1) Object-Types with semantic mapping functions

(2) ID and Property characterization of Object-Types

(3) Constructors to build new Object-Types

(4) Context Dependency to define Instance-to-Instance relationships

Through the use of constructors and context dependency, this model has achieved the semantic expressiveness similar to EER, IFO, and OSAM*. A demonstration of semantic richness is detailed in Section 9.3. Because the organization of these semantic constructs are orthogonal, each can be applied uniformly and extended independently. For example, new constructors, such as Bag, Tree, and Semantic Sequences, can be added; constructor arc types can be extended with Ordered_By arc, as in Sequence constructor; and "total role participation" can be defined as reverse source dependency. Directions for extensions are less clear in the other semantic data models, since their framework was not as well defined as the EOM.

We also proposed a graphical notation for the model that is based on the orthogonal characterization. In addition, the notation is oriented top-down and left-to-right, which improves readability and lowers the possibility of "getting lost", which is commonly seen with the omnidirectional arcs associated with other data models (see Appendix C.1). The use of instance association diagrams is a form of instance semantics. Since the association diagrams are reflections of the Extensible Object schema, we now have an implementation independent view of information location.

With the separation of definition from declaration, the information captured by a schema is also localized, which reduces the amount of information one needs prior to comprehension. A model based on an orthogonal semantics, i.e. two object symbols + three arc types + three contexts, also reduces the number of interactions one has to learn in order to effectively use it. Therefore, this model has an additional advantage in simplicity.

Using the semantic information flow model, where type names are based on semantic mapping functions of encoding and interpretation, we have created a natural enforcement of language usage. Therefore, each constructor/arc/context semantic can be verified against the real world phenomenon. Certain guidelines are listed in Appendix C.

9.3. Comparison with Other Data Models

The semantic expressiveness of the EOM can be demonstrated by restricting its full expression and by mapping constructs from the other models. There are two aspects to this model mapping. One is construct equivalence where the constructs from both models are similar, which results in a schema with the same number of objects, i.e. there is a one-to-one schema object equivalence. The second is implementation equivalence where several constructs from one model are used to "implement" a specific construct in another model. Although, the information captured remains the same, the new objects require new integrity enforcements. Only the first aspect will be considered to be a demonstration of semantic expressiveness. We now map the features of the traditional and semantic data models using the constructs from the EOM.

The relational model can be modelled by only using the Composition constructor. In addition, all source components to the Composition constructor is of "primitive" implementations, e.g. integers, floats, and strings. The ID components correspond to the key attributes in the relational model. Target dependency is always local, peer dependency is always singular, and source dependency is either required or non-required depending on the relational null or non-null specification, respectively. This is construct equivalent, because a relational schema expressed in the EOM has the same number of "relations" and "fields" (Object-Types). The following is a schematic view of the mapping:

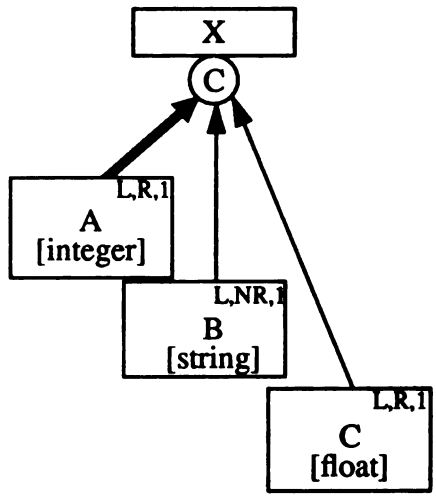
Relational Model: Extensible Object Model:

Relation X

Attribute A: Integer, Key, NonNull

Attribute B: String, Null

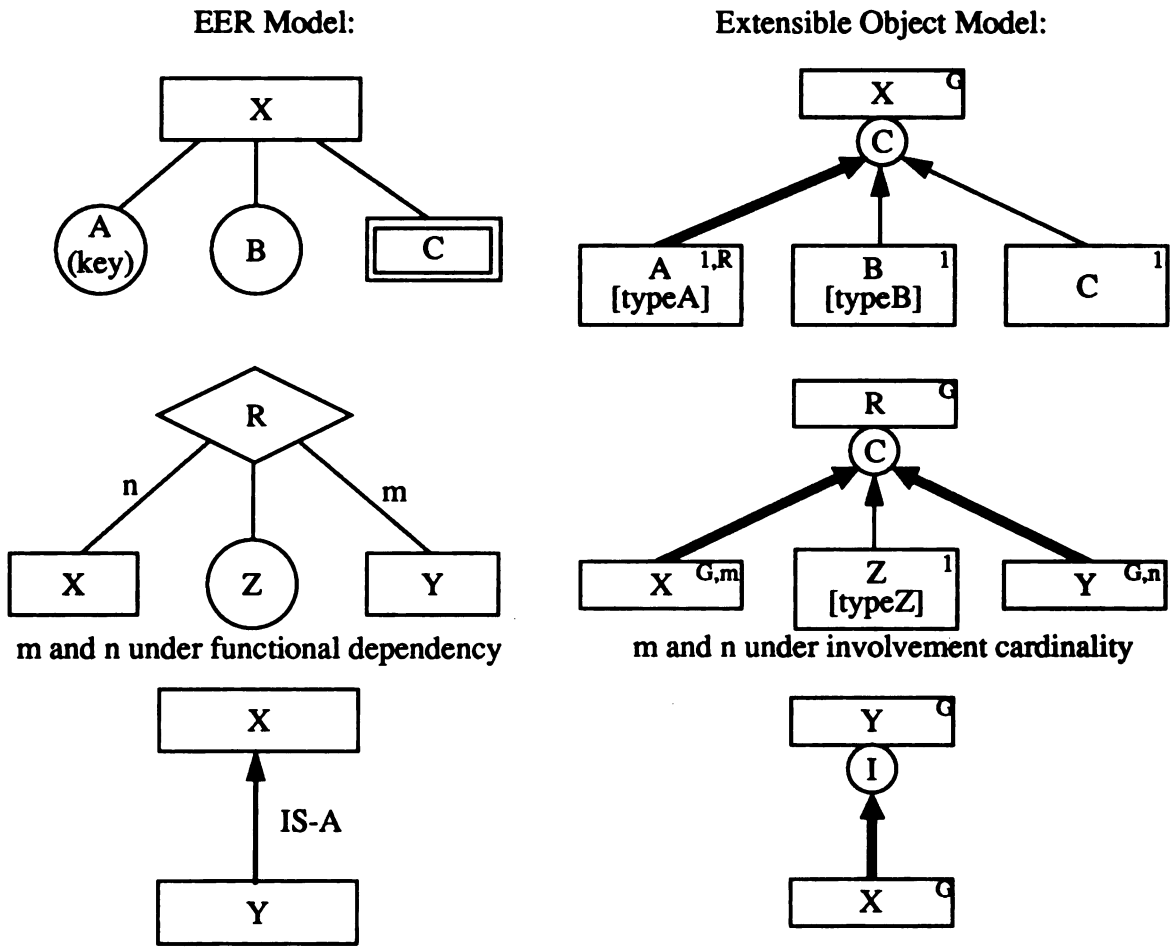
Attribute C: Float, NonNull



For extensions to the relational model, such as NFNF, we add the equivalent of the set or sequence construct. Therefore, the EOM subsumes the relational and NFNF models.

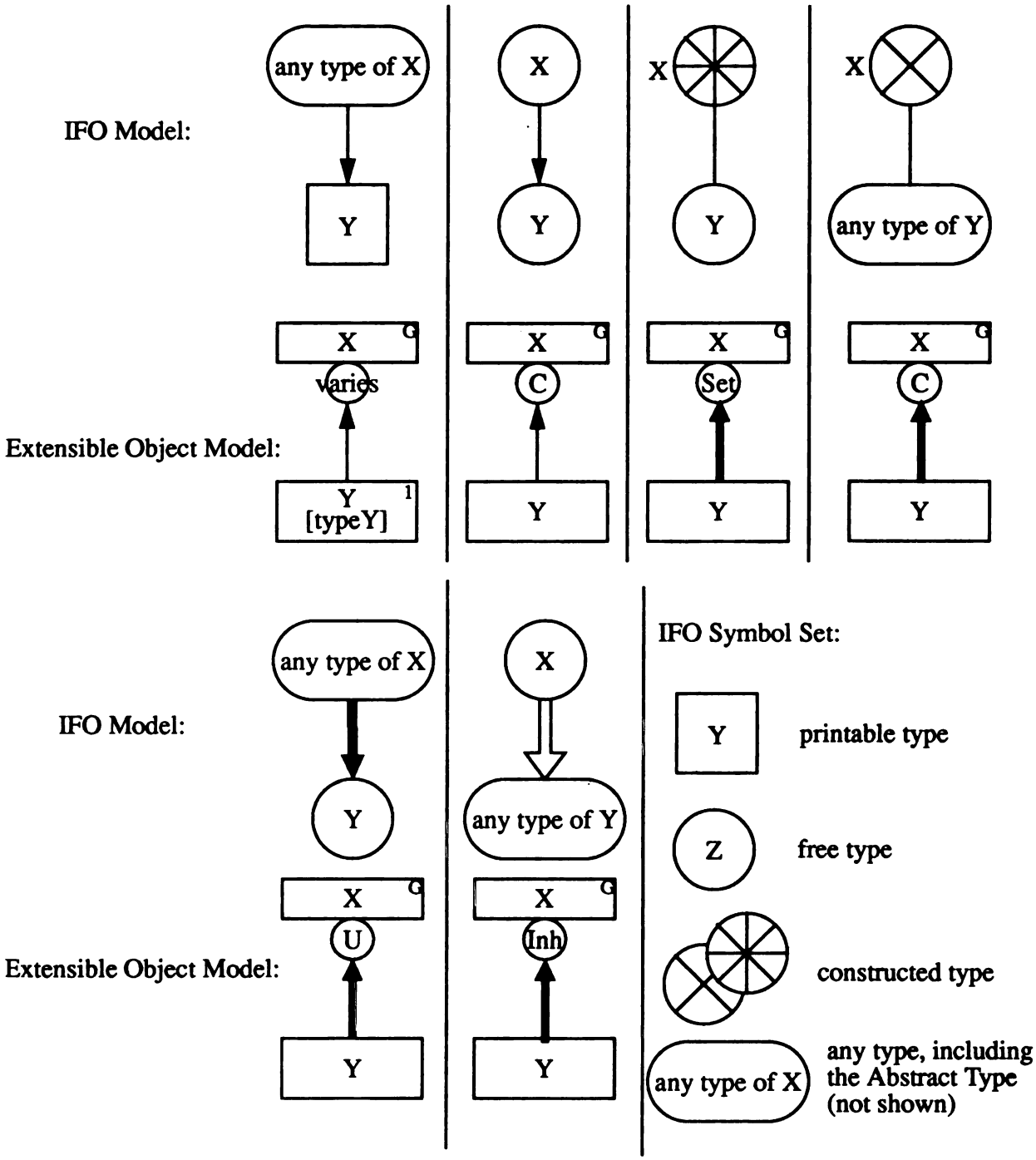
Similarly, the Extended Entity-Relationship model can be described with Composition and Inheritance constructors. Composition is applied in two situations. The first situation is Entity-Composition. It is used to model ER Entity, which is identical to the Composition in the relational model.

The second situation is Relationship-Composition. It is used to model ER Relationship, which has ID sources that are Entity-Compositions. To model Weak-Entities, simply attach the Weak-Entity-Composition as a Property source. For ER cardinality specification, we convert them into the corresponding peer dependency and for ER participation specification, source dependency is used. The direction of inheritance in EER is "reversed" with respect to the EOM Inheritance constructor, i.e. supertypes are sources to the Inheritance constructor. The following shows the mapping between the two models:



In newer EER models, relationships can be built from other relationships using a merged entity/relationship symbol, but this is handled by a cascade of Compositions in the EOM without additional symbols. If the ER model chooses functional dependency for ternary and higher-degree relationships, then the EOM will have to extend the simple involvement cardinality to the generalized constraint derived from the truth table for source dependency (see Appendix B). Other ER characteristics that would require EOM extensions are: range declaration for cardinality instead of the simple one or many and total role participation.

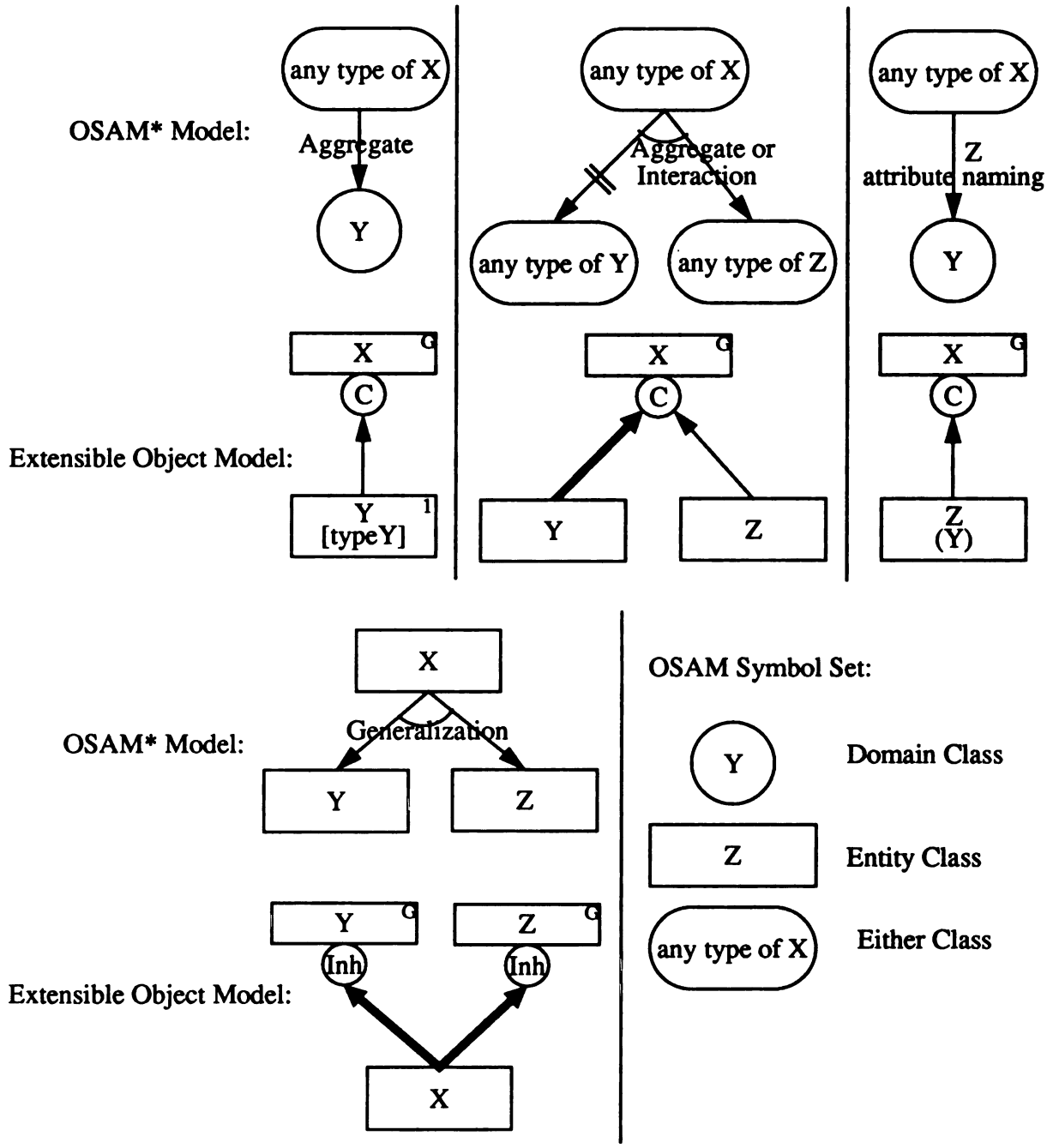
For the IFO model, the mapping from the EOM is still straight forward:



All features of the IFO model can be properly mapped by the EOM.

The mapping of the OSAM* model [62] to the EOM is limited to a subset of constructs. Direct support is available for Aggregate, Generalization and Interaction Associations. The following dia-

gram demonstrate the equivalent constructs:



OSAM* also supports the set association, which is equivalent to the Set constructor (not shown). For generalization, OSAM* provides Set-Exclusive and Set-Equality characterization which are not supported in the core EOM. In addition, several OSAM* constructs are not available in the core EOM: OSAM* Composition, which is an explicit instance-to-type classification, and Cross-product, which is a statistical aggregate.

Object-Oriented data models are based on existing OODBMS's. Therefore, in the sense of imple-

mentation independence, they are not conceptual models. However, the constructs available from the underlying language, namely C++ libraries [61] and Smalltalk Classes [27], are very useful for conceptual modelling. A subset of these constructs can be mapped to the EOM constructors, namely Composition, Set, Sequence, and Inheritance. Other OODBMS constructs that are not supported in the core EOM are: Bags, Queues, Stacks, and Trees. However, these can be provided by extensions to the EOM (see Section 9.5.6).

Some standard constraints can be declaratively stated in OODBMS's, e.g. cardinality, but in most other cases, they are user-defined procedures added to the DBMS. OO philosophy is based on the encapsulation of both data and operations, therefore, as one defines a class, equivalent to the EOM Object-Type, one includes the operations which manipulate the instances of that class and also maintains the integrity of the instance collections. The EOM allows this operational definition in a schema by abstracting them as an Object-Type whose implementation is "procedural". Because our constructors are data definitional, there is no support to further define a procedure at the schema level, i.e. specifying the actual code. But at the query language level, one can fully define the operational specification of a procedure (see Part III).

9.4. Differences From Other Models

The strengths of the EOM are based on the unique features described in Section 5.4. The following table summarizes the differences between the EOM and other semantic models. The categories are Abstraction, Constructors, Constraints, and Graphical Representation. Abstraction covers model features such as extensible framework, model simplicity, implementation independence, and domain invariance. Constructors and Constraints are aspects of semantic richness. Graphical Representation deals with comprehensibility of a schema. Clearly, some models carries additional features not listed.

Characteristics	Extensible Object	Extended ER	IFO	Object-Oriented	OSAM*
Abstraction:					
Extensible Framework	✓			✓	
Model Simplicity	✓			✓	✓
Implementation Independence	complete	mostly	mostly	limited	mostly
Domain Invariance	Collections	Collections	Collections + Instances	Collections	Collections + Instances
Existing Constructors:					
Composition	✓	✓	✓	✓	✓
Set	✓		✓	✓	
Sequence	✓			✓	
Inheritance	✓	✓	✓	✓	✓
Union	✓		✓		✓
Existing Constraints:					
ID/Property	✓	✓	✓		✓
Cardinality	✓	✓			✓
Contexts	✓			✓	
Graphical Representation:					
Nodes	unique	unique	unique	none	unique
Arcs	unique	ambiguous	unique	none	unique
Simplicity	✓	✓			

9.4.1. Extensible Framework

Extensibility provides the ability to extend the model in an organized fashion. In the EOM, we have grouped certain semantic features into categories and "load" them onto distinct representations. For example, the constructor carries the "what" information and the context dependency carries the "where" information. They form two orthogonal categories, each are extensible independent of the other. This is missing in other data models where constructs of different semantic nature are developed in *ad hoc* fashion. Consequently, without a framework for organized extension, these models are "characteristically closed", i.e. extensions are made in *ad hoc* fashion. The extensions of the EOM are based on the two semantic categories.

(1) Constructors

In other models, when a new semantic association is to be added, the typical approach is to add a new object type, e.g. entity and attribute. This creates a consistency problem because one needs to verify the interaction between the new type and the existing types in the model. Our model resolves this problem by defining a new constructor, whose instances represent the information captured by the new association, independent of the previously defined constructors. Once this is achieved, instances from this new Object-Type behave like all other instances, free to participate in any other constructors.

(2) Contexts

In other models, constraints are designated between object types. However, no framework is given with the constraint types. Therefore, it is difficult to add new constraints in a systematic fashion. On the other hand, we have classified the constraints into three distinct context groups: target, source, and peer. Technically, only target context is true context designation, while source and peer contexts are constraints. Nevertheless, future constraint extensions can be added along any of the three groups, independent of each other.

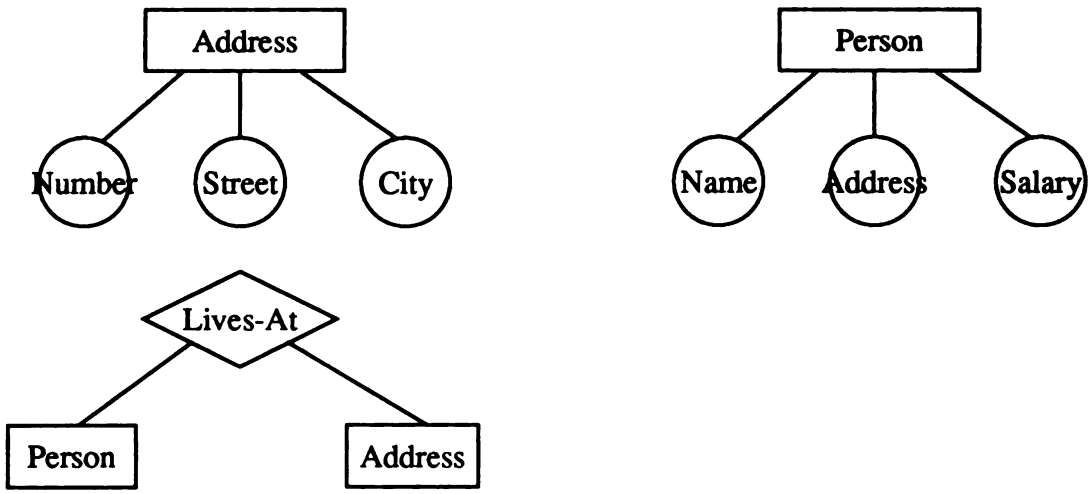
OODBMS present a special case for extensibility. Native OODBMS normally do not support the Union constructor. Therefore, to extend an OODBMS with the Union constructor would be equivalent to constructing a Bag or Queue constructor, i.e. these are really implemented in a class library and is not a feature of the underlying language. In fact, only Composition and Inheritance constructors are provided as native language features, everything else is implemented. This method of extension is just as powerful as our framework.

9.4.2. Model Simplicity

Model simplicity is based on orthogonality and uniformity. Clearly, a benefit of orthogonality is the extensibility framework discussed previously. However, the main advantage is still user comprehensibility achieved through uniformity. This can be demonstrated by the following ER example. The ER model has nonuniform semantics and rules for construct adjacency, e.g. a full ranked

entity (non-weak) can not be attached to another entity without going through a relationship (see Section 5.3.3). For example, an Address can be considered as an entity, composed of Number, Street, and City. On the other hand, it can be considered as a string attribute to another entity, e.g. Person. In the former, it is a complex object, in the latter, a primitive object. In the ER models, the entity Address can only be attached to Person entity via a relationship, while the attribute Address can be attached directly. This forces the schema designer into choosing one form or another:

Address as Entity: vs. Address as Attribute:



The EOM eliminates this dilemma by allowing the schema designer to consider an Address as an abstract object. then the decision for choosing which schema to use is based on the semantic factors in the domain, i.e. whether the relationship between Person and Address is containment (composition) or peer association (through Lives-At), and not by the non-uniformity of model construction.

Other semantic models disallow specific construct-pairs because they also lack uniformity. This introduces complexity into the model, because each special case must be defined separately. OO data models are uniform, but they occur at the implementation level. Therefore, the implementation-driven problem of the Address example is still present. For OSAM*, uniformity came at the cost of orthogonality. While all the constructors can be uniformly paired, there was a proliferation of similar constructors, e.g. aggregation and interaction. Consequently, it became difficult to find the appropriate constructor to use.

