

This thesis is dedicated to the memory of my father.

"the soul never thinks without an image"

Aristotle

A Database Query Language for Operations on Graphical
Objects

Submitted in Partial Fulfilment of the Requirements of the
CNAA

Degree of Ph.D.

in

Computer Science

by

A.W. Wakelin

Dept. of Mathematics and Computer Studies
Dundee College of Technology
Bell St.
Dundee
DD1 1HG
Scotland

Abstract of the Dissertation

A Database Query Language for Operations
on Graphical Objects

A.W. Wakelin

Dundee College of Technology

1988

ABSTRACT

The motivation for this work arose from the recognised inability of relational databases to store and manipulate data that is outside normal commercial applications (e.g. graphical data).

The published work in this area is described with respect to the major problems of representation and manipulation of complex data. A general purpose data model, called GDB, that successfully tackles these major problems is developed from a formal specification in ML and is implemented using the PRECI/C database system. This model uses three basic graphical primitives (line segments, plane surfaces - facets, and volume elements - tetrons) to construct graphical objects and it is shown how user designed primitives can be included.

It is argued that graphical database query languages should be designed to be application specific and the user should be protected from the relational algebra which is the basis of the database operations. Such a base language (an extended version of DEAL) is presented which is capable of performing the necessary graphical manipulation by the use of recursive functions and views. The need for object hierarchies is established and the power of the DEAL language is shown to be necessary to handle such complex structures.

The importance of integrity constraints is discussed and some ideas for the provision of user defined constraints are put forward.

Acknowledgments

Thanks must go to my two supervisors Dr. W.B. Samson and Prof. S.M. Deen for the quality of their guidance and ideas. Also many thank to R. Sadeghi who wrote the PRECI/C DEAL implementation as part of a closely related research project. Useful discussions were had with Dr. D.A. Duce (Rutherford Appleton Laboratory) and Dr. P. Robinson (Cambridge University Computing Laboratory) and my thanks go to them for giving up their time.

I am indebted to my colleagues at Dundee College of Technology for their interest and support during the duration of this work.

The author would also like to thank SERC for funding this work (Grant No. GR/D/03574).

Table of Contents

Chapter 1: Introduction	1
1.1. Aims	2
1.2. Structure of Thesis	3
Chapter 2: Literature Review	5
2.1. Graphical Database Models	5
2.2. Issues in Formal Specification	23
Chapter 3: Overview of Proposed Graphical DataBase Model (GDB)	34
3.1. Introduction	34
3.2. Representation	39
3.3. Query Language	50
Chapter 4: Formal Specification of GDB	59
4.1. Introduction	59
4.2. Specification of a Relational Database	59
4.3. Specification of the GDB Graphical Operators	68
Chapter 5: Implementation	73
5.2. C Language Version	73
5.3. The VMS/RDB Version	74
5.4. The PRECI/C Version	75
Chapter 6: Example Graphical Databases	83
6.1. Database (2-D) with Object Hierarchy	83
6.2. Database (3-D) with Matrix Operations	90

Chapter 7: Conclusions	97
7.1. Assessment of GDB Model	97
7.2. Further Work	99
Chapter 8: Summary	113
Appendix A: Introduction to Computer Graphics	115
Appendix B: The Specification Language ML	120
Appendix C: Specification Theory	125
Appendix D: Correctness Proof of SML Function from its Axiom	131
Appendix E: The DEAL Query Language	136
Appendix F: Complete BNF for DEAL	148
Appendix G: Standard ML Specification of Matrix operations	154
Appendix H: Example DEAL Programs for Display	160
Appendix I: C Language Implementation	170
Appendix J: RDB Implementation	173
Appendix K: Hardware	177

Table of Figures

2.1.	Checklist for reviewed Texts	22
3.1.	Points relation	41
3.2.	Lines relation for a cube	41
3.3.	Facets relation for a cube	42
3.4.	Tetrons relation for a cube	42
3.5.	Compound object	47
3.6.	Higher order compound object	48
3.7.	Matrix relation	49
3.8.	Hierarchy relation	49
3.9.	Compound object	49
3.10.	Display relation	53
3.11.	Matrix as a relation	55
4.1.	Graphical predicates	71
4.2.	Functions	71
6.1.	Hierarchy relations	84
6.2.	Functions for display of hierarchy	84
6.3.	Hierarchy of the 'House'	87
6.4.	Screen dump of 'House'	89
6.5.	Relations for cube	90
6.6.	Functions and views for the cube	90
6.7.	Screen dump of 'Cube'	95
7.1.	GDB compared to 3-D models	97

7.2.	Constraint symbol table	102
7.3.	Object relation	104
E.1.	An Overview Syntax of DEAL	137
E.2.	Part Explosion Database	144
I.1.	ML specification for Union	170
I.2.	C implementation of Union	170

CHAPTER 1

Introduction

The graphical presentation of data on all visual media is widely recognised as one of the best ways of communicating complex ideas. The widespread interest in computer graphics reflects this fact and more and more software and supporting hardware becomes available every year.

Unfortunately, these graphical systems are not generally compatible for a variety of reasons involving both software and hardware. The advent of computer graphics standards (Core, GKS, PHIGS, IGES) has attempted to rectify this problem by establishing software standards that enable graphics to be portable between machines running software that conforms to the appropriate standard.

At the same time, databases have become increasingly important for all large data handling situations and this increasing use has focussed attention on their shortcomings. Among these is the lack of support for complex data structures such as those used in graphics.

It is the purpose of this thesis to show how the relational database model can be adapted to accommodate data which is more complex in nature than simple strings and

numbers.

1.1. Aims

The aim of this research is to produce a graphics database that provides a general purpose data model upon which specific applications could be built. The three main requirements are :-

- (1) A representation of graphics that would need no special data structures i.e. the relations holding the data could be treated in exactly the same way as non-graphical relations in the same database, as it is desirable to integrate all the data of an organisation into one database to fully utilise the benefits of reduced data redundancy and central control that accrue from the relational model. This would enable existing database software to be used with minimum modification. Therefore, we seek to produce a model for a graphical database that allows both graphics and normal information to be held in a single logical database.
- (2) A query language that is flexible enough to adequately express the queries required of a graphical database. These include the commands for producing on-screen graphical displays of the results of queries.
- (3) A formal specification of the database model that would enable it to be implemented unambiguously.

1.2. Structure of Thesis

The thesis continues in Chapter 2 with a review of relevant literature. This is split into two sections. The first deals with graphical database models that have been devised for use in a number of application areas. The drawbacks of each as a general purpose model is noted with reference to the model developed here. The second section deals with formal specification in the areas of databases and graphics, and surveys the different techniques used.

In Chapter 3 the issues to be tackled and their solutions are introduced informally in the context of current database theory and graphical systems. The subjects of representation, data input, query languages and graphical transformations are discussed in turn and it is explained how the problems associated with each is tackled.

A formal specification of the graphical database is presented in Chapter 4. A detailed explanation of each part of the specification is given (despite the commonly held view that specifications should be easy to read - only simple ones actually are!).

Three different approaches to the implementation of the graphical database are presented and discussed in Chapter 5. These are an implementation in the C language of the database specification in SML, an implementation in VMS/RDB (the proprietary relational database system produced by DEC)

using its PASCAL interface to encode the queries and special functions required, and a part implementation using the PRECI/C database system with the DEAL language implementation developed by Dr. R. Sadeghi.

Chapter 6 gives examples of applications that were used to evaluate the model as implemented in PRECI/C. Conclusions are drawn in Chapter 7 together with ideas for further work and a Summary forms Chapter 8.

Matters of a general nature are described in the Appendices where computer graphics, the Standard ML language, Specification theory and the three implementations are described, together with details of the hardware used in this project.

Literature Review

2.1. Graphical Database Models

The subject of graphical databases has been studied for many years and the following review of this work is mostly restricted to those papers utilising the relational model. Other models have been advocated, particularly the hierarchical approach because of the naturally hierarchical structure of many graphical representations.

The papers reviewed here are split into sections which reflect their intended application area. Computer Aided Design (CAD) is the most widely used graphics application and this has been the focus of much research although most of this has been directed to specialist database models that are not among the commonly used commercial models. Geographical Information Systems have also attracted a lot of attention as mapping is a major activity in many different organisations and is a very data hungry task that demands efficient data handling.

The remaining sections look at novel approaches to the graphics database problem and at pictorial databases.

The major issues in graphical databases are the choice of representation, the query language to be used and the use of integrity constraints. The impact of the research papers on these issues is discussed.

2.1.1. Application Directed Models

2.1.1.1. Computer Aided Design

Lorie, Plouffe and others [1,2,3,4,5] have recognised the need for more complex objects to be stored in relational databases for engineering purposes such as CAD/CAM and geographical systems [6]. They build up complex objects from tuples derived from different relations by means of a root tuple to which they are all referenced. The referencing is achieved by a system of pointers and 10-byte identifiers. The queries to such a database are in an extended SQL which allows easy access to complex objects by using predefined "cursors" which extract the data hierarchy from the database and place them in a program data space ready for manipulation by a host program.

They also introduce extensions to the query language called "implicit join" and "key index". The implicit join is designed to facilitate the retrieval of complex objects without the user having to specify complex join predicates. The key index allows the user to specify his own key values rather than rely on the system generated ones used to con-

struct the complex tuple structures.

This work concentrates on building a relational model that can operate as a hierarchy and they have chosen a simple 2-D points and lines representation for graphics. There is no mention of integrity constraints or their role in maintaining the "complex object" structure.

Other advocates of the object oriented approach are Batory and Kim [7]. They call their concept "molecular objects" as opposed to Lorie's "complex objects" and use the design of VLSI chips as their example. The idea is similar to that described above in that it defines a high level entity in terms of aggregations of lower level entities.

They introduce integrity checking as an important part of their model but ignore the graphical representation problem and its associated language. They do look at version control and its problems, however.

An interesting application of graphical databases is the molecular modelling systems as described by Todd, Morffew et al. [8,9].

Molecular modelling systems are finding widespread use in the pharmaceutical industry for the discovery of new pharmacologically active chemicals by modelling the shape of molecules accurately and matching them with models of other molecules or active biological sites. Using the Peterlee

Relational Test Vehicle (PRTV) as their database system and the Winchester Graphics system (WGS) to display the results of graphical queries, they have six base relations ("Atoms", "Residues", "Linktypes", "Radii", "Ideal" and "Diagram") which hold 3-D coordinate data and linkage information about atoms. In addition they have some pre-defined functions, two of which are effectively integrity constraints.

This is a fixed purpose graphical database that has no graphical operators as part of the language (the relational algebra). All the graphical transformations such as changing viewpoint are handled by the WGS using data that is transferred from the database.

A database model for graphics is proposed by Tikkanen et. al. [10]. who developed a specialised geometric data manager in C for a 3-D CAD application. They choose to represent solid objects by linked lists of faces, edges and vertices which are connected by pointers. The decision to implement in C rather than using a conventional DBMS is on the grounds of efficiency. Clearly there can be no argument here that a specialised software package for a specific purpose is likely to be faster in operation but you sacrifice generality and the opportunity for integration with the rest of the data in a normal database.

Another approach to describing 3-D objects is Constructive Solid Geometry [11]. This uses a small number of basic

units to build up a tree which defines a new object. An object may be a combination of other constructed objects (i.e. defined in terms of the basic units) thus forming a complex tree with just the basic units at its leaves.

The basic units can be combined in a number of ways by using the set operators (union, intersection and difference) and/or specifying translational and rotational motion to the primitives thus describing its absolute position in space relative to the other basic units.

This method is used by Lee and Fu [12] as the basis for a CAD/CAM DBMS. They utilise the relational model with the ideas of Smith and Smith [13]. Lee and Fu have extended SEQUEL to allow relations to be defined detailing generic relations (i.e. relations which are to be considered as the same type) and aggregate entities (i.e. the attributes of the relation).

```
e.g.  Var Primitive :
      generic
      GC = (Cube, Cylinder, Cone)
      of
      aggregate [object#]
      object# : identification number ;
      .
      .
      .
      GC : geometric category ;
      end
```

This defines the basic units for this CSG database as cubes, cylinders, and cones as relations of type "primitive".

They also feature the use of views (relations derived from base relations) and triggers (automatic updates to relations upon detection of a specified condition) and integrity assertions (predicates which must be satisfied before the triggering action is allowed to proceed).

This latter feature is essential to maintain the integrity of the database where there are many semantic associations as in Lee and Fu's model. The disadvantage of such a complicated system is the complexity of the constructs that are required for comparatively simple queries.

e.g. List all primitives used in the object "shaft".

```
Select *
From primitive
Where O# =
    Select O#
    From object X, object Y
    Where X.CC = "primitive"
    And X.own_O# = Y.O#
    And Y.O# =
        Select O#
        From mech_parts
        Where name = "shaft"
```

Their motivation for using such a complex scheme is that it obviates the need for null values which would be required to terminate the branches of a CSG tree. Also it gives the user information about the type of primitive used at each position in the tree and therefore eases query formulation (sic).

The model proposed in this thesis also uses the idea of building objects from a small set of primitives but does not use a CSG tree. The resultant queries are easier to formulate than those associated with Lee and Fu's model.

The SAM* database (Su [14]) is a semantic network approach to CAD/CAM databases that is represented as a set of nodes pointed to by a number of directed arcs. Each node is characterised by an association and defines a domain of values which are the data in the database. There are seven association types (membership, aggregation, interaction, generalisation, composition, cross-product and summarisation) and these are used to express the semantic connection between nodes in the database.

He argues that a relational (or indeed any other) model cannot provide the range of semantics or datatypes required for a fully integrated manufacturing database. Also included are rules as a domain in the membership type of node and thus provide for a wide range of constraints for integrity and security purposes.

Shenoy and Patnaik [15] have devised a graphical database (ARDBID) that uses base relations "point", "line", "curve", "surface" and "volume" to describe 3-D objects using wire frame graphics. Thus the "line" relation has attributes "start point" and "end point" which are defined as

references to the key attributes in the "point" relation. The "surface" relation has attributes that reference the key attributes in the "line" relation together with some display attributes (line type). Curved lines are handled by using the parametric representation of Bezier or B-splines which enables curves to be specified by a few control points. This imposes a computational overhead as the data for the line must be calculated at display time.

The graphical description of an object is stored in relations with unique names that have the extension that defines the type of relation (e.g. name.lin, name.cur for line and curve relations for the object called "name").

The representation chosen by Shenoy and Patnaik is not unlike the one presented in this thesis, but they do not handle geometric transformations within the DBMS. All graphical operations are performed on data extracted from the relations (files) and processed as arrays in the host language (PASCAL). Updates are performed by writing these arrays back to the relations (files). There is no mention of whether integrity issues have been addressed.

2.1.1.2. Geographical Information Systems

The systems reviewed here are all concerned with mapping either from satellite data or ground survey data.

An alternative to the usual query language interface is proposed by Chang and Fu [16] who introduce "Query-by-Pictorial-Example". This extends the ideas of Zloof [17] where queries are expressed by putting example elements into tabular representations of the relations in a database to compose complex predicates. This implementation uses a graphical database derived from satellite pictures of the Earth which have the major features such as towns, rivers and roads translated into coordinate data. The graphical relations are all concerned with points and lines, and each tuple contains the 2-D coordinates of the end points of a line which represents part of a feature from the original picture.

Their database does not, however, extend to any general purpose graphical application and no graphical transformations are allowed on the pictures created. A link is maintained to the original satellite pictures which means that these can also be retrieved as the result of a query.

Frank [18] proposes a query language called "Mapquery" based on SQL for a Geographic Information System without detailing any precise storage strategy. He argues that the result of a query should be a graphical display which shows the area of interest. For multiple answers the system should automatically split the viewing surface into separate windows and display an answer in each. He highlights a number

of similar issues concerning the graphical display of results of graphical queries which the system should handle automatically. This matter is discussed later with reference to the model proposed in this thesis.

Frank demonstrates that more work is required to establish a standard form of presentation for results of graphical queries taking into account a variety of factors that are usually labelled Man Machine Interface.

Another approach to representing maps is to use a standardised grid onto which the mapped features are overlaid. The individual grid sections are then assigned to features on the map. An implementation of such a system is PICDMS [19]. It is used for storing and manipulating satellite photographs in a general purpose database system. The database model used is a stacked image structure which is stored sequentially. Each record represents one grid location and each field is the value of that location for each image stored. The motivation for this design choice is that new images of the same region are added by the addition of a new field to the data record rather than by adding a new record. This gives the database a compact format that does not waste storage with blank records. although this does give problems with the physical storage of variable length records. A data dictionary keeps track of the current structure in a user transparent way. A new language

is defined by giving examples of its use in a sample database. This is a procedural language that resembles BASIC with the addition of data manipulation keywords such as "ADD", "DELETE", and "DISTANCE".

2.1.1.3. Pictorial Databases

A number of models for pictorial databases have been proposed (e.g. [20]) but these are concerned with indexing images (e.g. video frames) and do not allow manipulation of objects within the picture so they are not considered here.

Kunii et al. [21] use a relational database to store representations of images by decomposing them into the objects depicted therein. Thus relations store data on the image and the index to the picture together with a title. Objects within each picture are itemised in another relation together with a measure of their distance from the observer. Each object is further decomposed into parts which are ranked for distance as before. Finally a relation called "region" is used to describe the colour of the components of each picture with a definite boundary. The boundaries are stored in a relation which holds the 2-D coordinates for each.

This is a top-down approach to picture decomposition that requires a lot of processing (either human or machine). The system is a complex index to a set of images but as no

sample queries were given and no system for the reproduction of the image was intimated it is difficult to divine the authors' intended application area.

An integrated Database Management System for a pictorial database is developed by Tang [22] using a matrix representation for the pictures (bit maps) and an extended SEQUEL for the query language. This is essentially an indexing system that allows the retrieval of the pictures by a number of keys ("text", "sub-picture" or "video frame") and displays them on specified or default output devices.

Some decomposition of the picture is possible by selecting parts of them to be indexed (i.e. in a picture of a face the nose and eyes may be referenced in different relations). Input of the pictures is by some unspecified camera-like device and a method of matching two pictures is assumed.

2.1.2. Previous Work into General Purpose Models

A. van Dam [23] gives a very early example of the use of non-homogeneous tuples as a data structure for computer graphics. This was only two years after Codd's seminal paper [24] and he didn't advocate the use of a relational database by name. van Dam proposes a simple representation using tuples to describe points, lines and pictures and presents a few functions that act over them. The data

structure he uses is hierarchical in nature with pointers to rings of similar objects in a backing store page. Thus he is concerned with physical devices and access problems as much as with a good graphical representation.

Weller and Williams [25] have a general purpose graphical system as their main aim using a relational database to store graphical data on which they build a hierarchical structure.

This extends the earlier work by Williams [26] where he proposed a graphical database built from points and using ordered tuples to provide the linkage of those points by lines. Their new model allows duplicate keys and an attribute can have the name of another relation as value thus allowing a hierarchy of relations to be constructed. This is a higher order idea that leads to difficulties in processing queries which are akin to the non-normal form relational systems that are being studied at the present time.

The graphical data is stored as point coordinates which have graphical actions associated with each one. Thus by using ordered tuples a shape can be drawn by having a relation with coordinates and the operation "line" in each tuple. Clearly re-ordering the tuples will produce a different (and unexpected) shape.

Crampes et. al. [27] developed a data model for a

database that would hold all the information required for an organisation and might contain any or all of data, text, image and graphical types. The pictorial part of their model is split into two types ; drawings and fixed images. The first is the class of graphics represented by points, lines and arcs. The second are images (bit mapped screens). They do not explain how drawings are stored or reproduced in the database but fixed images are stored on magnetic or optical media for recall when requested by the database query language. The pictorial information is represented in the relations as pointers to the stored image.

This type of representation of graphics avoids the issues of graphical representation suitable for manipulation and query by anything less than a whole image.

The need for integration of applications and the sharing of data is discussed by Spooner [28] who proposes a general purpose interface for a interactive graphical database system. He follows the philosophy of Lorie [3] in constructing hierarchies of tuples to form complex objects. The graphical data is held as 2-D coordinates in a single "Points" relation and all the entities are associated with points tuples by key values. He indicates that such a mechanism for representing geometric information is not flexible enough for most purposes and a better scheme is required.

His novel feature is the use of a relation called "Semantics" which holds semantic values for each attribute in each relation. The semantic values are used by the system for displaying the entities they refer to. So when an entity is to be shown graphically the "Semantics" relation is consulted to find the "meaning" of each attribute and the graphical data is processed accordingly. He also advocates the use of high level database model such as DAPLEX [29] for graphical interfaces to other data models.

A novel approach to database structuring is proposed by Hardwick and Sinha [30]. They use a non-normal form relational model to construct objects from a set of primitives (e.g. polygon, line). The resulting relation has one attribute for each primitive but each tuple has only one attribute which contains any value; the others contain nulls. These they call heterogeneous relations to distinguish them from first normal form (homogeneous) relations.

This structure is very much the heterogeneous list as used in LISP and its use in a relational environment is inappropriate. It is not clear how the relational model is used at all as the normal advantages conferred by that model cannot apply here.

A different approach to modelling graphical objects is used by Shapiro et. al. [31] for use in scene analysis. They propose a system for describing 3-D objects

approximately by using a few general purpose objects they call "blobs". These are parameterised objects that can be used to describe an object of interest by defining a set of relationships between them. Thus a table might be represented as a circular "plate" supported by four "sticks" and the database will hold the values for the sizes and angles between these "blobs". This data is held as 10 relations which are linked to form a hierarchy of sub-objects that construct the object of interest.

The objects held in the database are grouped by the values held in the root relation and new objects to be identified from a 2-D scene are matched with a group of known objects to restrict the search space before proceeding on to more detailed matching.

This is another example of a specific application that is not applicable to the general purpose graphical database although the "blob" concept is quite appealing.

Garrett and Foley [32] have looked to a graphical database system to increase programmer productivity by using a database to hold data from an interactive graphics system. They hold the data in relations that are analogous to the procedures in the CORE graphics standard. Thus calls to the graphics system are made by taking data from the appropriate relation and passing that to the CORE procedure. In addition they rely on defining dependencies between the relations

which control the update of the database via a mechanism of production rules they call "Continuously Evaluated Qualified Updates" (CEQUs). These cause data to change in the database as changes are made graphically by the user.

This use of a non-procedural mechanism to specify the operations on the database is close to a formal specification and gives a good approach to the difficult problem of graphical input to a database. This system is not restricted to the relational model or a production rule system and is therefore a general purpose concept.

The PROBE project [33] aims to increase the range of complex objects that can be handled by databases and research is continuing on a variety of topics, one of which is graphics. They have adopted the grid approach to representing graphical objects. (This is where a regular grid is superimposed over the object to be represented and the locations in the grid that coincide with the object are identified in the database.)

They utilise Approximate Geometry (AG) to process spatial queries. This is not an imprecise procedure, as the approximation relates to the coarseness of the grid used to define the object i.e. a large grid (few locations) is less precise than a fine grid (many locations). Each grid location is encoded by a "z-value" to facilitate the processing of a particular class of query ; the range query i.e.

queries where the information required is retrieved by an attribute value which lies within a range of values specified by an upper and lower bound.

The advantage of this approach is that all range queries can be reduced to a one dimensional problem i.e. is the attribute value between the bounds. This works for any dimension of object due to the nature of the coding process where the coordinates of the grid positions are converted to binary and interleaved to give a binary number that is unique. The sequence of these numbers traces out a regular path through the grid which means that, in the case of three dimensions, a volume can be described by two "z-values" and a three-dimensional "contains" query is reduced to a one-dimensional range query.

They define a new operation - the "spatial join" - which utilises this feature and present evidence that this improves the processing of such spatial queries.

The following table (Figure 2.1) shows how the various models compare with each other.

	Representation			Graphics	Integrity	2/3 D
	Pts	Grid	Obj	Language		
Lee (12)			x		x	3
Spooner (28)	x					2
Shenoy (15)	x				x	2
Chang (16)	x			x		2
Garrett (32)			x		x	2
Lorie (3)	x			x	x	2
Shapiro (31)			x			3
Morffew (8)	x				x	3
Williams (26)	x			x		2
Orenstein (33)	x	x		x		2
Hardwick (30)	x		x	x		2
Tang (22)			x			2
Kunii (21)			x			2
Chock (19)		x		x		2
Frank (18)	x			x		2
Su (14)			x			3
Tikkanen (10)	x			x		3
Batory (7)					x	2
van Dam (23)	x			x		2
Crampes (27)			x			2

Figure 2.1: Checklist for reviewed Texts

The "x" denotes features discussed by the authors in their papers.

2.2. Issues in Formal Specification

The use of formal specification methods as a precursor to the implementation of complex computer systems has been shown to be of benefit in the speed of implementation and in promoting easy maintenance of the resulting system. The specification also serves as a medium of communication between designer, programmer and customer.

In this thesis the formal specification of the proposed model is a major objective to promote understanding the ideas developed in the model. The sections that follow review the extant work in the area of database specification and graphical system specification in order to provide a basis for the presentation of the formal specification in Chapter 4.

2.2.1. Formalisation of Database Models

The use of formal specification methods to describe any system is the first stage in a generally larger task. This means that the choice of formalism and the structure of the specification must be geared towards the overall task. The papers reviewed in this section demonstrate different styles and formalisms without showing the purpose for which the specifications are written. Indeed, in many cases the purpose is an academic one that demonstrates the use of a particular formalism.

The objectives of formal specification are discussed more fully in Appendix C, but briefly a specification should give a precise description of an abstract syntax and semantics of the system being specified with the aim of allowing implementors to achieve a working system more easily and giving users a clear idea of the functionality of the system without pre-judging implementation issues [34]. A specific type of specification method is advocated by Lockemann et

al. [35] who define a database specifically (i.e. as a cartesian product of named domains) to demonstrate the use of the concept of data abstraction in databases.

In Tompa [36] there is an algebraic definition of quotient relations and the operations permissible upon them. This is based on sets and relations which are defined as abstract data types together with a set of operations on them. These operations include the usual set operations and the relational algebra operations. He also gives a commentary on how he arrived at his specification and the design decisions taken.

Quotient relations are relations that are permanently arranged according to a "Group-by" operation on a set of attributes. Thus a relation can be defined as $R(a,t,a')$ where a is the set of attributes, t is the set of tuples and a' is the set of "group-by" attributes.

For example, his definition of relational union is :

```

type    relation

syntax
  parts   : relation      -> set[attribute]
  attribs : relation      -> set[attribute]
  union   : relation x relation -> relation

constraints
  union(r,r') >> attribs(r)=attribs(r')
                AND parts(r)=parts(r')

equivalences
  union(R(a,t,a'),R(a,t',a')) = R(a,tUt',a')
  attribs(R(a,t,a')) = a
  parts(R(a,t,a')) = a'
end

```

where the functions "parts" returns the "group-by" attribute set and "attribs" returns the full attribute set.

The datatype being used is given in the type statement and the functions acting on that type are listed in the "syntax" section which details just the domain and range types. The "constraints" section is in effect the preconditions that must hold before the function "union" can be applied. The constraints are expressed as logical combinations of functions and constructor patterns. Finally the semantics of the operations are given in the "equivalences" section.

Another algebraic approach to defining data model semantics is described by Brodie [37] who uses a set of properties for each type to describe the semantics of each operator for the relational model. This means that the properties are given by the tuple :

the technique [40,41].

Another formalism used to specify a database is the attribute method [42] as described by Niemi. He defines a relational tuple to be a 5-tuple $\langle r, X, AN, Ix, Fx \rangle$ where r is the data in a tuple, X is the type of that data, AN is the set of attribute names, Ix is a set of indices and Fx is a function which maps attribute names to indices. For example,

```
( <1112,Smith,5819.20>,  
  int x char x real,  
  {Ecode,Ename,Salary},  
  {<1>,<2>,<3>},  
  {f(Ecode)=<1>,f(Ename)=<2>,f(Salary)=<3>}  
)
```

is a tuple in an employee database.

A relation is defined as a six-tuple $\langle r, X, RN, AN, Ix, Fx \rangle$ with the same meanings as before and RN as the relation name. Here the author defines an abstract syntax for a relational database and then defines the semantic and checking attributes for that model. The abstract syntax is a series of structural productions that includes the relational algebra together with the constructors for a relational database. The attributes express the precise semantics of the productions (semantic attributes) together with restrictions (checking attributes) on the possible set of objects that can be generated by them.

Stemple and Sheard [43] develop a specification that

incorporates integrity constraints as part of the main model to ensure that the results of all operations yield valid database states. The main aim of this specification exercise is to prove theorems about the behaviour of transactions on constrained databases. The relations are defined explicitly for specific cases (as are some of the transactions). The two datatypes used are finite sets (fset) and tuples. The axioms for the set functions are given in if-then-else format as shown below.

```
member : element X fset -> boolean
member(e,s) = if s = emptyset then false
              else if e = choose(s)
                  then true
              else member(e,rest(s))
```

This is a recursive definition of set membership that uses the function "choose" which selects a member of the set determined by an internal ordering.

The tuple axioms they define to construct and select attributes are written in a shorthand style which stands for a whole family of axioms for selecting and constructing tuples.

This style of specification is continued in a further paper [44] where they use the language ADABTPL to express their database specification. They illustrate how novel datatypes can be incorporated into a database system and used in database transactions.

This specification is executable and can be analysed using a Boyer-Moore style theorem prover to provide semantic checking.

2.2.2. Formalisation of Graphical Systems

The major work in this area is that of W.R. Mallgren [45] whose objective is to provide a means of reasoning formally about graphics programming languages and he gives examples showing proofs for the equivalence of two programs using the specification as rewrite rules.

He starts by identifying three main problem areas with existing graphics programming languages :

- (1) Specialised constructs are used that are not found in general purpose languages.
- (2) These constructs are not fully understood and are therefore difficult to specify properly.
- (3) Graphics programs often involve direct interaction with the user which is difficult to specify.

He contends that a graphical language should mirror traditional languages by being based on a small set of well understood concepts (e.g. datatype, assignment statements and flow control statements). As there are no universally fundamental graphical concepts Mallgren chooses an arbitrary

set on which to base his specification ("region", "picture", "graphical transformation", "hierarchical picture structure" and "user interaction") and provides an algebraic specification based on these. A "region" is simply a set of points which represents a plane in 2-D space or a volume in 3-D space. A "picture" is defined as a partial function mapping a set of points to values that can be considered as colours. (In the simplest case 0 for black and 1 for white). A "graphical transformation" is a function that maps a picture into another picture. A "hierarchical picture" is one that is composed of sub-pictures each of which may itself be composed of sub-pictures. Finally "user interaction" is the process of the user communicating in some way with the graphics program. Each of these concepts has an associated set of operations.

He defines abstract datatypes called "point", "region", "picture" and "transformation" and operations upon them which constitute the formal specification he aims to derive.

He also tackles the difficult problem of user interaction which requires a variation on the normal specification method to express the semantics adequately (He uses the notions of shared datatypes, semaphores and concurrent processes. These concepts are not relevant to this thesis).

The other authors who have tackled formal specification of graphics systems [46,47,48] have taken different

approaches. Duce and Fielding have written specifications for parts of the already implemented GKS system. Their aim was to provide an understandable description of features of the standard GKS system that were difficult to describe in plain English.

Ayra does not provide a formal specification of a graphics system as such but by using HOPE as his graphics language he shows how manipulation can be achieved in a functional environment.

Carson provides an axiomatic specification of the old ANSI draft standard PMIG (Programmers Minimal Interface to Graphics) which has now been superceded by the GKS standard. The import of the paper is to show how formal specification can be used as an aid to providing an unambiguous description of a system.

2.3. Conclusions

This review of the extant work in the areas covered by this thesis has shown that there is no clear cut candidate for a generally useful graphical database although Lorie's ideas seem to approach one. Also, there is a variety of specification styles which try to encompass the semantics of relational databases and graphics systems but do not combine the two.

In the next chapter a model is proposed that will attempt to overcome some of the problems identified in the work reviewed above.

Overview of Proposed Graphical DataBase Model (GDB)

3.1. Introduction

There are three main issues to be addressed in the field of graphical databases; representation, query languages, and graphical transformations.

Firstly, a suitable representation of graphics must be devised. This is important because it will determine the range and style of queries on the database. For dedicated systems the representation chosen can reflect the user's expected queries and give the bonus of efficient operation. This limits the opportunities to apply the model to other application areas where queries may become more difficult to construct and some evaluation overheads may be incurred.

A general purpose representation should be applicable to many tasks with a suitably flexible query language to handle any foreseeable query. The chosen representation should also allow the integration of graphics with the more usual data held in databases and therefore allow complex queries to yield graphics and text as a result.

Secondly, the query language devised to accompany a graphical (or indeed any) database must be expressive enough

to extract any subset of the stored data without having a complex syntax or having overlapping constructs (i.e. two distinct syntactic constructs for the same semantic meaning). For a graphical database this means that the ability to use graphical concepts for selection predicates is required (e.g. retrieve all objects within a specified volume of space). New constructs are needed to handle the graphical display of the results of a query.

Finally, the subject of graphical transformations must be considered. The DBMS must support graphical transformations to allow the user to manipulate objects represented in the database as well as providing the means to display objects by the choice of a suitable viewing projection method.

The types of graphical objects to be stored in the database is an important issue that must be clarified before a suitable database model can be devised. There is no clear distinction between the everyday use of the terms graphics and images. One might use the definitions that graphics are line drawings, sketches, commercial artwork while images are photographs or electronic (video) frames. Unfortunately commercial artwork cannot be wholly classified into either category because it may mix a variety of styles and media. These two varieties of non-textual representation (to avoid the use of possibly ambiguous terms) must, therefore, be

considered in different ways as far as graphical databases are concerned.

The philosophy adopted in this work is to make the distinction between images (for want of a better generic term) that are to be decomposed in some way and those that are to be viewed as a whole. Thus for the first type there must be some method of decomposing the image into some useful primitives (or, conversely, some way of constructing an image). The CAD environment is typical of this type of application.

For the second type, consider the example of the integration of photographs in a database (for security applications perhaps). This requires the database to act simply as an index into the store of photographs. i.e. a query of the form "Give me the photo of John Smith" will display the required photo by finding the value of an attribute in the "John Smith" tuple which is passed to the photo store held on a separate medium (perhaps video disc, CDRom). In this application the user is not concerned with how the photo is constructed because he does not want to decompose it in any way. Some matching with a frame from a video camera would be performed by digital methods rather than by decomposition into constructional primitives.

Applications such as CAD or geographic information systems (GIS) demand a constructional approach because they require the facility to examine small parts of a whole

object and possibly create a new object from them. Again the overlap of the categories occurs with image processing where photographs or video frames need to be stored in such a way that allows them to be decomposed into salient features. This may mean that two representations of the image are required ; one of each format.

The approach taken in this thesis is that the constructional approach is the lowest level representation possible and if a suitable candidate can be found then applications can be built on top using only those aspects of the model they require. Therefore, simple indexing databases or pixel based images for image processing applications are not considered as these can be handled adequately by other software and stored in a constructional model if required by the application of some suitable conversion algorithm using edge detection and similar methods.

Some of the problems associated with current graphical databases are highlighted by Abel [49] in which he describes the CORGIS database system used for mapping applications. He cites four deficiencies of spatial databases (my comments are given in parentheses) :

- (1) There is no high level view of entities. (The user must be able to deal with entities and not with the primitives used to construct those entities. There is a

need, therefore, for grouping primitives into user definable objects and manipulating them as such).

- (2) The query language lacks spatial operators. (The query language must support some operators which allow the user to move entities relative to one another etc. This includes such transformations as rotation, translation and scaling).
- (3) There is no graphical display of the results of queries. (There must be some way of seeing graphically the results of queries ; a list of coordinates on its own is virtually meaningless).
- (4) Physical access methods are inefficient for spatial queries. (If graphical data can be incorporated in an existing database system then the DBMS can handle the disc accesses. Some queries involving spatial relationships might be handled more efficiently if the data were organised on disc in some suitable fashion but the mapping from the logical data in the database to the physical data on disc would be difficult. See Chapter 5).

The one feature that must be present in all graphics systems is a conversion of the graphical representation to a set of 2-D coordinates/instructions that can be presented to

an output device such that it can display the required image. This means that (for 3-D graphics) the chosen representation must be converted (if necessary) to some 3-D coordinate system and then transformed to a set of 2-D instructions by selecting a projection style. The consequence of this is that any use of more abstract models than points etc. will require extra processing to obtain a set of 3-D coordinate data before the viewing projection may be applied. Clearly the balance between the usability of an abstract representation and the processing time must be made.

3.2. Representation

The crucial issue in this work is the choice of a representation for graphical data that will be compatible with the data structure provided by the relational model. As noted in Chapter 2 a number of representations have been adopted by workers in this field.

The use of such schemes as CSG which use solids as the primitives for constructing objects does not allow the user the freedom to design difficult objects in a natural way. It is easy to decompose an object into primitive solids (such as cubes and cylinders) but it is much more difficult to construct a new object from such primitives. A more natural way to do such a job is to first create an outline and then fill in more detail later. It is also possible to create

"impossible" objects (such as the drawings of Escher) using a points based model so this type of system will be applicable over a wide range of applications.

Another model noted in Chapter 2 uses the database to store a series of calls to a graphical system. In this way an object can be displayed by passing the selected data to the graphical system. This has the merit of being fast but there is no possibility of being able to manipulate objects as the data stored is only the display of the object not a construction of that object.

The fixed format of the GKS metafile offers the facility of storage in relations but it involves varying length records of varying scheme that would demand the use of multiple relations (one for each type of metafile record).

The representation devised for this thesis is based upon primitive constructors which are all defined in terms of points in space. These points are defined in turn on the cartesian coordinate system.

The reasons for this choice are :

- (1) The manipulation of cartesian coordinate systems is widely understood and used.
- (2) The point is the smallest graphical unit. More complex

primitives can easily be constructed from points.

- (3) It allows existing systems to be interfaced to this design with ease.

The basic GDB model consists of three primitives that are all based on points (Figure 3.1). These are line segments, triangular surfaces ("facets") and tetrahedral volumes ("tetrons").

Line segments can be defined by specifying the two endpoints (See Figure 3.2). This allows the construction of "wire frame" graphics.

Points

ptnum	x	y	z
p001	0.0	0.0	0.0
p002	10.0	0.0	0.0
p003	10.0	10.0	0.0
p004	0.0	10.0	0.0
p005	0.0	0.0	10.0
p006	10.0	0.0	10.0
p007	10.0	10.0	10.0
p008	0.0	10.0	10.0

Figure 3.1: Points relation

For more complex images the primitive called a "facet" (Figure 3.3) can be used. This is a triangular surface that is defined on three points which specify the positions of

Lines

lnum	pt1	pt2
1101	p001	p002
1102	p002	p003
1103	p003	p004
1104	p005	p006
1105	p006	p007
1106	p007	p008
1107	p004	p001
1108	p008	p005
1109	p001	p005
1110	p002	p006
1111	p003	p007
1112	p004	p008

Figure 3.2: Lines relation for a cube

the vertices in 3-D space.

The extension to provide a volume primitive is quite natural and a tetrahedral volume of space called a "tetron" (Figure 3.4) is used.

One of the advantages of defining facets and tetrons in this way is that they are not restricted to being fixed size or shape which reduces the number of primitives required when compared to fixed size representations [50] which are wasteful of storage where large areas or volumes are being represented. The tetron model allows any precision to be used for any shape by using more smaller primitives at places where high precision is required.

Facets

-----	fnum	pt1	pt2	pt3	
	f001	p001	p002	p003	
	f002	p003	p004	p001	
	f003	p005	p006	p007	
	f004	p007	p008	p005	
	f005	p001	p002	p006	
	f006	p006	p005	p001	
	f007	p002	p006	p007	
	f008	p007	p003	p002	
	f009	p003	p004	p008	
	f010	p003	p007	p008	
	f011	p001	p005	p008	
	f012	p001	p008	p004	

Figure 3.3: Facets relation for a cube

Tetron

-----	tnum	pt1	pt2	pt3	pt4	
	t001	p001	p002	p004	p005	
	t002	p002	p008	p004	p005	
	t003	p002	p003	p004	p008	
	t004	p002	p003	p006	p005	
	t005	p006	p003	p007	p005	
	t006	p005	p008	p003	p007	

Figure 3.4: Tetrons relation for a cube

The primitives outlined so far are sufficient to construct a wide range of plane faced objects and by using a large numbers of small line segments or facets, curved lines and surfaces can be approximated. To represent more regular shapes, however, it is convenient to use algebraic formulae

as shorthand. For example, the circumference of a circle might be represented as a set of line segments but defining a large number of them is cumbersome and tedious. The more natural way is to define a circle as the set of points generated by the formula $y^2=r^2-x^2$ (or similar) where x can range from $-r$ to $+r$ to generate the required set of 2-D coordinates for a circle of radius r .

One can think of any number of such primitives e.g. spheres, parametric cubics (Bezier, B-spline), cylinders etc.

The representation chosen must also provide the user with a consistent method of manipulation. This means that any new primitives used should be able to be handled in the same way as the predefined ones. For this representation based on points all other primitives must be defined on points and not in terms of scalar quantities. For instance, lines could be defined in terms of one end point, a length and an angle of rotation about a pre-defined axis. Transformations applied to point based primitives can be handled by matrix methods which deal with 3-D coordinates. This method is not possible on scalar valued objects where the operations to rotate, translate and scale must be performed separately by different mechanisms. The points based primitives can combine the three transforms into a single matrix and apply them all in one operation.

The following shows the different models; the points based and the scalar based.

Points based Model

```
Point : real * real * real      - x,y,z coordinates
Line  : point * point          - end points
Facet : point * point * point  - vertices
Tetron: point*point*point*point- vertices
Circle: point * point          - centre + curcumf. pt
Sphere: point * point          - centre + curcumf. pt
```

Alternative Model

```
Point : real * real * real      - x,y,z, coordinates
Line  : point * real * real     - point, length, rot'n
Facet : line * line * line     - three lines
Tetron: facet*facet*facet*facet - four facets
Circle: point * real           - centre and radius
Sphere: point * real           - centre and radius
```

There would be the added complexity of integrity checking for the alternative model to ensure that the component primitives created the desired primitive. e.g. constructing the facet in the alternative model the three lines would have to join at the ends - in the points based model the three points always produce a triangle.

One benefit of a points based model is the ease with which properties such as area and volume can be calculated because the objects are already tessellated.