9.4.3. Implementation Independence

With explicit semantic mapping between physical implementation and domain concepts (see Section 6.1), the EOM maintains itself at the conceptual level. In other data models, some link with the logical or physical level always remained through the use of printable types or primitives. This detracts from focusing on the real issue of capturing conceptual semantics of the application domain. Furthermore, this forces a user to prematurely commit to an implemented type. The EOM

does not suffer from either of these side effects.

An additional benefit of implementation independence is evolvability and maintenance. Data models with links to the underlying system are much more difficult to evolve or maintain as either the underlying system changes or when the domain semantics changes. OO data models, by the virtue of being logical models, suffers the most when one needs to migrate a schema from one database format to another.

A consequence of the semantic mapping is the elimination of the distinction between "attribute" and "entities". In most models, a distinction is made because it follows the predominant paradigm of mental modelling, which is one of dominant/dependent pairing. However, we took another approach because we needed a uniform model, and there wasn't sufficient justification for the distinction. Shifting the paradigm away from the dominant/dependent conceptualization to one of information flow, the EOM permits completely uniform usage of constructs. As one can see, the decisions made in one area, e.g. implementation independence, also have an impact in other areas, e.g. model simplicity.

9.4.4. Domain Invariance

Domain invariance in the EOM determines which properties of the domain are to be modelled. This ensures the consistency of the schema constructs and their interpretation. This is not seen in instance-oriented data models, e.g. OSAM*, where both instances and collections have representations on the same schema. In addition, uniform instances and types allow Object Types, as representations of instance collections, to be arbitrarily associated with other Object Types without restriction. This is not observed in other data models that also have "constructor"-like features, but lacks explicit domain invariance.

9.4.5. Semantic Richness

The number of constructors and constraints determines semantic richness. For the applications in the HGP, our core model is sufficiently rich. The entries in the table indicate that it is a superset of most other models. Clearly, other models could also be extended to include features in the EOM, but without a framework for extensions, it is less clear of how uniform the extensions will be.

9.4.6. Representation Uniqueness

Semantic uniqueness is controlled by the definition of model constructs. Since our model was based on a perspective of information capture and flow, it is very specific about how a schema should be constructed. For example, most semantic models, e.g. ER, separate composition from relationship, but the EOM does not. The reason is, under the information flow perspective, a composition is simply an information container box, holding the association information of its sources, and a relationship is also a container box, holding the association information of its sources. Therefore, the behaviors of the two constructs are identical and the distinction is not made in the model

itself, but left to the user's interpretation. Furthermore, with separate constructs for composition and relationship, other semantic models requires additional distinguishing definitions, which unnecessarily complicates the model.

These considerations have an impact on the graphical form of the model. Since the EOM is a simpler model, the symbols used have unique meaning. In other data models where symbols are overloaded, there will always be ambiguities that must be resolved by context information. However, it is often the user who is faced with this task of collecting the sufficient amount of the context information.

9.4.7. Feature Summary

There is no single feature that distinguishes the EOM from others, instead, it is the gestalt of all these features that makes this model unique and powerful. Currently, it also contains certain weaknesses. For example, it lacks a verification for mathematical completeness and lacks a sufficient set of support tools. But these can be remedied with future research and development.

9.5. Open Topics

There are several open topics for the EOM.

9.5.1. Classification

Classification is the explicit formation of types from instances. Traditional data models, e.g. hierarchical, network, and relational, are collection-based (or type-based), i.e. schema objects reflect collections of instances and specific instances are not denoted. Therefore explicit classification of instances is not supported in these models. On the other hand, OODBMS and certain semantic data models, e.g. OSAM* and SDM [28], are instance-based, i.e. explicit instance references are provided and classification is supported. The EOM Object-Type is based on the domain invariance of collections and therefore, lacks explicit instance references. In addition, if Classification is supported in the EOM, we will have to introduce a new symbol for instances and a schema will no longer have uniform abstraction. Furthermore, if a model provides the ability to cross the instance-type boundary at one level, then it could also cross multiple levels for full generality. This opens up the problem of Metatyping.

9.5.2. Metatyping

Metatyping is the formation of types based on a type-description schema, i.e. a metatype, which represents a collection of instances where each instance is a Type. If the data model provides Classification, then Metatyping is supported, because the "instance Types" are explicitly denoted and grouped into Metatypes by the use Classification. The problem of invariance is now carried to the Meta-Type level. If the database, by necessity, spans several levels, then the choice of the abstraction level for its schema may not be freely decidable. For example, a database for taxonomic classification of biological organisms may fluctuate in both instances, types, and metatypes. In other words, none of the three are invariant within the domain. Therefore, a schema for this database should be at the highest abstraction (or meta-metatype) level.

Strict metatyping is supported by some OOPL, but not supported in database data models. Consequently, several concepts have been used as substitutes: IS-A hierarchy, subject-term hierarchy, and parametric typing. These provide various amount of organization for metatype relationships. Subject-term is an implicit classification method to form types and type-level characteristics, e.g. the partitioning of organisms into kingdoms, phyla, etc.. Parametric typing allows structurally similar types to be collectively referenced. But none of these provide the true semantics of metatyping. For more details on metatypes and inheritance, see [6].

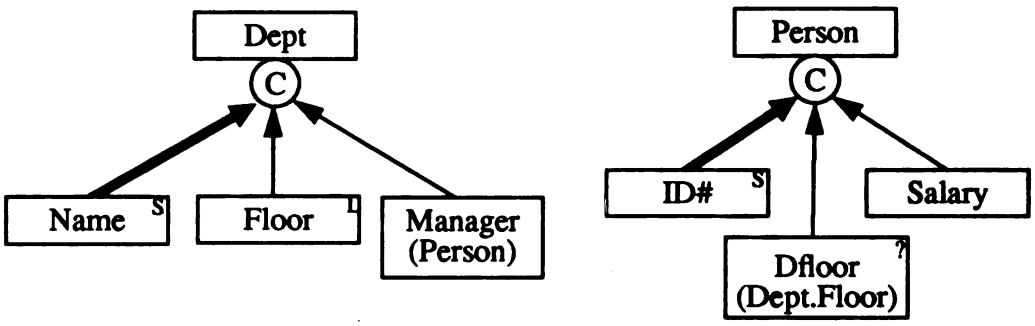
9.5.3. Overloading and Naming

One extension to Object-Type naming is to allow overloading. In the EOM, because Object-Type names defines the semantic mapping of real world concepts to symbolic values, overloading was

disallowed (see Section 7.5.3). However, overloading is a concept strongly defended by OO paradigms because of two major arguments: first is the limited natural language expressions for unique objects, and second is the ability to resolve similar names by context information. If a conceptual model is to provide real world semantics by using natural language constructs, then overloading should be added to the EOM. On the other hand, if a conceptual model is to bridge real world semantics and database implementations, then it would not be necessary to use overloading. The EOM can be extended to support name overloading, using the traditional name resolution strategies [61].

9.5.4. Extended Target Contexts

We have specified three target contexts. However, there may be situations where an unrelated Object-Type is a source component. For example, in a schema, one can attach Dept.Floor under a Person as Property:



But where does the Dept.Floor instance comes from when the user accesses the instance Person.-Dfloor? Since Dept.Floor is localized to the context of Dept instances, what should be its target context under Person? When this type of schema is mapped to the instance association diagrams, source instances to a constructed instance may refer to an instance that is not global nor related to the constructed Object-Type. In the core EOM, it should be the Dept that is directly attached to the Person with global target context, then there is no ambiguity when Person.Dept.Floor is accessed. Semantically, this is the correct schema. On the other hand, Dfloor may be a useful shorthand for generating schema projections. The nature of this extension has not been worked out and is reserved for future research.

9.5.5. Constraint Arcs

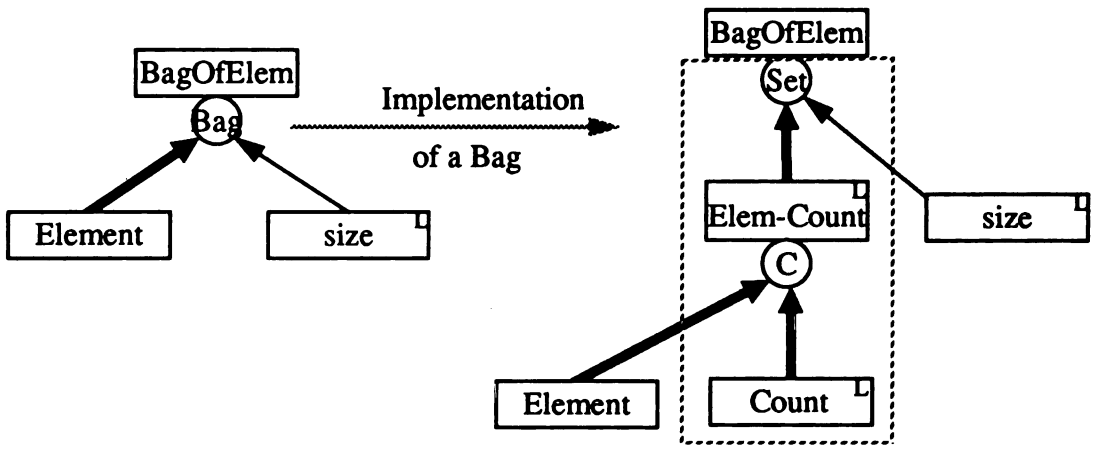
In addition to ID and Property arcs, we used the Ordered-By arc for sequence constructors. This is a special case of a constraint arc. A future extension would change Ordered-By arc to a generalized constraint arc, where it can be sourced from any other Object-Type. For example, an Ordered-By arc that is sourced by an object not directly connected to the Sequence constructor. It is an open question whether orthogonality can be maintained, i.e. is there a combination of contexts that pre-

clude a constraint arc? This is similar to the previous problem of attaching "unrelated" Object-Types.

9.5.6. Additional Constructors

Now that a framework for building constructors has been erected, we can explore other constructors. Certain constructors are implicitly defined by our core constructors. For example, a Relationship constructor is subsumed by the Composition constructor and the Subset is subsumed by Inheritance. However, if relationship or subset constructors have clearly distinguishable semantics, then these can exist as independent, primary constructors.

Certain constructors can be modelled by using the core constructors. For example, a Bag constructor can be built from a Set of a Composition, where one element of the composition is its multiplicity and the other element is the actual ID component to the Bag:



Therefore, a method for constructor extension is by defining partial schema, i.e. a new constructor is instantiated by placing its source objects in the places defined by the partial schema. Although some constructors can be defined by partial schema, it may be simpler to consider them as primary constructors. For example, Queues and Stacks can probably be considered as primary than structures containing a mixture of Sequence and Composition constructors.

There is also a need for domain specific constructors. For example, a statistical cross-product or subject-term constructor will be useful in population and classification studies. Constructors that relate spatial values would be useful in CAD/CAM databases. These are reserved for future research.

9.5.7. Cyclic ID's

In Section 7.8.3, we concluded that Object-Type cycles on ID arcs are ambiguous and undefinable. However, if we now consider the possibility that the ID of an instance is determined by the path (graph) of the instance association links, then the cycles are permitted. In this extension, identity

of an instance is graph based and can be uniquely determined. If so, then ID() has to be redefined as the instance graph that is traced out by descendent components and instance association links (both associative and IS-A). This possibility remains to be analyzed.

9.5.8. Instance Equivalence → Property Relationship

The Fundamental Relationship is based on ID equality to determine Property equality. We can change this relationship to instance equivalence instead of ID equality. The resultant relationship will permit instances with the same ID value, but with different identities due to different contexts, to have different Properties. This extension addresses the problem of assigning unique annotations to sequence and set elements.

PART III. EXTENSIBLE OBJECT QUERY LANGUAGE

Part III of this work will be concerned with a demonstration of the Extensible Object Model. We will implement a query processor for a language, named the "Extensible Object Query Language" (EOQL), that features the main properties of the EOM. Chapter 10 discusses the design criteria for the query language and its processor. Chapter 11 is a user reference manual for the query language.

CHAPTER 10. DESIGN OF A QUERY PROCESSOR

A query language provides access and modification to a database based on the data model. The data model provides a certain amount of abstraction, which removes the user from the detailed knowledge of the database implementation. The query language should also maintain this abstraction and information hiding. This goal is identical to any other computer languages. For example, assembly language replaces address references with labels and effectively hides the need to know about absolute addresses. Higher languages, such as Pascal and C, hides the details of control flow, space allocation, and register arithmetic. A query language for EOM should hide the implementation details of the constructors and context dependency.

There are many aspects to a query language design [6]. However, in the context of this work, we are only interested in providing a core language that demonstrates the main features of the EOM. Therefore, the design criteria we chose may not be the ones to use for another environment, e.g. commercial. A query language has two components: data definition and data manipulation.

10.1. Data Definition Criteria

All query languages provide a syntax to specify schemata for databases. Although the EOM is specified graphically in Part II, a lexical grammar will be given. In particular, the EOQL will support the specification of:

(1) Definitions by Implementation and Alias,

(2) Definitions by Constructors,

(3) ID and Property Components, and

(4) Context Dependencies

10.1.1. Model Extensions

Since the EOM is an extensible model, the language should also be extensible. Therefore, the grammar must be organized so that new constructors and contexts can be added to the language gracefully.

10.1.2. Storage Directives

In Part II, the conceptual model was explored in depth, from which a location abstraction was

defined through the use of instance association diagrams. However, there is insufficient information to map the location abstraction to real, physical database locations. This mapping is important for performance optimizations. Therefore, in the EOQL, the language is extended with storage directives. In real world databases, these directives would include access methods, indices, explicit clustering, etc.. However, we will only provide three storage directives to be assigned to constructor components, similar to the assignment of context dependency. These directives are orthogonal to the EOM constructs and they are listed as follows:

(1) Direct

The information content of the component instance is stored with the target instance. In programming languages, this is equivalent to packing a value in a record, i.e. the field in the record holds the actual value.

(2) Indirect

The information content of the component instance is stored elsewhere, i.e. independent of the target instance. In programming languages, this is equivalent to packing a "pointer" in a record, i.e. the field in the record holds a reference to the actual value. In relational databases, this is similar to the use of a foreign key.

(3) Virtual

The information content of the component instance is not stored in the database. It is created from context information by the query processor or by procedures.

The EOQL grammar will include these storage directives for the underlying system. Since these directives are part of the implementation strategy, they must be separated from the conceptual schema. Grammatically, this can be done by separate statements or by enclosing storage tokens in special markers.

10.2. Data Manipulation Criteria

The data manipulation component of a query language has to handle changes in the state information of a database and manage information flow for the user in a local session. Four main criteria stands out: declarative nature, state information, control flow, and operations.

10.2.1. Declarative Nature

Since the development of SQL for the relational model, other query languages have been compared to its declarative approach that allows the user to specify "what" a query is and not "how" to do it. The main aspect to this declarative nature of SQL is the abstraction of the selection process. SQL provides a single selection statement: "SELECT ... FROM ... WHERE ...", which hides the details of access methods and selection strategies. The advantage of this approach is that storage implementations are effectively hidden from the user and the maintenance of applications is simplified when the underlying storage methods or strategies are changed. The EOQL will also use the declarative approach, by providing an abstraction for the selection process. However, the EOQL SELECT command is not considered as an operator which returns a value (set of tuples), therefore, it could not be used in a nested selection.

10.2.2. State Information

In conjunction with the declarative approach, SQL also removed dependency on state information inside a query. The virtue of stateless query is the fact that it can be invoked anytime. However, this removal of state information places two burdens on the user. The first burden is that a user need to represent a query strategy as a stateless SQL statements. The second is, in some situations, domain complexity may force the user to reconstruct state information by using temporary tables and tuples. Consequently, many SQL extensions include variables to hold state information [30,63]. The EOQL will also provide variables.

10.2.3. Control Flow

Secondary to state information is control flow, i.e. the only useful purpose for state information is to have an impact on the next operation. Basic SQL does not provide control flow, but many extensions do. The EOQL will also provide basic control flow capability. Control structures allow the user to specify "how" and not "what" and, therefore, this seems antagonistic to the declarative approach. In fact, they are independent features, because the control flow specification occurs at a level above the access methods, i.e. control flow in the EOQL is domain driven, not implementation driven. In addition, control structures provides a popular model for users to phrase their domain-specific queries.

10.2.4. Operations

The simplicity of the relational model permits all the operations to be defined on the "relation" type

alone. Consequently, the result of one operation can be used in any other operation, e.g. one can embed a SELECT command whereever a tuple set (relation) is used. Since the EOM has different constructors, operational uniformity of the relational model cannot be enforced, i.e. there are operations that are meaningful in one constructed type but not in another. In fact, each constructor defines a set of generic operators for any instances derived from that constructor. These constructor-specific operators are well-defined and their implementations can be hidden, but they cannot be used in arbitrary combinations.

A core set of the constructor-specific operators will be provided by the query language. In addition, the EOM allows each Object-Type to specify operations from the domain semantics, which are outside the model. The query processor will provide hooks to access these domain-specific operations.

10.2.5. Strongly Typed

The EOQL will be strongly typed, i.e. every statement is verified for consistent type usage prior to execution. Although we will not pursue the development of a query compiler, type checking is one of the fundamental components to building a compiler. The advantage of type checking for a query interpreter is the prevention of potentially dangerous results.

10.3. Implementation Criteria

The previous design criteria are for the query language itself, i.e. the features that language should have. However, it is the query processor that implements the operations defined by the language and it is the query processor that changes information in the database. Associated with design criteria of the EOQL, there are also design criteria for the Extensible Object Query Processor (EOQP).

10.3.1. Physical Independence

The EOM is a conceptual model and consequently, EOQL should be a conceptual language. Therefore, EOQP should remain independent of the implementation of the database. In fact, with the abstract location defined by instance association diagrams, one can build EOQP on top of any set and tuple-based database engine. For this particular demonstration, we have chosen Sybase DBMS with binary large object support as the underlying database engine. However, the code for EOQP should be designed so that one can replace the Sybase DBMS with another relational engine or an Object-Oriented database engine with minimal code reworking.

10.3.2. Domain Specific Operators

Since the EOM permits domain specific operators, the query processor must be organized so that users can define their own operations and integrate it into the QP. It is assumed that the user-defined operators are robust and free of side-effects because it is nearly impossible to verify this requirement independently. Therefore, the user is responsible to maintain and correct any errors in their operators.

10.4. Design Limitations

Because this implementation is just a demonstration of the EOM, not all features associated with a complete query language will be available. The limitations are listed below.

10.4.1. Constructors

Only Composition, Set, and Sequence constructors are implemented because of their direct applicability to the HGP. Inheritance and Union constructors are reserved for future development. In the EOQL, one can specify Inheritance and Union constructions. However, they can only be used as Composition constructions, i.e. the semantics of IS-A is not implemented.

10.4.2. Primitive Types

Only a few primitive types are supported: integer, float, and character string. In addition, user specified enumerated types are supported. Other types, e.g. boolean, date, and time, are reserved for future development.

10.4.3. Types and Storage Directives

The supported interaction between types and storage directives is listed in the following tables:

		Storage Directives		
		Direct	Indirect	Virtual
Component Type	Primitive	✓		✓
	Enumerated	✓		✓
	Procedural			✓
	Constructor	(see text)	✓	✓

ID component to Set and Sequence constructors can be Direct if the ID component is of fixed size and target dependency is Local. Set or Sequence Object-Types whose the ID component is target Global and peer One are not supported. These are secondary to the current implementation of Set and Sequence constructors. In addition, context dependency are not verified for virtual components.

10.5. Backend Limitations

The choice of Sybase DBMS with binary object support as the underlying system also creates some limitations.

10.5.1. Object-Type to Relational Mapping

The mapping of constructed Object-Types is straightforward:

(1) All non-Virtual components are mapped to a column, using the same name. For the Set and Sequence constructor, the ID component is mapped to a Sybase Image data type with the name "_set" and "_sequence", respectively. Primitive data types are mapped accordingly. Components which are constructed are mapped to their object id (integer).

(2) An Object_ID and Context_ID column are added. They will hold the object id and context id (for Shared context), respectively.

(3) If the constructor is of an IS-A type, the a Valid column is added. This determines which ID sources are present.

(4) If the constructor is Set or Sequence, a Length and Hash column are added. The Length column holds the count of elements in the set or sequence instance. The Hash column holds a hash value for faster lookups.

10.5.2. Object_ID and Context_ID Generation

Since Sybase DBMS does not provide unique row id or tuple id, we are forced to create and manage our own ids. A separate ID daemon is built to supply ID's for Object_ID and Context_ID columns. Object_ID is always unique. Context_ID is an ID for the set that contains the instance, this is used for Shared context dependency to determine identity. We do not "garbage collect" discarded ID values in this version. A simple but effective algorithm for managing Context_ID is as follows:

(1) Upon insertion, an instance is given a new object id from the ID daemon

(2) If it is global, its context is set to 0.
If it is local, its context is set to its parent's object id.
If it is shared, its context is looked up by the ID daemon for the combination of
[parent's type name , parent's context id]

(3) The combination of [instance's type name , instance's context id] is assigned a unique context id for future reference.

Since ID's are assigned only when the instances are added to the database, this is the only time when new ID values are used up. For lookups and identity determination, only Step (2) is needed.

10.5.3. Binary Large Object (BLOB) Support

All the instances that forms a Set or Sequence instance is packed inside a Sybase Image data type.

For components with fixed size storage, a Direct storage is feasible. In this situation, the values of the elements are packed into consecutive bytes of a block in the image data. When accessed, this block of bytes is unpacked into an element instance. For elements with variable size storage, e.g. has components which are character strings, sets, or sequences, only Indirect storage is possible. In this situation, only the object id is stored in the image data. When accessed, the id is retrieved and another retrieval to the relation holding the elements is made.

These methods are rudimentary but simple. Clearly, with image data type, one can construct embedded indices, along with the data, to improve access and hold variable size elements. However, the current version of EOQP is only concerned with correct operations and not performance.

CHAPTER 11. QUERY LANGUAGE REFERENCE

The Extensible Object Query Language is separated into four parts: administration, data definition, instance manipulation, and selection/control flow. We will provide a description of each of these parts for the Extensible Object Query Language. The full grammar is given in Appendix D.

11.1. Notations

We will use the following formats for describing the formal language:

KEYWORD	- keywords of the EOQL are denoted in upper-case letters.
name	- names are denoted in lower-case letters, they are user supplied.
token	- a token has additional structure, e.g. see *scoped_name.*
...	- ellipsis is for repetition of the previous construct
<u>XYZ</u>	- underline encloses optional phrase

Each formal language token can be entered into the query processor in different ways:

KEYWORDS	- they can be entered in either upper or lower case letters.
name	- any character strings, other than keywords, that start with an alphabetic character: a-z, A-Z, and '_' (underline). They may include either alphabetic or numeric characters: 0-9.

The commands are first described with the formal notation. Then examples of usage are shown using the actual notation.

11.2. Expressions

A basic feature of the EOQL is the construction of expressions for data modification or control flow.

11.2.1. Values

Basic values in the EOQL can be of integer, float, or string type:

value = integer
| float
| string

The vertical bar '|' indicates an alternative using the standard BNF notation. Integer and floating values are based on the underlying architecture. Typically, they are four bytes wide. Floating point value conversion is also dependent on the underlying architecture.

Examples:

12	- integer value 12
123.45	- floating point value that corresponds to 123.45
"test"	- string value of "test"

A string can enclose '"' (double quote) by escaping them with '\' (backslash). For example: **"a \"new\" world"**.

11.2.2. Variables

A variable or instance is referenced as follows:

scoped_name = name
| func_name *(* expression , ... *)*
| *scoped_name*.name
| *scoped_name*.func_name *(* expression , ... *)*

The ellipsis "..." denotes zero or more *expression*'s. If the function does not expect any arguments, then they are optional.