The problems of how to display the results of queries are many. The implementor would have to choose between show-

ing graphically all the objects retrieved by a query in a number of windows on the screen or giving the user a list of objects from which he can opt for graphical display of one or more. In many cases the queries will only return a single object in which case the options are more easily reconciled.

3.2.1. Object Hierarchies

The model detailed so far is sufficient for defining objects that exist in isolation. An enhancement would be the ability to define objects in terms of other objects. This is important from the point of view of re-usability and ease of use. This embodies the principle of top-down design where the whole problem is split into smaller parts and these parts are similarly decomposed until the parts are small enough to solve individually. So if the user wishes to define a complex object (e.g. a car) in the Graphical Database it is much easier to individually define each component part and then define the car as a collection of these parts.

This is the well known parts tree data structure which must be extended in the case of GDB to include some graphical information along with the expected attributes of "superior part", "inferior part" and "quantity". The graphical information necessary describes the spatial relationship of the inferior to its superior (i.e. their relative positions in space). There must also be a component that converts the world coordinate system of the inferior into the coordinate

system of the superior. (Clearly as each individual part is defined in the database the scale used will be user definable and it is pointless to demand that the user maintains a constant coordinate system over all the parts. This negates the objective of re-usability as each part would have to be redefined for each major object it was a component of).

There are several ways that this graphical information may be expressed in relational terms.

- (1) Option one is to use 9 attributes to hold values for the rotational, translational and scaling degrees of freedom (Figure 3.5). This could be converted into the necessary matrix which would be used to transform the

points associated with that inferior part.
Compound Object

```

-----
| Snum | Inum |Rx|Ry|Rz|Tx|Ty|Tz|Sx|Sy|Sz|
-----
| o1   | o8   | ...
| o1   | o9   | ...
| o1   | o9   | ...
| o8   | o10  | ...
| o8   | o11  | ...
| o9   | o12  | ...
| o9   | o14  | ...

```

Figure 3.5: Compound object

This is only feasible for a single level hierarchy. If the inferior part is itself composed of other parts, then these parts will each have a transform matrix which must be multiplied together to give a total transform matrix. These operations are possible but at

some intermediate stage the matrix must be stored in a relation as the 9 attributes and this mapping involves factoring matrices and deriving eigenvalues and eigenvectors to generate the scaling and rotational values respectively.

- (2) Option two is to store the relation name of the matrix relation in the compound object relation (Figure 3.6) which would allow matrices to be stored and multiplied as required, but this model is then not in normal form (because the values are not foreign keys but relation names. This means that the normal join operation is not applicable) and, as such, is outwith the scope of current database technology (although research into higher order or non-normal form relations is underway at a number of institutions at the moment).

Compound Object

```

-----
| Snum | Inum | MatrixRname |
-----
| o1   | o9   | matrix2     |
| o1   | o8   | matrix1     |
...

```

Figure 3.6: Higher order compound object

(Note: values of MatrixRname are names of relations, not normal attribute values. This is a higher order structure which is foreign to the relational model.)

- (3) Option three is to store all the matrices in one large relation (Figure 3.7) which has a composite key of unique matrix number and tuple number (The latter is necessary to ensure that the matrix is constructed in the correct order).

Matrices

MatrixID	row	one	two	three	four
m1	1	1.0	0.0	0.0	0.0
m1	2	0.0	1.0	0.0	0.0
m1	3	0.0	0.0	1.0	0.0
m1	4	0.0	0.0	0.0	1.0
m2	4	2.0	0.0	0.0	1.0
m2	3	2.0	1.0	0.0	1.0
...					

Figure 3.7: Matrix relation

The hierarchy that results from this approach is shown in Figure 3.8.

Object hierarchy

Snum	Inum	MatrixID
o1	o9	m2
o1	o8	m1
...		

Figure 3.8: Hierarchy relation

- (4) Option four is to define a new datatype "matrix" within the DBMS to allow matrices to be stored as attribute values (Figure 3.9).

```

Compound Object
-----
|Snum|Inum|                Matrix                |
-----
| o1 | o9 |(1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1)|
| o1 | o8 |(2,0,0,0,0,2,0,0,0,0,2,0,0,0,0,2)|
...

```

Figure 3.9: Compound object

This has the advantage of making matrix manipulation much faster as it can be coded in the implementation language (C in the case of DEAL/PRECI-C). However, this means that an interface between matrices and relations must be established to support the transform operation that takes a matrix and the points relation to produce the transformed points relation.

The mechanism of graphical transforms is considered later in this chapter.

3.3. Query Language

The query language devised for a graphical database must have all the normal relational operations plus the specialised graphical ones. The primitives used to describe objects graphically would not be available for normal query purposes. All manipulation of the points, lines, facets and tetrons relations is left to the built-in operators "gjoin" and "show". The former produces a new relation that is a subset of the main points relation which contains only those points that describe a selected object. The latter produces

a relation containing all the transformed point data necessary for display purposes. This in turn is hidden by the user operator defined to select an object and display it on a specified portion of the screen using a specified projection style and orientation.

The user query language will be designed for the application and will map from the users conceptual model into calls to "show" and "gjoin". (The gjoin function is in fact a family of functions that are defined for each type of primitive defined in the database).

The relational operations embodied by the operator "gjoin" are expressed in the following SQL-like statements to transform the points for the object "Bowl". (T1-T9 are temporary relations.)

```

T1 = SELECT [onum]
      FROM object
      WHERE name = Bowl

T2 = SELECT [lnum]
      FROM T1,linelink
      WHERE T1.onum = linelink.onum

T3 = SELECT [spt,ept]
      FROM T2,lines
      WHERE T2.lnum = lines.lnum

T4 = SELECT [spt]
      FROM T3

T5 = SELECT [ept]
      FROM T3

T6 = T4 ++ T5

SubPoints = SELECT *
            FROM points,T6
            WHERE T6.spt = points.ptnum

```

Thus a call of

```

SubPoints = GJOIN
            WHERE object.name = Bowl

```

would achieve the same end. Note that gjoin for an object defined in terms of facets requires one additional project and one additional union operation. Similarly for an object defined in terms of tetrons, two additional project and two additional union operations are required.

The resulting points relation can be transformed to produce a relation suitable for display purposes. This involves the viewing transform as well as the transformation

needed to change from world coordinates to device coordinates (see Appendix A). The viewing transform is specified in ML in Appendix G, as is the "transform" function which is used to apply that transform to a points relation.

The transformed points relation is then joined twice to the lines relation (three times for a facet and four times for a tetron) to produce a relation which contains the three coordinates for each end of the line (Figure 3.10). This relation is then processed by the device driver to draw the object defined in the query.

```

-----
|lnum| xs  | ys  |zs  | xe  | ye  | ze  |
-----
|L101|120.3|130.4|0.0|400.5|130.4|0.0 |
|L102|400.5|130.4|0.0|800.9|200.6|10.0|
...

```

Figure 3.10: Display relation

The equivalent operations for the "show" operator in SQL syntax are :

```
T7 = TRANSFORM SubPoints
    WITH Viewmatrix
```

```
T8 = SELECT *
      FROM T7,T3
      WHERE T7.ptnum = T3.spt
```

```
T9 = SELECT *
      FROM T8,T3
      WHERE T8.ptnum = T3.ept
```

DISPLAY T9

where DISPLAY is the device driver for the output device

selected for display purposes (usually the screen). This is the equivalent of

```
SHOW "Bowl"  
USING VIEWMATRIX
```

The operations for compound objects (i.e. object hierarchies) are similar except that the hierarchical structure must first be "flattened". The "gjoin" must contain all the points for all the sub-objects of the selected object. Given a compound object relation "parts" with scheme (sup,inf) where "sup" is the superior object and "inf" is the inferior object, the SQL-like syntax for the flattening operation is :

```
temp1 = SELECT [inf]  
        FROM part  
        WHERE sup = onum  
  
temp2 = SELECT [inf]  
        FROM temp1,part  
        WHERE temp1.inf = part.sup  
  
temp3 = SELECT [inf]  
        FROM temp2,part  
        WHERE temp2.inf = part.sup  
  
etc.  
  
Result = temp1 ++ temp2 ++ temp3 ++ ...
```

This sequence of operations must be continued until the resulting relation is empty (i.e. the "leaves" of the tree have been reached. Clearly SQL is limited in that it can not

adequately express such operations in a simple syntax. (A DEAL function for this operation is given in Appendix H). The relation "Result" can be substituted in the query for T2 above, in place of T1.

3.3.1. Graphical Transforms

Graphical transformations of points based models by matrix methods is described in Appendix A. What follows is a discussion of the possible options available to handle matrices and relations. The essence of graphical transforms is the multiplication of vectors (representing 3-D coordinate points) by a matrix which embodies the desired transformations. The 4x4 matrix is not a built-in datatype so some provision must be made for its inclusion within the database. One possibility is to store the matrices as relations with five attributes which represent the four columns of the matrix together with an extra attribute which defines the row of the matrix. Such a relation would look like Figure 3.11.

Matrix1

Line	one	two	three	four
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Figure 3.11: Matrix as a relation

It is possible to define matrix operations in terms of relational operations given a suitably flexible query language (e.g. DEAL). This is likely to be quite slow as many temporary relations are created for operations such as matrix multiplication. The other possibility is to define matrices as data types within the DBMS and include the appropriate operations implemented in a suitable language (e.g. C) which are accessible from the query language.

What is also required if the latter method is adopted is a mechanism for accessing the points relations such that the coordinate data can be operated upon by the matrices. The facility to define the appropriate matrices is also required which means that the query language must have some of the attributes of a conventional programming language and has operations that are not relational in nature.

From the purist's point of view the former option is preferable even if it is slower than the latter, since it does not require extensions to the language.

A related problem involves the display of objects that are the results of queries. A decision must be made as to where the graphics device driver resides. The two obvious choices are in the query language or within the DBMS. The former will allow flexibility to use different display devices (e.g. screen, printer, plotter) as the user will be able to write the appropriate routines in the query

language. The latter is more elegant in that the low level operations are hidden from the user and the query language is not cluttered with non-relational constructs.

The ability to manipulate objects by routines embedded within the DBMS is essential to the efficient operation of a graphical database. The other option is to interface the DBMS with a specialised graphical system that would handle all the graphical transforms. This would involve extracting data from the DBMS and converting it into the data structure appropriate for the host language. If graphical data input is required then the reverse operation is also required. The data from the graphics system must be converted into the relational structure.

If the transformation and graphical display facilities reside within the DBMS and are controlled by an extended query language, then an interface must be included to define output devices and suitable software to utilise them. If these functions are left to a host language program then the burden of driving devices is placed with the applications programmer but the result of a query must be a set of data in a standard form which can be processed by the external program.

Choosing this latter option would allow the use of a standard graphics system (such as GKS) to produce output. This has obvious benefits in allowing the graphical database

to be widely used and married easily to existing graphics systems.

Including the graphical transformations and graphics calls within the DBMS does facilitate the production of graphics for the specified device giving a speed advantage.

For this thesis the choice between these options is not critical as the model is expounded fully and the implementor is at liberty to choose a suitable implementation.

3.4. Conclusion

The model developed informally in this chapter has tackled the problems of devising a representation suitable for inclusion in a database that contains the usual commercial data. A discussion of the query language required has shown that few new operators are needed to produce a working graphical database system.

In the following chapter this model is placed on a sound footing by providing a formal specification of the model proposed in this chapter.

Formal Specification of GDB

4.1. Introduction

The motivation for this part of the thesis stems from the papers by Wong and Samson [51] and Mallgren [52] who used algebraic specification methods to specify a database system and a graphics system respectively (see Chapter 2).

Wong and Samson use the applicative language HOPE [53] to express the semantics of the PRECI [54] relational database and its algebraic language PAL. This involved specifying (among other entities) schemes, tuples and relations as abstract datatypes together with the operations on these types (such as union compatibility, difference, join etc.). The specification shown here was devised independently as the notion of schemes and their inherent naming problems was not considered an imperative part of the present specification. (See Appendix C for the theoretical basis of specification methods. Note that the decision to exclude schemes illustrates the point made in the appendix about how a level of abstraction is decided upon by individual modellers depending on the problem under consideration.)

4.2. Specification of a Relational Database

The following is a detailed explanation of the specifi-

cation for a graphics database written in Standard ML (SML) and this has been executed with suitable test data and shown to be consistent. The proof of any specification is a matter of conforming to the higher level axioms that govern its behaviour. A sample proof is given in Appendix D.

The style of presentation of the specifications that follow is to provide axioms in the manner advocated by Sannella [55] using predicate logic to specify the properties of the functions. These axioms are only comments and do not form part of the executable specification. The SML functions follow. (See Appendix B for an explanation of the syntax of SML).

The first step is to establish the constructors for the datatype of interest (in this case "relations"). The "datatype" construction of ML is used and this consists of the following structure :

```
datatype < type name > = < constructors >
```

The constructors are the keywords that allow members of the datatype to be constructed. So in this case, the constructors for a relation are "mt" (pronounced "empty") and "tcons". (The "'a" is a type variable. This is instantiated to a type at execution time and allows polymorphic specifications to be written i.e. the functions can be applied to any type.)

```
datatype 'a relation = mt |
                    tcons of 'a * 'a relation
```

Thus the new type "relation" can have any scheme which is established at run-time by the SML type checking system according to the type used in the data.

The specification includes a number of general utility functions of which "ismt", "member" and "card" are three. They are used by other functions rather than invoked by the user directly.

```
(*****)
(* Axiom forall R => R = mt <==> ismt(R) *)

fun ismt(mt ) : bool = true
| ismt(tcons(t,r)) = false

(*****)
(* Axiom forall t,R=> R=mt ==> member(t,R) = false *)
(* Axiom forall t,R => member(t,tcons(t,R)) = true *)

fun member(tuple:'a , mt : 'a relation) = false
| member(tuple: 'a , tcons(tup:'a, r: 'a relation)) =
if tuple = tup
then true
else member(tuple,r)

(*****)
(* card is normal cardinality function *)
(* Axiom => card(mt) = 0 *)
(* Axiom forall t,R=> card(tcons(t,R))=1 + card(R) *)

fun card(mt : 'a relation ) = 0 : int
| card(tcons(t,r)) = 1 + card(r) : int
```

Ismt is a predicate that returns true if the relation has no tuples and false otherwise. The usual set membership

function is embodied in "member" which is constrained to take only relations as parameters. The "card" function returns the cardinality or count of the number of tuples in the given relation.

Two further functions are "add" and "mkrel".

```
(*****)
(*Axiom forall t,R=> member(t,R) ==> add(t,R) = R *)
(*Axiom forall t,R=> not(member(t,R)) ==>          *)
(*          add(t,R) = tcons(t,R) *)

fun add(t,r) = if member(t,r) then r else tcons(t,r)

(*****)
(* Axiom mkrel(nil) = mt *)
(* Axiom forall h,t =>          *)
(*          mkrel(h::t) = add(h,mkrel(t)) *)

fun mkrel(nil : 'a list) = mt
| mkrel(h::t) = add(h,mkrel(t))
```

These functions provide the only method of building a relation in SML. The function "mkrel" takes a list of tuples and places them in a relational structure after using "add" to ensure that no duplicate tuples are created.

The set functions as used in most relational database languages are specified below.

```

(*****)
(* Axiom forall t: tuple; R,S : relation =>      *)
(* member(t,R) or member(t,S) ==>              *)
(*          member(t ,union(R,S)) *)

fun union(mt,r) = r
| union(tcons(t,r),rr) = if member(t,rr)
then union(r,rr)
else tcons(t,union(r,rr))

(*****)
(* Axiom forall t: tuple; R,S : relation =>      *)
(* member(t,R) and not(member(t,S)) <==>        *)
(*          member(t ,diff(R,S)) *)

fun diff(mt,r) = mt
| diff(tcons(t,r),rr) = if member(t,rr)
then diff(r,rr)
else tcons(t,diff(r,rr))

(*****)
(* Axiom forall t: tuple; R,S : relation =>      *)
(* member(t,R) and member(t,S) <==>            *)
(*          member(t ,intersect(R,S)) *)

fun intersect(mt,r) = mt
| intersect(tcons(t,r),rr) = if member(t,rr)
then tcons(t,intersect(r,rr))
else intersect(r,rr)

```

These are the usual union, difference and intersect functions constrained to apply only to relations. Note that they all use the member function to check for the presence of a specified tuple in a relation and thereby determine whether that tuple is a member of the result relation. The "member" function is polymorphic so that it will cope with any scheme of relation.

The functions for the relational algebra operations select and project are given below. These functions are

higher order functions i.e. they take functions as parameters as well as the values to be used by those functions. These functions are very like the "MAPCAR" function in LISP which takes a list and a function as arguments and applies the function to each element of the list and returns a list as a result, which contains the same number of elements, each of which has been transformed by the function. The design of the function passed to these functions is critical to their operation. The signature of the required function is shown in the specification. The boolean returned by the function passed to "select" determines whether the tuple is to be included in the result relation or ignored. So for a selection function the boolean will be true only for tuples that are found which satisfy the selection predicate.

```
(*****  

(* Axiom forall t:tuple;f:function;R:relation => *)  

(* member(t,R) and f(t) <==> member(t,select(R,f))*
```

```
fun select(mt, f:( 'a -> bool)) = mt : 'a relation  

| select(tcons(tup: 'a ,r: 'a relation), f) =  

  if f(tup)  

  then tcons(tup,select(r,f))  

  else select(r,f)  

end
```

```
(*****  

(* Axiom forall t:tuple;f:function;R:relation => *)  

(* member(t,R) <==> member(f(t),project(R,f)) *)
```

```
fun project(mt, f:( 'a -> 'b)) = mt : 'b relation  

| project(tcons(tup: 'a ,r: 'a relation), f) =  

  add(f(tup),project(r,f))
```


For example, given a relation " parts " and the selection function "pselect" the boolean returned will only be true for those parts which satisfy ' colour = "red" '.

```
val parts = mkrel([ ("p1","nut","red"),
  ("p2","bolt","blue"),
  ("p3","cam","blue"),
  ("p4","bolt","blue"),
  ("p5","screw","green") ])

fun pselect(pnum,name,colour) = (colour = "red")

fun projfun(pnum,name,colour) = (name,colour)
```

The use of "add" is not required in the "select" function because the input relation does not contain any duplicate tuples so there is no need to check or remove them in the result relation. On the other hand, the "project" function may generate duplicate tuples as a result of removing attributes from the input relation. So, for example, the function "projfun" when used with the above relation will create two identical tuples one of which will, of course, be ignored by "add".

Using a function which is isomorphic to "project" it is possible to specify the "extend" operation. This has been suggested as a useful addition to the relational algebra. Its purpose is to create a new relation by adding one or more columns to an existing relation as the result of a function applied to one or more of the existing columns. (e.g. adding an age column derived from a date of birth).

This is defined below.

```
(*****  
(* Axiom forall t:tuple ;f:function;R:relation => *)  
(* member(t,R) <=> member(f(t),extend(R,f))      *)  
  
fun extend(mt, f:('a -> 'b)) = mt : 'b relation  
| extend(tcons(tup: 'a ,r: 'a relation), f) =  
    add(f(t),extend(r,f))
```

The other relational algebra operations "cartesian product" and "join" can be specified in similar fashion by using a "MAPCAR"-like function which takes two relations and a function as arguments and returns a relation which is the result of the application of the function to the two argument relations. The body of the "cartesian product" and "join" functions are similar. The "join" function requires a predicate that determines whether the combined tuples will be included in the result relation whereas the "cartprod" function needs no predicate as all combinations of tuples appear in the result relation.

```

(*****)
(* The parameters of "combine" are a tuple,      *)
(* a relation and a function giving              *)
(* a 'a*'b relation                              *)
*)

fun combine(tup,mt,f: ('a*'b -> bool)) = mt
| combine(tup ,tcons(tt,rr), f) =
if f(tup,tt)
then tcons((tup,tt),combine(tup,rr,f))
else combine(tup,rr,f)

(*****)
(* Axiom forall t,s:tuple;R,P:relation =>      *)
(* member(t,R) and member(s,P) and f(t,s) <==> *)
(* member((t,s),join(R,P))                    *)
*)

fun join(mt,r,f) = mt
| join(tcons(t,r),s,f) =
union(combine(t,s,f),join(r,s,f))

(*****)
(* Ccombine is similar to "combine" except that *)
(* there is no function required                *)
*)

fun ccombine(tup: 'a, mt:'b relation = mt
| ccombine(tup ,tcons(tt,rr)) =
tcons((tup,tt),ccombine(tup,rr))

(*****)
(* Axiom forall t,s:tuple;R,P:relation =>      *)
(* member(t,R) and                              *)
(* member(s,P) <==> member((t,s),prod(R,P))    *)
*)

fun cartprod(mt,s) = mt
| cartprod(tcons(t,r),s) =
union(ccombine(t,s),cartprod(r,s))

```

This specification is sufficient to provide an executable relational database that provides the base level functions normally expected. There is no concept of schemes or integrity embodied in this specification as they were not crucial to the prototyping of the Graphical Database. Integrity issues are discussed in Chapter 7.

4.3. Specification of the GDB Graphical Operators

It is not practical to expect any user to formulate a query in terms of the primitives (such as lines, facets or tetrons) and explicitly specify the series of graphical joins necessary to generate the display of the object(s) that form the result of the query. Thus the query language is tailored to the application and deals with entities at a higher level. The commands in the user language are transformed into a sequence of commands in the database language. The following language specification defines the low level functions that support the user oriented language.