Examples:

Person	- **Person** instance
Person.Age	- **Age** instance in **Person** instance
Dept.Manager.Age	- **Age** instance in **Manager** instance in **Dept** instance
add(5,4)	- Add 5 and 4
Person.Age.increment()	- Increment **Person.Age**

11.2.3. Expressions

An expression is composed of binary operations on values and variables:

expression = *scoped_name*
| *value*
| *expression* binop *expression*
| - expression
| (expression)

A binop is an arithmetic operator: + , - , * , and / , which stands for addition, subtraction, multiplication, and division, respectively.

Examples:

`3 + 5`	- add 3 and 5
`Person.Salary * 1.2`	- add 20% to `Person`'s `Salary` instance
`2 + length(dnaseq)`	- add 2 to length of `dnaseq`
`dnaseq.length() / 2`	- divide the length of `dnaseq` by 2

A boolean expression is used in control flow:

condition = *condition* OR *condition*
| *condition* AND *condition*
| NOT *condition*
| (*condition*)
| *expression* boolop *expression*

A boolop is a comparator: < , > , and = , which stands for less than, greater than, and equal to, respectively.

Examples:

`3 > 5`	- returns False
`length(dnaseq) > 5`	- returns True if length of `dnaseq` is greater than 5
`(Person.Age < 40) AND (Person.Salary = 40000)`	- returns True if `Person`'s `Age` is less than 40 and `Salary` is equal to 40000

11.3. Administration

These commands are mainly directed at the backend of the EOQP, i.e. Sybase DBMS. Since this is implementation specific, certain limitations are visible here.

11.3.1. Start-Up

Sybase start-up requires a password for each user, this is maintained in the EOQP.

11.3.2. Database Creation and Destruction

CREATE DB db_name ;

DESTROY DB db_name ;

If the user has the correct privileges, then he can create and destroy databases.

11.3.3. Default Database

USE DB db_name ;

This sets the default database to db_name. Subsequent operations goes to the database db_name, until the next USE DB command.

Examples:

CREATE DB test ;	creates a database named "test"
USE DB test ;	sets the default database to "test"
DESTROY DB test ;	destroys the database named "test"

11.3.4. Direct Backend Access

BYPASS string ;

This allows direct command submission to the backend. The string enclosed in double quotes is sent without modification.

Examples:

BYPASS "sp_help" ;	asks for help from Sybase
BYPASS "select * from test" ;	sends the SQL query "select * from test" to the backend

11.3.5. Table Management

The EOQP maintains several tables. They can be viewed by:

DISPLAY pool ;

DISPLAY pool name ;

DISPLAY pool string ;

The value for pool is TYPE , FUNC , VAR , or VALUE. The DISPLAY command reports on the defined types, functions, variables, and values, respectively. The string is used for regular expres-

sion selection of table entries for output. A special case exists when pool is VALUE and the string format is: "name:start-end". This only prints the values from start to end in the value buffer.

Examples:

```
DISPLAY TYPE ;          display all the type definitions
DISPLAY FUNC compl ;    display the function definition of compl
DISPLAY VAR "^a" ;      display all variables with names that start with 'a'
DISPLAY VALUE "integer:3-5" ;
                        display currently used integers in buffer 3 to 5.
```

11.3.6. Scripts

The EOQP can read in a previously stored script:

READ string ;

The string determines the filename of the script. At this point, the EOQP process the commands in the designated file. It returns after the end of file is reached.

Examples:

```
READ "define_all" ;  read the EOQL script named "define_all"
```

11.3.7. Comments

comments

Comments are any text that follows a '#' character and up to the next line, except when they occur in strings.

Examples:

```
# count is 0            - comment
count = count + 1; # increment count
                        - the increment command followed by a comment
```

11.4. Type Definition

11.4.1. Primitive Types

INTEGER, FLOAT, and STRING

The basic types are given above. They should be considered as keywords.

11.4.2. Implementation

DEFINE name IS IMPL_BY impl_name ;
DEFINE name IS IMPL_BY PROC func_name (type_name , ...) ;
DEFINE name IS IMPL_BY ENUM : enum_label , ... ;

An Object-Type can be implemented by another Object-Type. In this case, the Object-Type name has the same structure as the Object-Type impl_name. An Object-Type can also be implemented as a function, then accessing an instance of this type is equivalent to calling the defined function. Finally an Object-Type can be an enumeration. Enumeration labels can be either name or string. Enumerated types are equivalent to primitive types. A special type_name for the function is the string "..." (ellipsis in quotes), this represents an indeterminate number of arguments.

Examples:

```
DEFINE position IS IMPL_BY INTEGER ;
```

Type position is structurally similar to INTEGER

```
DEFINE length IS IMPL_BY seq_length ( "..." ) ;
```

Type length is the function seq_length that takes an indeterminate argument.

```
DEFINE base IS IMPL_BY ENUM: A, C, G, T, Pu, Py, N ;
```

Type base is an enumeration.

11.4.3. Aliasing

DEFINE name IS ALIAS_OF alias_name;

An Object-Type can be an alias of another Object-Type, i.e. the instances of this type are actually from the aliased type.

Examples:

```
DEFINE manager IS ALIAS_OF person ;
```

Type manager is really type person

11.4.4. Construction

DEFINE name IS constructor
{
ID comp_name : target, source, peer [storage] ;
...
PROPERTY comp_name : target, source, peer [storage] ;
...
ORDERD_BY scoped_name : target, source, peer [VIRTUAL] ;
} ;

constructor is	COMPOSITION, SET, SEQUENCE, INHERITED, or UNION
target is	LOCAL, SHARED, or GLOBAL
source is	REQUIRED or NONREQ
peer is	ONE or MANY
storage is	DIRECT, INDIRECT, or VIRTUAL

The "..." after ID and PROPERTY components indicate that more of them can be specified. PROPERTY components are not required. For the SET and SEQUENCE constructors, only one ID component is allowed. However, for the UNION constructor, at least two ID components are needed. The order of component specification is not important.

Only one ORDERED_BY component can be specified with the SEQUENCE constructor. Although it is specified with *scoped_name*, it should be either ID_name, comp_name, or ID_name.comp_name, where ID_name is the name of the ID component and comp_name is a component under the ID component. Since the ORDERED_BY component is a constraint, its context dependencies are ignored and its storage directive should be VIRTUAL.

Context dependencies default to LOCAL, NONREQ, and MANY. Storage directive defaults to INDIRECT.

Examples:

```
DEFINE posbase IS COMPOSITION
{ ID position : REQUIRED, ONE [ VIRTUAL ] ;
  ID base : REQUIRED, ONE [ DIRECT ] ;
} ;
DEFINE dnaseq IS SEQUENCE
{ ID posbase : REQUIRED, ONE [ DIRECT ] ;
  ORDERED_BY posbase.position [ VIRTUAL ] ;
  PROPERTY length [ VIRTUAL ] ;
```

```
  PROPERTY complement [ VIRTUAL ] ;
} ;
DEFINE ds_dna IS COMPOSITION
{ ID dnaseq : SHARED, REQUIRED, ONE [ INDIRECT ] ;
  PROPERTY source : GLOBAL, NONREQ, ONE [ INDIRECT ] ;
  PROPERTY mapped : LOCAL, NONREQ, ONE [ DIRECT ] ;
} ;
```

The first example defines a positioned-base. Note that the position component is virtual. Therefore, it is not stored in the database. This is a special case where the sequence operators will fill the position value based on the context of the positioned-base instance. The second example defines a single stranded DNA sequence, made of positioned-bases. The properties of **length** and **complement** are procedure types, to be defined elsewhere. The third example defines a double stranded DNA.

11.4.5. Undefine Types

UNDEFINE type_name ;

This will allow one to undefine an Object-Type. If the type holds existing variables (see Section 11.5.1), then this command will not complete.

11.4.6. Storage for Types

CREATE STORE type_name ;

DESTROY STORE type_name ;

The definition of Object-Types is not sent to the backend (Sybase) until explicit storage creation. This command will recursively create all component types if they are also constructed. If the backend already have an existing instance store (relation) with the same name, this command will not complete.

11.5. Instance Manipulation

11.5.1. Variable Creation and Destruction

CREATE VAR var_name IN *scoped_name* ;

DESTROY VAR var_name ;

These commands create and destroy variables, which are instances of a given type, but they exist only in the query processor and not in the permanent store. The *scoped_name* is either nested type_name's, which traverses the type definition, or it can start with a variable name. In the former case, var_name is simply a variable of the last type_name encountered. In the latter case, var_name is a variable to a descendent component of the first variable in *scoped_name*. Variable destruction is transitive, i.e. any variables dependent on the destroy one will also be destroyed.

Examples:

CREATE VAR x IN ds_dna ;

Variable **x** is an instance **ds_dna** in local session

CREATE VAR y IN ds_dna.dnaseq ;

Variable **y** is an instance of **dnaseq**

CREATE VAR z IN x.dnaseq ;

Variable **z** is the instance of **dnaseq** in **x**

DESTROY VAR x ;

Variables **x** and **z** (if created by the previous statement) are destroyed

11.5.2. Storage Effects

INSERT *scoped_name* ;

UPDATE *scoped_name* ;

DELETE *scoped_name* ;

In order to change the state of the database, we use the above commands. The *scoped_name* is used as a reference to the actual instance that will be operated on.

Examples:

INSERT d.manager ;

insert the instance referred by **d.manager** into its collection.

UPDATE p.age ; update the value of **age** in instance **p**.

DELETE x.dnaseq ;

delete the instance reference by **x.dnaseq** from its collection.

If the instance references by *scoped_name* is an ID instance of a Set or Sequence instance, then

the Set or Sequence instance value will be affected.

Example:

```
CREATE VAR x in dnaseq ;
... # fetch an x instance
CREATE VAR y in x.posbase ;
... # build a value of y
INSERT y ;
```

This will cause **y** to be inserted into the sequence referenced by **x** and the instance of **y** will be inserted to its collection. Therefore, the instance **x** is also affected by this command.

11.5.3. Printing Values

OUTPUT *expression* , ... ;

This statement prints the expressions in order. The format is based on the type of the expression:

Type	Output
raw string	the string, without quotes
calculated value	the calculated value, strings are quoted
constructed	comma separated component instances: if primitive → the actual value if constructed → the object id
set value	semicolon-separated elements in braces, "{ }" if direct → the actual value(s) if indirect → the object id
sequence value	semicolon-separated elements in angle brackets, "< >", otherwise the same as set

Examples:

```
OUTPUT "the answer to 3 + 4 is " , 3 + 4 ;
```

prints: `the answer to 3 + 4 is 7`

```
OUTPUT x ;
```

prints: `243 , 345 , False`

```
OUTPUT x.dnaseq ;
```

prints: `< a ; c ; g ; t ; c >`

```
OUTPUT x.dnaseq.dna2str() ;
```

prints: **"acgtc"**

11.5.4. Value Assignment

scoped_name = *expression* ;

scoped_name := *expression* ;

To effect changes in variables, we have value copy ("=") and instance assignment (":="). In both situations, if the instance referenced by scoped_name is a primitive type, only the value is copied to the destination, and if the instance is a constructed type, then all the component values are copied to the destination. The object id and context id values are copied only if it is an assignment.

Examples:

x = y ;	copy **y** value to **x**
x := y ;	assign identity of **y** to **x** and copy **y** value to **x**
p.age = 50 ;	copy the value 50 into **p.age**
p.age := 50 ;	copy the value 50 into **p.age**

11.6. Selection and Control Flow

11.6.1. Selection Statement

FOREACH
var_name IN *scoped_name* ,
...
WHERE (condition)
PERFORM statement ;

This is the select operation for instance collections. The ellipsis, "...", indicates multiple variables can be iterated in one statement. The *scoped_name* is identical to the one used for variable creation (Section 11.5.1). The statement is executed for every combination of instances that satisfy the WHERE clause. If the WHERE clause is not specified, then the statement is executed for all instances.

Examples are:

```
FOREACH p IN person
     WHERE ( p.age > 40 )
     PERFORM OUTPUT p ;
FOREACH pb IN x.dnaseq
     WHERE ( "c" = base2str(pb.base) )
     PERFORM count = count + 1 ;
FOREACH m IN dnaseq , n in dnaseq
     WHERE ( overlap ( m , n ) > 5 AND NOT ( m = n ) )
     PERFORM OUTPUT m , n ;
```

The first example prints out all Persons whose age is greater than 40. The second example counts the number of c's (cytosine) in the DNA sequence of x. The third example prints all pairs of distinct DNA sequences whose overlap is greater than 5.

11.6.2. If-Then-Else Statement

IF (*condition*) statement ; ELSE statement ;

The standard If-Then-Else control flow.

11.6.3. While Statement

WHILE (*condition*) statement ;

The standard While control flow.

11.6.4. Compound Statement

statement = { statement ; ... }

This permits multiple statements to be executed as a unit for the selection or the control flow statements.

Examples:

```
FOREACH p IN person
     WHERE ( p.age < 40 )
     PERFORM
{
     OUTPUT "Person is " , p.name , " in dept ", p.dept.name;
     count = 0 ;
     FOREACH c IN p.children
          WHERE ( c.age > 3 )
          PERFORM count = count + 1;
     OUTPUT "number of children > 3 years is ", count ;
} ;
```

This prints all the persons who are younger than 40 years old, the department they worked in, and the number of children greater than 3 years old.

11.6.5. Other Flow Control

BREAK ;

CONTINUE ;

EXIT ;

These allow redirection of control flow. BREAK will terminate the execution of any compound statement in a selection or control flow statement. This implies the termination of FOREACH and WHILE statements. CONTINUE will restart the execution of the compound statement. In the FOREACH statement, the next instance combination satisfying the WHERE clause will be fetched and in the WHILE statement, the condition is checked to see if another round of execution is warranted. EXIT will terminate the session.

11.7. Operators

The query processor will support a basic set of operators for the primitive and constructed types. All of these can be accessed as prefix function calls, e.g. `f(a,b)`, or if the first argument is a *scoped_name*, as infix function calls, e.g. `a.f(b)`.

11.7.1. Primitive Types

Numeric operators for integer and floating point values:

add(*value* , *value*)	- add two values
sub(*value* , *value*)	- subtract the second value from the first
mult(*value* , *value*)	- multiply two values
div(*value* , *value*)	- divide the first value by the second
neg(*value*)	- negate the value

scoped_name can also be used if the referenced instance is implemented by a primitive type. If needed, these operators will promote integer to float prior to execution.

Comparison operators for integers, floats, and strings:

gt(*value* , *value*)	- returns true if the first value is greater than the second
lt(*value* , *value*)	- returns true if the first value is less than the second
equal(*value* , *value*)	- returns true if the first value is equal to the second

The result for strings is based on standard lexicographic ordering.

Standard boolean operators:

and(*bool_value* , *bool_value*)	- returns true if both values are true
or(*bool_value* , *bool_value*)	- returns true if either values is true
not(*bool_value*)	- returns true if the value is false

They only operate on boolean types derived from a result of a comparison.

All these operators have an infix binary operator described in Section 11.2.3.

Examples:

`add ( 4 , 3 )`	returns 7
`add ( 4 , 3.5 )`	returns 7.5, 4 is promoted to 4.0 prior to addition
`div ( 3 , 0 )`	returns error
`gt ( "abc" , "abd" )`	returns false
`or ( equal(x,y) , lt(3,4))`	returns true

11.7.2. Constructed Types

The following operators applies to all constructed instances:

equal(*scoped_name* , *scoped_name*) - returns true if instance values are equal
equiv(*scoped_name* , *scoped_name*) - returns true if they are the same instance

Examples:

```
equal ( x , y )        - returns true if x equal y
equiv ( x , y )        - returns true if x is y
```

11.7.3. Set and Sequence Operators

To affect change in a set or sequence instance:

insert (*set_or_seq* , *scoped_name*) - insert instance into a set or a sequence
update (*set_or_seq* , *scoped_name*) - update instance in a set or a sequence
delete (*set_or_seq* , *scoped_name*) - delete instance from a set or a sequence

These operations will not flush the *scoped_name* instance to its collection store, unlike the INSERT, UPDATE, and DELETE commands. The distinction is very important if *scoped_name* is a transient indirect instance. This characteristic is subject to change in future versions.

Sequence specific operators are:

length (*sequence*) - returns the length of sequence
concat (*sequence* , *sequence*) - concatenate two sequences
range (*sequence* , *start_position* , *end_position*) - returns subsequence
index (*sequence* , *position*) - returns element at position
append (*sequence* , *element*) - append element to sequence
overlap (*sequence* , *sequence*) - find maximal overlap position

The append() will affect the sequence, i.e. it modifies the argument *sequence*. All others returns a copy of the value. They can also be applied to character strings, except for index() and append(), since the type CHAR is not supported.

Set specific operators are:

union (*set* , *set*) - returns the union of two sets
diff (*set* , *set*) - returns the difference of the first set from the second
intersect (*set* , *set*) - returns the intersection of two sets
issubset (*set* , *set*) - returns true if the second set is a subset of the first
ismember (*set* , *element*) - returns true if the element is in the set

Examples:

```
insert ( dnaseq , x )   insert x into dnaseq based on position in x
length ( x )            returns the length of x
x.concat ( y )          concatenates x and y, using infix function call
```

Conclusions

Review

This thesis has three major parts: 1) sequence modelling for Human Genome applications, 2) generalized conceptual object modelling as a framework to incorporate sequence models, and 3) a query language for the above models and their implementations.

The purpose of Part I is to provide an understanding of the structure of sequences encountered in the Human Genome Project (HGP) and operations over them. In addition to traditional data types, the HGP also produces a large amount of information as sequences. Therefore, a first step in building genomic databases is to understand the nature of this sequence information. The informatics requirement and the biology of the HGP are described in Chapter 1.

As described in Chapter 2, there is no consensus on how to model sequences from current data models. Therefore, we started with the most primitive and abstract sequence model: a set of ordered pairs, with one component being the position and the other component being the content. The next step was to characterize the real world types for position, which were characterized in terms of their metric, granularity, and atomicity properties. Then, by analyzing the real world interactions of these position characteristics, we added semantic operations to the sequence model. This provided a framework of the basic sequence model. We followed that, at the end of Chapter 2, with the definition of some abstract sequence operations using only the equality operation provided by the content type.

In Chapter 3, we verified the expressiveness of the sequence framework by using it to model the basic types of sequences found in the HGP. In addition, we built new operations specific to the HGP by using operations derived from domain semantics, such as the complement and match_cut operations. Finally, in Chapter 4, we summarized our sequence model results and concluded with remarks for future extensions to the framework.

Part II of this work dealt with generalized conceptual data modelling. For the HGP, the primary reason for developing a new data model was the ability to integrate the sequence framework defined in Part I. Another reason was the fact that a conceptual model is the easiest form for porting an existing database to new database technologies. Because of the long life-time of the HGP, database technology will invariably evolve over time. Therefore, porting of genomic databases can best be done at the conceptual level, similar to porting programs written in high-level vs. low-level languages. External to the needs of the HGP, an objective of this model was to bring forth a new approach to conceptual modelling. This was embodied in its information flow perspective and its extensible nature. In addition, this model was developed with the potential as a conceptual model for Object Oriented (OO) DBMS.

In Chapter 6, we defined the basic tenets of the Extensible Object Model. It includes the definition of Object-Types and Instances. In Chapter 7, Object-Type constructors were defined to provide semantic richness. These gave the ability to encapsulate multiple distinct information units into a single unit. Specifically, it provided a framework for defining complex objects that have multiple components (information units), some of which can be complex as well. In Chapter 8, context dependencies were defined to manage semantic integrity. These provided the ability to specify constraints for instances at the type level. The result was a rich conceptual model that is highly uniform and orthogonal in the sense of increasingly richer layers of object structures and their semantics (see further discussion on this point in the section "Common Themes" below). Furthermore, a graphical representation was given to simplify schema design and improve its comprehensibility. In addition, the concept of physical locality has been abstracted by the use of instance association diagrams. This allowed the EOM to be implemented on top of any primitive database engine, without sacrificing its semantic richness and power. In Chapter 9, the EOM was compared with other data models and the previously stated goals in Chapter 5. Finally, various extensions were informally discussed for future research.

The main focus of Part II has been the data definition component of the EOM. As with all models, the EOM is incomplete without a data manipulation language. The third part of this work was to show that a working query language (EOQL), based on the EOM, can be developed. Therefore, we designed a query language that contained the major features of the EOM and implemented the query processor (EOQP) for such as language. Since this was only meant as a demonstration of the features unique to the EOM, the language did not include many features common to all query languages, such as transaction management. These design criteria and limitations were discussed in Chapter 10. A commercial DBMS, Sybase, was chosen for the backend engine to implement the instances. In Chapter 11, we briefly described the major language constructs and provided example usage.

Below we discuss briefly features of the sequence model, the object model, the query processor, and common themes between the sequence and object models.

The Sequence Model

The construction of the sequence model framework is an initial step in the unification of many differently named, but related sequence constructs in separate application domains. This unification removes the burden of reinventing the sequence model for each domain specific database model. Since this framework is at the conceptual level, it properly hides the implementation of the sequence operators. Consequently, the user only has to model domain concepts and not be coerced into specific constructs by implementation details. For example, research in temporal data modelling often deals with optimizing its operations for the relational implementation. However, as

newer database technologies evolve, the strategies discovered in the relational implementation may not be applicable. On the other hand, a physical implementation of the model should not be oblivious to these optimized strategies. Therefore, in our EOQP implementation for sequences, an optimized temporal joins, for example, can be used for the operator SEQ_APPLY() if the implementation was on top of a relational system.

The Object Model

The EOM was constructed with OO principles in mind. However, borrowing OO concepts does not mean we have to bind the model to a specific OODBMS. Therefore, the resultant conceptual model was independent of any specific OODBMS, but still provides the "feel" of an OO model. Because of the its semantic richness, this model also subsumed many features from the traditional and semantic data models. In addition, it also extended certain abstractions to a higher conceptual level, e.g. context dependencies. Consequently, the EOM can be used as a transition model for evolving databases by first mapping the existing database schema to an EOM schema and then converting the EOM schema to the next generation database schema.

The Query Processor

Although the query processor was meant as a demonstration of the EOM features, it has many powerful constructs, such as functional overloading and procedural types. In addition, the query processor has a code frame that supports future extensions and has isolated the underlying database engine. This is well suited for replacing the underlying engine, which will provide the opportunity to test and compare new back-end technology, e.g. OODBMS, without reconstructing the front-end programs. The EOQL and EOQP still lack some of the common features of query languages, but these can be borrowed from available technologies.

Common Themes

Three common themes were present in Part I and Part II. The first theme is the separation of semantics from symbolics. In Part I, this was used to separate position values from its real world values. In Part II, this was used to separate model constructs from real world objects. Both models employ the use of semantic mapping functions that are defined by the application domain and not by the model. This allows each model to work completely in the symbolic domain and yet retain its real world characteristics.

The second theme is the framework for extensions. It is not realistic to think that one can build the end-all model for all purposes and domains. Therefore, it is better to construct an extensible framework so that future additions can be made without redoing all the previous work. Both the sequence model framework and the object model framework were designed to be extensible.

The third theme is the orthogonal layering of concepts. In Part I, three specific layers of sequence

models were built: order, simple and semantic. In Part II and Part III, three layers of object characterization were defined: constructor, context, and storage. Each outer layer is an elaboration of the inner layer with more semantic information. Each inner layer is an abstraction of the outer layer. The type of information abstracted at each level is independent of the other layers. This results in modelling by successive refinement in both conceptual abstractions and conceptual objects.

While the specific models will find uses in their respective domains, the three themes for building the models are much more crucial to the understanding of data model building. If applied appropriately, they serve to create models that can gracefully evolve with both the application domain and the underlying database technology. In addition, they create models that are simple to understand and easy to use.

Bibliography

Abbreviations Used:

DOOD	Deductive and Object Oriented Databases
OOPSLA	Object-Oriented Programming Systems, Languages, and Applications
SIGACT	ACM Special Interest Group on Automata and Computability Theory
SIGART	ACM Special Interest Group on Artificial Intelligence
SIGMOD	ACM Special Interest Group on Management of Data
TODS	ACM Transaction on Database Systems
TOIS	ACM Transaction on Information Systems
VLDB	Very Large Data Bases
ECOOPS	European Conference on Object Oriented Programming Systems

1. Abiteboul, S. and Bidoit, N., "Non-First Normal Form Relations to Represent Hierarchically Organized Data". Proceedings of SIGACT-SIGMOD Symposium on Principles of Database Systems, p. 191-200, 1984.
2. Abiteboul, S. and Hull, R., "IFO: A Formal Semantic Database Model". TODS, 12(4): 525-565, December 1987.
3. Albano, A., Ghelli, G., and Orsini, R., "A Relationship Mechanism of a Strongly Typed Object-Oriented Database Programming Language". Proceedings of VLDB 1991 Conference, p. 565-575, September 1991.
4. Alberts, B., Bray, D., Lewis, J., Raff, M, Roberts, K, and Watson, J.D. Molecular Biology of the Cell, 2nd Edition. Graland Publishing: New York, New York: 1989.
5. Agrawal, R. and Gehani, N.H., "ODE (Object Database and Environment): The Language and the Data Model". Proceedings of SIGMOD 1989 Conference, p. 36-45, June 1989.
6. Bancilhon, F. and Buneman P. Eds. Advances in Database Programming Languages. ACM Press/Addison-Wesley: Reading, Massachusetts. 1990.
7. Barker, W.C., George, D.G., and Hunt, L.T., "Protein Sequence Database". Methods in Enzymology, 183:31-49, 1990.
8. Batory, D.S., Leung, T.Y., and Wise T.E., "Implementation Concepts for an Extensible Data Model and Data Language". TODS, 13(3): 231-262, September 1988.
9. Beech, D. "A Foundation for Evolution from Relational to Object Databases". Advances in Database Technology - EDBT 1988, Lecture Notes in Computer Science, #303, p. 251-270.

Springer-Verlag: Berlin, 1988.

10. Beeri, Catriel, "Formal Models for Object Oriented Databases". Proceedings of DOOD 1989 Conference, p. 405-430, December 1989, Elsevier: North Holland. 1990.

11. Bilofsky, H.S., Burks, C, Fickett, J.W., Goad, W.B., Lewitter, F.I., Rindone, W.P., Swindell, C.D., and Tung, C.S., "The GenBank Genetic Sequence Data Bank". Nucleic Acids Research, 14(1): 1-4, 1986.

12. Bouzeghoub, M. and Metais, E., "Semantic Modeling of Object Oriented Databases". Proceedings of VLDB 1991 Conference, p. 3-14, 1991.

13. Brunn, C., personal communications. The Genome Data Base, Welch Library, John Hopkins University, Baltimore, Maryland.

14. Burks, C., et. al., "Genbank: Current Status and Future Directions". Methods in Enzymology, 183: 3-22, 1990.

15. Carey, M.J., et. al., "The Architecture of the EXODUS Extensible DBMS". Proceedings of Object-Oriented Database Workshop, p.52-65, 1986.

16. Carey, M.J., et. al., "The EXODUS Extensible DBMS Project: An Overview". Readings in Object Oriented Database Systems. Eds. Zdonik, S.B.and Maier, D., p.474-499, 1989.

17. Cattell, R.G.G., Object Data Management. Addison-Wesley: Reading, Massachusetts. 1991.

18. Chen, P.P.S., "The Entity-Relationship Model - Toward a Unified View of Data". TODS, 1(1): 9-36, january, 1976.

19. Clemons, Eric K.. "Data Models and the ANSI/SPARC Architecture". Principles of Database Design, Vol I: Logical Organizations. Ed. Yao, S.B. Prentice-Hall: Englewood Cliffs, New Jersey. 1985.

20. Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto". SIGMOD Record, 19(3): 31-44, September 1990.

21. Courteau, Jacqueline, "Genome Databases". Science, 254:201-207, 1991.

22. Date, C.J. An Introduction to Database Systems. 5th Edition. Addison-Wesley: Reading, Massachusetts. 1990.

23. Enderton, H., A Mathematical Introduction to Logic. Academic Press: Orlando, Florida. 1972.

24. Fickett, J.W. and Burks C., "Development of a Database for Nucleotide Sequences". Mathematical Methods of DNA Sequences, p. 1-34. CRC Press: BocaRaton, Florida. 1989.

25. Frenkel, Karen A., "The Human Genome Project and Informatics". CACM, 34(11): 41-51, 1991.

26. Gogolla, M. and Hohenstein, U., "Towards a Semantic View of an Extended Entity-Relationship Model". TODS, 16(3): 369-416, September, 1991.

27. Goldberg, A. and Robinson D. Smalltalk-80: The Language and Its Implementation. Addison-Wesley: Reading, Massachusetts. 1983.

28. Hammer, H. and McLeod D., "Database Description with SDM: A Semantic Database Model". TODS, 6(3): 351-386, September 1981.

29. Hull, R. and King R., "Semantic Database Modeling: Survey, Applications, and Research Issues". ACM Computing Survey, 19(3): 201-260, September 1987.

30. Informix-OnLine Programmer's Manual. Version 4.0. Informix: Menlo Park, California. March, 1990.

31. Ingres/SQL Reference Manual. Release 6.3. Ingres: Alameda, California. January, 1990.

32. Jackson, M. A. Principles of Program Design. Academic Press: Orlando, Florida. 1975.

33. Joseph, J., Thatte, S., Thompson, C., and Wells, D., "Report on the Object-Oriented Database Workshop". SIGMOD Record, 18(3): 78-101, September 1989.

34. Kahn, P. and Cameron, G., "EMBL Data Library". Methods in Enzymology, 183:23-31, 1990.

35. Kent, William, "Limitations of Record-Based Information Models". TODS, 4(1): 107-131, March 1979.

36. Khoshafian, S.N. and Copeland, G.P., "Object Identity". Proceedings of OOPSLA 1986 Conference, p. 406-416, 1986.

37. Kilov, Haim, "Reviews of Object-Oriented Papers". SIGMOD Records, 18(4): 50-55, 1989.

38. Kim, Won, "Research Directions in Object-Oriented Database Systems". Proceedings of SIGACT-SIGMOD-SIGART 1990 Conference, p. 1-15, 1990.

39. Lander, E.S., Langridge, R., Saccocio, D.M., "Mapping and Interpreting Biological Information". CACM, 34(11): 33-39, 1991 or IEEE Computer, 24(11):6-13, 1991.

40. Lawrence, C.B., "Data Structures for DNA Sequence Manipulation". Nucleic Acids Research, 14(1): 205-216, 1986.

41. Lecluse, C., Richard, P., Velez, F., "O2, an Object-Oriented Data Model". Proceedings of SIGMOD 1988 Conference, p. 424-433, 1988.

42. Liskov, B. and Guttag, J., Abstraction and Specification in Program Development. MIT Press: Cambridge, Massachusetts. 1986.

43. Lohman, G.M., Lindsay, B., Pirahesh, H., and Schiefer, K.B., "Extensions to Starburst: Objects, Types, Functions, and Rules". CACM, 34(10): 94-109, October 1991.

44. Loomis, M.E.S., Shah, A.V., Rumbaugh, J.E., "An Object Modeling Technique for Conceptual Design". Proceedings of 1987 ECOOPS, p. 192-202, 1987.

45. Maier, D. and Zdonik S., "Fundamentals of Object Oriented Databases". Readings in Object Oriented Database Systems. Morgan Kaufman: San Mateo, California. 1990.

46. McGee, William C., "On User Criteria for Data Model Evalutaion". TODS, 1(4): 370-387, 1976.

47. National Research Council, Mapping and Sequencing the Human Genome. National Academy Press: Washington, D.C., 1988.

48. Nomenclature Committee of the International Union of Biochemisty, "Nomenclature for incompletely specified bases in nucleic acid sequences". Journal of Biological Chemistry, 261(1): 13-17, January 5, 1986.

49. Osborn, S.L. and Heaven, T.E., "The Design of a Relational Database System with Abstract Data Types as Domains". TODS, 11(3): 357-373, September, 1986.

50. Pearson, P. L., Maidak, B., Chipperfield, M., and Robbins, R., "The Human Genome Initiative - Do Databases Reflect Current Progress?". Science, 254:214-215, 1991.

51. Peckham, J. and Maryanski, F., "Semantic Data Models". ACM Computing Surveys, 20(3): 153-189, September 1988.

52. Pistor, P. and Anderson F., "Designing a Generalized NF2 Model with an SQL-type Language Interface". Proceedings of VLDB 1986 Conference, p. 278-285, 1986.

53. Pistor, P. and Traunmeuller, R., "A Database Language for Sets, Lists, and Tables". Information Systems, 11(4): 323-336, 1986.

54. Roberts, Fred. Discrete Mathematical Models. Prentice Hall: Englewood Cliffs, N.J.. 1976.

55. Roth, M.A. and Korth, H.F., "The Design of ¬1NF Relational Databases into Nested Normal Form". Proceedings of SIGMOD 1987 Conference, p. 143-159, 1987.

56. Segev, A. and Shoshani, A., "Logical Modeling of Temporal Data". Proceedings of SIGMOD 1987 Conference, p. 454-466, 1987.

57. Soo, Michael D., "Bibliography on Temporal Database". SIGMOD Record, 20(1): 14-23, 1991.

58. Sowa, J., Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley: Reading, Massachusetts. 1984.

59. Stevens, W.P., Myers, G.J., and Constantine, L.L, "Structured Design". IBM Systems Journal, 13(2): 115-139, 1974.

60. Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., Guttman, A., "Document Processing in a

Relational Database System". TOIS, 1(2), 1983.

61. Stroustrup, B. The C++ Programming Language. Addison-Wesley: Reading, Massachusetts. 1986.

62. Su, S.Y.W., Krishnamurthy, V., Lam, H."An Object Oriented Semantic Association Model (OSAM*)". Artificial Intelligence: Manufacturing Theory and Practice, Eds. Kumara, S.T., Soyster, A.L., and Kashyap, R.L.. Chapter 17. IIE/IEM Press: Norcross, Georgia. 1989.

63. Sybase Transact-SQL Users Guide. Release 4.0. Sybase: Emeryville, California. May, 1989.

64. Teorey, T.J. Database Modeling and Design: The Entity-Relationship Approach. Morgan Kaufman: San Mateo, California. 1990.

65. Teorey, T.J., Yang, D.Q., Fry, J.P., "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model". ACM Computing Surveys, 18(2): 197-222, June 1986.

66. Tsichritzis, D.C. and Lochovsky, F.H., Data Models. Prentice Hall: Englewood Cliffs, New Jersey. 1982.

67. U.S. Congress, Office of Technology Assessment, Mapping Our Genes—The Genome Projects: How Big, How Fast. OTA-BA-373. U.S. Government Printing Office: Washington, D.C., April 1988.

68. U.S. Department of Health and Human Services, "Understanding Our Genetic Inheritance: The First Five years", The U.S. Human Genome Project. N.I.H. Publication No. 90-1590. April, 1990.

69. Vossen, Gottfried, "Bibliography on Object-Oriented Database Management". SIGMOD Record, 20(1): 24-46, March 1991.

70. Watson, J.D., Hopkins, N.H., Roberts, J.W., Steitz, J.A., Weiner, A.M.. Molecular Biology of the Gene, 4th Edition. Benjamin Cummings: Menlo Park, California. 1987.

Glossary

Biology

This glossary is based on [68].

A , C , G , T - DNA nucleotides adenosine, cytidine, guanosine, and thymidine, respectively. Also stands for the bases adenine, cytosine, guanine, and thymine.

A , C , G , U - RNA nucleotides, same as DNA nucleotides except thymidine has been replaced with uracil. In addition, the deoxyribose sugar molecule in each nucleotide is replaced by a ribose.

amino acids - A class of small molecules which can be chained together to form proteins.

anneal - The process by which two complementary single stranded nucleotide molecules (DNA or RNA) are hydrogen bonded into one double stranded molecule.

autosomal chromosome - A chromosome that has been identified with a number between 1 and 22. There are paired in normal human cells.

base - A molecule whose nonionic form is basic (hydrogen ion receptor), as in acids and bases. In molecular biology, they refer to a class of 5 molecules.

base pair - Two nucleotides (A and T or G and C) held together by weak (hydrogen) bonds. Two strands of DNA are held together in the shape of a double helix by the bonds between base pairs.

centimorgan - A unit of measure of recombination frequency. One centimorgan is equal to a 1-percent chance that a genetic locus will be separated from another marker due to recombination in a single generation.

chromosome - A structure found in the cell nucleus and containing the genes. Chromosomes are composed of DNA and proteins. They can be seen in the light microscope during certain stages of cell division.

complement - the pairing of nucleotides, A with T (or U) and C with G.

cytological mapping - Mapping of genes using DNA probes that bind to the chromosome at the site of the gene and are visible in a light microscope.

DNA (deoxyribonucleic acid) - The molecule that encodes genetic information. DNA is normally a double-stranded molecule held together by weak (hydrogen) bonds between pairs of nucleotides on opposite strands.

DNA sequence - The order of base pairs, whether in a stretch of a DNA, a gene, a chromosome, or an entire genome.

double helix - The shape in which two linear strands of DNA are hydrogen bonded together.

electrophoresis - A method of separating large molecules from a mixture by differentiating mobility in a medium under an electrically charged field.

exon - The piece of primary transcript RNA that is kept for mRNA.

gel - A medium used for electrophoresis. It is commonly made of either agarose (complex sugar) or polyacrylamide (organic polymer).

gene - The fundamental physical and functional unit of heredity. A gene is an ordered sequence of nucleotides located in a particular position on a particular chromosome.

gene mapping - Determining the relative locations of different genes on chromosomes.

genome - All the genetic material in the chromosomes of a particular organism. Its size is generally given as the total number of base pairs.

Human Genome Initiative - An initiative whose goal is to map and sequence the human genome.

Human Genome Project - The implementation of the concepts proposed as the Human Genome Initiative.

Human Genome Program - The individual programs, such as those at the DOE and the NIH, that make up the Human Genome Project.

intron - The piece of primary transcript RNA that is removed to form the mRNA.

locus - The position of a marker on a chromosome or piece of DNA.

marker - An identifiable physical location on a chromosome, e.g. restriction enzyme cutting site, gene, RFLP marker, whose inheritance can be monitored.

mRNA (messenger RNA) - A class of RNA whose role is to carry the genetic code from the chromosome to the ribosome, the site of protein synthesis.

nucleotide - A unit of DNA or RNA consisting of a nitrogenous base molecule, a phosphate molecule, and a sugar molecule. DNA nucleotides is usually represented as A, C, G, and T and RNA nucleotides is usually represented as A, C, G, U.

physical map - A map of the locations of identifiable landmarks (from markers) on DNA. Distance is measured in base pairs. The lowest resolution is the banding pattern on the chromosome, the highest resolution would be the complete nucleotide sequence.

primary transcript RNA - An exact RNA copy of the DNA in the coding region of a gene.

protein - A molecule that is a sequence of amino acids. They serve as messengers, metabolic processors, and structural support for cellular activity.

purine - A type of base molecule, either A or G.

pyrimidine - A type of base molecule, either T, U, or C.

recombination - The process by which portions of DNA are exchanged or deleted. Recombination occurs naturally between or within chromosomes, particularly during the formation of sperm and egg cells.

restriction enzyme (RE) - An enzyme that recognizes a specific base sequence (usually four to 6 base pairs in length) in a double stranded DNA molecule and cuts both strands of the DNA molecule at every place where this sequence appears.

restriction enzyme cutting (recognition) site - A specific nucleotide sequence of DNA at which a RE cuts the DNA. Some are frequent, e.g. every several hundred base pairs, some are infrequent, e.g. every 10,000 base pairs.

RFLP (restriction fragment length polymorphism) - The presence of two or more variants in the size of DNA fragments from a specific region of DNa that has been exposed to a particular RE. These fragments differ in length because of an inherited variation in a RE recognition site.

sex chromosome - Either X or Y chromosome. A normal human male has one X and one Y chromosome and a female has XX.

transcription - The process by which a RNA copy is made from the DNA for the expression of the gene.

translation - The process by which an amino acid sequence (protein) is synthesized by reading an mRNA sequence.

Data Modelling

abstraction - The process by which information concerning an object is converted into a symbol in a language or model.

attribute - A characteristic, usually of a data model entity or an object.

cardinality - The number of instances in a set or collection.

conceptual data model - a data model that primarily focuses on the concepts of the application domain and is not involved with the details of databases.

data model - A descriptive language that is used to express concepts from whatever domain it is modelling. Like a language, its richness determines its expressiveness.

database - a depository of organized information.

Database Management System (DBMS) - the collection of computers and programs that manages a database via the storage, retrieval, and updates of information.

entity - A common abstraction used in data modelling to represent objects (or collection of objects) in the real world.

Entity-Relationship Model - A simple conceptual data model that is based on the abstractions of Entities, Attributes, and Relationships between Entities.

First Normal Form - A relational characteristic where each field in the tuple holds only one value.

hierarchical model - A logical database model whose objects are organized into single parents and its children, i.e. tree-like.

informatics - The study of the application of computer and statistical techniques to the management of information.

logical data model - A data model for databases that does not contain any implementation specific characteristics.

methodology - A set of rules and methods for achieving a stated goal.

network model - A logical database model whose objects are organized into multi-parents and their children, i.e. direct acyclic graphs.

NonFirst Normal Form - A data model that is an extension of the relational model where the First Normal Form assumption is not held, i.e. a field may contain several values.

Normal Form - A characteristic of a relation in the relational model which determines the amount of data cohesion. There are several types of normal forms.

Object Oriented - Any process or methodology that describes the static and dynamic characteristics of a concept into a single unit of abstraction.

physical data model - An implementation-specific data model for databases, i.e. the actual DBMS specification.

query language - A language used to access and update database information.

query processor - A program that processes the commands of a query language and effect the changes in the database.

relational model - A logical database model whose objects are organized in sets and tuples.

schema - A description of the data collections and their characteristics in a database. Usually expressed using a data model language.

semantics - The meaning of something, as in the meaning of a value.

semantic data model - A data model that is rich in semantics, i.e. real world concepts.

stepwise continuous - a function (sequence) whose range (content) value is held constant until the next event. For example, salary history.

traditional data model - A hierarchical, network, or relational data model.

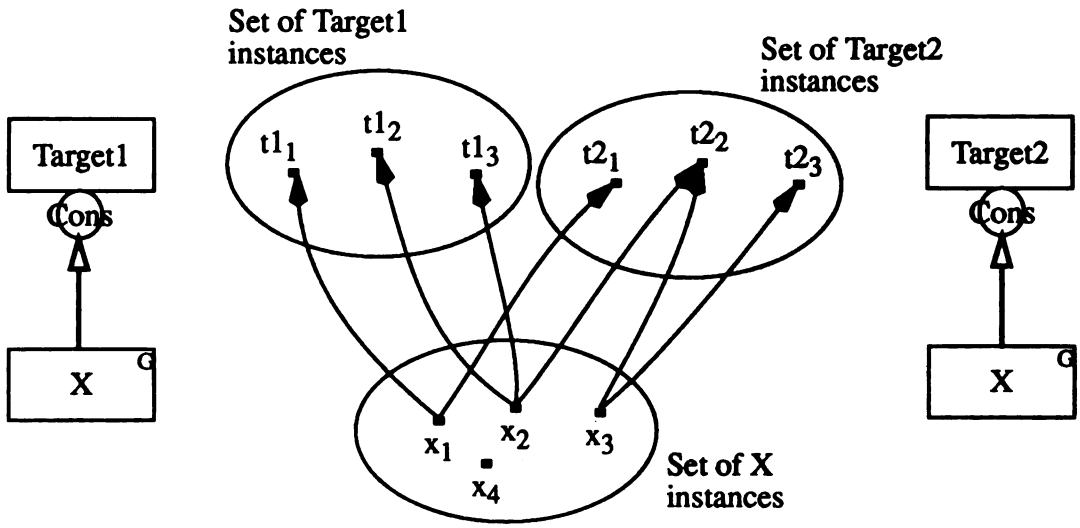
APPENDIX A. Context Interactions

In this Appendix, we will briefly describe the interactions of context dependencies and their corresponding instance association diagrams. We will first look at target context dependency alone, then add source dependency, and finally add peer dependency.

A.1. Target Dependency

For the purpose of enumerating the possible combinations, we will assume that the Target Object-type defines a strict set of instances. The three contexts have the following instance association examples.

A.1.1. Global-context

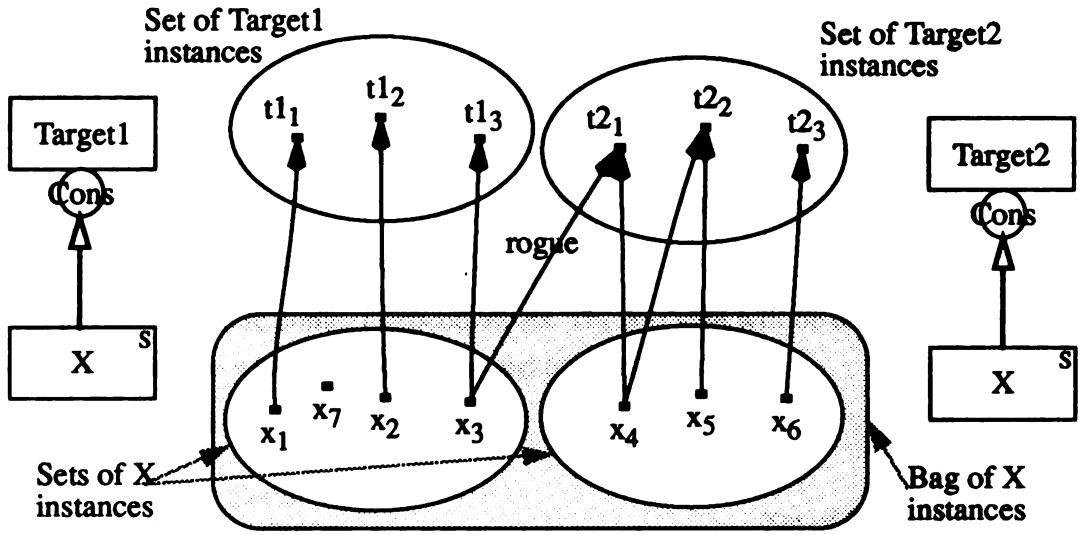


Comments:

(1) Since X is global-context, its instances are independent of target Object-Types. Thus x_4 is not associated with any target instances and x_1 can be associated with both $t1_1$ and $t2_1$.

(2) No distinction is made on the arc, they could be ID or Property. No distinction is made on the constructor, it could be Composition, Set, or Sequence. Thus $t2_2$ can have both x_2 and x_3 be associated with it.