As discussed in Chapter 3 there is one operator ("show") designed to handle the display of objects stored in the database as lines, facets or tetrons relations. This relies on the "gjoin" function which is a complex function that has declared within it a number of local functions designed to produce data in intermediate relations for later processing.

```

(*****)
(* Axiom forall pt:point;r:relation;o : object => *)
(*      member(o,r) <==> subset_of(gjoin(o),points)*)

fun gjoin(objname,transmat) =
let val T1 = let fun sfun(name) = (name=objname) in
let fun pfun(onum,name) = onum in
project(select(object,sfun),pfun) end end

let val T2 = let fun pfun2(onum1,onum2,lnum) = lnum
in
let fun jfun2(onumj,(onum,lnum)) = (onumj=onum) in
project(join(T1,linelink,jfun2),pfun2) end end in

let val T3 = let fun pfun3(t2lnum,llnum,spt,ept) =
(spt,ept) in
let fun jfun3(t2lnum,(llnum,spt,ept)) =
(t2lnum=llnum) in
project(join(T2,lines,jfun3),pfun3) end end in

let val T6 = let fun pfunl6(spt,ept) = spt in
let fun pfun26(spt,ept) = ept in
union(project(T3,pfunl6),project(T3,pfun26))
end end in

let val pts =
let fun jfunl((ptnum,x,y,z),pt) = (ptnum = pt) in
join(points,T6,jfunl) end in

let val transpts = transform(pts,transmat) in

let val inter = let
fun jf((pt,x,y),(l,spt,ept)) = (pt=spt) in
let fun pf(pt,x,y,l,spt,ept) = (l,x,y,ept) in
project(join(transpts,lines,jf),pf) end end in

let fun jf2((pt,x,y),(l,x1,y1,ept)) = (pt=ept) in
let
fun pf2(pt,x,y,l,x1,y1,ept)=(l,x,y,x1,y1) in
project(join(transpts,inter,jf2),pf2) end end ... ;

fun display(disprel) =
if card(disprel) = 0 then 1
else let val tcons(t,r) = disprel in
let val (l,txs,tys,txe,tye) = t in
(plot(4,txs,tys);plot(5,txe,tye);display(r))
end end ;

fun show(object, viewmat) =
display(transform(gjoin(object),viewmat()));

```

The overall function of this somewhat unwieldy specification is explained in Chapter 3 where, in the section on query language the "gjoin" function is explained in terms of SQL like statements.

The function "plot" is based upon two functions ("byte" and "word") which send control codes to the graphics display device (see Appendix K). In this way the graphical output of the proposed database model can also be checked and refined at the specification stage. These functions are shown below.

```
fun putchar(c : int) = output(std_out,chr(c)) ;
fun byte(c : int) = let val c = if c<0 then 0
                             else c in
                    if c<=15 then (putchar(c+96))
                    else if c<=95 then (putchar(c)) else
                    (putchar(112+(c div 16));
                     putchar(96+(c mod 16)))
                    end ;
fun word(c: int ) = (byte(c mod 256);
                    byte((c div 256) mod 256));
```

The coding of the "byte" function is determined by the graphics chip installed in the BBC micro (used as graphics terminal).

4.3.1. Predicates

The inclusion of novel data types (such as graphics) into a database must be accompanied by a set of predicates that enable members of the data type to be compared. Some,

like "=", are usually overloaded such that they can compare two members of any data type, but others (e.g. ">") have little currency outside reals, integers and characters. A suitable set of predicates can be defined to enable queries to be expressed easily (Figure 4.1).

```

share_edge      : facet X facet -> bool
share_vertex    : facet X facet -> bool
within         : point X line -> bool
enclosed_by    : point X facet -> bool
enveloped_by   : point X tetron -> bool
cross          : line X line -> bool
intersects     : line X facet -> bool
overlaps       : facet X facet -> bool
interpenetrate : tetron X tetron -> bool
pass_thru      : line X tetron -> bool

```

Figure 4.1: Graphical predicates

There is also a set of simple functions which can be defined that will be widely used (Figure 4.2).

```

length      : line -> num
fun length((x,y,z),(x1,y1,z1)) <=
    sqrt(sq(x-x1)+sq(y-y1)+sq(z-z1)) ;
area       : facet -> num
volume    : tetron -> num

```

Figure 4.2: Functions

4.4. Conclusion

This formal specification of the graphical database GDB

has established the functionality in such a way that an implementation should follow with few problems. The next chapter describes the three implementations based on the above specification.

Implementation

5.1. Introduction

It is necessary to implement the ideas formulated in this thesis for two reasons. Firstly, it is desirable to prove that the thesis is valid and secondly, to prompt further refinements and improvements to the model. There were three implementations of the model :

- (1) The C language implementation of the SML specification.
- (2) The VMS/RDB implementation using PASCAL as host language.
- (3) The PRECI/C implementation from the SML specification.

The three versions were attempted as software became available and was deemed appropriate for the task.

5.2. C Language Version

The SML specification of the graphical database model described in this thesis was used as the basis of an implementation in C. The process of transformation from functional specification language to implementation language was an interesting exercise in itself but was not pursued to a fully functional database management system as this would

have detracted from the work on the query language and the GDB model. The source code is described in Appendix I where the SML axioms are shown for one of the relational operations (union) together with the C language implementation.

5.3. The VMS/RDB Version

RDB is the proprietary Relational Database Management system marketed by the Digital Equipment Corporation. This was used as the host database on a VAX 11/750/780/8530 cluster running the VMS operating system. As it is possible to call this database from within VMS/Pascal a graphics interface was built on top of the database to display the results of queries in GDB on a colour graphics terminal. (See Appendix K for hardware details).

The advantage of using VMS/RDB was that a result could be achieved very quickly while having a secure database system controlling the data. Thus, the model could be explored and substantially prove its worth. The provision of a built-in integrity constraint system was also considered an important feature which could be used as a test-bed for further work.

The main disadvantage with using RDB was that there was only a limited interactive interface via PASCAL to RDB so all the queries had to be expressed in terms of RDB commands and pre-compiled. This and the cumbersome query language (RDO) meant that an alternative vehicle was sought. (The

RDB schema and PASCAL programs are given in Appendix J).

5.4. The PRECI/C Version

The deductive database query language for PRECI/C called DEAL (DEductive ALgebra) [56] was implemented by Dr. Sadeghi at Dundee College of Technology. This work was done partly as a vehicle for HQL (Historical Query Language) [57] and included extensions to DEAL to allow the user to express temporal queries. The "date" attribute type was added to the PRECI/C repertoire of integer and character string and a set of operators for type date were included.

The main features of the language are the user defined functions and views, which can be recursive. Imperative features such as assignment and while loops were also provided. These features allow very complex queries to be formulated relatively easily. There is also the possibility of a straightforward translation from ML specifications into DEAL functions and views which allows ML to be used a prototyping language for database queries.

In order to utilise DEAL as the language for the Graphical Database substantial extensions to the existing implementation were required and this work is described here.

The primary requirement was for the attribute type "real" to allow real numbers to be stored and manipulated. This is not straightforward because the PRECI/C implementa-

tion stores all the data as files of integers. This is a strategy that produces a compact database with quick access times provided the only attribute types are integers and character strings, but precludes the possibility of storing floating point numbers. The solution chosen was to convert all real numbers into character strings prior to insertion into a PRECI/C relation and convert the strings back into floating point again when arithmetic operations were required. This conversion uses the C language functions "sscanf" and "sprintf".

In addition, a new datatype (called "dble") was introduced into the DEAL language and the normal arithmetic operators (already defined for integer operations) were overloaded to accept floating point numbers as operands and results. Having established a viable scheme for handling real numbers in both PRECI/C and DEAL the necessary operators for the Graphical Database were defined. These included sine, cosine, atan, floor, square root etc. with the usual C meaning.

To perform the necessary matrix operations which form the basis of graphical transformations (Appendix A) it is necessary to take the points data from the relation and perform a matrix multiplication with the transform matrix. This transforms the x,y and z coordinates into values that are suitable for the viewing operation. There are two methods

of performing this operation. Firstly one could move the points data into a matrix structure and, after performing the necessary operations, insert the data into a new relation. The second option entails storing the matrices as relations and performing all the operations by using a suitably modified database language.

The latter approach was adopted as the expressive power of DEAL was capable of supporting the operation required by the use of functions and views. The first option would be more suitable if an implementation using an imperative host language was attempted as the requisite data structures would be readily available.

The necessary matrices can be created from DEAL and exist in the database as empty relations until some graphical transforms are required, at which time data is inserted by the use of DEAL views. As this data is not committed to the database it exists only for the duration of the query session. In this way there is no storage overhead for large numbers of relations with only four tuples. There is clearly a computational overhead in calculating and inserting data at run time but this is a small price to pay for the flexibility such a system provides to the user.

All the necessary matrix operations can be written in DEAL and, as intimated above, are translations from the ML specification. The full DEAL programs are given in Appendix

H but some of the typical operations are described here.

Firstly consider the translation matrix. This is usually written

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{matrix}$$

where T_x , T_y , T_z are the translation factors for each axis. This is created as a PRECI/C relation with following DEAL statement.

```
create table trans ( number int(1) nonnull ,
                    one real,
                    two real,
                    three real,
                    four real);
```

Note that a key attribute is included because the relational model has no natural ordering of tuples while, clearly, the matrix has precisely ordered rows. The attribute names were chosen for the same reason. The matrix operations can be defined easily using the keys and attributes names without the need for sorting the relation prior to performing any transformation operation.

The data can be inserted into this relation by the use of a DEAL parameterised view.

```

view translate(Tx:db1e,Ty:db1e,Tz:db1e) as {
insert into trans values ( 1,1.0,0.0,0.0,0.0);
insert into trans values ( 2,0.0,1.0,0.0,0.0);
insert into trans values ( 3,0.0,0.0,1.0,0.0);
insert into trans values ( 4,Tx,Ty,Tz,1.0);
        translate := trans ;
};

```

The four insert statements place the data into the relation "trans" and this is returned as the result of invoking this "view". In this way the required relations (matrices) can be constructed when needed. The use of the insert statement within a view was not a feature of the original HQL implementation and was added to facilitate this type of operation.

Two other major operators are "matrix multiplication" (required for composing a total transformation matrix) and "point transformation" (the application of a matrix to a points relation). Both these view definitions are supported by function definitions for low level functions to extract values and perform arithmetic operations.

The view definition "matmult" shows a recursive structure which has the side effect of inserting a tuple into the result relation for each pass through the "else" branch of the body. The operators "first" and "rest" are the relational equivalent of the selector functions "head" and "tail" for lists and provide a means of terminating the recursion by testing for the empty relation using the

built-in function "card" which returns the cardinality of the relation.

```
view matmult(matrixa:rel, matrixb:rel, result:rel)
as {
  if (card(matrixa) = 0) {
    matmult := result;
  } else {
    fr := first(matrixa);
    amatval := #(fr [number]);
    rc1 := rcmult(fr,matrixb,one);
    rc2 := rcmult(fr,matrixb,two) ;
    rc3 := rcmult(fr,matrixb,three);
    rc4 := rcmult(fr,matrixb,four);
    insert into result values (amatval,rc1,rc2,rc3,rc4);
    matmult := matmult(rest(matrixa),matrixb,result) ;
  }
};
```

The symbol '#' is an operator that when applied to a relation with a single tuple with a single attribute extracts the value which is then assigned to the variable on the left hand side of the assignment.

The view definition for "transformpts" is similar in structure to the matrix multiplication view but requires a points relation as first parameter rather than a matrix (relation).


```

view transformpts(pts:rel, matrix:rel ,result:rel)
as {
if (card(pts) = 0) {
transformpts := result;
} else {
fr := first(pts);
ptval := #(fr [ptnum]);
rc1 := rcpts(fr,matrix,one);
rc2 := rcpts(fr,matrix,two) ;
rc3 := rcpts(fr,matrix,three);
insert into result values ( ptval, rc1,rc2,rc3) ;
transformpts:=transformpts(rest(pts),matrix,result);
}
};

```

In practice it was found that although it was possible to write DEAL functions to perform matrix multiplication, the implementation of PRECI/C was unable to cope with the large number of relations and variables created. This was due to its simplistic heap management and garbage collection algorithms whose performance degrades rapidly as the allocated space is used and released space becomes more fragmented. Increasing the heap space has little effect.

Consequently, it was necessary to build in the matrix multiplication operation into the DEAL language (using the symbol |x|) which would allow two relations representing matrices to be multiplied and return the result in a temporary relation. An example DEAL view is shown below.

```

view test(mat1 : relation, mat2 : relation) as {
test := mat1 |x| mat2 ;
}

```

5.5. Conclusion

In this chapter the implementation issues involved in providing a vehicle for exploring the GDB model was given based upon the formal specification which appears in Chapter 4. The PRECI/C implementation using DEAL as a query language was the main development system used as its expressive power and flexibility proved ideal for the purpose. This work is exemplified in the next chapter where a trial application is described which uses the PRECI/C - DEAL system.

Example Graphical Databases

Two different examples were implemented to demonstrate the facilities of the proposed database model. The first was a simple database of a two-dimensional hierarchical object to illustrate the "flatten" function which is necessary to produce relations suitable for the "gjoin" and "show" functions. The second was a database of a cube without hierarchy to demonstrate the matrix transformations in action for a three dimensional object.

The examples are designed to show different features of the proposed system and are therefore slightly artificial. A real application would combine all the features of flattening, viewing matrices, perspective transforms and display functions. The separation of these features in these examples is intended to clarify the explanation of the functions.

6.1. Database (2-D) with Object Hierarchy

A simple picture of a house with a window and a door was chosen to illustrate an object hierarchy. The relations are shown in Figure 6.1 and the associated functions to produce a display are shown in Figure 6.2.

Object

onum	oname
o1	roof
o2	door
o3	window
o4	wall
o5	house
o7	frame
o8	window2

Hierarchy

sup	inf
o5	o1
o5	o2
o5	o3
o5	o4
o6	o1
o6	o2
o6	o3
o6	o4
o2	o7
o2	o8

Figure 6.1: Hierarchy relations

```

(*****)
(* Getall recursively produces all the leaves of *)
(* the hierarchy *)

view getall (recrel : rel , part : rel) as {
    if ( card(recrel) = 0 ) { getall := recrel ;}
    else {
temp := (( recrel (inf,sup) part )
          rename [0:=sup,1:=inf] ) ;
getall :=(recrel--(temp [sup])) ++
          getall(temp [inf],part) ;
    }
};

(*****)
(* Flatten is top level procedure to decompose *)
(* a hierarchical object *)

view flatten (onum : char , part : rel) as {
    templ := (part where sup = onum) [inf] ;
    flatten := getall(templ,part) ;
};

(*****)
(* This is the gjoin for an object defined only *)
(* in terms of line primitives *)

view gjoin(objname:char) as {
nrel := ( xobject where oname = objname ) [onum] ;
frel := flatten(#(nrel),xhier);
lrel := ((frel ( inf,onum ) xlinelink)
          rename [1:=lnum]) [lnum];
l2rel := (lrel ( lnum,lnum ) xlines )
          rename [0:=lnum,1:=spt,2:=ept];
pt1 := (l2rel (spt,ptnum) xpoint )
        rename [0:=lnum,2:=ept,3:=sx,4:=sy];
pt1tmp := pt1 [lnum,ept,sx,sy] ;
pt2 := (pt1tmp (ept,ptnum) xpoint )
        rename [0:=lnum,2:=sx,3:=sy,4:=ex,5:=ey];
fin := pt2 [lnum,sx,sy,ex,ey] ;
gjoin:= fin ;
};

(*****)
(* Display takes the final relation of x,y pairs *)
(* and produces screen output using plot *)

func display(scrpts :rel ) {
    if (card(scrpts) = 0 ) { display := 1 }
    else {
        fr := first(scrpts);
    }
};

```

```

        stx := #(fr [sx]);
        sty := #(fr [sy]);
        enx := #(fr [ex]);
        eny := #(fr [ey]);
        ds := plot(4,floor(stx),floor(sty));
        de := plot(5,floor(enx),floor(eny));
        display := display(rest(scripts));
    }
};

(*****)
(* Show selects the object to be displayed and *)
(* calls gjoin and display to do the job.      *)

func show(objname : char) {
    d := setup();
    show := display(gjoin(objname));
};

(*****)
(* Actual call to "show" for the object "House" *)

show("house");

```

Figure 6.2: Functions for display of hierarchy

The main DEAL views of interest here are "flatten" and "getall". As discussed briefly in Chapter 3, the hierarchy of objects and sub-objects must be broken down into a flat structure which contains only the lowest level objects i.e. those which are not composed of any other objects. The example used here depicts a simple two dimensional house which is composed of a wall, a roof, a window and a door. The door is composed of a frame and a small window. The coordinates were chosen so that no workstation transformation was necessary (see Appendix A for definition).

From the hierarchy relation the selection of inferior

objects to "o5" yields objects "o1", "o2", "o3", and "o4". Object "o2" ("door") also appears in the "sup" attribute of the hierarchy relation so must be decomposed to get the ultimate objects "o7" and "o8". A diagram of the hierarchy is shown in Figure 6.3. The final relation must contain objects "o1", "o3", "o4", "o7" and "o8". This operation of decomposition is performed by "flatten" which obtains the immediate sub-parts of the chosen object and then passes that relation to "getall" which recursively joins the derived relations to the hierarchy relation until there are no sub-parts remaining.

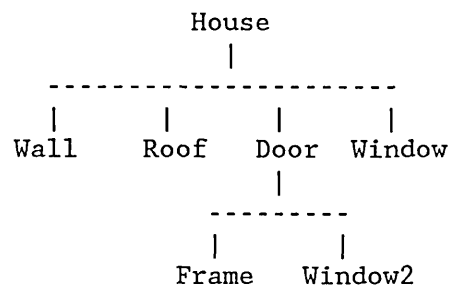


Figure 6.3: Hierarchy of the 'House'

These two views make use of the "rename" facility of DEAL which allows attributes to be renamed. This is necessary because the naming convention of PRECI/C after a relational join is to prefix the attribute name with the parent relation names to create unique names for the attributes of the result relation. To write recursive views it is

necessary to have consistent and predictable names so the "rename" facility is essential. The attribute number (normally internal to PRECI/C) is used to identify the attribute to be renamed and the identifier after the "!=" is taken to be the new name.

The view "gjoin" has the meaning described in Chapter 3 and produces a relation suitable for display after joins with the "line_link", "lines" and "points" relations. The function "display" extracts the values from the resulting relation and uses the function "plot" to produce graphical output (see Figure 6.4). The function "setup" is used to initialise the display device to give a graphics window and a small text window.

The full details of the DEAL program for this section are given in Appendix H.