A.1.2. Shared-context

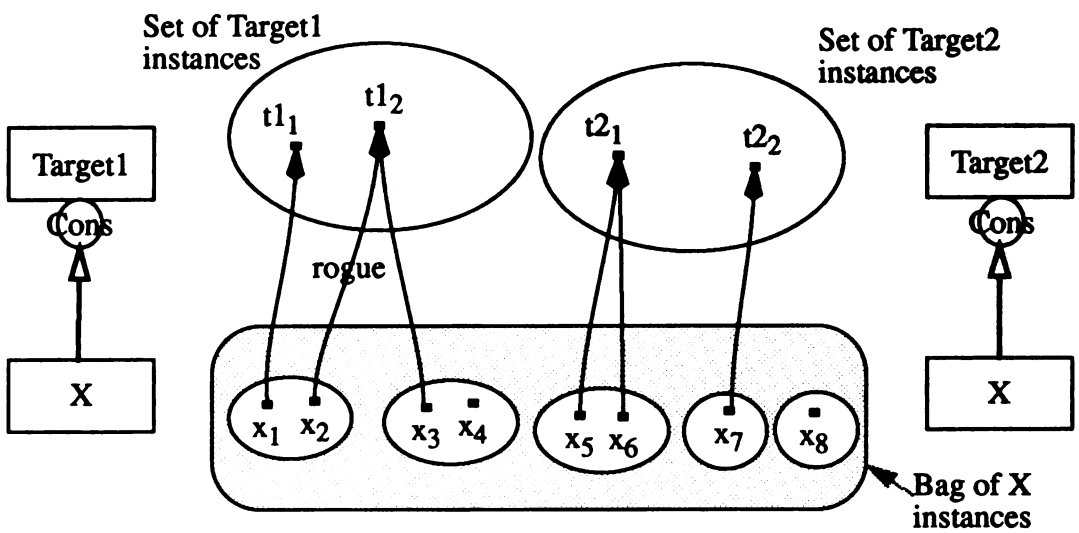


Comments:

(1) Since X is shared-context, each set of Targets is associated with its own set of X instances. Instance x_7 is an "orphan" because it is not associated with a Target1 instance. This can be flagged as a violation or be "forgiven", depending on the database designer. The link between $t2_1$ and x_3 is a "rogue", it is a violation.

(2) Instance x_1 under Target1 can be value-equal to instance x_4 under Target2. However, shared-context will ensure that they are different instances. Within each set, the values must be distinct, i.e. values of x_1, x_2, x_3, and x_7 must be distinct, same for x_4, x_5, and x_6.

A.1.3. Local-context



Comments:

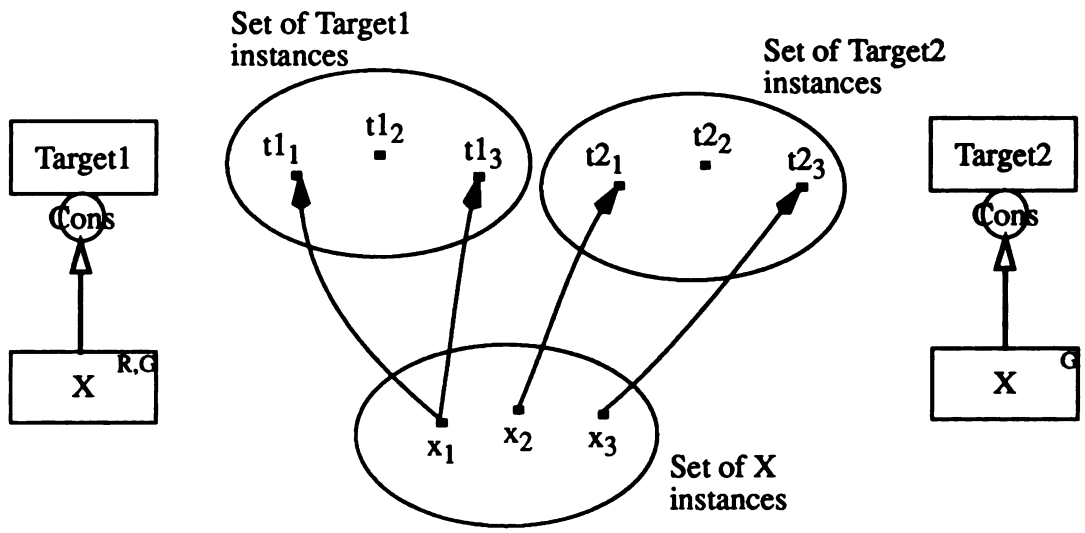
(1) Since X is local-context, each Target instance define a set of X instances. Instance x_4 is an orphan. The link between $t1_2$ and x_2 is a rogue and a violation. The set enclosing x_8 is an orphan set and a violation.

(2) Instance values within each set have to be unique, but can overlap with instances from other sets. For example, x_1 and x_3 can be value-equal but remain non-identical.

A.2. Target and Source Dependency

Now there are two possible context dependencies. The left side will be of Required source dependency and the right side will be of Non-required (default).

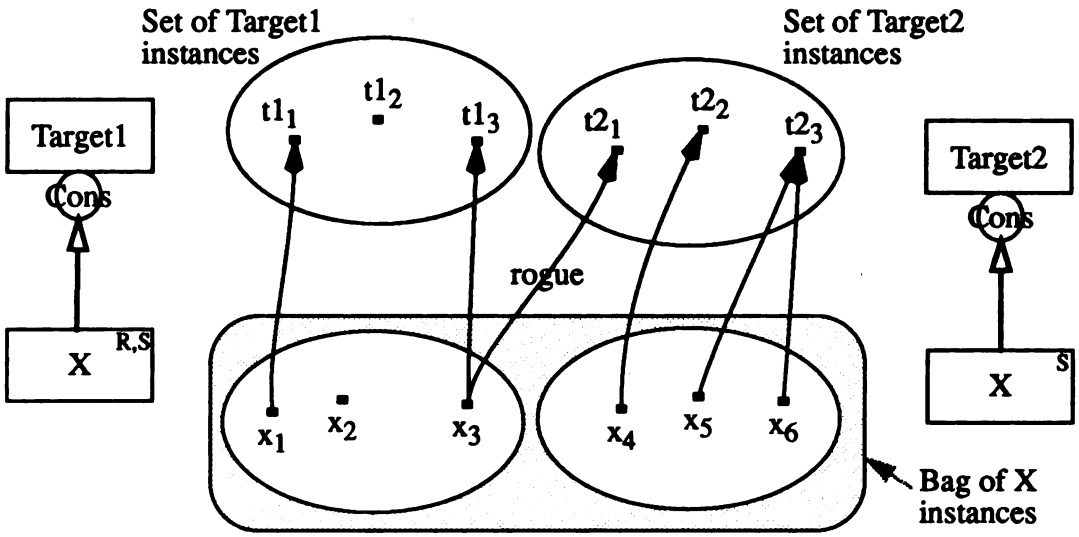
A.2.1. Global-context



Comments:

(1) With respect to Target1, each instance of Target1 must have an association arc from a source X instance. The instance $t1_2$, with a "dangling reference", cannot exist, because it is a violation of source dependency, while $t2_2$ can exist and is valid.

A.2.2. Shared-context

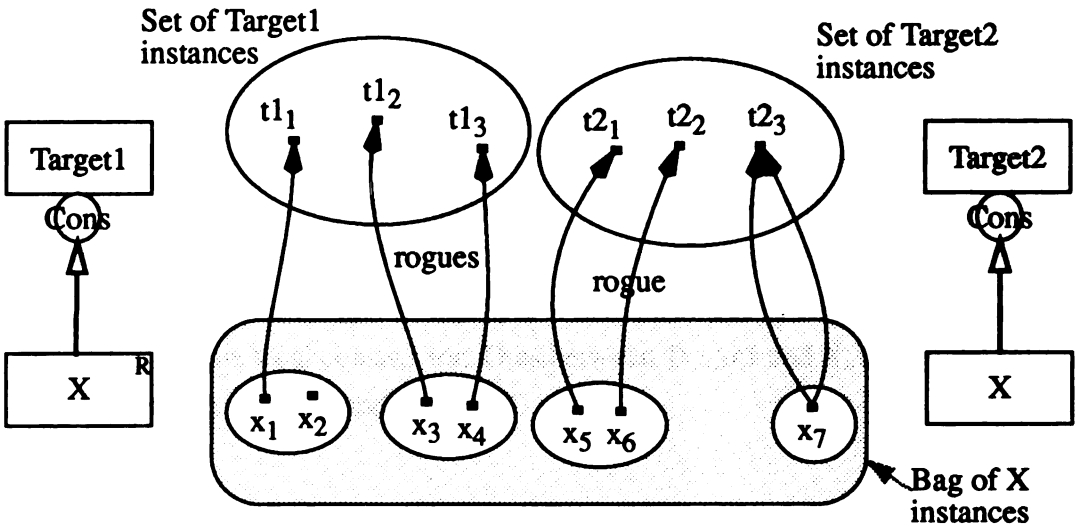


Comments:

(1) Between X and Target1, each instance of Target1 must receive an arc from a X instance. Therefore, $t1_2$ is a violation. Instance x_2 is an orphan and its disposition is dependent on target dependency.

(2) The link between x_3 and $t2_1$ is a rogue, and is a violation of target dependency, but $t2_1$ can have valid existence because it is non-dependent on X.

A.2.3. Local-context



Comments:

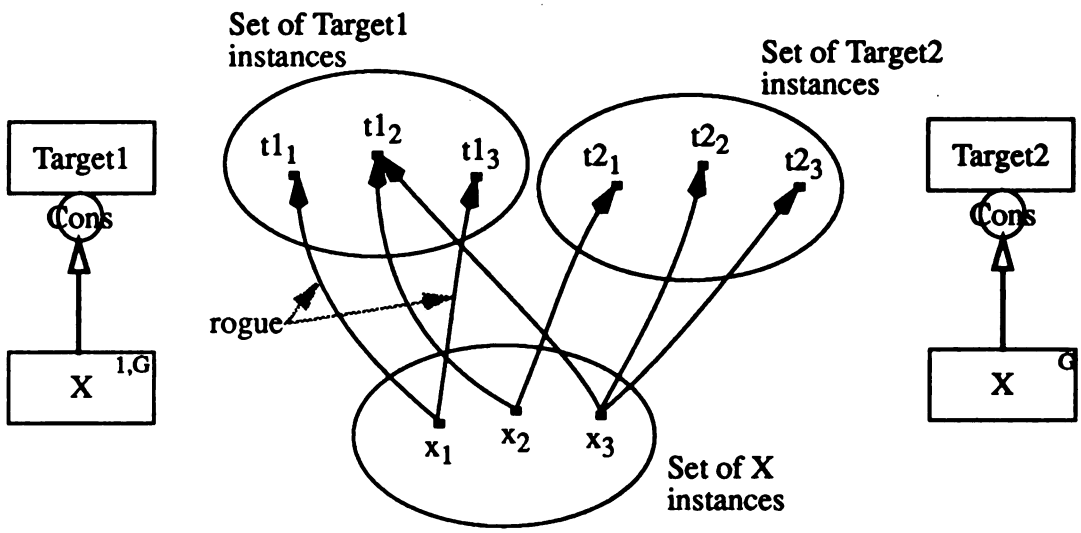
(1) One of the links, between x_3 and $t1_2$ or between x_4 and $t1_3$, is a rogue. Removing one will create a violation of the source dependency on the other Target1 instance.

The link between x_6 and $t2_2$ is a rogue because it violates target dependency. However, $t2_2$ is a valid instance in Target2, because it is not dependent on any X instance.

A.3. Target, Source, and Peer Dependency

Now we have all three context dependencies. There are $3 \times 2 \times 2$, or 12, cases. We are going to block them into 6 diagrams, within each diagram, the left side has involvement cardinality of "1" and the right side has "M" (default). As the following diagrams demonstrate, the actual number of semantic distinctions between target, source, and peer interaction is only 3 + 2 + 2, or 7, because these dependency rules are independent of each other and they can be interpreted separately.

A.3.1. Global/Non-Required context

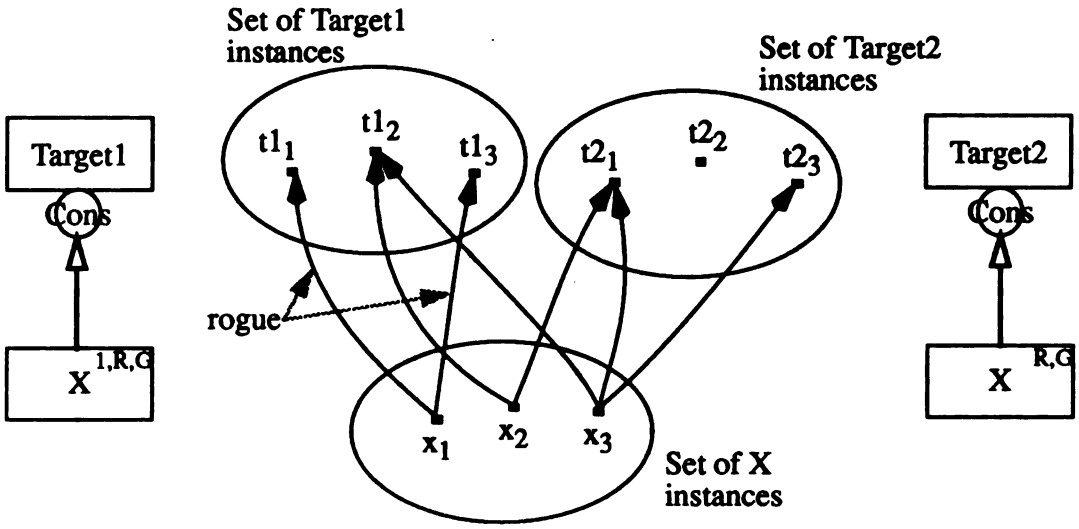


Comments:

(1) With respect to Target1, each instance of X can only originate one association arc. If we accept the link between x_1 and $t1_1$, then the link between x_1 and $t1_3$ is a rogue and a violation. Because of non-dependency, $t1_3$ is still valid after the removal of the link. The link between x_3 and $t1_2$ is valid, depending on the type of constructor.

(2) Each instance of X can originate multiple arcs to instances in Target2. The cardinality only applies toward the local constructor. Thus between Target1 and X, the source cardinality is 1, while between Target2 and X, it is M.

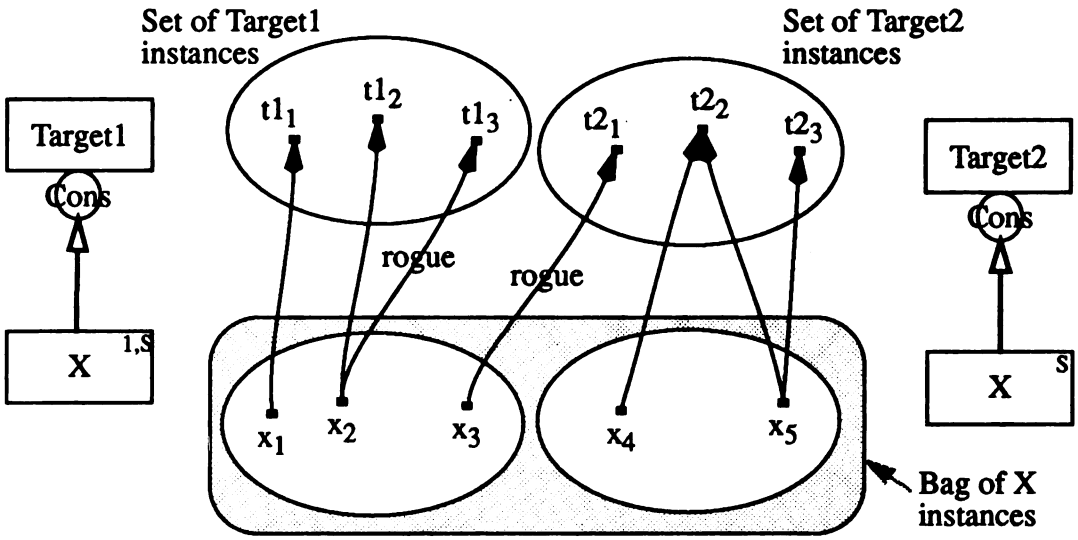
A.3.2. Global/Required context



Comments:

(1) One of the links emanating from x_1 is a rogue. Therefore, a correction by removing one of them will cause the other target instance to violate source dependency. Instance $t2_2$ is a direct violation of source dependency.

A.3.3. Shared/Non-Required context



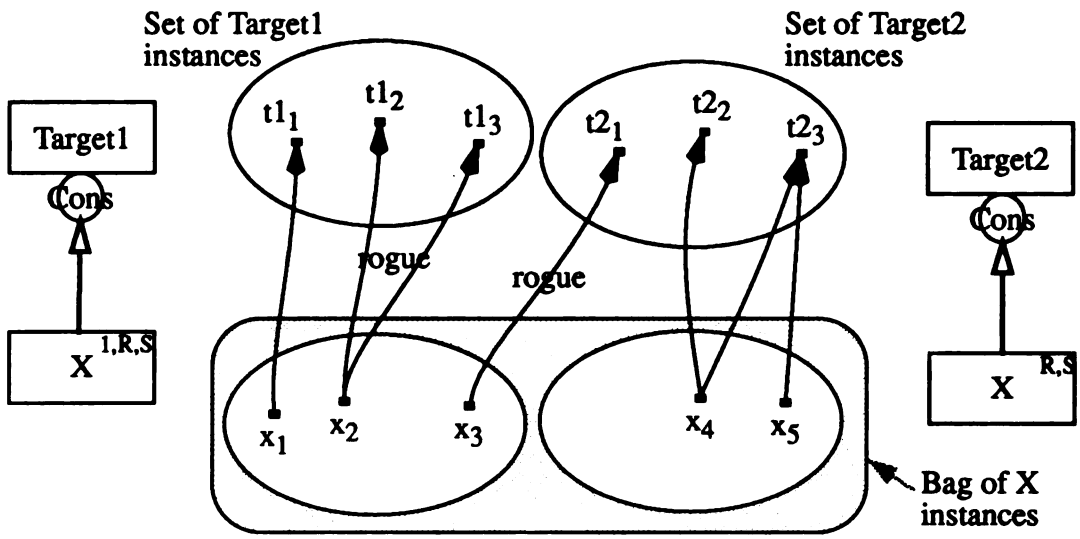
Comments:

(1) Between X and Target1, each instance of X should only originate one arc, thus the link between x_2 and $t1_3$ is a rogue and a violation of peer dependency. The link between x_3 and $t2_1$ is a rogue, but it is a violation of target dependency.

(2) Between X and Target2, the cardinality is M and because of shared-context, this set of X

instances are isolated from Target1's set of X instances. Both $t1_3$ and $t2_1$ are valid because of non-required source dependency.

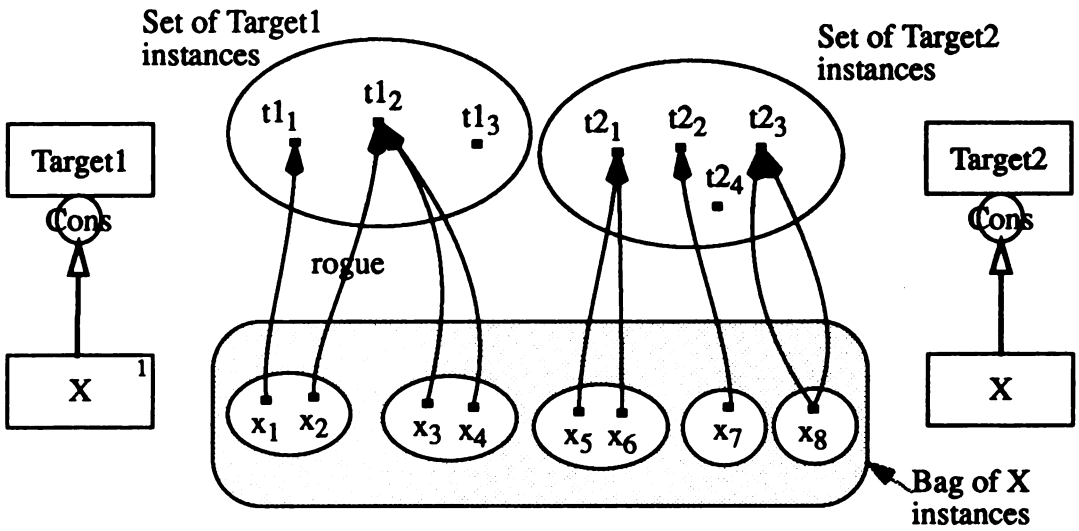
A.3.4. Shared/Required context



Comments:

(1) One of the links emanating from x_2 is a rogue and removing it will cause the other target instance to violate source dependency. The link between x_3 and $t2_1$ is a rogue, but it is a violation of target dependency. Removing this link will also cause $t2_1$ to violate source dependency.

A.3.5. Local/Non-Required Context



Comments:

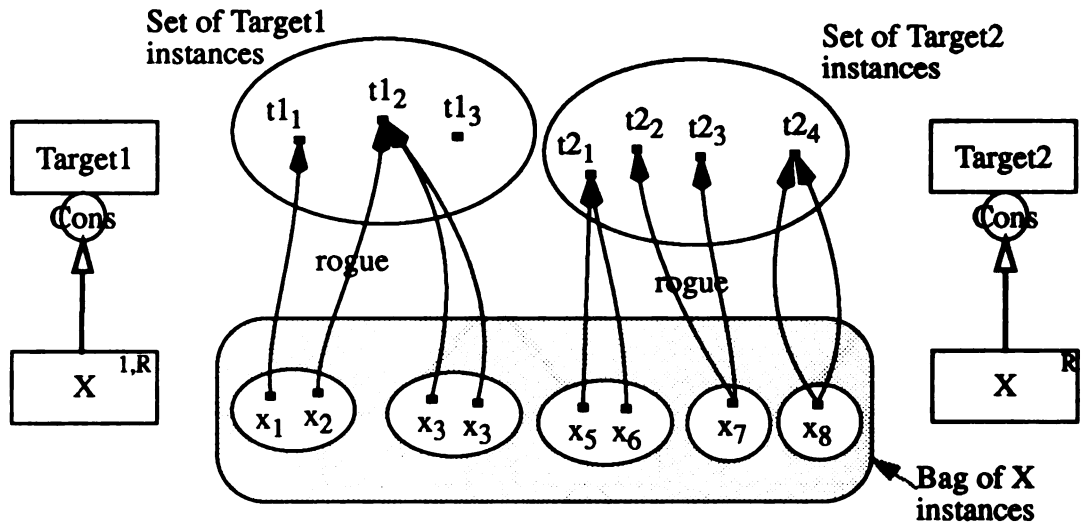
(1) Since X is local-context, there is only one target for each X instance. The link between x_2

and $t1_2$ is a rogue and a violation of target dependency.

(2) For the case where cardinality is M, still only one link can originate from each X instance because of the local-context. A redundant link, e.g. one of the two links between x_8 and $t2_3$, is invalid because instance association diagrams do not support multiplicity. However, if a Bag constructor is defined, this might become possible, indicating the multiplicity of occurrence of an instance in its target. If so, this would generalize to all contexts and dependencies.

(3) Despite their dangling references, $t1_3$ and $t2_4$ are valid because of source non-dependency.

A.3.6. Local/Required Context



Comments:

(1) Instance $t1_3$ is a direction violation of source dependency. The link between x_2 and $t1_2$ is a rogue and a violation of target dependency. Removal of it will not affect $t1_2$.

(2) One of the arcs emanating from x_7 is a rogue because of target dependency violation. Removal will cause the other target instance to violate source dependency.

APPENDIX B. Ternary Relationships

The truth table mechanism for peer dependency was introduced in Section 8.4.1. As a demonstration of its generality, we shall use it to analyze ternary relationships. The basic questions are:

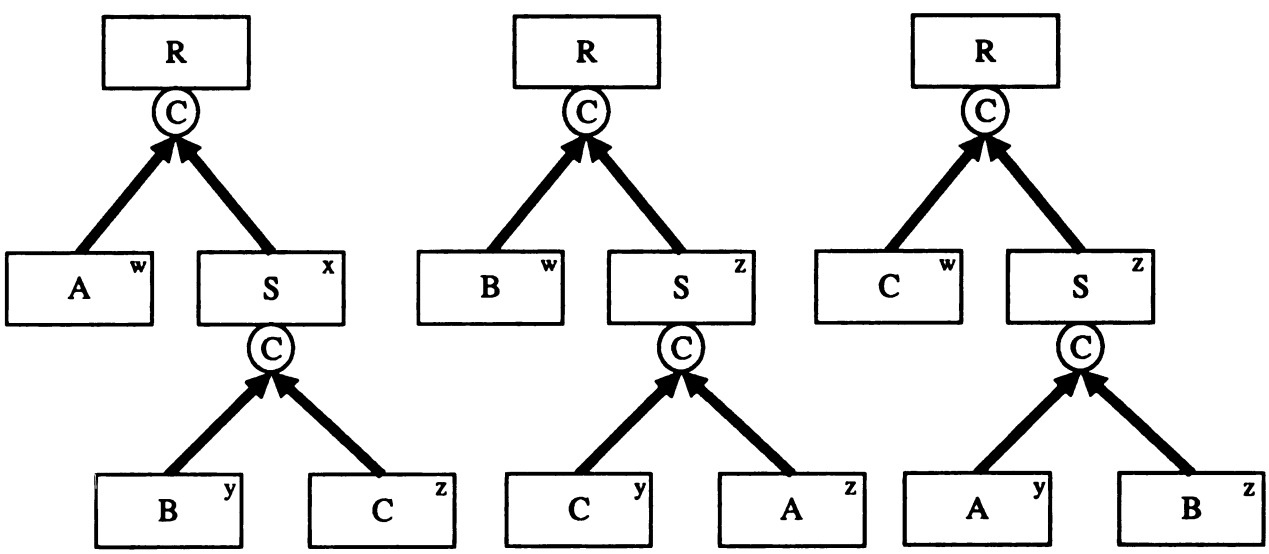
(1) Under what circumstances is a ternary relationship truly ternary, i.e. can it be decomposed into a cascaded binary relationship?

(2) What differences exist between functional dependency and involvement cardinality constraints on ternary relationships?

B.1. Binary Cascades

Assuming target dependency is not a factor in this analysis, i.e all elements are global-context, we will consider the simple case of three ID sources. There are three possible cascades:

Cascade I: Cascade II: Cascade III:



For each cascade, the values of w, x, y, and z could be either 1 or M to form 2^4 or 16 possible trees. Binary relationships based on functional dependency is the "converse" of involvement cardinality, thus 1–M becomes M–1 while 1–1 and M–M remains the same. Therefore, all cases of binary functional dependency can be mapped onto the cases for involvement cardinality and do not need to be analyzed separately. From this point, we can safely let our cardinality numbers represent involvement cardinality. Cascade II is a rotation of Cascade I, i.e. A → B → C, and Cascade III is another rotation, A → C → B. Thus, once we have the truth table for Cascade I, we can generate the truth table for Cascades II and III by rotation of A, B and C, which is equivalent to row permutation. We now focus on Cascade I.

B.2. Cascade Predicates

To determine whether a new instance can be inserted is determined by the relationship between A and S and the relationship between B and C. The relationships can be expressed as the following predicates:

W	X	Predicate
1	1	$A \leftrightarrow S$
1	M	$A \rightarrow S$
M	1	$S \rightarrow A$
M	M	None

Y	Z	Predicate
1	1	$B \leftrightarrow C$
1	M	$B \rightarrow C$
M	1	$C \rightarrow B$
M	M	None

where $A \equiv (a_1=a_2)$, $B \equiv (b_1=b_2)$, $C \equiv (c_1=c_2)$, and $S \equiv (B \text{ AND } C)$

We now have the 16 cases for Cascade I:

Case	W	X	Y	Z	Equivalent Predicate
1	1	1	1	1	$(A \leftrightarrow S)$ AND $(B \leftrightarrow C)$
2	1	1	1	M	$(A \leftrightarrow S)$ AND $(B \rightarrow C)$
3	1	1	M	1	$(A \leftrightarrow S)$ AND $(C \rightarrow B)$
4	1	1	M	M	$(A \leftrightarrow S)$
5	1	M	1	1	$(A \rightarrow S)$ AND $(B \leftrightarrow C)$
6	1	M	1	M	$(A \rightarrow S)$ AND $(B \rightarrow C)$
7	1	M	M	1	$(A \rightarrow S)$ AND $(C \rightarrow B)$
8	1	M	M	M	$(A \rightarrow S)$
9	M	1	1	1	$(S \rightarrow A)$ AND $(B \leftrightarrow C)$
10	M	1	1	M	$(S \rightarrow A)$ AND $(B \rightarrow C)$
11	M	1	M	1	$(S \rightarrow A)$ AND $(C \rightarrow B)$
12	M	1	M	M	$(S \rightarrow A)$
13	M	M	1	1	$(B \leftrightarrow C)$
14	M	M	1	M	$(B \rightarrow C)$
15	M	M	M	1	$(C \rightarrow B)$
16	M	M	M	M	none

B.3. Cascade Truth Table

The truth table for the sixteen cases of Cascade I are:

Truth Values for A, B, C		Cases															
A	B	C	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
T	T	F	F	F	F	F	F	F	F	F	F	F	T	T	F	F	T	T
T	F	T	F	F	F	F	F	F	F	F	F	T	F	T	F	T	F	T
T	F	F	F	F	F	F	F	F	F	F	T	T	F	T	T	T	T	T
F	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
F	T	F	F	F	T	T	F	F	T	T	F	F	F	T	F	F	T	T
F	F	T	F	T	F	T	F	T	F	T	F	T	T	T	F	T	F	T
F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

To generate Cascade II, we perform permutation on the truth values of A, B, and C:

Truth Values for Cascade II			Equivalent Cascade I			
A	B	C	A	B	C	
T	T	T	T	T	T	no change
T	T	F	T	F	T	row 3 of Cascade I
T	F	T	F	T	T	row 5
T	F	F	F	F	T	row 7
F	T	T	T	T	F	row 2
F	T	F	T	F	F	row 4
F	F	T	F	T	F	row 6
F	F	F	F	F	F	no change

After row permutation, the first eight cases of Cascade II would be the following:

Truth Values for A, B, C			Cases for Cascade II							
A	B	C	1	2	3	4	5	6	7	8
T	T	T	T	T	T	T	T	T	T	T
T	T	F	F	F	F	F	F	F	F	F
T	F	T	F	F	F	F	T	T	T	T
T	F	F	F	T	F	T	F	T	F	T
F	T	T	F	F	F	F	F	F	F	F
F	T	F	F	F	F	F	F	F	F	F
F	F	T	F	F	T	T	F	F	T	T
F	F	F	T	T	T	T	T	T	T	T
Cascade I equivalent:			1	9	2					10

B.4. Results

Note that several cases between Cascade I and Cascade II overlaps. After fully enumerating all 48 possible cases (3 cascades with 16 cases each), only 32 are unique. Each of the standard ternary cases under involvement cardinality can be matched to a case in a cascaded binary relationship. On

the other hand, the ternary case 1-1-M and 1-M-M under functional dependency are not decomposable. In a ternary table, there are 2^3 - 2 or 6 rows (excluding all T and all F), each of which can be T or F, resulting in 2^6 or 64 unique columns or cases. We have already enumerated 32 such cases from the cascaded binary compositions, thus leaving 32 cases of pure ternary relationships that are not decomposable into cascades. Of the 32, two are identified by ternary functional dependency.

In general, we can fully describe a n-ary relationship as a truth table column using $O(2^n)$ space. We also can decide if a relationship is truly n-ary by comparison against tables constructed from relationships of lesser degree. In theoretical terms, this is equivalent to predicate transformation and logic synthesis, both of which have a vast amount of literature. Since it is outside the scope of this thesis, we will not pursue the connection further.

APPENDIX C. Examples of EOM Schema

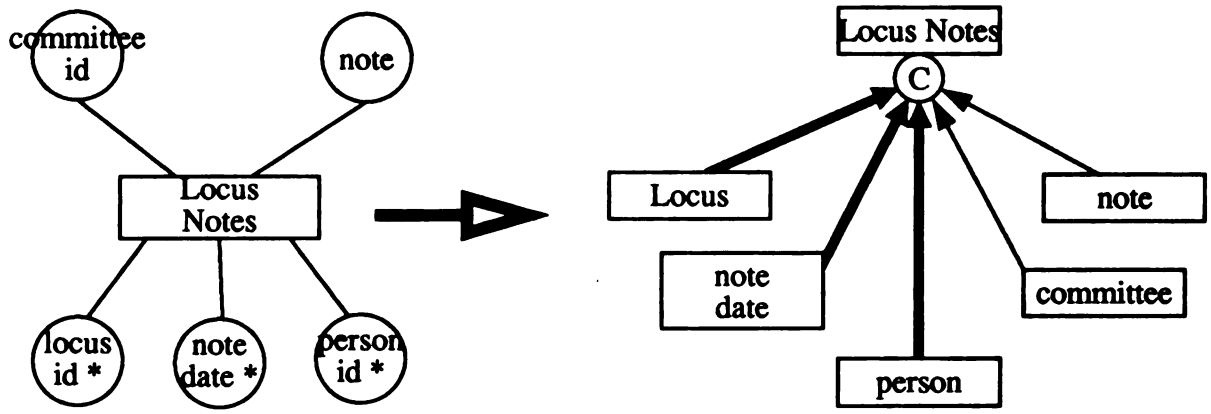
C.1. Transformation of ER Diagrams

An example of a large but basic ER diagram is shown in Figure C.1. This is taken from Genome Data Base (GDB) [13]. It represents the conceptual model associated with locus information. In this exercise, we will take the ER diagram and convert it into an Extensible Object schema. There are three basic stages: structure, context, and storage.

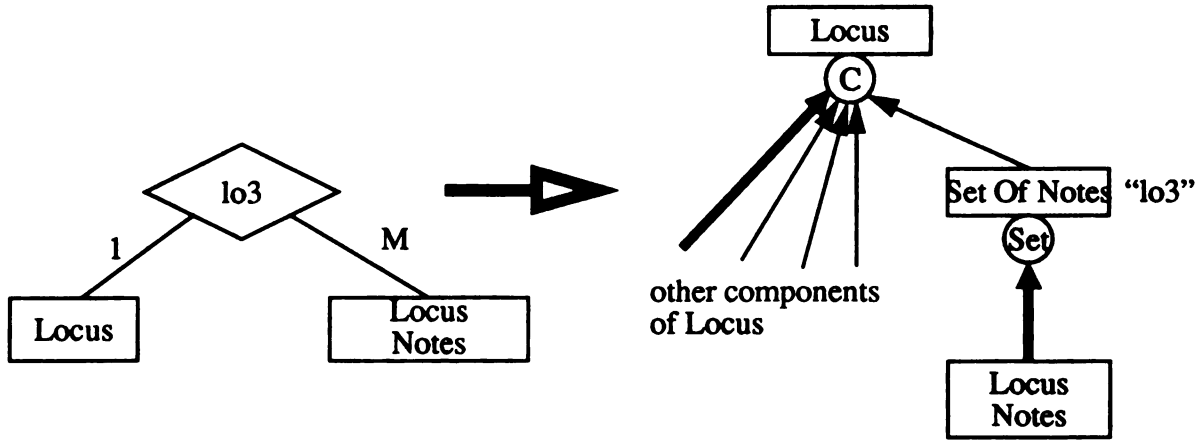
C.1.1. Structure Determination

The first stage determines structural relationships between Object-Types. For ER conversion, it consists of the following steps:

(1) Map all the entities and attributes into composition constructors. All key attributes (marked with "*") are converted to ID arcs. Since EOM is not bound by implementation specific types, all references to "id" is replaced by the original object. For example:



(2) Examine 1-to-M relationships and decide if a set constructor is more appropriate. If so, then redundant ID components can be dropped. For example:



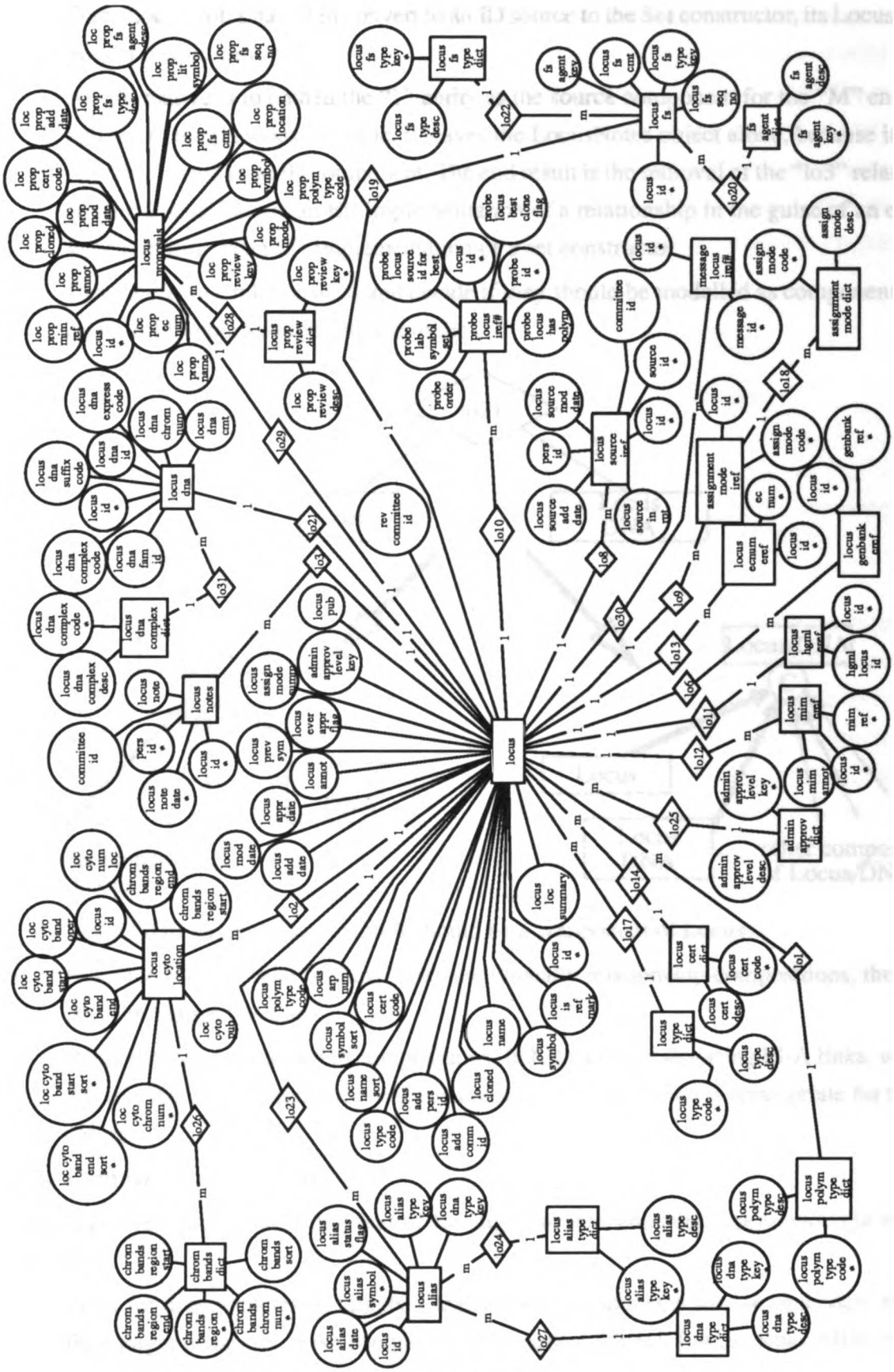
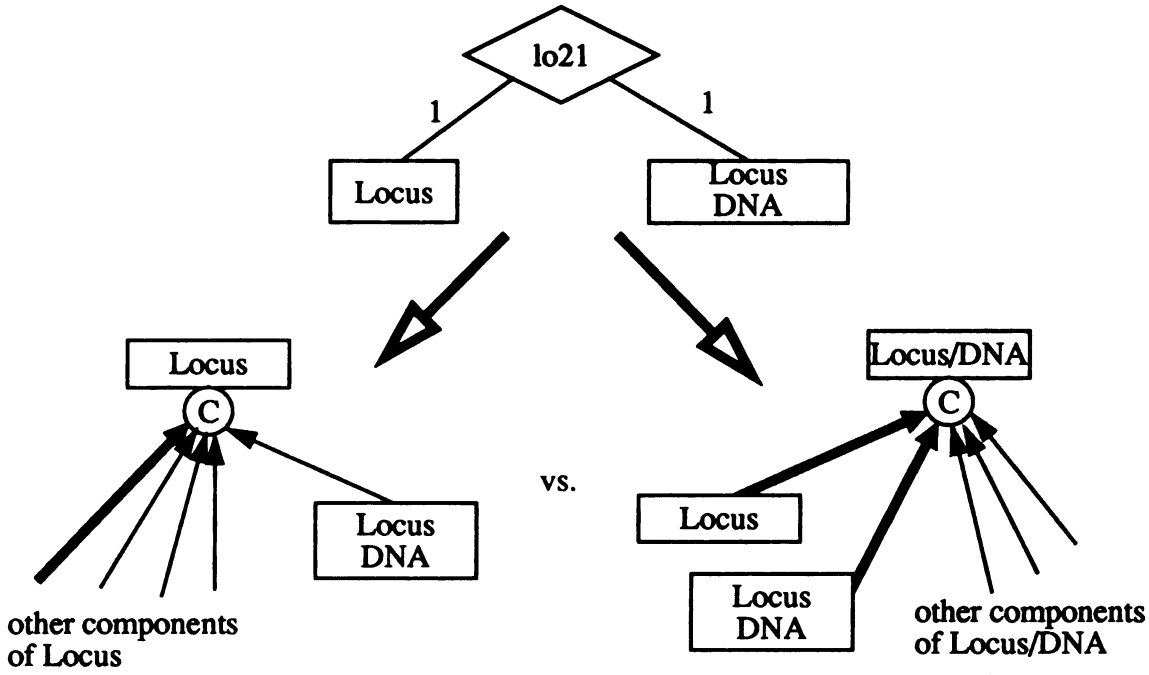

locus
locus mnemonic
locus dna
locus cyto location
locus notes
locus dna complex dict
chrom bands dict
locus alias
locus alias type dict
locus dna type dict
locus polym type dict
locus type dict
locus cert dict
admin approv dict
locus genbank cref
locus hgml cref
locus mim cref
locus ecnum cref
assignment mode dict
assignment mode iref
locus source iref
probe locus iref#
message locus iref#
locus fs
fs agent iref
locus fs type dict
locus prop review dict
locus prop fs
locus prop lit
locus prop dna
locus prop ec
locus prop name
locus prop mnemonic
locus prop polym type code
locus prop location
locus prop symbol
locus prop fs cmt
locus prop review key
locus prop review desc
locus prop mode
locus prop mod date
locus prop add date
locus prop cert code
locus prop closed
locus prop annot
locus prop mim ref
locus prop fs seq no
locus prop fs type desc
locus prop fs agent desc
locus dna express code
locus dna chrom num
locus dna cmt
locus dna id
locus dna suffix code
locus dna complex code
locus dna fam id
locus dna complex desc
locus dna complex key
locus note
locus note date
locus pub
locus assign mode num
locus prev sym
locus ever appr flag
locus annot
locus appr date
locus mod date
locus add date
locus cyto num loc
chrom bands region end
chrom bands region start
loc cyto band oper
loc cyto band start
loc cyto band end
loc cyto band start sort
loc cyto chrom num
loc cyto band end sort
loc cyto num
chrom bands region
chrom bands chrom num
chrom bands sort
locus alias status flag
locus alias symbol
locus alias date
locus alias type key
locus dna type key
locus alias type desc
locus polym type desc
locus polym type code
locus dna type desc
locus type desc
locus type code
locus cert desc
locus cert code
locus mim annot
locus genbank ref
hgml locus id
mim ref
admin approv level key
admin approv level desc
locus loc summary
locus is ref mark
locus symbol
locus name
locus cloned
locus add pers id
locus add comm id
locus type code
locus symbol sort
locus name sort
locus arp num
locus fs type key
locus fs type desc
locus fs cmt
locus fs seq no
fs agent key
fs agent desc
probe locus best clone flag
probe locus source id for best
probe lab symbol set
probe order
probe locus has polym
probe id
message id
assign mode code
assign mode desc
committee id
rev committee id
source id
locus source mod date
locus source add date
locus source in mt
pers id
ec num
locus id
lo1
lo2
lo3
lo6
lo8
lo9
lo10
lo11
lo12
lo13
lo14
lo17
lo18
lo19
lo20
lo21
lo22
lo23
lo24
lo25
lo26
lo27
lo28
lo29
lo30
lo31

Figure C.1

Once LocusNotes has been convert to an ID source to the Set constructor, its Locus ID component can be removed.

An alternative is to embed the "1" entity as the source component for the "M" entity. If applied to the previous example, it leaves the LocusNotes object alone, because it already contains Locus as an ID component. The end result is the removal of the "lo3" relationship. In fact, LocusNotes is an ER implementation of a relationship in the guise of an entity, which, in turn, is an ER implementation of a set constructor.

(3) Examine 1-to-1 relationships and decide if they should be modelled as components or as a relationship-composition. For example:



In the GDB example, LocusDNA should be a component of Locus.

(4) Examine M-to-M relationships. They are typically relationship-compositions, therefore, they are mapped as such.

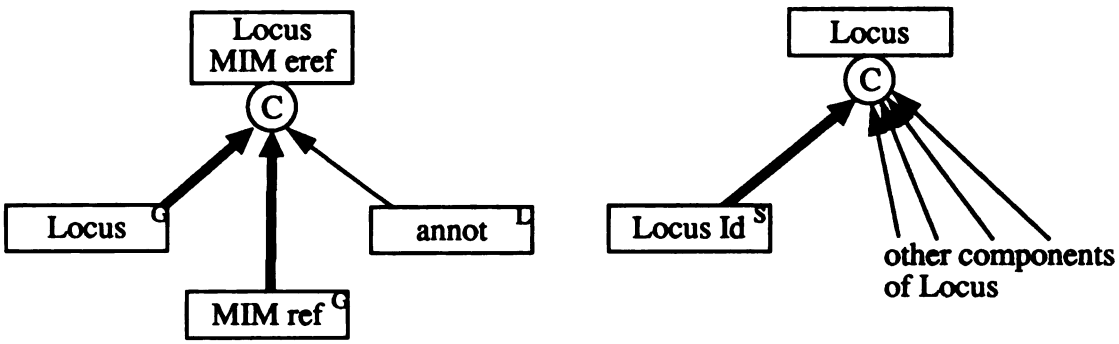
(5) Weak entities are mapped to compositions. In ER models supporting IS-A links, we would convert them into inheritance or union constructors, whichever is appropriate for the application.