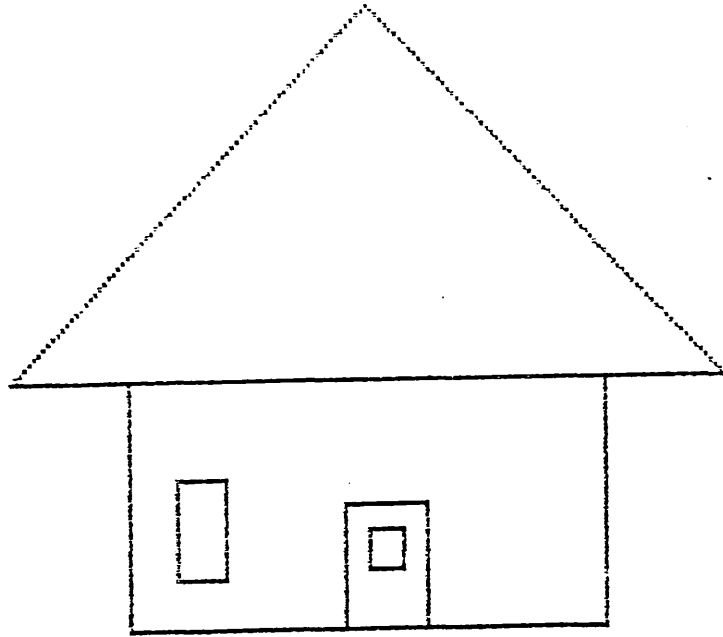


Figure 6.4: Screen dump of 'House'

6.2. Database (3-D) with Matrix Operations

The relations for a simple cube are shown in Figure 6.5 and its associated functions and views in Figure 6.6.

bpts

num	x	y	z
1	0.0	0.0	0.0
2	50.0	0.0	0.0
3	50.0	50.0	0.0
4	0.0	50.0	0.0
5	0.0	0.0	50.0
6	50.0	0.0	50.0
7	50.0	50.0	50.0
8	0.0	50.0	50.0

clines

lnum	spt	ept
"11"	1	2
"12"	2	3
"13"	3	4
"14"	4	1
"15"	5	6
"16"	6	7
"17"	7	8
"18"	8	5
"19"	1	5
"110"	2	6
"111"	3	7
"112"	4	8

Figure 6.5: Relations for cube

```

(*****)
(* Getcol, rval, cval and rcpts extract data      *)
(* values from the points and matrix relations  *)
(* and multiply them together to give the       *)
(* transformations required.                    *)

view getcol(arel :rel,num :at) as {
    getcol := arel [number,num] ;
};
func rval(pts :rel, col :at) {rval := #(pts [col]);
};
func cval(xmat :rel, cname :at,n :int){
cval := #((getcol(xmat,cname)) [cname]
           where number = n);
};
func rcpts(pts:rel,bmat:rel,cname:at) {
    rcpts := rval(pts,x) * cval(bmat,cname,1)
           + rval(pts,y) * cval(bmat,cname,2)
           + rval(pts,z) * cval(bmat,cname,3)
           + 1.0 * cval(bmat,cname,4) ;
};

(*****)
(* Transformpts calls the above functions to      *)
(* perform the transform of a points relation by *)
(* the matrix relation                            *)

view transformpts(pts :rel,amatrix :rel,c :rel) as{
if (card(pts) = 0) {
    transformpts := c; } else {
fr := first(pts);
ptval := #(fr [ptnum]);
rc1 := rcpts(fr,amatrix,one);
rc2 := rcpts(fr,amatrix,two) ;
rc3 := rcpts(fr,amatrix,three);
vsx := (511.5 * (rc1/rc3)) + 511.5 ;
vsy := (511.5 * (rc2/rc3)) + 511.5 ;
insert into c values ( ptval, vsx,vsy,0.0) ;
transformpts := transformpts(rest(pts),amatrix,c) ;
    }
};

(*****)
(* Simplified gjoin that doesn't use linking      *)
(* relations.                                     *)

view gjoin(npts:rel) as {
pt1 := (clines (spt,ptnum) npts )
       rename [0:=lnum,2:=ept,3:=sx,4:=sy];
ptltmp := pt1 [lnum,ept,sx,sy] ;
pt2 := (ptltmp (ept,ptnum) npts )
};

```

```

        rename [0:=lnum,2:=sx,3:=sy,4:=ex,5:=ey];
gjoin := pt2 [lnum,sx,sy,ex,ey] ;
};

(*****
(* This produces the viewing matrix          *)

view viewmat(vx:db1e,vy:db1e,vz:db1e,dx:db1e,
            dy:db1e,dz:db1e,azrot:db1e) as {
if (vx=dx) { vdx := 1.0 ; } else { vdx := vx-dx ; }
if (vy=dy) { vdy := 1.0; } else { vdy := vy-dy ; }
nvx := 0.0 - vx ;
nvy := 0.0 - vy ;
nvz := 0.0 - vz ;
m1 := translate(nvx,nvy,nvz) ;
m2 := rotx(90.0,xtmp1) ;
n3val := 0.0 - (180.0+(atan((vdx)/(vdy))
                    *180.0/3.142)) ;
m3 := roty(n3val);
n4val := 0.0 - (atan((vz-dz)/
                    (sqrt((vdy*vdy)+(vdx*vdx))))*180.0/3.142);
m4 := rotx(n4val,xtmp2) ;
n1 := 0.0 - 1.0 ;
m5 := scale(1.0,1.0,n1,stamp1) ;
m6 := scale(4.0,4.0,1.0,stamp2) ;
viewmat := (((m1 |x| m2) |x| m3) |x| m4)
            |x| m5) |x| m6;
};

view vmat() as {
vmat := xviewmat(100.0,200.0,100.0,0.0,
                0.0,0.0,135.0,xtmp);
};

(*****
(* Show selects the object to be displayed and *)
(* calls gjoin and display to do the job.      *)

func showx(pts:rel) {
d := setup() ;
showx:=display(gjoin(transformpts(bpts,vmat,cpt)));

(*****
(* Actual call to "show" for the object "Housel" *)

showx(bpts);

```

Figure 6.6: Functions and views for the cube

As described in Appendix A, the transformations neces-

sary to produce coordinates suitable for a display device involve constructing a matrix from a number of matrices that embody the desired transformation. The complexity is mainly due to the viewing transform which requires a view point and a direction vector which is used to determine the apparent position in space of each point in the object being viewed.

The DEAL views which perform these functions in this example are "viewmat" and "transformpts". The view "viewmat" produces a matrix relation that represents the viewing transformation for the scene from the designated view point passed as parameters vx, vy and vz. The parameters dx, dy and dz are the coordinates of a point that, when joined to the view point produces the viewing direction vector. The parameter "azrot" fixes the final degree of freedom to produce the desired orientation of the final image.

The individual matrix relations for the viewing transform are specified as parameterised views previously defined (see Chapter 5 for an example). The total viewing matrix relation is produced by multiplying together the individual matrix relations using the "|x|" operator described in the previous chapter.

The viewing matrix relation is used by the view "transformpts" to change the coordinate values of the points for the 3-D object of interest to simulate the view from the view point. The perspective transform is performed to

produce the final 2-D points (vsx and vsy) which can be displayed on the output device by the function "display". The result of these operations is shown in Figure 6.7.

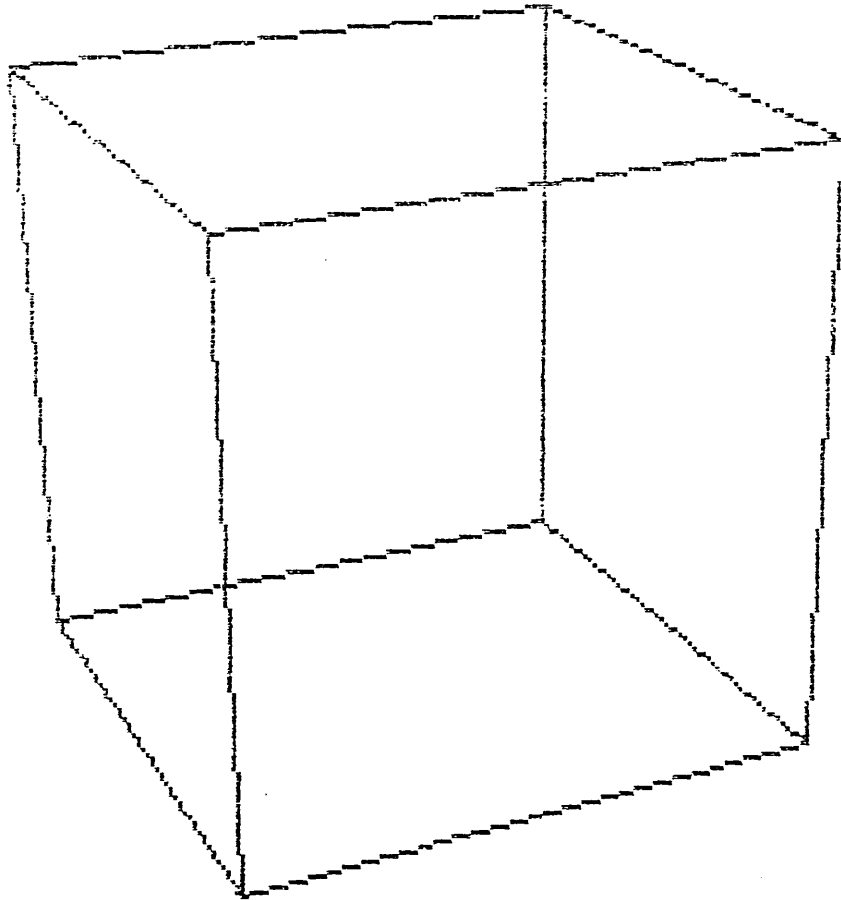


Figure 6.7: Screen dump of 'Cube'

6.3. Conclusion

In this chapter it has been shown how object hierarchies and viewing transforms are defined in DEAL and how the "gjoin" operation is defined in practice. The combination of all the ideas embodied in this chapter would implement a system as described in Chapter 3. In the next chapter the related topics not investigated in this thesis are discussed with a view to providing a plan for further work.

CHAPTER 7

Conclusions

7.1. Assessment of GDB Model

The GDB model as described in this thesis provides a graphical database with a powerful query language (DEAL) that is capable of accessing hierarchically built objects and displaying the results of queries graphically under user control. This thesis addresses the major issues cited in Chapter 1 of representation and expressive query languages and provides an overall solution to this type of database that the reviewed texts do not.

The table in Figure 7.1 shows how the GDB model compares with the other 3-D models reviewed in Chapter 2. Using a points based representation the GDB model provides a query language (DEAL) which enables graphical manipulation together with a database query language of great expressive power. None of the other languages with graphical operators can perform such complex queries with any ease.

	Representation			Graphics	Integrity	2/3 D
	Pts	Grid	Obj	Language		
Lee (12)			x		x	3
Shapiro (31)			x			3
Morffew (8)	x				x	3
Su (14)			x			3
Tikkanen (10)	x			x		3
GDB	x			x	x	3+

Figure 7.1: GDB compared to 3-D models

Of the reviewed work only that of Garrett and Foley [32] treat integrity as a fundamental issue and the GDB model matches that work with several possible solutions to providing integrity checking in DEAL. (See section 7.2.2)

The necessity of expressing objects as hierarchies of sub-objects has been explored by Lorie et. al. Their "complex object" model addresses the problems of hierarchical structures but they do not provide a display of the stored objects as DEAL is capable of doing. Their use of an enhanced SQL as query language reflects the growing importance of this as the standard database language despite its deficiencies in solving complex problems such as parts explosions [57].

This thesis has shown that a more powerful language is

necessary to cope with such "real life" problems.

7.2. Further Work

The following issues have only been considered briefly and constitute areas where future research can be directed. Each of the following sub-sections introduce the topics of interest and show how they affect the model proposed here.

7.2.1. Physical Access Methods

As noted in Chapter 2 there may be benefits to be had from organising the DBMS to hold the graphical representation of a single object on contiguous sectors of a disc to improve the speed of access. It is beyond the scope of this thesis to delve deeply into how this might be achieved but a brief discussion follows.

Abel's proposition [49] is that a query that has spatial properties is slow to answer if the data referring to that space is not clustered in some way on the storage device. Similar views are expressed by Frank [58] who points out the problem of artificial clusters of data which produce fixed size areas for retrieval only provide fast access if the query returns areas of that size.

7.2.2. Integrity Constraints

Integrity constraints are an essential part of the relational model and are used to ensure that the database always contains information that is consistent. This is important after any operation that changes the data stored in the base relations. For example, if we add a new line to the lines relation this includes two references to the points relation and these values must exist for the new line to be accepted. If the the values are not present in the points relation then the line cannot be displayed as there will be no data available concerning its position in 3-D space.

Similarly when deleting from the points relations there can be no deletions of points that are referenced in other relations in the database as these primitives would then be rendered meaningless.

In any real application the user would want to define integrity constraints on the objects of interest to him, so a mechanism must be available for him to do so. In the current implementation of DEAL there are problems with implementing integrity constraints as the language does not yet support the delete and rollback functions. Constraints can be explored for the insert function which is supported. Three possible solutions are :

- (1) Include integrity checking into the appropriate operations. This is simply a matter of including boolean

functions which will prevent any semantically irregular operations. For example :

```
func line-insert(Lnum : id,spt : id,ept : id){
    if card((points where ptnum=spt) +
        card(points where ptnum=ept))= 0
    { error("Line invalid") }
    else insert(ptnum,lines);
```

- (2) Establish an "Object oriented" implementation where each relation inherits some subset of the basic operators as well as its own set of operators that include integrity checking. Thus the only interface to members of that class is by the declared specialist operators. The following is a simple syntax for the declaration of relations.

```
declare relation <name>(<attribute list>)
    of class <class name>
    <attribute name> : <type>
    ...
    with operations <function list>
```

For example,

```

declare relation map-points(ptnum,x,y,z)
  of class point
  ptnum : identifier
  x      : real
  y      : real
  z      : real
  with operations union,
                intersection,
                difference,
                select,
                point-project,
                point-insert,
                point-delete,
                join.

```

where the "point" prefixed operations are inherited from the class of points relations. Any operations specific to this instance of a points relation can be defined at this time.

- (3) A third option is to use parametrised integrity constraints (whose syntax would be similar to functions) and would be triggered by the invocation of an update, delete or insert function on the target relation. Such a system might use a symbol table as in Figure 2.

```

-----
| relation | constraint |
-----
| point   | delpt     |
| lines   | delln     |
...

```

Figure 7.2: Constraint symbol table

The constraints could be "activated" or "deactivated" by commands from the language which would install or

remove them from the table. Only those constraints that were in the table would be executed when a update statement was encountered and only those constraints pertinent to the relation would be activated.

7.2.2.1. Object Level Integrity

It is convenient to classify objects that might be stored in a graphical database into four groups : rigid, semi-rigid, scaleable and malleable.

- (1) A rigid object is any object that has no articulated parts and cannot change its size or shape. Therefore, only rotation and translation are permissible operations.
- (2) A semi-rigid object is one where there are a number of constituent parts which are rigid but are connected in some well defined way such that there is constrained relative motion between them (e.g. a hinged lid on a box). The transformations allowed on these objects must preserve the shape and size of the parts while allowing the parts to be moved relative to each other. (e.g. open the lid of the box). Possible articulations might include hinges, pins, sliding fit. Another form of connecting two objects is by gluing. Clearly operations must preserve the juxtaposition.

- (3) A scaleable object can be scaled but only by equal amounts in all axes. Thus a sphere will still be spherical under all allowable transforms. Rotations and translations are allowed.
- (4) A malleable object can be conceived to be made of rubber that can be deformed in any way. Thus any transformation is permissible and such objects would be used as templates for constructing other objects by suitable use of transformations. (e.g. a standard cube could be used as the template for all regular parallelepipeds which can be derived by selective scaling operations.)

The objects relation will have one attribute that contains the type of object as shown in Figure 3.

Object

onum	oname	material	type	...
o1	nut	steel	rigid	
o2	screw	steel	malleable	

Figure 7.3: Object relation

These ideas have scope for further development in the light of the GDB model proposed in this thesis.

7.2.3. Triangulation

The use of the primitives "facet" and "tetron" for constructing objects that are to be stored and manipulated in the Graphical Database raises the question of how these primitives are created. It would be inconceivable to expect the user to construct an object using these primitives as they are not the usual intuitive method of describing objects. There must be some automatic way of transforming the user's input (probably in terms of the boundaries of the object) into the necessary primitives.

Such methods have been developed for use in the field of Finite Element Analysis (FEA - a technique for the analysis of stress on objects by decomposing the surface into a finite number of elements which can be analysed individually and the results summed to give a model of the performance of the whole surface). These primarily produce triangular elements for 2-D analysis and tetrahedra for 3-D analysis although other polygons and polyhedra can be generated.

The different algorithms for FEA are reviewed in a recent paper by Ho-Le [59] who classifies the methods by element type, element shape, mesh density control and efficiency.

One important difference between the algorithms for FEA

is the requirement for small regularly shaped polygons (i.e. the elimination of long thin triangles - all angles should be approximately equal). This is not a constraint for the GDB where the requirement is simply for the least number of primitives to accurately describe the object regardless of its topology provided they are triangular (or tetrahedral if solid objects are required).

Thus it should be possible to develop a simplified algorithm based upon those used for FEA ignoring the testing and processing used to eliminate poorly shaped elements.

7.2.4. Extension to n-Dimensions

The extension of this system to more than three dimensions is facilitated by the fact that the primitives are all based upon points. There are two situations to consider; 3-D primitives in a n-D world and n-D primitives in an n-D world.

It is intuitively obvious that one does not need primitives that extend in all the possible dimensions to describe and object exists in those dimensions e.g. a 3-D object (cube) may be represented as a collection of 1-D primitives (lines). Thus we can postulate a 4-D world represented by 1-, 2- or 3-D primitives with the points relation extended to include values for the fourth dimension.

This notion is easily extended to n-D by increasing the degree of the points relation to n+1 while retaining the set of primitives as described for the 3-D world.

To extend the model to cope with n-D primitives then suitable primitives must be found and expressed as relations containing references to the expanded points relations.

The graphical transformations are just as easily handled by increasing the number of rows and columns in the transformation matrices to maintain them both at n+1 for a n-D world.

It may be argued that this extension is of little practical value as it does not describe the real world as we perceive it but nevertheless, multiple dimensions are mathematically possible and find application in the field of sub-atomic physics (super-strings). The model proposed here is capable of storing and manipulating data describing such a situation.

7.2.5. Data Input

The input of data to a database system as described here is not straightforward. A number of possibilities exist at the present time. The most awkward method is to type the coordinate data into the system manually. Clearly this is of limited use when trying to construct a new object interac-

tively, although some facility for reading data files will be required to enable data from other systems to be incorporated into the graphical database.

The use of a digitising table is acceptable for the copying of 2-D objects such as maps and drawings. A suitable system could be devised to accept engineering drawings (three view) and create the 3-D coordinate data from those. Three dimensional digitising is possible for models of objects but clearly has limitations.

To allow the user to design some object, a suitable graphical drawing package is required but the subsequent translation of a pixel based image into precise 3-D coordinate data is not easy although some recent research has produced a scheme for providing such data from perspective views of objects.

In addition to the problems with supplying the data in a suitable format the sequence of operations required to establish those data in the requisite relations must be considered. To input a new object into the Graphical Database requires a cascade of functions to achieve the insertion of tuples that maintains the model as described above.

For example, to insert a new object called "bracket" into the objects relation, not only must a new tuple be created in that relation with an unique identifier (value

for the "onum" attribute), but new tuples must be created in the link and primitives relations as well as the points relation. So if "bracket" is based on tetrons then the tetron-link and tetron relations must be updated to reflect the existence of the new object.

The sequence of operations is important if integrity constraints are enforced for each stage of this operation and is typically :

- (1) Insert set of points into Points relation.
- (2) Insert tetron data into Tetron relation.
- (3) Insert tuples into Tetron-link relation.
- (4) Insert new tuple into Objects relation.

As noted above, the collection of point data is not trivial and may require 3-D coordinate values to be typed in by hand. The generation of tetrons can be performed by existing mesh generation techniques from the points data. Once all the data is available the other steps are automatic.

7.2.6. Deletions

To delete an object from the Graphical Database requires a similar scheme of operations to those described in the previous section except that the order is reversed.

Also it is possible to utilise the normal relational set operations to perform the deletion.

To delete the object bracket requires the following :

- (1) Delete tuple from Object relation.
- (2) Delete tuples from Tetron-link relation
- (3) Delete tuples from Tetrons relation.
- (4) Delete tuples from Points relation.

A complication can arise if the sharing of primitives is allowed. This would be adopted to reduce the data stored by removing redundant tuples which referenced the same primitive in more than one object. Thus after the creation of a new object the database can be checked for redundant tuples and the references from each object rationalised to reduce the storage overhead.

For instance if a two objects both used the line from (0,0,0) to (2,3,4) they would initially each have a reference to this line called "L23" and "L77" respectively. Each line would reference the same coordinates but they would be called differently (e.g. pt267,pt268 and pt299,pt300). Thus the lines relation hold one extra tuple and the points relation two extra tuples. To rationalise this situation the line would be represented only once as "L23" and the link-relations for the two objects would be updated accordingly.

7.2.7. Object Oriented models

As indicated above, the use of object classes (rigid, semi-rigid etc.) suggests investigation into more object oriented database models following the ideas of SMALLTALK-80 where objects have precisely defined "interfaces" (permissible queries) and a hierarchical structure. Work in this field has already been published with reference to ordinary database applications [60,61].

7.2.8. Temporal models

There is application in animated films for the use of a graphical database to store the data for the construction of "frames" that would then be transferred to film for display purposes. The sequence of frames is of importance so there must be some time attributes stored with the other data.

For example, consider the making of a film concerning an archaeological site showing the changes in the boundaries and buildings over a period of years. The data for each "frame" will be mostly the same with only a few changes. The database would hold data (e.g. lines) that would have a time attribute attached that describes the time span of that line. The "frame" can be constructed by selecting only those lines that are valid for a certain time period.

The work for temporal queries has already been imple-

mented in DEAL [57,62] and the combination of temporal and graphical queries has yet to be investigated.

CHAPTER 8

Summary

Having established the need for integrated graphics databases that can incorporate data from traditional commercial style databases, this thesis shows that the published literature offers no global solution to this problem.

The work described here establishes a model for a graphics database based on a boundary representation using points, lines, facets, and tetrons as primitives to describe objects. This representation can be stored in a relational database along with the usual character based data. User defined primitives can also be defined together with the necessary functions for their manipulation.

A formal specification of this system is given using Standard ML as a specification language and the usefulness of specification prior to implementation is demonstrated by the initial C language version of the database and by the easy translation of ML functions into DEAL functions and views.

A language to manipulate and display the objects held in the database is described and the power of the language and its suitability for answering complex queries has been demonstrated. (No absolute method exists to quantify the

expressibility of a language so the suitability of DEAL, as described here, is a matter of judgement on the part of the author.) The ideas expounded here have been explored in several implementations of relational database management systems.

The problems of complex object hierarchies and integrity constraints are discussed and possible strategies are suggested. The use of object oriented models and the extension of the model to more than three dimensions is also discussed. The problems associated with data input and deletion are highlighted as areas for further work.

APPENDIX A

Introduction to Computer Graphics

Computer graphics is the display, on a suitable device, of the geometric data describing an object. The sophistication of the image formed is dependent on the data and computer power available.

The subject can be split into two separate parts; the geometric description of an object and the display of that object.

The geometric description of an object consists of a set of boundary and surface parameters expressed in a coordinate system. Of the possible coordinate systems the cartesian system with its three mutually perpendicular axes is the most commonly used. The three axes are usually labelled x , y and z , and any point in this space can be described precisely by an ordered triple of numbers representing distances along each axis from a designated origin where the three axes meet.

Edges can be expressed as line segments joining two points. Thus an object can be described by a list of edges which denote its boundary. This can be extended to describe a solid as a list of polygonal surfaces (or indeed a list of volume elements).

There are a variety of graphical output devices including dot matrix printers, plotters, bit mapped and vector graphics CRTs.

To display a 2-D object on such devices all that is required is a mapping to be applied to the list of points describing the object which converts the values to the range of values addressable on the device.

For the display of a 3-D object there is more computation required as the mapping must now remove the data from the unwanted third dimension. This could be performed by simply ignoring one member of the triplet (z) and plotting the x and y values as before by using the mapping described above. This leads to a parallel projection of the object which gives an unsatisfying image on the output device. This is due to the nature of human vision which gives a perspective view of a scene i.e. an object viewed from a short distance will appear larger than if viewed from a longer distance and parallel lines appear to converge as the distance from the viewer increases. This type of transformation is easily applied to the 3-D data, and involves dividing the x and y values by the depth value (z) and then multiplying them by a factor which is a ratio of the screen size and the optimum distance of the screen from the eye.

To represent the required mappings and transformations concisely, a matrix is commonly used. A 3-D point can be

represented by a row vector (x y z h) and the transformation by a 4-by-4 matrix. The extra row and column are required to express the 3-D point in a homogeneous coordinate system which allows all the transformations to be applied by the multiplication of the vector by the matrix (otherwise translations would be performed by addition of matrices while the others - rotation and scaling - would use multiplication).

The translation matrix is

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{matrix}$$

where T_x , T_y , T_z are the translation factors for each axis.

The scaling matrix is

$$\begin{matrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

where S_x , S_y , S_z are the scaling factors for each axis.

The x-axis rotation matrix is

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & R_1 & R_2 & 0 \\ 0 & R_3 & R_4 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

where R1, R2, R3, and R4 are the rotation factors.

R1 = cos(a)
R2 = -sin(a)
R3 = sin(a)
R4 = cos(a)
a = angle of rotation around the x-axis
measured in an anti-clockwise direction
when viewed from the origin.

The y-axis rotation matrix is

R1 0 R2 0
0 1 0 0
R3 0 R4 0
0 0 0 1

where R1, R2, R3, and R4 are the rotation factors.

R1 = cos(a)
R2 = sin(a)
R3 = -sin(a)
R4 = cos(a)
a = angle of rotation around the y-axis
measured in an anti-clockwise direction
when viewed from the origin.

The z-axis rotation matrix is

R1 R2 0 0
R3 R4 0 0
0 0 1 0
0 0 0 1

where R1, R2, R3, and R4 are the rotation factors.

R1 = cos(a)
R2 = -sin(a)
R3 = sin(a)
R4 = cos(a)
a = angle of rotation around the z-axis
measured in an anti-clockwise direction
when viewed from the origin.

In addition to these operations, account must be taken of the different coordinate systems of the representation and the output devices. The established approach (CORE, GKS) is to perform several transformations to convert from the "world coordinates" used to specify the objects to the "device coordinates" used by the display hardware. This is achieved by creating an intermediate "normalised device" coordinate (NDC) system.

The NDC space is defined as 0 to 1 in both vertical and horizontal axes. The normalisation transformation maps a predefined window in world coordinates onto a viewport in the NDC space (windows and viewports are rectangular regions of 2-D space). The workstation transformation maps a NDC window onto a display device ("workstation") viewport. In this way several objects can be mapped onto the NDC space and the whole space mapped onto the workstation thus allowing the output to be "composed" on the NDC device before being displayed.

APPENDIX B

The Specification Language ML

There are several publications by the research group who devised Standard ML [63,64] which are available from Edinburgh University's Laboratory for the Foundation of Computer Science. In addition there is an excellent introduction to the language by Wikstrom [65] which sadly does not go into the more difficult area of modules.

ML is a strongly typed functional programming language which is similar in ethos to HOPE [53] and, although not directly descended from HOPE, they do share similar constructs. Functional programming languages in general can be contrasted with the more conventional languages (e.g. FORTRAN, PASCAL, COBOL etc.) in that they are not procedural. That is, the programmer does not specify the procedure to adopt to solve a particular problem. A functional solution comprises of a collection of functions which will each solve a small part of the problem and can be combined into a single function call that will provide a solution (or a partial solution) to the problem.

Such languages are not particularly efficient when compared with procedural languages (although the advent of parallel machine architectures may enable them to become

viable alternatives) but they enable the programmer to specify the solution to a problem in a form that can more easily be proven to be correct with respect to the original specification. Thus they can be used as a formal specification which, when proven correct, can be transformed into a procedural language by clearly defined steps that preserve its correctness. Thus a procedural solution can be produced that will be less error prone than that produced by conventional methods.

SML has been used in this thesis as a formal specification tool with which ideas could be explored quickly and with little programming effort. The specification includes a relational database specification together with a set of relational algebra functions sufficient to demonstrate the features of the graphical database. It was thus possible to provide a functional specification of the new relational expressions required for this work.

The facilities offered by Standard ML are many (including some imperative constructs such as while loops and case structures) but the specification detailed in this thesis is restricted to a subset of the language just sufficient for the purpose. A fuller treatment of the facilities offered can be found in the references quoted above.

A ML program can be considered as a collection of data-

types and a collection of functions which represent the functionality of the problem domain. So it looks schematically like :

```
datatype < name > = < constructors >
fun < name > < parameter list > = < body >
fun < name > < parameter list > = < body >
fun < name > < parameter list > = < body >
...
```

The datatype structure allows the user to define new types of his choosing and add these to the in-built ones (e.g. int, bool, real, list etc.). For instance :-

```
datatype queue = empty | enqueue of (alpha * queue);
fun add_to_end ( element : alpha, q : queue ) =
    enqueue(element,q) ;
fun service_q ( enqueue(e,q) ) = ( e , q ) ;
fun is_empty(empty) = true
    | is_empty(enqueue(e,q)) = false ;
```

The constructors for the type "queue" are "empty" and "enqueue" (the "|" is pronounced "or" and separates the possible constructors). This means that members of the type queue can only be built from these keywords. Three functions are shown

- (1) `add_to_end` : adds new elements to the queue
- (2) `service_q` : to get the head of the queue
- (3) `is_empty` : tests for the occurrence of an empty queue and returns one of the in-built boolean values true or false.

The latter function demonstrates the pattern matching capability where two axioms (separated by "|") are given to cope with the two possible constructors. This feature allows the programmer to provide axioms for all the possible occurrences of a datatype by considering all the constructors in turn. Thus, an application that requires the structure of a queue can use these three functions to implement a queue.

The body of a function can have a variety of constructs which are illustrated in the examples below.

A specification for the operation of a supermarket might include a function that describes the function of the check-outs.

```
fun shopping(q : queue) = if is_empty(q)
    then shut_down
    else let val ( s, q1 ) = service(q) in
        (check_out(s),shopping(q1))
    end ;
```

The two possible courses of action are programmed as the

branches of an if-then-else construct depending on whether the queue is empty or not. (Note that in this example pattern matching on the constructors of a queue is a possible alternative). The functions "check_out" and "shut_down" will have been defined beforehand to perform the required actions (In the strongly typed environment of ML the result types of both these functions must be the same). The "let-val-in-end" construct is also shown here that allows the separate parts of the result of function "service" to be obtained by pattern matching.

Constant values (i.e. data) can be defined as functions (possibly parameterised) or as "variables".

```
fun data = ( item1, item2, item3, ... )  
val data = ( item1, item2, item3, ... )
```

The effect of both options is to establish data that can be used as parameters to function calls.

The Standard ML language also contains "abstypes", which are abstract data types with defined public interfaces and "structures", which embody object oriented ideas via imprecise public interfaces ("signatures") and modifying functions ("functors"). These constructs were not utilised in the specification of the GDB model and are, therefore, not described here. The Standard ML manual should be consulted for further information.

APPENDIX C

Specification Theory

In general terms the formal specification of a system is a concise, precise description of its structure and behaviour. It is essentially a model of the system that can be used to explore parts of the construction and its behaviour without the penalty of investing many hours of effort in coding in a procedural language. The modelling of any real world system by some formal specification method must be preceded by the modeller making a decision about the level of abstraction required to achieve the task in hand. At some stage the specification will be transformed into an implementation in an appropriate procedural language, so decisions must be made then concerning low level implementation details (such as optimisation, I/O devices), but initially the specification can be written independently of such issues. This high level approach is one of the major benefits of formal specification methods as they free the modeller from low level problems which can be tackled later after the major behaviour of the system as a whole has been modelled to the desired precision.

Much of the pioneering work on algebraic specification systems was performed by Guttag and his co-workers [66,67,68,69] and the so-called ADJ group [70] who

have established the mathematical basis of specification languages and have demonstrated the benefits of such formalisms. The advantages of formal specification have also been investigated by Duce and Fielding [71], who detail its benefits with respect to the chosen application.

Other major workers in this field are Liskov and Zilles who give a clear definition of abstract data types as

... a class of abstract objects which is completely characterised by the operations available on those objects. ... When a programmer makes use of an abstract data object, he is concerned only with the behaviour which that object exhibits, but not with any details of how that behaviour is achieved by means of an implementation.

The basis of most specification methods is some form of abstraction together with modularity. The former allows complex systems to be described in a simple way by hiding unnecessary details. This method of "black boxing" a complex subject is natural in everyday language and has been used to great effect in algebraic specification techniques using "abstract data types". The use of modularity has long been recognised as a useful technique for tackling large problems by breaking them down into more manageable sub-tasks.

The objective of using a formal specification is to produce a good software product quickly with the minimum of errors undetected.

The production of a correct program is a difficult concept to quantify, however, as it is not possible to determine whether a program is completely correct. The only definite quantification is "if a program is used without error for all of its life then it is probably correct". This is still not absolute because there may be paths through the code which were never taken. This definition is, of course, of little use when designing software as it demands hindsight. What is really required is a method that will predict the correctness of a program before it is used. This is where the current research into formal specification methods is leading.

It is a commonly held belief that a formal specification of a system can be written quickly and modified easily as well as being easy to understand. Thus a "correct" specification can be attained more quickly than a "correct" program. The testing of a specification can be achieved more easily if the formalism used allows the specification to be executable (i.e. the specification can be "run" like a program with test data to ensure it's syntactic and, in some part, its semantic correctness).

The choice of specification formalism is not an easy one as each different method has its drawbacks. Executable formalisms are useful (while admitting that some features of a system cannot be modelled) because of the increased confidence in the resultant specification. An executable specification method is thus desirable if the benefits of execution outweighs the lack of modelling abilities.

Examples of constructional (non-executable) specification languages are the Vienna Development Method (VDM) and Z which are both model based methods. They are built up of data objects representing the inputs, outputs and states of the system together with operators which manipulate these objects. Essentially, these methods consist of a series of statements about the system which are specified in terms of predicates about the data objects.

Examples of the different styles of specification are shown below where each formalism is used to specify a graphics system that consists of pictures (i.e. a screen) and two operations (add a line and delete a line).

First the VDM approach which defines the state as the picture with the two operations defined in terms of pre- and post-conditions.


```

State :: picture : line set
       line = (p1,p2) where p1 and p2 are points

addline ( L : line )
Ext     picture : line set

Pre     p1 <> p2 where L = ( p1,p2)

Post    picture' = union(picture, L)

del-line ( L : line )
Ext     picture : line set

Pre     member(L,picture)

Post    picture' = picture - L

```

Algebraic specification languages such as Standard ML [63] and OBJ [72], both of which are executable, (although not all algebraic methods are) can be used to check the syntax of the specification as well as form the basis of a working prototype of the system. These methods establish a set of axioms which define the relationship between operators on the data objects rather than defining a model.

Firstly the ML version.

```

datatype picture = nopic |
                 mkpic of ( line * picture );

fun addline(L,p) = if ok(L) then mkpic(L,p) else p;

fun del-line(L :line,nopic :picture):picture = nopic
  | del-line ( L, mkpic(L1,p)) = if equal (L,L1)
    then p else
      mkpic ( L1,del-line(L,p)) ;

```

The OBJ version can be seen to be similar to the ML version but is not as compact.

obj Picture

sorts picture

ops

```
nopic   :                -> picture
mkpic   : line picture -> picture
addline : line picture -> picture
del-line: line picture -> picture
```

vars

```
  L      : line
  pt,pt1 : point
  p,p1   : picture
```

eqns

```
(addline(L,nopic) = mkpic(L,nopic))
(addline(L,mkpic(L1,p)) = mkpic(L,mkpic(L1,p)))
(del-line(L,nopic) = nopic)
(del-line(L,mkpic(L1,p)) = p if equal(L,L1))
(del-line(L,mkpic(L1,p)) = mkpic(L1,del-line(L,p1))
                           if not(equal(L,L1)))
```

jbo

A comparison of the VDM and OBJ formalisms was made by Duce and Fielding [73] who found both methods useful in that VDM encourages a more top-down approach, while OBJ (and algebraic methods in general) promotes a more bottom-up approach to problem solving. They view the ability to execute a specification as of great benefit.

APPENDIX D

Correctness Proof of SML Function from its Axiom

As noted in Chapter 4 a SML specification can only be shown to be consistent by execution, a proof can only be made against a more abstract specification. In this case the specification is an implementation of axioms expressed in a predicate calculus.

The function "union" is proved by showing that the implementation in SML supports assertions made in the axiom. The procedure is to take each side of the axiom and show that the implementation supports the implication of the other side of the axiom.

The following proof uses induction to show that the axioms can be shown to hold for the SML equations in each of the cases which together constitute all the possible cases.

The parts of the implementation are numbered for ease of reference in the following proof.

Equations used in SML implementation :

```

union(mt,r2) = r2                                {1}
union(tcons(h,r1),r2) = union(r1,r2)   if member(h,r2)  {2}
                        or tcons(h,union(r1,r2))   if not member(h,r2) {3}

member(t,mt) = false                            {4}
member(t,tcons(h,r)) = true                    if t = h      {5}
                        or member(t,r)          if t < h     {6}

```

Axiom :

```
member(t,r1) V member(t,r2) <=> member(t,union(r1,r2))
```

Proof that axiom can be deduced from the equations :

Show first

```
member(t,r1) V member(t,r2) => member(t,union(r1,r2))    {7}
```

** Case 1 r1 = mt

```

member(t,mt) V member(t,r2) = false V member(t,r2)      From {4}
                             = member(t,r2)
                             = member(t,union(r1,r2))    From {1}

```

** Case 2 r1 < h

Using induction, assume without loss of generality, that {7} holds for r1 with n members and show that for tcons(h,r1) with n+1 members the following holds.

```

member(t,tcons(h,r1)) V member(t,r2) =>
member(t,union(tcons(h,r1),r2))      {8}

```

* Case 2.1.1 t = h and member(h,r2)

```

member(t,tcons(h,r1)) V member(t,r2) =
true V member(t,r2) From {5}
= member(t,r1) V member(t,r2)    {9}

```

```

member(t,union(tcons(h,r1),r2)) = member(t,union(r1,r2))    {10}
From {2}

```

Substituting into {4} from {9} and {10} gives

```

member(t,tcons(h,r1)) V member(t,r2) =>
member(t,union(tcons(h,r1),r2))

```

Hence {8} is true by induction in this case.

* Case 2.1.2 $t = h$ and $\text{not member}(h, r2)$
 $\text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) = \text{true} \vee \text{member}(t, r2)$ From (5)
 $\text{member}(t, \text{union}(\text{tcons}(h, r1), r2)) =$
 $\text{member}(t, \text{tcons}(h, \text{union}(r1, r2)))$ From (3)
 $= \text{true}$ From (5) &
case assumption

and so (8) holds in this case.

* Case 2.2.1 $t \triangleleft h$ and $\text{member}(h, r2)$
 $\text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) =$
 $\text{member}(t, r1) \vee \text{member}(t, r2)$ (11) From (6)
 $\text{member}(t, \text{union}(\text{tcons}(h, r1), r2)) =$
 $\text{member}(t, \text{union}(r1, r2))$ (12) From (2)

Substituting from (11) and (12) into (7) gives
 $\text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) \Rightarrow$
 $\text{member}(t, \text{union}(\text{tcons}(h, r1), r2))$

hence (8) by induction in this case.

* Case 2.2.2 $t \triangleleft h$ and $\text{not member}(h, r2)$
 $\text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) =$
 $\text{member}(t, r1) \vee \text{member}(t, r2)$ (13) From (6)
 $\text{member}(t, \text{union}(\text{tcons}(h, r1), r2)) =$
 $\text{member}(t, \text{tcons}(h, \text{union}(r1, r2)))$ From (3)
 $= \text{member}(t, \text{union}(r1, r2))$ (14)
From (6) &
case assumption

Substituting from (13) and (14) into (7) gives
 $\text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) \Rightarrow$
 $\text{member}(t, \text{union}(\text{tcons}(h, r1), r2))$

hence (8) by induction in this case.

So (8) is true in all cases.

Secondly show

$\text{member}(t, \text{union}(r1, r2)) \Rightarrow \text{member}(t, r1) \vee \text{member}(t, r2)$ (15)

** Case 1 $r1 = mt$

$\text{member}(t, \text{union}(mt, r2)) = \text{member}(t, r2)$ From (1)
 $= \text{member}(t, r1) \vee \text{member}(t, r2)$
From def'n
of 'OR'

therefore (15) holds in this case.

** Case 2 $r1 \triangleleft mt$

Using induction, assume without loss of generality, that {15} holds for $r1$ with n members and show that for $tcons(h,r1)$ with $n+1$ members the following holds.

$member(t, union(tcons(h,r1), r2)) \Rightarrow$
 $member(t, tcons(h,r1)) \vee member(t, r2)$ {16}

* Case 2.1.1 $t = h$ and $member(h,r2)$

$member(t, union(tcons(h,r1), r2)) =$
 $member(t, union(r1,r2))$ From {2}
 $= true$ {17}
From inductive assumption

$member(t, tcons(h,r1)) \vee member(t, r2) = true \vee true$ {18}
From {5} &
case assumption

Substituting {17} and {18} into {16} gives
 $true = true \vee true$
hence {16} holds by induction in this case.

* Case 2.1.2 $t = h$ and not $member(h,r2)$

$member(t, union(tcons(h,r1), r2)) =$
 $member(t, tcons(h, union(r1,r2)))$ From {3}
 $= true$ {19} From {5} &
case assumption

$member(t, tcons(h,r1)) \vee member(t, r2) =$
 $true \vee true$ {20} From {5} &
case assumption

Substituting {19} and {20} into {16} gives
 $true = true \vee true$
hence {16} holds by induction in this case.

* Case 2.2.1 $t \triangleleft h$ and $member(h,r2)$

$member(t, union(tcons(h,r1), r2)) = member(t, union(r1,r2))$ {21}
From {2}

$member(t, tcons(h,r1)) \vee member(t, r2) =$
 $member(t, r1) \vee member(t, r2)$ {22}
From {6} &
case assumption

Substitute {21} and {22} into {16} gives {15}

$$\text{member}(t, \text{union}(r1, r2)) \Rightarrow \text{member}(t, r1) \vee \text{member}(t, r2) \quad \{15\}$$

hence {16} holds by induction.

* Case 2.2.2 $t \triangleleft h$ and not $\text{member}(h, r2)$

$$\begin{aligned} \text{member}(t, \text{union}(\text{tcons}(h, r1), r2)) &= \\ &\quad \text{member}(t, \text{tcons}(h, \text{union}(r1, r2))) \text{ From } \{3\} \\ &\quad = \text{member}(t, \text{union}(r1, r2)) \quad \{23\} \\ &\quad \quad \text{From } \{6\} \text{ \&} \\ &\quad \quad \text{case assumption} \end{aligned}$$

$$\begin{aligned} \text{member}(t, \text{tcons}(h, r1)) \vee \text{member}(t, r2) &= \\ &\quad \text{member}(t, r1) \vee \text{member}(t, r2) \quad \{24\} \\ &\quad \quad \text{From } \{6\} \text{ \&} \\ &\quad \quad \text{case assumption} \end{aligned}$$

Substitute {23} and {24} into {16} gives {15}

$$\text{member}(t, \text{union}(r1, r2)) \Rightarrow \text{member}(t, r1) \vee \text{member}(t, r2) \quad \{15\}$$

hence {16} holds by induction.

So {16} is true in all cases,
but {8} is also true in all cases hence

$$\text{member}(t, r1) \vee \text{member}(t, r2) \Leftrightarrow \text{member}(t, \text{union}(r1, r2))$$

(which is the original axiom). QED.

Thus it is shown by induction that the implementation satisfies the axiom.

The DEAL Query Language

DEAL (Deductive Algebra) [56] is a proposed extension of a relational language, capable of supporting user-defined functions, recursions and deductions based on the full first-order predicate calculus. DEAL was intended to provide a unified framework for all database processing - conventional and deductive.

In the design of DEAL, special attention was given to the orthogonality of its constructs. The advantage of orthogonality is that it leads to a coherent language, one which is simple, clean, and with a consistent structure. It is based on the belief that orthogonality must be the guiding principle in language design.