C.1.2. Context Determination

In this stage, we attach context designators for target, source, and peer dependencies in three separate steps:

(1) Look for dependencies on targets. Clearly, some components are not dependent at all on their target instances, these components are marked Global. For example, MIM_ref under

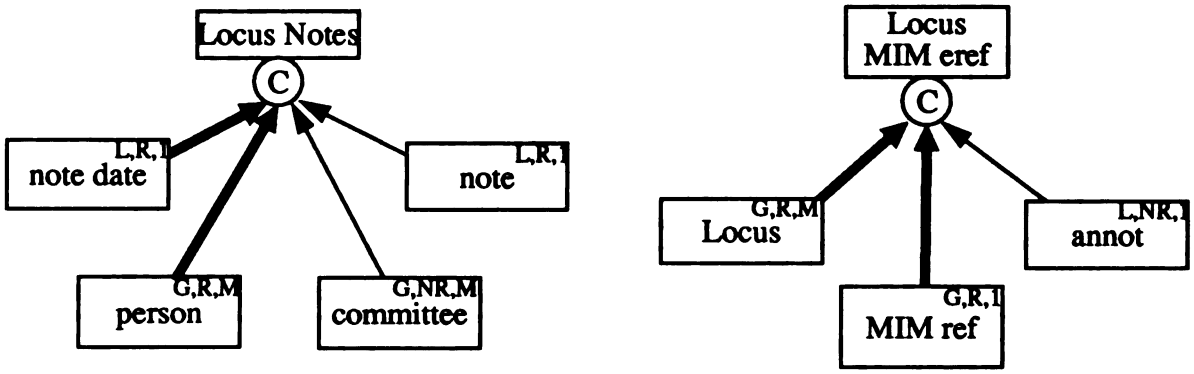
LocusMIM_eref is Global. Other components are dependent and specific to their target instances, they are marked Local. For example, LocusMIM_annot in LocusMIM_eref is specific to each target instance, therefore, they should be considered Local. Shared components are the hardest to determine. Invariably, they are represented in ER models as "id" values. For example, Locus_id is dependent on Locus instance, but the set of Locus_id values is shared among all Locus instances and the id-to-instance integrity will be maintained only through source and peer dependency. The result of these examples is diagrammed as follows:



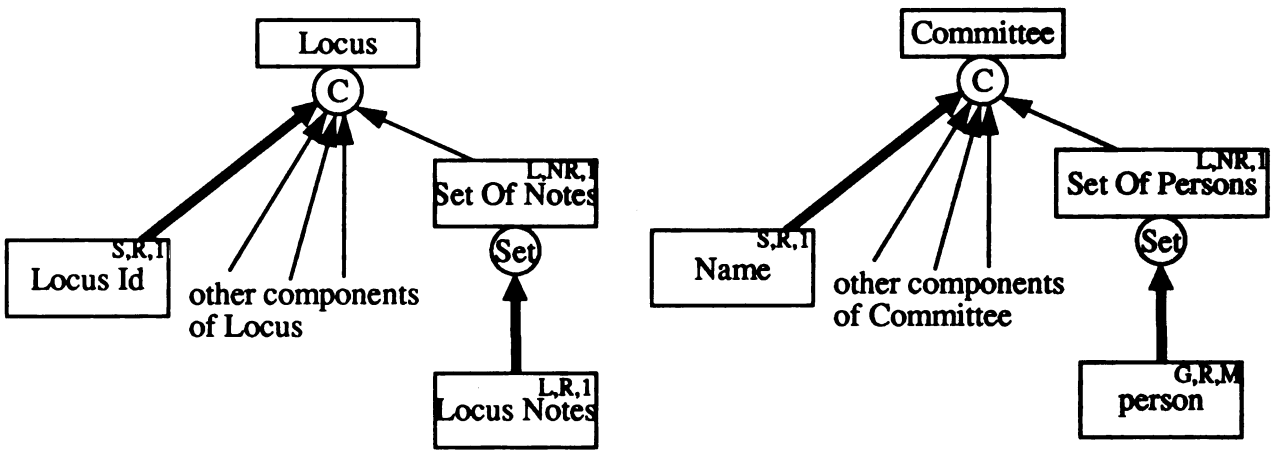
(2) Determine source dependencies, i.e. whether a component is required or not required. An Object-Type that is the only ID component of a target is typically required, unless one wishes to preserve a possible "null" value for other uses, e.g. defaults or summaries. Since EOM does not distinguish null vs. non-null in terms of value, this source dependency relates to domain integrity. The ER diagram provided from GDB does not designate null or non-null, therefore, the conversion will be based on guessing the domain semantics.

(3) Finally, we convert cardinality constraints from ER functional dependency to Extensible Object Model involvement cardinality. For binary relationships, this amounts to the reversal of "1" and "M". In general, Local components for Composition, Inheritance, and Union are set to "1" because, in their target context, they are unique.

Using the previous examples, the Extensible Object schema is diagrammed as follows:



The target dependency of SetOfNotes under Locus should be Local, since an instance of SetOfNotes is only associated with one Locus instance. LocusNotes, now as an ID component to the SetOfNotes, can be either Local or Shared, because SetOfNotes is already partitioned into disjoint sets of one element each. If one wish to associate a LocusNote instance to several Locus instances, then LocusNotes will have to be global and its cardinality set to "M". This is demonstrated in the Committee Object-Type (not derived from the GDB diagram).

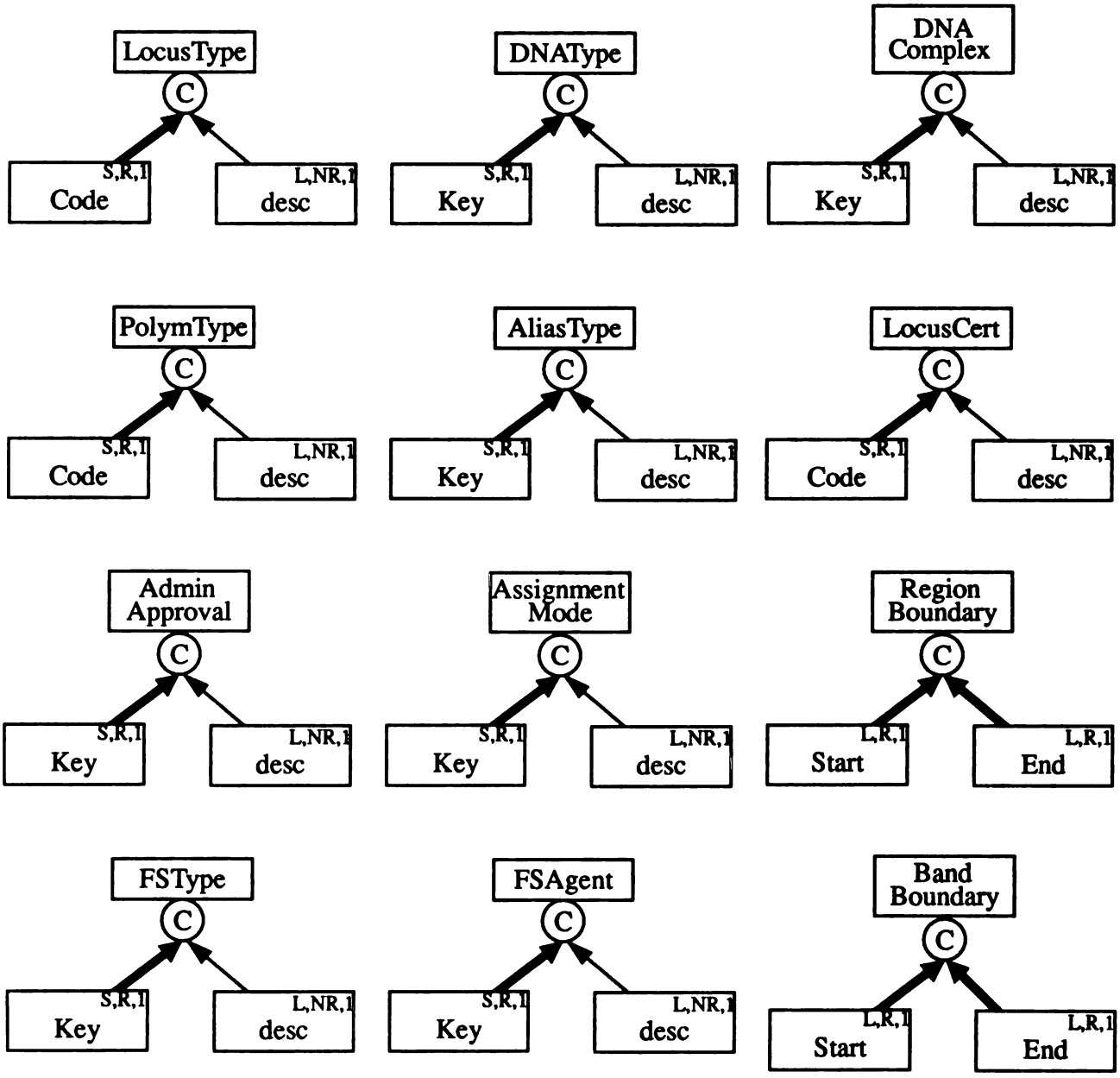


C.1.3. Storage Determination

The last step is determining the storage characteristics. This stage is highly implementation specific. Therefore, EOM does not permit a graphical representation for them. However, in the query language, there are provisions for specifying storage attributes so that physical optimization may be carried out (see Section 10.1.2). Since we do not designate storage directives in the EOM schema. This step will be ignored.

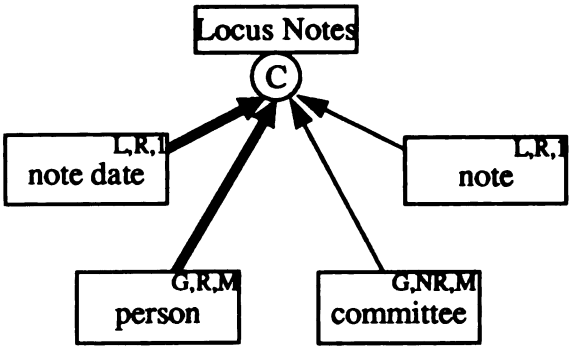
C.1.4. Result of Transformations

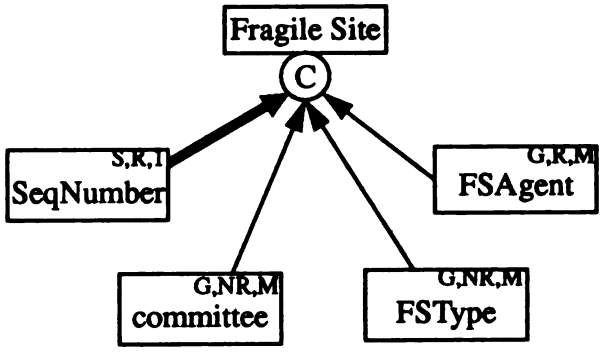
The simple entities are mapped to the following:

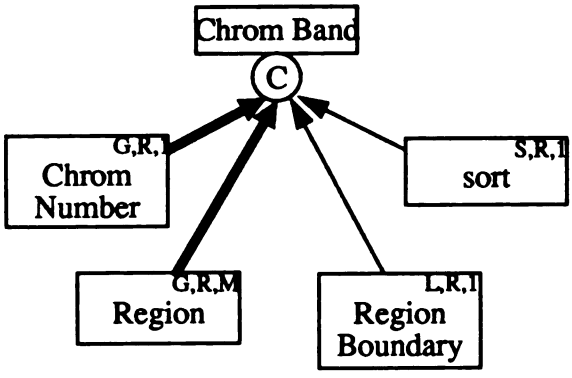


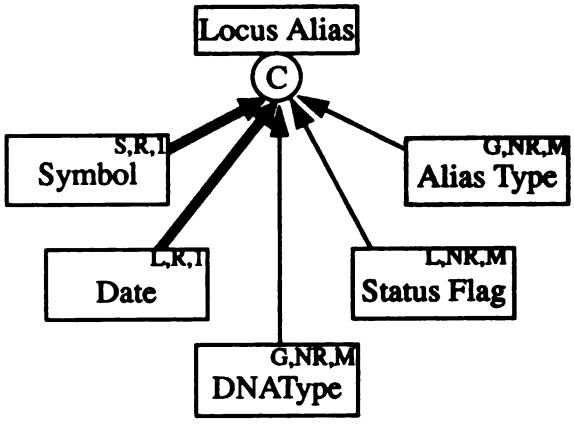
The reason these are "simple" is because all their characterization is encapsulated into the "desc" Object-Type. Conceivably, these can be further defined into more details.

The more complex entities are mapped to the following:

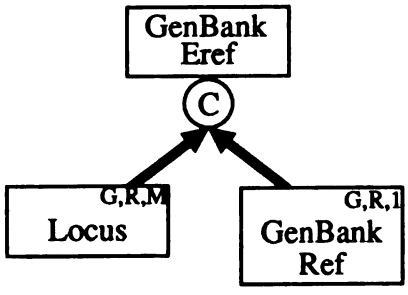


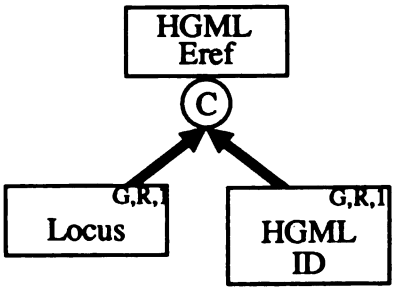


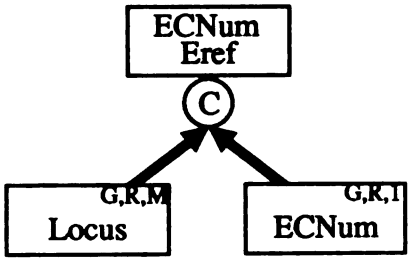


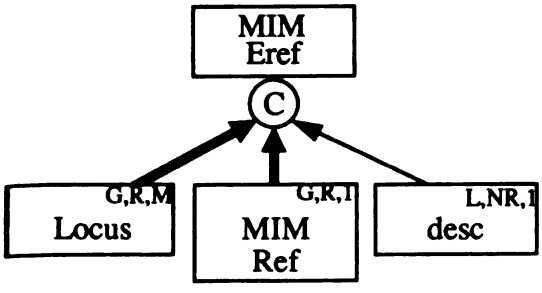


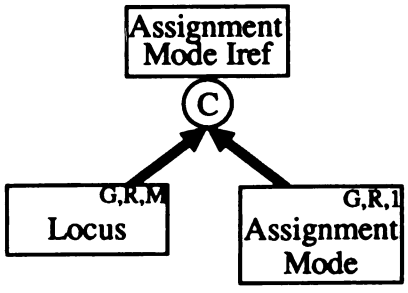
The simple relationships are:

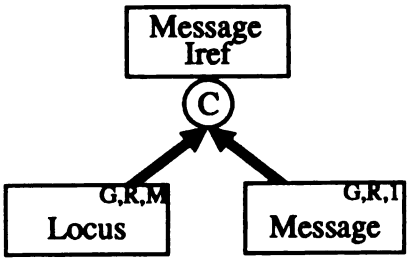








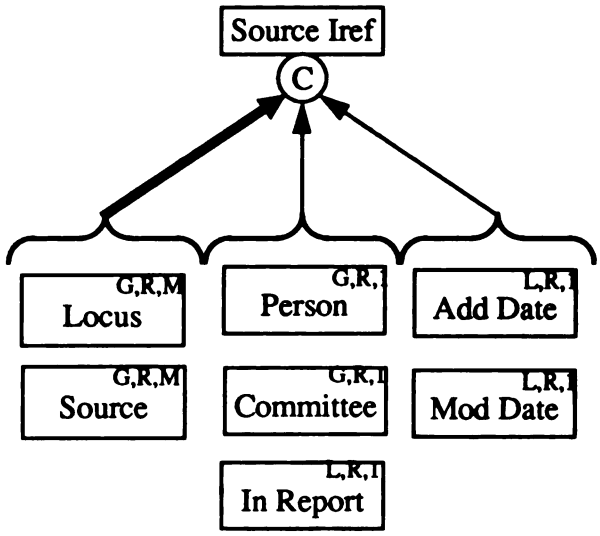


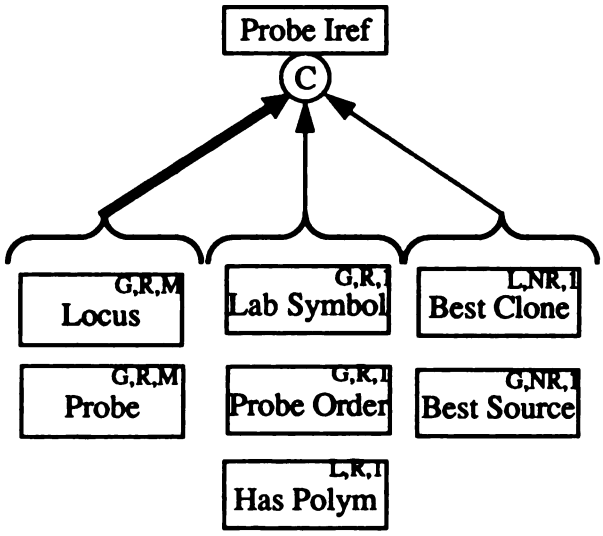


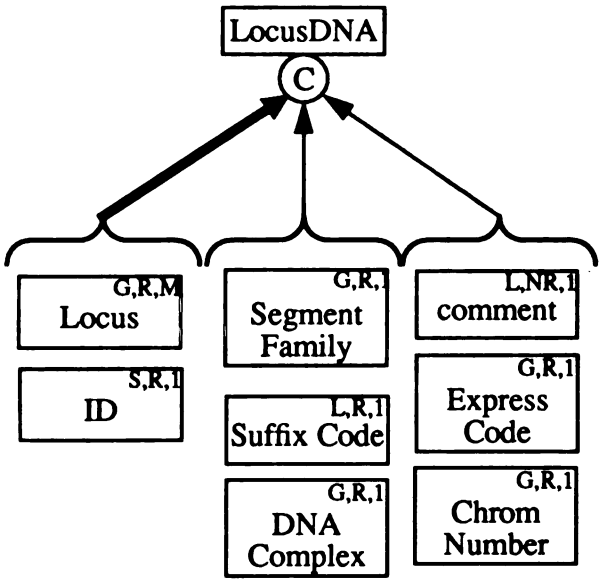
The more complex relationships are mapped to the following:

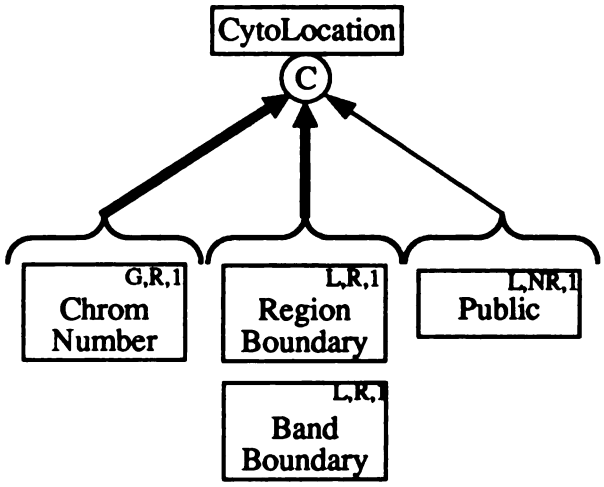
Notation: A horizontal "}" represent multiple arcs of the same type:

This allows grouping of similar components together to improve organization.

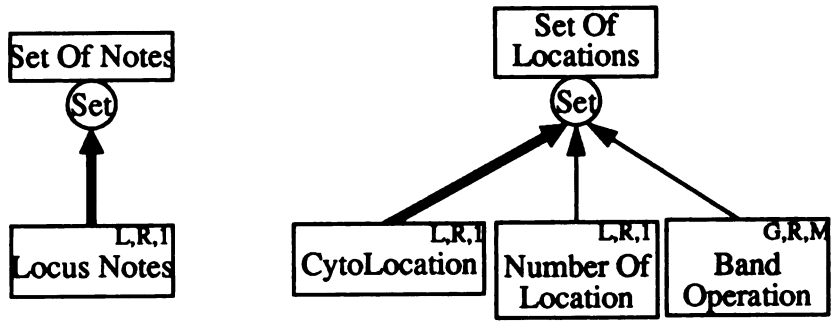




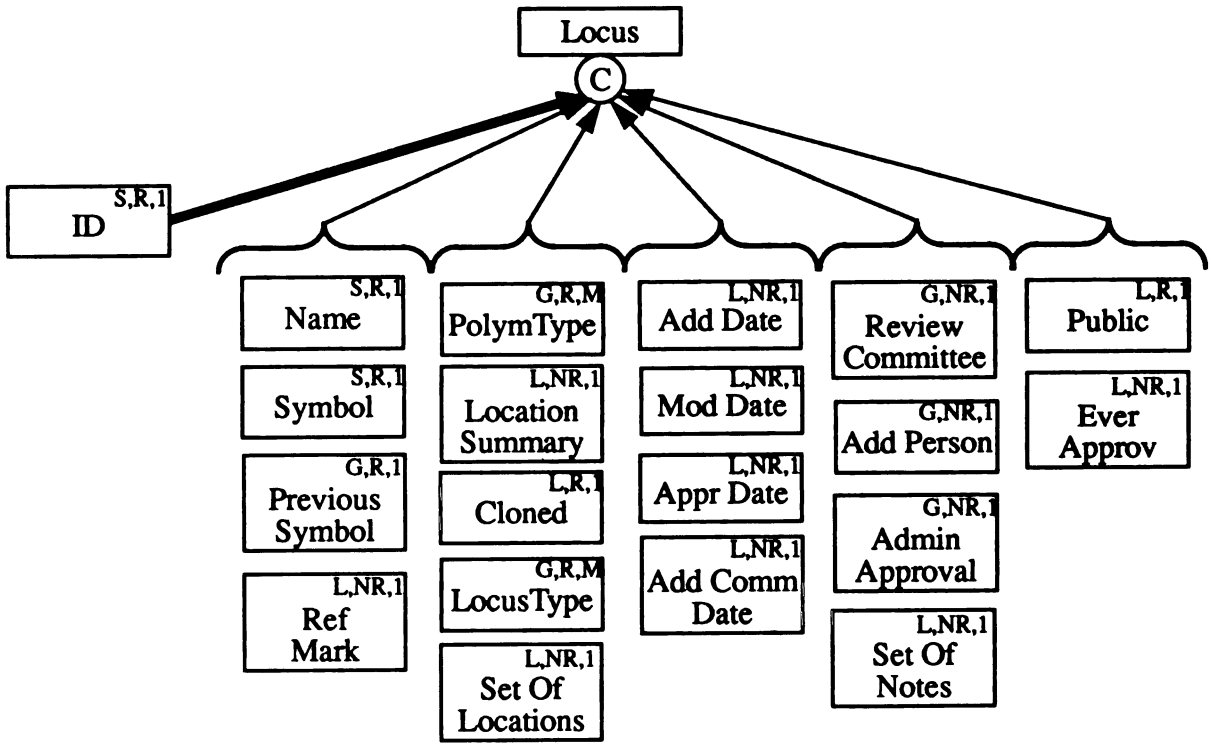




The mapped set constructors are the following:



Finally, we have the following:



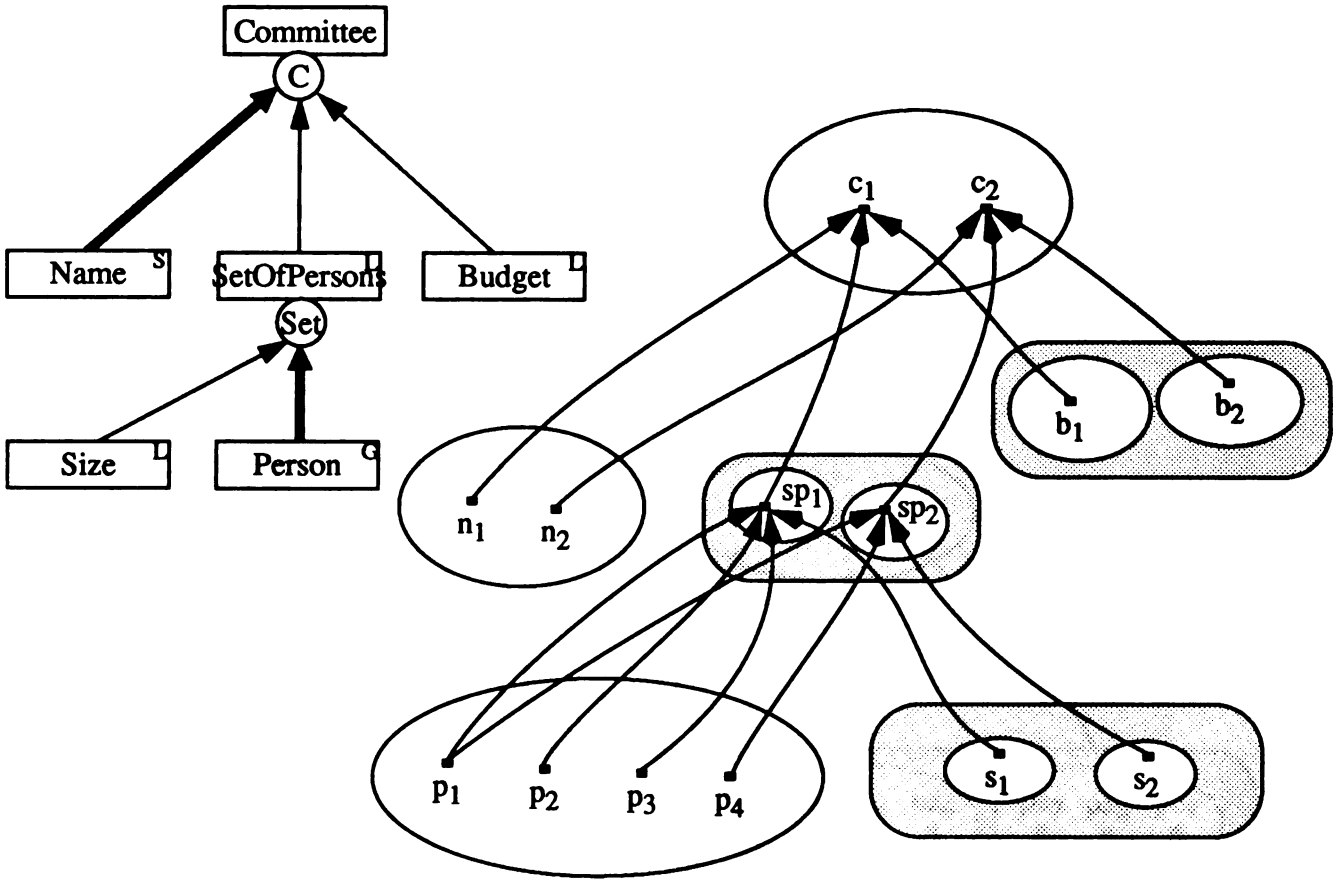
"Locus Proposal" is left as an exercise for the reader.