E.1. Relational Operations

The syntax of a DEAL query is

```
base-expression [attribute-spec]
where selection predicate
```

The base-expression can be a relational expression, or another DEAL query. As a relational expression, it may include any valid relational operations. The "attribute-spec" is a list of attributes that will appear in the result relation (c.f. project) and the "selection predicate"

determines which tuples will appear in the result relation (c.f. select).

The principal operations allowed within a base-expression are

Cartesian product (**), Union (++)
Outer union (+?), and Difference (--)

The definitions of these relational operations is given in [74].

Figure-E.1 gives the main part of the DEAL syntax. The complete syntax of DEAL is given in Appendix F.

```
query      ::= query_expr
           | func-defs

query-expr ::= query-block
           | query-expr set-op query-block

query-block ::= rel-expr
            [ '[' selection-list ']' ]
            [ where condition ]

rel-expr   ::= relation-name
           | ( query-expr )
```

Figure E.1: An Overview Syntax of DEAL

The syntax of DEAL compared with SQL is illustrated below.

DEAL	SQL
Emp[name,city] where enum > 2	select name,city from Emp where enum > 2
Emp	select * from Emp
Emp where city = 'london'	select * from Emp where city = 'london'

DEAL, unlike SQL, supports nesting at the external or user's level, for example the DEAL query

```
(Emp [enum,name,salary] where city = 'London')
where salary > 50k;
```

can not be directly translated into a SQL query without defining a view (for London employees first). The advantage of DEAL, in this instance, is that queries are expressed in a natural way and are therefore easier to understand.

A query to find the employee names and their department names for departments in Paris may be written in DEAL as:

```
(Emp (edno , dno) Dept) [name,dname]
where loc = 'Paris';
```

The base-expression contains a join of Emp and Dept over the common domains edno and dno.

E.1.1. Retrieval involving a subquery

As an example, consider Dates well-known supplier-parts-shipment database [75].

The following example queries show the use of subqueries which are not permitted in SQL.

- (1) Get supplier numbers for suppliers who are located in the same city as supplier s1.

```
supplier [snum]
where city =
    ( supplier [city]
      where snum = s1 );
```

- (2) Get suppliers with above average status.

```
supplier
where status >
    ( average( supplier [status]) );
```

E.1.2. Query involving Set operations

Get part numbers for parts that either weigh more than 16 pounds or are supplied by supplier s2 (or both).

```
(parts [pnum] where weight > 16)
++
(shipment [pnum] where snum = s2);
```

E.1.3. Views

Views in DEAL are represented by function-like constructs (see the following section for more detailed discussion of functions). Like functions, views are not executed

when they are created but they are merely stored in the system. For example, london suppliers could be defined as a view as follows:

```
view london_supplier() as {  
london_supplier := supplier where city = 'london';  
}
```

where '(' and ')' are used to indicate the beginning and end of the function body.

Unlike SQL, any relational expression is permitted in the definition of a view in DEAL.

E.2. Functions and Recursion

The power of any query language is considerably enhanced by allowing constructs for looping and conditional execution. In the past this was achieved by embedding the query language into a high level programming language such as PL1 or PASCAL. In DEAL however these constructs and some others such as assignments and print statements are incorporated in the query language. Functions may also be defined, allowing recursive queries to be expressed. Given the built-in function "first" (first is view which takes a relation and returns a relation with a single tuple from the original relation) we can define the function "rest" as follows:

```

view rest(x:rel) {
    rest := x -- first(x) ;
}

```

Other complicated functions can also be defined with the help of built-in view "first" and the function "empty". For example there follows two versions of the aggregate function max. The first version is iterative and the second version is recursive.

```

func max(x:rel) {
    MAX := 0;
    if (empty(x)) max := 0;
    while ( not(empty(x)) )
        f := first(x);
        if (#f > MAX) {
            MAX := #f;
        }
        x := x -- f;
    }
    max := MAX;
}

```

is used to cast a relation with one tuple and one attribute to its value.

```

func max(x:rel) {
    if (empty(x)) {
        max := 0;
    } else {
        if (#first(x) > max(rest(x))) {
            max := #first(x);
        } else max := max(rest(x));
    }
}

```

For example the query "find employees whose salary is greater than max salary of associate profs." is

```
employee where salary > max(employee [salary]
    where position = 'associate');
```

Other examples which demonstrates how DEAL can be used for answering queries which cannot be handled by conventional query languages are illustrated below.

E.2.1. Ancestor Problem

Given the relation parents, we can retrieve the paternal ancestors of a given person by using the following two functions which use recursion.

```
parents(dad,mum,pname)

      dad      mum      pname
derek  jane     joe
fred   mary     jim
greg   alison   jane
jim    beth     derek
john   joan     beth
```

It is assumed that people's names are unique.

```
view father(x:char) as {
    father := parents [dad] where pname = x;
}

view ancestor(x:char) as {
    if (empty(father(x))) ancestor := null; else {
        ancestor := father(x) ++ ancestor(father(x));
    }
}
```

"null" is a built-in relation with no tuples and arbitrary columns.

The call ancestor('joe') will then return a relation with three tuples which are paternal ancestors of joe i.e. derek, fred and jim.

E.2.2. Parts Explosion Problem

The parts explosion problem, which arises quite frequently in a practical context, is well known as a problem that is beyond the capabilities of classical relational algebra. It is also well known that it is impossible to formulate a parts explosion as a single expression in the relational algebra or relational calculus. The current relational query languages such as SQL or QUEL therefore can not handle this problem without some comparatively major extensions [76].

In DEAL however, the parts explosion problem can be solved using a set of user-defined functions which use recursion.

The following two relations parts and link are assumed:

parts (pnum,cost)

link(supp,inf,qty)

A given part may contain any number of other parts as immediate components and may itself be an immediate component of any number of other parts. In other words, relation link represents a many-to-many relationship between

parts and parts.

Figure-E.2 is an extension of relations parts and link.

parts	
pnum	cost
1	10
2	15
3	20
4	12
5	8
6	15
7	20
8	30

link		
sup	inf	qty
1	2	2
1	4	5
1	7	5
2	4	3
3	6	3
4	7	5
5	3	1
5	6	8
6	1	9

Figure E.2: Part Explosion Database

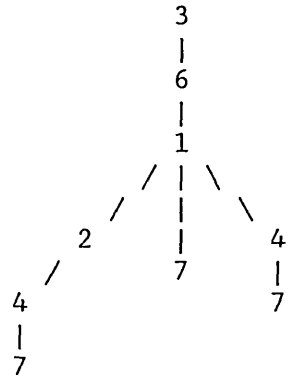
given the above relations, the parts explosion problem can now be stated as follows:

- Find the total cost of some given part to all levels

Or equivalently:

- Produce the bill of material for any given part.

The relation 'link' can be regarded as a collection of trees, for example part number 3 could be represented as:



To produce the total cost of a given part, one would evaluate each branch of the tree completely and sum the cost over all branches.

The DEAL functions used to perform the task are as follows:

```

view cost_of(pno:int) as {
    cost_of := parts [cost] where pnum = pno
}

view allparts(pno:int,z:rel) as {
    allparts := z where sup = pno;
}

view rest(pno:int,m:rel) as {
    rest := m -- first((allparts(pno,m)));
}

view quant(pno:int,z:rel) as {
    quant := first((z [qty] where sup = pno));
}

view infer(pno:int,z:rel) as {
    infer := first((z [inf] where sup = pno));
}

func link_cost(pno:int,z:rel) {
    if (empty(allparts(pno,z)) ) {
        link_cost := cost_of(pno);
    } else {
        link_cost := quant(pno,z) *
            link_cost(infer(pno,z),z) +
            link_cost(pno,(rest(pno,z)));
    }
}

func total_cost(pno:int) {
    total_cost := link_cost(pno,link);
}

```

In [76] two approaches are given based on extending a relational language such as SQL or QUEL to deal with the parts explosion problem. These highlight the inadequacy of current programming languages to deal with the parts explosion problem.

The second approach gives a series of changes to SQL to enable it to handle the "tree" type problems and the proposals include user defined functions and increased functionality. DEAL implements these ideas and the functions

quoted above illustrate the validity of these ideas.

APPENDIX F

Complete BNF for DEAL

The following is a complete BNF for the DEAL language which supports both historical [57] and graphical queries. The lines marked with a '{G}' represent those productions that are required for graphical operations. The lines marked '{*}' indicate the changes required to enable more complex functions and views to be constructed.

```

input ::= input ';'
      | input generals ';'
      | input defn ';'
      | input ddl ';'
      | input dml ';'
      | input query ';'
      | input function ';'
      | input error ';'

query ::= query_expr
      | let_in

query_expr ::= query_block
            | query_expr WHEN simple_t_pred
            | query_expr DIFF query_expr
            | query_expr INTER query_expr
            | query_expr UNION query_expr
            | query_expr CART query_expr
            | query_expr join query_expr
            | query_expr MM query_expr {G}
            | '(' query_expr ')'
            | '(' query_expr ')' block
            | '(' query_expr ')' rename

simple_t_pred ::=
            BEFORE expr
            | AFTER expr
            | MEETS expr
            | OVERLAP expr
            | STARTS expr
            | SAME expr
            | FINISHES expr
            | CONTAINS expr
            | BETWEEN expr
            | LONGER expr
            | SHORTER expr
            | EQUAL expr

query_block ::= rel_name
            | rel_name block
            | query_block rename
            | FIRST_GROUP query_block BY
              '[' selection_list ']'

block ::= '[' selection_list ']'
      | WHERE condition
      | '[' selection_list ']'
        WHERE condition

rename ::= RENAME '[' rename_list ']'

```

```

rename_list ::= const BEQ attr_name
              | rename_list ',' const BEQ attr_name

join        ::= '(' attr_name ',' attr_name ')'

asgn        ::= as_name BEQ expr
              | ARG BEQ expr

as_name     ::= VAR
              | L_VAR
              | NUMBER
              | RELATION

stmt        ::= procname BEQ expr
              | dml          {*}
              | generals
              | defn
              | asgn
              | PRINT prlist
              | while cond stmt end
              | if cond stmt end
              | if cond stmt end ELSE stmt end
              | '(' stmtlist ')'

cond        ::= '(' predicate ')'

if          ::= IF

while       ::= WHILE

end         ::= /* nothing */

begin       ::= /* nothing */

stmtlist    ::= stmtlist ';'
              | stmtlist stmt

prlist      ::= list_item
              | prlist ',' list_item

list_item   ::= QSTRING
              | ARG
              | L_VAR
              | DOUBLE

selection_list ::=
              expr
              | selection_list ',' expr

condition   ::= predicate
              | condition AND predicate

```

```

defn ::= FUNC funcname '('paramlist')' stmt
      | VIEW funcname '('paramlist')' AS stmt

paramlist ::= param ':' declaration
           | paramlist ',' param ':' declaration

param ::= VAR
       | ARG

declaration ::= INT
            | DBLE
            | REL
            | AT
            | CHAR

header ::= '(' arglist ')'

arglist ::= expr
         | arglist ',' expr

procname ::= FUNCTION
         | VIEWDEF

funcname ::= VAR
         | FUNCTION
         | VIEWDEF

predicate ::= expr
          | predicate GT predicate
          | predicate GE predicate
          | predicate LT predicate
          | predicate LE predicate
          | predicate NE predicate
          | predicate EQ predicate
          | predicate C_AND predicate
          | predicate C_OR predicate

expr ::= arith_term
      | expr '+' arith_term
      | expr '-' arith_term

arith_term ::= arith_factor
           | arith_term '*' arith_factor
           | arith_term '/' arith_factor

arith_factor ::=
           primary
           | '-' primary %prec UNARYMINUS

primary ::= const
        | field_spec
        | function

```

```

        |      query_expr
        |      '#' primary
        |      '(' expr ')'

let_in ::= LET as_name EQEQ expr
        IN query

const ::= NUMBER
        |      DBLE           {G}
        |      DOUBLE        {G}
        |      QSTRING
        |      time_point
        |      NOW
        |      time_interval

time_point ::= NUMBER '/' NUMBER '/' NUMBER

time_interval ::=
        '[' expr ',' expr ']'
        |      MONTH NUMBER
        |      MONTH
        |      ':' NUMBER

function ::= FUNCTION begin '(' arglist ')'
        |      BLTINFUNC '(' arglist ')'
        |      DBLTINFUNC '(' arglist ')' {G}

field_spec ::= attr_name

dml ::= INSERT INTO rel_name
        VALUES '('list_tup')'

list_tup ::= tup
        |      list_tup ',' tup

tup ::= QSTRING
        |      ARG
        |      L_VAR
        |      BLTINFUNC '(' arglist ')'
        |      DBLTINFUNC '(' arglist ')' {G}
        |      DOUBLE {G}
        |      NUMBER

ddl ::= create_table

create_table ::= CREATE TABLE VAR '(' f_d_l ')'

f_d_l ::= f_d
        |      f_d_l ',' f_d

```



```

f_d      ::=   attr_name atype
           |   attr_name atype  NONULL

atype    ::=   CHAR '(' NUMBER ')'
           |   REAL   {G}
           |   INT   '(' NUMBER ')'
           |   DATE

attr_name ::=  VAR

rel_name ::=  RELATION
           |   VIEWDEF begin header
           |   BLTINVIEW '(' expr ')'
           |   ARG
           |   L_VAR

generals ::=  TRACE ON
           |   TRACE OFF
           |   DISPLAY
           |   DISPLAY NUMBER
           |   COMMIT
           |   MAKEHIST '(' RELATION ')'
           |   SHELL
           |   CLOSE   {*}

```

This forms the basis of the grammar for the YACC parser generator.

APPENDIX G

Standard ML Specification of Matrix operations

The following Standard ML specification encapsulates the multiplication of matrices and the mixed operations necessary to transform a points relation by use of a matrix composed to reflect the desired operations. e.g. viewing transforms.

```
datatype matrix = null
| matcons of ((real list) * matrix) ;

exception mkmat : unit

fun mkmat((rows : real list)::rest)=
    matcons(rows,mkmat(rest))
  | mkmat((nil : real list)::rest) = raise mkmat
  | mkmat(nil) = null ;
```

The datatype "matrix" is defined as "null" (an empty matrix) or as a number of lists of real (as opposed to integer) values. This allows matrices of any size to be accommodated. The main construction function is "mkmat" which takes a list of lists of reals (i.e. rows of the resulting matrix) and applies the matrix constructor to them to create the matrix. There is no checking to ensure that the matrices are rectangular.

```

(* matrix -> row Returns next row from matrix *)
fun getrow(null) = nil
  | getrow(matcons(r,m)) = r ;

(* matrix -> col Returns next column from matrix *)
fun getcol(null) = nil
  | getcol(matcons((nil:real list),m)) = nil
  | getcol(matcons(r,m)) = hd(r) :: getcol(m) ;

(* matrix -> matrix *)
(* strips row and returns rest of matrix *)
fun rows(null) = null
  | rows(matcons(r,m)) = m ;

(* matrix -> matrix *)
(* strips col and returns rest of matrix *)
fun cols(null) = null
  | cols(matcons(r,m)) = matcons(tl(r),cols(m)) ;

```

The above functions serve to deconstruct a matrix and thus allow access to the individual rows and columns as required in the subsequent functions.

```

(* row X col -> num Multiplies a row by a column *)
fun rcmult(r:real list,nil:real list) = 0.0
| rcmult(nil:real list,c:real list) = 0.0
| rcmult(nil:real list,nil:real list) = 0.0
| rcmult(h::t,hl::tl) = h*hl + rcmult(t,tl) ;

(* Creates row of result matrix by multiplying *)
(* row by each column of matrix *)
(* row X matrix -> row *)
fun mkrow(r,null) = nil
| mkrow(r,matcons(nil,m)) = nil
| mkrow(r,m) = rcmult(r,getcol(m)) ::
                mkrow(r,cols(m)) ;

(**** Tests if two matrices are      ***)
(**** multiply compatible            ***)
(* matrix X matrix -> boolean *)
fun mult_compatible(m1,m2) =
    length(getrow(m1)) = length(getcol(m2)) ;

(* matrix X matrix -> matrix ;*)
fun mm(null,m1:matrix) = null
| mm(m1:matrix,m2:matrix) =
    if mult_compatible(m1,m2) then
        matcons(mkrow(getrow(m1),m2),mm(rows(m1),m2))
    else null;

```

These functions perform the multiplication of two matrices after checking that they are compatible for the multiplication operation. (The "length" function is built-in a returns the number of members of a list).

```

(* converts a tuple into a matrix *)
fun mkptmat(x,y,z):matrix = mkmat([x,y,z,1.0]::nil)
                                end;

(* Degree to radian conversion *)
fun dtor(n : real) = n * (3.14159 / 180.0) ;

(* num X num X num -> matrix *)
fun translate(x:real,y:real,z:real) =
    mkmat([[1.0,0.0,0.0,0.0],
           [0.0,1.0,0.0,0.0],
           [0.0,0.0,1.0,0.0],
           [x,y,z,1.0]]) ;

(* num X num X num -> matrix *)
fun scale(x:real,y:real,z:real) =
    mkmat([[x,0.0,0.0,0.0],
           [0.0,y,0.0,0.0],
           [0.0,0.0,z,0.0],
           [0.0,0.0,0.0,1.0]]) ;

(* num -> matrix *)
fun zrot(n:real) =
    mkmat([[cos(dtor(n)),0.0-sin(dtor(n)),0.0,0.0],
           [sin(dtor(n)),cos(dtor(n)),0.0,0.0],
           [0.0,0.0,1.0,0.0],
           [0.0,0.0,0.0,1.0]]) ;

(* num -> matrix *)
fun yrot(n:real) =
    mkmat([[cos(dtor(n)),0.0,sin(dtor(n)),0.0],
           [0.0,1.0,0.0,0.0],
           [0.0-sin(dtor(n)),0.0,cos(dtor(n)),0.0],
           [0.0,0.0,0.0,1.0]] ) ;

(* num -> matrix *)
fun xrot(n:real) =
    mkmat([[1.0,0.0,0.0,0.0],
           [0.0,cos(dtor(n)),0.0-sin(dtor(n)),0.0],
           [0.0,sin(dtor(n)),cos(dtor(n)),0.0],
           [0.0,0.0,0.0,1.0]]) ;

```

The functions above produce the transform matrices necessary for rotation, scaling and translation operations. They can be combined into a single matrix by multiplication.

```

fun sqr(n : real) = n*n ;

(*-----*)
(*real Xreal Xreal Xreal XrealXrealXreal -> matrix*)
(*vx    vy    vz    dx    dy    dz  arbitrary  *)
(*  view point      target point  rotation    *)

fun viewmat(vx,vy,vz,dx,dy,dz,azrot) =
let val vdx = if vx=dx then 1.0 else vx-dx in
let val vdy = if vy=dy then 1.0 else vy-dy in
let val m1  = translate(~vx,~vy,~vz) in
let val m2  = xrot(90.0) in
let val m3  = yrot(~(180.0+(arctan((vdx)/(vdy))*
                        180.0/3.142))) in
let val m4  = xrot(~(arctan((vz-dz)/
                        (sqrt(sqr(vdy)+sqr(vdx))))*180.0/3.142)) in
let val m5  = scale(1.0,1.0,~1.0) in
let val m6  = zrot(~azrot) in
      mm(m1,mm(m2,mm(m3,mm(m4,mm(m5,m6))))))
      end end end end end end end ;
\

```

The viewing matrix is constructed by multiplying together the transform matrices necessary to perform the viewing function. The precise details of this function can be found in standard texts on graphics (e.g [77].).

```

(* ----- *)
(* This function takes a single row matrix and      *)
(* produces a tuple suitable for insertion into     *)
(* a points relation                               *)
fun mkptup(key:string,m:matrix) =
  let val one :: two :: three :: r = getrow(m) in
    (key,one,two) end ;

(* ----- *)
(* relation X matrix -> relation                    *)
(* takes a subset of the main points relation      *)
(* and returns a x,y points relation               *)

fun transform( r:'a relation, m:matrix):'a relation=
  if ismt(r) then nil else
  let val (k,pt) = first(r) in
    add(mkptup(k,mm(mkptmat(pt),m)),
        transform(rest(r),m))
    end ;

```

The "transform" function performs the mixed type operation of applying a matrix to a points relation such that the resulting relation contains points whose coordinates have been transformed by multiplication with the matrix.