C.2. Set and Sequence Constructors Under Target Context

In the following examples, we will examine some of the complex interaction between constructors and target contexts. The question we will ask is: are all target dependencies meaningful for the ID source of a Set or Sequence constructor?

C.2.1. Global Context

Going back to the Committee example in Section 7.2.1, if the Person is set to global-context and the SetOfPersons is local-context, we would end up with this diagram:



Comments:

(1) Each instance of Committee is composed of an instance of Name, SetOfPersons, and Budget. SetOfPersons, Budget, and Size are local-context, thus each instance value is isolated from the other values. If there is another instance of Committee, c_3, and it is composed of an instance of SetOfPersons, sp_3, such that sp_3 is composed of $\{p_1, p_2, p_3\}$, then sp_3 is value-equal to sp_1, but is a separate and distinct instance by context.

(2) Person is global, thus the SetOfPersons instances are derived from a common set of Person

instances. The Person instance, p_1, of sp_1 (ID = $\{p_1, p_2, p_3\}$) and of sp_2 (ID = $\{p_1, p_4\}$) are identical, i.e. same person.

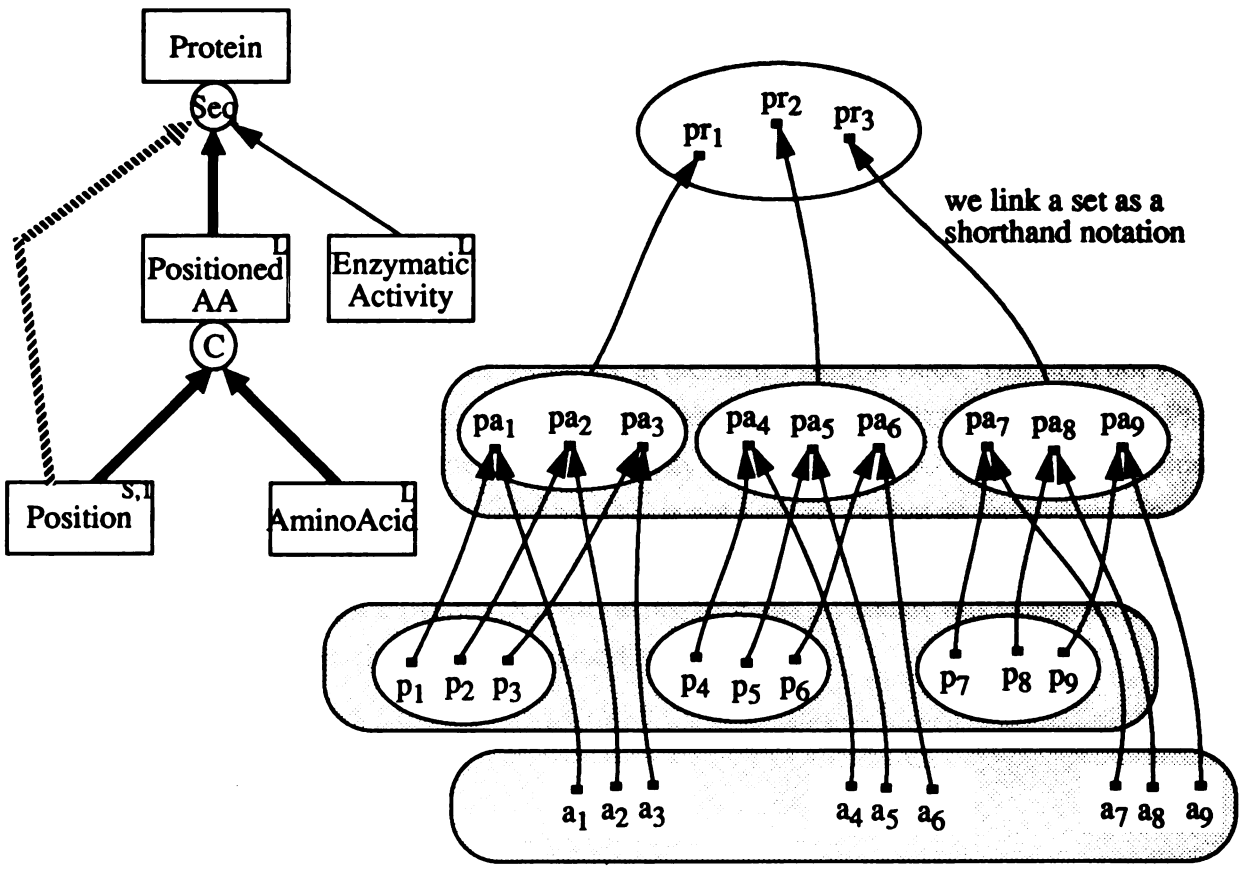
The result of this combination of contexts is that the same set of persons can function as different Committees and operate with a separate budget where the nature of the committee is probably associated with the Name given.

C.2.2. Shared Context

Shared Context is similar to Global context because the elements for a Set or Sequence is drawn from the same set. The only difference is that they are partitioned based on the target instance (set or sequence instance) sets.

C.2.3. Local Context

Using the protein example in Section 7.3.2, we need to assign the context dependencies such that changes in a protein's sequence can be made without affecting other proteins, at the same time retaining the ability to verify against previously inserted protein sequences in the database. Assigning global-context to Protein and local-context to Positioned-AA satisfies these requirements. Here is how the schema and its instance association diagram would look:



Comments:

(1) The instance set for Enzymatic Activity is not shown.

(2) Each instance of Protein is associated with an independent set of PositionedAminoAcids. There are three proteins: $< a_1 , a_2 , a_3 >$, $< a_4 , a_5 , a_6 >$, and $< a_7 , a_8 , a_9 >$. As long they are distinct sequences, repeated values are permitted. For example, pr_1 could have the value < Val , Phe , Tyr >, pr_2 could have <Val , Phe , Phe >, and pr_3 = < His , Phe , Tyr >. The position is implicit in the order of the amino acid occurrence.

(3) The set boundaries between each instance of AminoAcids has not been drawn for brevity. The shaded area around the Positioned-AA, Position, and AminoAcid are "bags".

(4) Position is shared-context in Positioned-AA to maintain correct set membership relationship between Positioned-AA and Position. In this case, values of $p_1=p_4=p_7=1$, $p_2=p_5=p_8=2$, and $p_3=p_6=p_9=3$. As a set, p_1, p_2, and p_3 are distinct.

(5) AminoAcid is an ID component of local-context, therefore we can have multiple equal-valued instances. For example, < Val , Phe , Phe >.

Using the example values, we can change the 3rd amino acid of pr_1 to His, without affecting the 3rd amino acid of pr_3 because of the local-context separation of AminoAcid. The result protein instances are: < Val , Phe , His >, < Val , Phe , Phe >, and <His , Phe , Tyr >. However, we are prevented from changing the value of a_3 to Phe, because the ID(pr_1) is < Val , Phe , Tyr > before the change and < Val , Phe , Phe > afterwards, which would now equal ID(pr_2). This integrity is maintained by the fact that ID is based on value only and the new ID value of pr_1 would now be equal to ID(pr_2).

Another change we can perform is Position value, if we change the value of p_3 to 2, we could create a protein with two different amino acids, a_2 and a_3, at the same Position value. This does not violate the construction of Positioned-AA. For example, the value of pr_1 now becomes { [1,Val] , [2,Phe] , [2,Tyr] } and is a unique value. However, now we only have { p_1 , p_2 } as the set of pr_1 positions and two association arcs will originate from p_2. This is not a sequence, but an ordered set. To maintain semantic consistency with sequence, we need to add a peer dependency constraint, so that only one arc is permitted to originate from each position instance and invalidate this change of p_3 value. Therefore, Position is given the involvement cardinality of "1".

APPENDIX D. Query Language Syntax

The syntax listed in this Appendix is a stripped version. The actual one has numerous error trapping rules which are not relevant to this discussion. Therefore, some rules have been condensed together to improve readability. Keywords are in upper-case and is translated to unique numeric values by the lexical analyzer. However, NAME, NUMBER, and STRING are supplied by the lexical analyzer as special token nodes.

D.1. Chaining Statements

The starting point for the parser is `stmtlist`.

```
opt_stmtlist        : stmtlist
                    | /* NULL */
                    ;

stmtlist            : statement
                    | stmtlist statement
                    ;
```

D.2. Basic Statement Types

```
statement           : dba_statement ';'
                    | bypass_statement ';'
                    | read_statement ';'
                    | display_statement ';'
                    | define_stmt ';'
                    | undef_statement ';'
                    | storage_statement ';'
                    | dbinst_statement ';'
                    | assign_statement ';'
                    | query_statement
                    | flow_statement
                    | compound_statement
                    | ';'
                    ;
```

All `statements` are terminated by ';', except for `compound_statement`. Since `query_statment` and `flow_statement` ends with a `statement`, the ';' is unnecessary.

D.3. Administration

```
dba_statement       : CREATE DB NAME
                    | DESTROY DB NAME
```

```
                  | USE DB NAME
                  ;

bypass_statement  : BYPASS STRING
                  ;

read_statement    : READ STRING
                  ;
```

D.4. Viewing Internal Table

```
display_statement : DISPLAY TYPE opt_pattern
                  | DISPLAY FUNC opt_pattern
                  | DISPLAY VAR opt_pattern
                  | DISPLAY VALUE opt_pattern
                  ;

opt_pattern       : STRING
                  | NAME
                  | /* NULL */
                  ;
```

If `opt_pattern` is a `STRING`, it is considered as a regular expression pattern. If it is a `NAME`, then it is used as an exact match pattern.

D.5. Type Definition

```
define_stmt       : DEFINE scoped_name IS define_phrase
                  ;

define_phrase     : ALIAS_OF scoped_name
                  | IMPL_BY PROC func_call
                  | IMPL_BY ENUM ':' arglist
                  | IMPL_BY scoped_name opt_storage
                  | constructor_token '{' componentlist '}'
                  ;
```

Although `scoped_name` is used, they are actually restricted to one name in this version. The validation module checks for violations.

```
constructor_token : COMPOSITION
                  | SET
                  | SEQUENCE
                  | INHERITED
                  | UNION
```

```
                    ;
```

This is where future extension of constructors are added.

```
componentlist       : component
                    | componentlist component
                    ;
```

Must have at least one `component` in a `componentlist`.

```
component           : ID scoped_name opt_contextlist opt_storage ';'
                    | PROPERTY scoped_name opt_contextlist opt_storage ';'
                    | ORDERED_BY scoped_name opt_contextlist opt_storage ';'
                    ;

opt_contextlist     : ':' context_list
                    | /* NULL */
                    ;

context_list        : context_token
                    | context_list ',' context_token
                    | /* NULL */
                    ;

context_token       : LOCAL /* target */
                    | SHARED /* target */
                    | GLOBAL /* target */
                    | REQUIRED /* source */
                    | NONREQ /* source */
                    | ONE /* peer */
                    | MANY /* peer */
                    ;
```

The `contexts` are pooled, because they can occur in any order and can be left out. The validation module will manage them correctly. This is also the place where future extensions are added.

```
opt_storage         : '[' DIRECT ']'
                    | '[' INDIRECT ']'
                    | '[' VIRTUAL ']'
                    | /* NULL */
                    ;
```

Currently only one storage directive is permitted. However, future extensions can be added here.

```
undef_statement     : UNDEFINE NAME
                    ;
```

D.6. Storage Management

```
storage_statement : CREATE STORE NAME
                  | DESTROY STORE NAME
                  ;
```

D.7. Instance Manipulation

```
dbinst_statement  : CREATE VAR NAME IN scoped_name
                  | DESTROY VAR NAME
                  | INSERT scoped_name
                  | UPDATE scoped_name
                  | DELETE scoped_name
                  | dbioutput_stmt
                  ;
```

The INSERT, UPDATE, and DELETE commands have corresponding overloaded function calls, e.g. insert(). The command calls the function for effecting change in the instance's context, then if necessary, will perform the actual DBMS action.

```
dbioutput_stmt    : OUTPUT opt_arglist output_redirect
                  ;

output_redirect   : ADDTO primitive
                  | /* NULL */
                  ;
```

This allows the output to be directed into a file.

D.8. Instance Assignment

```
assign_statement  : scoped_name '=' expr
                  | scoped_name ':=' expr
                  | scoped_name
                  ;
```

The scoped_name by itself is a special case of referencing the last instance in the scoped_name. However, if there are functions with side-effects along the way, the side-effects will be carried out.

D.9. Select Command

```
query_statement   : FOREACH varlist opt_whereclause PERFORM statement
                  ;
```

```
varlist             : var_def
                    | varlist ',' var_def
                    ;
```

At least one variable must be defined

```
var_def             : NAME IN scoped_name
                    ;

opt_whereclause     : WHERE '(' condition ')'
                    | /* NULL */
                    ;
```

D.10. Control Flow

```
flow_statement      : WHILE '(' condition ')' statement
                    | IF '(' condition ')' statement opt_else
                    | BREAK ';'
                    | CONTINUE ';'
                    | EXIT ';'
                    ;

opt_else            : ELSE statement
                    | /* NULL */
                    ;

compound_statement: '{' opt_stmtlist '}'
                    ;
```

D.11. Basic Expressions

```
opt_arglist         : arglist
                    | /* NULL */
                    ;

arglist             : argument
                    | arglist ',' argument
                    ;

argument            : condition
                    | expr
                    ;

condition           : condition OR condition
```

```
                    | condition AND condition
                    | NOT condition
                    | '(' condition ')'
                    | expr '=' expr
                    | expr '>' expr
                    | expr '<' expr
                    ;

expr                : expr '+' expr
                    | expr '-' expr
                    | expr '*' expr
                    | expr '/' expr
                    | '-' expr        %prec UMINUS
                    | '(' expr ')'
                    | value
                    ;
```

Need to specify unary minus precedence to ensure proper evaluation.

```
value               : scoped_name
                    | STRING
                    | NUMBER
                    ;

scoped_name         : simple_name
                    | scoped_name '.' simple_name
                    ;

simple_name         : NAME
                    | func_call
                    ;

func_call           : NAME '(' opt_arglist ')'
                    | overload '(' opt_arglist ')'
                    ;

overload            : INSERT
                    | DELETE
                    | UPDATE
                    ;
```

This is for handling overloaded names that matches keywords.

APPENDIX E. Query Processor Modules

The major modules of the Extended Object Query Processor are described in this Appendix. If there is a module-specific structure, it is first described, followed by some of the major functions in the module, and, finally, any interesting files in the module are mentioned.

E.1. Evaluators

This module handles the evaluation of statements.

`evaluate()` - takes a parse tree made of Node's and evaluates the command. This call is reentrant. Therefore, it can be called for evaluating subtrees. A large `switch` statement is used to call the specific evaluation functions.

`evalvalid()` - takes a `scoped_name` Node tree and performs type checking. Since expressions are converted to a `scoped_name` Node tree, this handles all static type checking.

`eval()` - the callback for the parser.

`evaltest.c` - the main body of the program. If linked with the stubs library, the program is a debugging version. If linked with the Sybase DB library, the program is a working version.

E.2. Functions

This modules manages the operators (functions) visible to the user.

```
typedef struct {
      char *name;                    name of function
      char **args;                   type names of arguments, null terminated
      char *retval;                  type name of return value
      int (*func)();                 actual function
      char *(*inst_retval)();        type name of return value
} Func;
```

If `retval` is `sig_inst`, i.e. an arbitrary instance, then the type checking routine will call `inst_retval()` to determine the actual type of the returned value.

func_find() - find the index of a function in the function table for the given name.

func_call() - calls a function. Arguments are: `func_ptr`, `return_inst`, `arg_inst[]`. `func_ptr` is a pointer to `Func`, `return_inst` and `arg_inst` are pointers to

`Inst`, and the `arg_inst[]` is null-terminated.

func_varcall() - calls a function. Arguments are: `name`, `return_inst`, `arg0`, `arg1`, ..., `0`. This is a `var_args` implementation of `func_call()`.

`constructor` - submodule with all constructor specific functions.

`overload` - submodule with all overloaded functions.

`primitive` - submodule with all primitive type functions.

`primitive/template.c and template.h`
- template for adding compiled functions.

`user` - user defined functions

E.3. Grammar

This module handles all the grammar, it primarily calls functions in the Node module to create a parse tree.

`low` - submodule where all the EOQL grammars are located.

`low/*.y` - the grammars, with query.y being the top file. They have a standard form, all keywords are in uppercase, all other tokens are in lowercase. This is used to dynamically generate the `token.h` and `types.h`. See the `Makefile` in this module.

`low/core.y` - if-then-else evaluation is special, so that under immediate mode, the parser does not wait for the next token before evaluation.

`low/lex.l` - the lexical analyzer in standard `lex` format. We changed the `input()` macro to handle reading EOQL script files.

`typegen.c` - a program that generate a C file from EOQL type definitions.

E.4. Instances

This module manages the session instances, such as creation, destruction, reference, and evaluation.

```
typedef struct {
      char *name;                 name of the instance
      TypeVal *type;              actual type of the instance
      NodeInst *ref;              pointer to the referencing Node
      int status;                 status information
```

```
    struct _instance *parent;        parent instance
    struct _instance *sib;           sibling chain from same parent
    struct _instance *child;         points to child instance chain
    int num_flds;                    number of values in the instance
    int *values;                     indices into the value buffers
} Inst;
```

The `parent/sib/child` links allows the QP to track dependencies when values are changed. The node reference and the opposite link from the Node allow the `scoped_name` evaluator to use previously created Instances.

`inst_find()`	- returns the index of instance in the instance table with matching name.
`inst_eval()`	- evaluate an instance based on parent information.
`inst_ref()`	- returns the instance at the end of a `scoped_name` reference.
`inst_store.c`	- routines for searching instances in an image data, i.e. sets or sequences.
`inst_value.c`	- manages the values of an instance.

E.5. Nodes

This modules provides the basic structure for the parse tree.

```
typedef struct {
    NodeType type;
} NodeProto;
```

Each node type has different structure, but they all start with `NodeType type`.

`template.c and template.h` are used for making new node types.

E.6. Storage

This module is the main interface to the underlying database engine. It is the most implementation specific portion of the EOQP. If the engine is replaced, then this module has to be recoded.

```
typedef struct {
    int (*setcolumn)();          determine column locations of components
    int (*create)();             create the collection store
    int (*destroy)();            destroy the collection store
    int (*updateparam)();        updates the relational overhead columns
    int (*exist)();              determines if an instance exist in the store
```

```
    int (*read)();                  read the component values of an instance
    int (*update)();                update the component values of an instance
    int (*insert)();                insert an instance into the store
    int (*delete)();                delete an instance from the store
} StoreRec;
```

Each constructor will have a `StoreRec`.

The interface to the Store Module is through the generic `store_*()` call.

`impl@`	- a symbolic link to the actual implementation. It should contain routines for all the constructors.
`include@`	- a symbolic link to the include files of the underlying engine.
`physical.c`	- base routines for interfacing with the underlying engine.
`stubs.c`	- stubs for all the calls to the underlying engine, for testing purposes.
`rel-blob`	- submodule for the current implementation.
`serial`	- submodule for building serial ID daemon. It uses RPC for communication.
`serial/serial_wrapper.c`	- hides the details of the `get_serial()` call from rest of the system.

E.7. Support

Miscellaneous support routines.

`args.c`	- routines for managing arrays of character pointers, e.g. `char *args[]`.

E.8. Types

This module manages the types used in the session, such as creation, destruction, and lookups.

```
typedef struct {
    char *typename;                 name
    TypeStruct const;               type of type definition
    int inst_count;                 number of instances
    int defined;                    whether the type is defined
} TypeProto;
```

There are many types: primitive, enumerated, core, alias, implemented, and constructor. Each has its own structure, but all start off with the `TypeProto` definition.

`type_find()` - returns the index of the matching type in the type table.

`type_resolv()` - resolves a type fully, i.e. make sure the type is well-defined.

`typecompat.c` - routines to determine whether two types are compatible.

`typecore.c` - holds all the definitions of primitive and core types.

`typeproc.c` - handles look ups for procedural types.

`typestore.c` - checks for validity of a type for external storage.

`typevalue.c` - routines that manage references to the value buffers for its instances.

`user` - a submodule where users can define their own types based on EOQL type definitions. The program `typegen` from Grammar is used to build C files.

E.9. Validation

This module performs validation of parse tree to make sure certain semantic constraints are not violated. For example, it checks for conflicting context dependencies and incorrect number of ID/Property/Ordered_By arcs in Object-Type definition.

`validate()` - takes a parse tree made of Node's and validates the command. This call is reentrant. Therefore, it can be called for validating subtrees. A large switch statement is used to call the specific validate functions.

E.10. Value Buffers

The value buffers serve to manage the data used by the instances.

```
typedef struct {
      ValEnum valtype;                type of values stored
      char *name;                     name of the value type
      int val_count;                  the upper bound of existing buffers
      int store_size;                 unit size of a value in bytes
      unsigned char in_use[];         reference count for the usage of a value
} ValPool;
```

There are several types of value buffers: binary, boolean vector, byte, integer, float, and string. Each has a different structure, but all of them start with the same definition.

`valpool_find()` - find the `valpool` with the matching type name.

`value_new(), value_free()`
- allocate new buffer and free buffer.

`value_get(), value_set()`
- gets and sets a value. Setting a value is by copying.

`value_use()` - increment a value's reference count.

UNIVERSITY OF CALIFORNIA
UC
San Francisco
LIBRARY