APPENDIX H

Example DEAL Programs for Display

The following functions and views are common to both examples described in Chapter 6.


```

view rest(x:rel) as {
    rest := x -- first(x);
};

func mode(m:int) {
    d := byte(22);
    d := byte(m);
    mode := d;
};

func plot(st:int,x:int,y:int) {
    d := byte(25);
    d := byte(st);
    d := word(x);
    d := word(y);
    plot := d;
};

func setup() {
    d := byte(22);
    d := byte(1);
    d := byte(24);
    d := word(0);
    d := word(128);
    d := word(1279);
    d := word(1023);
    d := byte(28);
    d := byte(0);
    d := byte(31);
    d := byte(39);
    d := byte(28);
    d := byte(18);
    d := byte(0);
    d := byte(1);
    setup := 1 ;
};

func display(scripts :rel ) {
    if (card(scripts) = 0 ) { display := 1 }
    else {
        fr := first(scripts);
        stx := #(fr [sx]);
        sty := #(fr [sy]);
        enx := #(fr [ex]);
        eny := #(fr [ey]);
        ds := plot(4,floor(stx),floor(sty));
        de := plot(5,floor(enx),floor(eny));
        display:= display(rest(scripts));
    }
}

```

);

The following DEAL program shows the creation and filling of the relations for the "house" in Chapter 6.

```

create table object ( onum char(9) nonnull,
                    oname char(9) ) ;

insert into object values ("o1","roof");
insert into object values ("o2","door");
insert into object values ("o3","window");
insert into object values ("o4","wall");
insert into object values ("o5","house1");
insert into object values ("o6","house2");
insert into object values ("o7","frame");
insert into object values ("o8","port");

create table hierarchy ( sup char(9) nonnull,
                        inf char(9) nonnull) ;

insert into hierarchy values ("o5","o1");
insert into hierarchy values ("o5","o2");
insert into hierarchy values ("o5","o3");
insert into hierarchy values ("o5","o4");
insert into hierarchy values ("o6","o1");
insert into hierarchy values ("o6","o2");
insert into hierarchy values ("o6","o3");
insert into hierarchy values ("o6","o4");
insert into hierarchy values ("o2","o7");
insert into hierarchy values ("o2","o8");

create table linelink ( onum char(9) nonnull,
                       lnum char(9) nonnull ) ;

insert into linelink values ("o1","l1");
insert into linelink values ("o1","l2");
insert into linelink values ("o1","l3");
insert into linelink values ("o7","l5");
insert into linelink values ("o7","l6");
insert into linelink values ("o7","l4");
insert into linelink values ("o3","l7");
insert into linelink values ("o3","l8");
insert into linelink values ("o3","l9");
insert into linelink values ("o3","l10");
insert into linelink values ("o4","l11");
insert into linelink values ("o4","l12");
insert into linelink values ("o4","l13");
insert into linelink values ("o8","l14");
insert into linelink values ("o8","l15");
insert into linelink values ("o8","l16");
insert into linelink values ("o8","l17");

create table lines ( lnum char(9) nonnull,
                    spt char(9),
                    ept char(9) ) ;

```

```

insert into lines values ("l1","p1","p2");
insert into lines values ("l2","p2","p3");
insert into lines values ("l3","p3","p1");
insert into lines values ("l4","p11","p10");
insert into lines values ("l5","p10","p9");
insert into lines values ("l6","p9","p8");
insert into lines values ("l7","p12","p13");
insert into lines values ("l8","p13","p14");
insert into lines values ("l9","p14","p15");
insert into lines values ("l10","p15","p12");
insert into lines values ("l11","p4","p5");
insert into lines values ("l12","p5","p6");
insert into lines values ("l13","p6","p7");
insert into lines values ("l14","p16","p17");
insert into lines values ("l15","p17","p18");
insert into lines values ("l16","p18","p19");
insert into lines values ("l17","p19","p16");

```

```

create table point ( ptnum char(9) nonull,
                    x real,
                    y real,
                    z real) ;

```

```

insert into point values ("p1",100.0,600.0,0.0);
insert into point values ("p2",400.0,900.0,0.0);
insert into point values ("p3",700.0,600.0,0.0);
insert into point values ("p4",200.0,600.0,0.0);
insert into point values ("p5",200.0,400.0,0.0);
insert into point values ("p6",600.0,400.0,0.0);
insert into point values ("p7",600.0,600.0,0.0);
insert into point values ("p8",450.0,400.0,0.0);
insert into point values ("p9",450.0,500.0,0.0);
insert into point values ("p10",380.0,500.0,0.0);
insert into point values ("p11",380.0,400.0,0.0);
insert into point values ("p12",240.0,520.0,0.0);
insert into point values ("p13",280.0,520.0,0.0);
insert into point values ("p14",280.0,440.0,0.0);
insert into point values ("p15",240.0,440.0,0.0);
insert into point values ("p16",400.0,480.0,0.0);
insert into point values ("p17",430.0,480.0,0.0);
insert into point values ("p18",430.0,450.0,0.0);
insert into point values ("p19",400.0,450.0,0.0);

```

```

view getall (recrel : rel , part : rel) as {
    if ( card(recrel) = 0 ) { getall := recrel ;}
    else {
temp := (( recrel (inf,sup) part )
         rename [0:=sup,1:=inf] ) ;
getall := (recrel -- (temp [sup])) ++
         getall(temp [inf],part) ;
    }
}

```

```

};

view flatten (onum : char , part : rel) as {
    templ := (part where sup = onum) [inf] ;
    flatten := getall(templ,part) ;
};

view gjoin(objname:char) as {
nrel := ( object where oname = objname ) [onum] ;
frel := flatten(#(nrel),hierarchy);
lrel := ((frel ( inf,onum ) linelink)
          rename [1:=lnum]) [lnum] ;
l2rel := (lrel ( lnum,lnum ) lines )
          rename [0:=lnum,1:=spt,2:=ept];
pt1 := (l2rel (spt,ptnum) point )
        rename [0:=lnum,2:=ept,3:=sx,4:=sy];
pt1tmp := pt1 [lnum,ept,sx,sy] ;
pt2 := (pt1tmp (ept,ptnum) point )
        rename [0:=lnum,2:=sx,3:=sy,4:=ex,5:=ey];
fin := pt2 [lnum,sx,sy,ex,ey] ;
gjoin:= fin ;
};

func show(objname : char) {
    d := setup();
    show := display(gjoin(objname));
};

show("house1");

```

The following DEAL program details the relations and functions for the "cube" example in Chapter 6.

```

create table stmp1 ( number int(1) nonnull ,
                    one real,
                    two real,
                    three real,
                    four real);

create table stmp2( number int(1) nonnull ,
                    one real,
                    two real,
                    three real,
                    four real);

create table xtmp ( number int(1) nonnull ,
                    one real,
                    two real,
                    three real,
                    four real);

create table clines ( lnum char(9) nonnull ,
                      spt int(1),
                      ept int(1));

insert into clines values ("11",1,2);
insert into clines values ("12",2,3);
insert into clines values ("13",3,4);
insert into clines values ("14",4,1);
insert into clines values ("15",5,6);
insert into clines values ("16",6,7);
insert into clines values ("17",7,8);
insert into clines values ("18",8,5);
insert into clines values ("19",1,5);
insert into clines values ("110",2,6);
insert into clines values ("111",3,7);
insert into clines values ("112",4,8);

view getcol(arel:rel,num : at) as {
    getcol := arel [number,num] ;
};

view translate(x:dbler,y:dbler,z:dbler) as {
insert into trans values ( 1,1.0,0.0,0.0,0.0);
insert into trans values ( 2,0.0,1.0,0.0,0.0);
insert into trans values ( 3,0.0,0.0,1.0,0.0);
insert into trans values ( 4,x,y,z,1.0);
    translate := trans ;
};

view scale(x:dbler,y:dbler,z:dbler,st : rel) as {
insert into st values ( 1,x,0.0,0.0,0.0);

```

```

insert into st values ( 2,0.0,y,0.0,0.0);
insert into st values ( 3,0.0,0.0,z,0.0);
insert into st values ( 4,0.0,0.0,0.0,1.0);
        scale := st ;
};

```

```

view rotz(angle:dbl) as {
cosangle := cosine(angle);
sinangle := sine(angle);
negsinangle := - sinangle ;
insert into zrot values ( 1,cosangle,negsinangle,
                        0.0,0.0);
insert into zrot values ( 2,sinangle,cosangle,
                        0.0,0.0);
insert into zrot values ( 3,0.0,0.0,1.0,0.0);
insert into zrot values ( 4,0.0,0.0,0.0,1.0);
        rotz := zrot ;
};

```

```

view rotx(angle:dbl,xt:rel) as {
cosangle := cosine(angle);
sinangle := sine(angle);
negsinangle := - sinangle ;
insert into xt values ( 1,1.0,0.0,0.0,0.0);
insert into xt values ( 2,0.0,cosangle,
                        negsinangle,0.0);
insert into xt values ( 3,0.0,sinangle,
                        cosangle,0.0);
insert into xt values ( 4,0.0,0.0,0.0,1.0);
        rotx := xt ;
};

```

```

view roty(angle:dbl) as {
cosangle := cosine(angle);
sinangle := sine(angle);
negsinangle := - sinangle ;
insert into yrot values ( 1,cosangle,0.0,
                        sinangle,0.0);
insert into yrot values ( 2,0.0,1.0,0.0,0.0);
insert into yrot values ( 3,negsinangle,0.0,
                        cosangle,0.0);
insert into yrot values ( 4,0.0,0.0,0.0,1.0);
        roty := yrot ;
};

```

```

insert into bpts values ( 1,0.0,0.0,0.0);
insert into bpts values ( 2,50.0,0.0,0.0);
insert into bpts values ( 3,50.0,50.0,0.0);
insert into bpts values ( 4,0.0,50.0,0.0);
insert into bpts values ( 5,0.0,0.0,50.0);
insert into bpts values ( 6,50.0,0.0,50.0);

```

```

insert into bpts values ( 7,50.0,50.0,50.0);
insert into bpts values ( 8,0.0,50.0,50.0);

func rval(pts :rel, col : at) {
    rval := #(pts [col]);
};

func cval(xmat : rel, cname : at,n : int){
    cval := #((getcol(xmat,cname)) [cname]
              where number = n);
};

func rcpts(pts:rel,bmat:rel,cname:at) {
    rcpts := rval(pts,x) * cval(bmat,cname,1)
    + rval(pts,y) * cval(bmat,cname,2)
    + rval(pts,z) * cval(bmat,cname,3)
    + 1.0 * cval(bmat,cname,4) ;
};

view transformpts(pts:rel,amatrix:rel,c:rel) as {
if (card(pts) = 0) {
    transformpts := c;
} else {
fr := first(pts);
ptval := #(fr [ptnum]);
rc1 := rcpts(fr,amatrix,one);
rc2 := rcpts(fr,amatrix,two) ;
rc3 := rcpts(fr,amatrix,three);
vsx := (511.5 * (rc1/rc3)) + 511.5;
vsy := (511.5 * (rc2/rc3)) + 511.5;
insert into c values ( ptval, vsx, vsy,0.0) ;
transformpts := transformpts(rest(pts),amatrix,c);
}
};

view gjoin(npts:rel) as {
pt1 := (clines (spt,ptnum) npts )
      rename [0:=lnum,2:=ept,3:=sx,4:=sy];
ptltmp := pt1 [lnum,ept,sx,sy] ;
pt2 := (ptltmp (ept,ptnum) npts )
      rename [0:=lnum,2:=sx,3:=sy,4:=ex,5:=ey];
fin := pt2 [lnum,sx,sy,ex,ey] ;
gjoin:= fin ;
};

```



```

view viewmat(vx:double,vy:double,vz:double,dx:double,
             dy:double,dz:double,azrot:double) as {
if (vx=dx) { vdx := 1.0 ; } else { vdx := vx-dx ;}
if (vy=dy) { vdy := 1.0; } else { vdy := vy-dy ;}
nvx := 0.0 - vx ;
nvx := 0.0 - vx ;
nvz := 0.0 - vz ;
m1 := translate(nvx,nvy,nvz) ;
m2 := rotx(90.0,xtmp1) ;
n3val := 0.0 - (180.0+(atan((vdx)/(vdy))*
                    180.0/3.142)) ;
m3 := roty(n3val);
n4val := 0.0 - (atan((vz-dz)/(
                    sqrt((vdy*vdy)+(vdx*vdx))))*180.0/3.142) ;
m4 := rotx(n4val,xtmp2) ;
n1 := 0.0 - 1.0 ;
m5 := scale(1.0,1.0,n1,stmp1) ;
m6 := scale(4.0,4.0,1.0,stmp2) ;
viewmat := (((m1 |x| m2) |x| m3) |x| m4)
            |x| m5) |x| m6;
};

```

```

view vmat() as {
vmat := viewmat(100.0,200.0,100.0,0.0,0.0,0.0,
                135.0,xtmp) ;
};

```

```

func showx(pts:rel) {
d := setup();
showx:=display(gjoin(transformpts(pts,vmat,cpt)));
};

```

```

show(bpts);

```

APPENDIX I

C Language Implementation

The following shows a small part of the implementation in C from the ML specification of the graphical database. The ML specification (Figure I.1) and the C implementation (Figure I.2) for the "union" operation is shown.

```
fun union(mt,r) = r
| union(tcons(t,r),rr) = if member(t,rr)
then union(r,rr)
else tcons(t,union(r,rr))
```

Figure I.1: ML specification for Union

```

typedef union anything
{
    int num;
    char ch ;
    float real ;
    char *word ;
    char *attname;
    struct list *ll ;
} typevar ;

/** This C union mimics the polymorphic types of ML
by enabling a variable of type "typevar" to be of
any of the quoted types. **/

typedef struct list
{
    char type ;
    typevar item ;
    struct list *next;
} llist ;

/** This linked list structure can contain elements
of any type allowed in the "typevar" union. The type
of any item is indicated by the "type" field of this
structure. **/

typedef llist set ;

typedef llist tuple ;

typedef struct rel
{
    set *tuples;
} relation ;

relation *runion(rel1,rel2)
relation *rell,*rel2 ;
{
relation *temp;
temp = (relation *) malloc(sizeof(relation));
temp->tuples = (set *) sunion(rell->tuples,
                             rel2->tuples) ;
return(temp) ;
}

```

```
}

set *sunion(setone, settwo)
set *setone, *settwo;
{
    if (setone == NULL) return(settwo);
    if (settwo == NULL) return(setone);
    else return(add(setone->type, setone->item,
                    sunion(setone->next, settwo)));
}

/** The original ML function for "union" must be
split into two to allow for the allocation of
storage for the result of the "union" operation. */
```

Figure I.2: C implementation of Union

APPENDIX J

RDB Implementation

The following shows the RDB schema and PASCAL programs used for this implementation. The "define" statements of RDO (the DML/DDL for RDB) establish views which constitute the "gjoin" for a line segment representation.

```
define view cstart of
l in line_segs cross
p in newpoints with l.st_pt = p.ptnum .
lnum from l.lnum.
sx from p.x.
sy from p.y.
sz from p.z.
end cstart view.
```

```
define view cfin of
l in line_segs cross
p2 in newpoints with l.end_pt = p2.ptnum .
lnum from l.lnum.
ex from p2.x.
ey from p2.y.
ez from p2.z.
end cfin view.
```

```
define view clines of
s in cstart cross
e in cfin with
s.lnum = e.lnum.
lnum from s.lnum.
sx from s.sx.
sy from s.sy.
sz from s.sz.
ex from e.ex.
ey from e.ey.
ez from e.ez.
end clines view.
```

The following is the PASCAL program for selecting the object to be displayed and then using the device driver routines to perform the display operation. The "%include" files contain the device driver procedures and functions and those for matrix operations. The RDB interface is via the non-PASCAL statements which are converted into VMS/PASCAL by a pre-processor prior to compilation.

```

program gdb(input,output);

{ This is TERMULATOR version }

%include 'decls.inc'

database db = pathname 'cdd$top.mc.t.aww.graphic';

%include 'exter.inc'

procedure nerase ;
begin
    for n in newpoints ;
        erase n ;
    end_for ;
end;

procedure transform(tmat : matrix) ;
var mat,nmat : matrix ;
    num : integer ;
begin
    num:=0;
    for p in points cross o in op
        with (o.onum = "o004") and (o.pnum = p.ptnum) ;
            num := num + 1;
            mat[1,1] := p.x ;
            mat[1,2] := p.y ;
            mat[1,3] := p.z ;
            mat[1,4] := 1.0 ;
            nmat := matmult(mat,tmat,1,4,4,4) ;
            store n in newpoints using ;
            n.ptnum := p.ptnum ;
            n.x := nmat[1,1] ;
            n.y := nmat[1,2] ;
            n.z := nmat[1,3] ;
        end_store ;
    end_for ;
end;

procedure show(xmin,xmax,ymin,ymax,dist,
                scr:integer);
var    vsx,vsy,vcx,vcy : real ;
        xls,yls,x2s,y2s real ;
        sx,sy,sz,ex,ey,ez : real ;
        persp,nsz,nez : real ;

begin
    vsx := (xmax-xmin)/2 ;

```

```

vsy := (ymax-ymin)/2 ;
vcx := (xmax+xmin)/2 ;
vcy := (ymax+ymin)/2 ;
persp := dist/scr ;
for l in clines ;
  if l.sz = 0.0 then nsz:=1.0 else nsz:=l.sz;
  if l.ez = 0.0 then nez:=1.0 else nez:=l.ez;

  xls := vsx * persp * (l.sx/(nsz)) + vcx ;
  yls := vsy * persp * ((l.sy)/(nsz)) + vcy ;
  x2s := vsx * persp * (l.ex/(nez)) + vcx ;
  y2s := vsy * persp * ((l.ey)/(nez)) + vcy ;

  plot(4, trunc(xls), trunc(yls));
  plot(5, trunc(x2s), trunc(y2s));
  writeln ;
end_for;

end;

{Start of main program}
begin
  start_transaction read_write;
  startup ;
  transform(viewmat(39600.0,31800.0,1000.0
    , 39600.0,31800.0,0.0,135.0)) ;
  show(0,1200,128,1020,40,11);
  nerase;

end.

```


APPENDIX K

Hardware

The hardware used for this work is as follows.

- (1) VAX cluster (running VMS)
- (2) VAX 11/750 (running ULTRIXO)
- (3) Sun 3/160 (running SUN-OS)

Systems 1 and 2 use a BBC micro as a graphics terminal using the ACORNSOFT TERMULATOR and the Cambridge University Computing Laboratory GTERM graphics terminal emulator. An Epson LX-86 dot matrix printer was used for hardcopy of the screen dumps produced by the GDB system.

Porting the system to the SUN workstation proved troublesome as there were problems with the internal representation of integers and reals which forced a rewrite of some of the PRECI/C source code. The graphical interface to the SUN console is not functional at the time of writing.

0 ULTRIX and VAX are a trademark of Digital Equipment Corporation.

References

1. R.A. Lorie and W. Plouffe, "Complex Objects and their Use in Design Transactions", in ACM/IEEE Proc. Ann. Meet. : Engineering Design Applications, pp. 115-121, 1983.
2. R.L. Haskin and R.A. Lorie, "On Extending the Functions of a Relation Database System", in ACM SIGMOD Int. Conf. on Management of Data, pp. 207-212, 1982.
3. R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier, "Supporting Complex Objects in a Relational System for Engineering Databases", in Query Processing in Database Systems, ed. W. Kim, D.S. Reiner, D.S. Batory, pp. 145-155, S-V, 1985.
4. R.A. Lorie, "Issues in Databases for Design Applications", in File Structures & Databases for CAD, ed. Encarnacao J. & Krause F.L., North-Holland, 1982.
5. G. Hallmark and R.A. Lorie, "Towards VLSI Design Systems Using Relational Databases", in Proc. 28th. Int. Conf. IEEE Comcon, pp. 326-329, IEEE, 1984.
6. R.A. Lorie and A. Meier, "Using A Relational DBMS for Geographical Databases", Geo-Processing, vol. 2, pp. 243-257, 1984.
7. D.S. Batory and W. Kim, "Modelling Concepts for VLSI CAD Objects", ACM TODS, vol. 10, no. 3, pp. 322-346, Sept 1985.
8. A.J. Morffew, S.P. Todd, and M.J. Snelgrove, "The Use of a Relational Database for Holding Molecular Data in a Molecular Graphics System", Comp. and Chem., vol. 7, no. 1, pp. 9-16, 1983.
9. S.J.P. Todd, A.J. Morffew, and J. Burrige, "Application of Relational Database and Graphics to the Molecular Sciences", in Proceedings of the Third British National Conference on Databases, ed. Longstaff J., C.U.P., 1984.
10. M. Tikkanen, M. Mantyla, and M. Tamminen, "GWB/DMS: A Geometric Data Manager", in Eurographics '83, ed. P.W. ten Hagen, pp. 99-111, 1983.
11. A.A.G. Requicha, "Representations for Rigid Solids", ACM Comp. Surv., vol. 12, no. 2, pp. 437-464, Dec. 1980.

12. Y.C. Lee and K.S. Fu, "A CGS Based DBMS for CAD/CAM and its Supporting Query Language", in Proc. Annual Meeting - Engineering Design Applications - Database Week, pp. 123-130, May 1983.
13. J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation and Generalisation", ACM TODS, vol. 2, no. 2, pp. 105-133, June 1977.
14. S.Y.W. Su, "Modelling Integrated Manufacturing Data with SAM*", Computer, vol. 19, no. 1, pp. 34-49, Jan 1986.
15. R.S. Shenoy and L.M. Patnaik, "Data Definition and Manipulation for a CAD database", CAD, vol. 15, no. 3, pp. 131-134, May 1983.
16. N.S. Chang and K.S. Fu, "A Query Language for Relational Image Database Systems", in Proc. Workshop on Picture Data Descriptions and Management, pp. 377-397, Aug 1980.
17. M.M. Zloof, "Query-by-Example - A Database Language", IBM Sys. J., vol. 16, no. 4, pp. 324-343.
18. A. Frank, "MAPQUERY: Database Query Language for Retrieval of Geometric Data and their Graphical Representation", ACM Computer Graphics, vol. 16, no. 3, pp. 199-207, July 1982.
19. M. Chock, A.F. Cardenas, and A. Klinger, "Database Structure and Manipulation Capabilities of a Picture Database Management System (PICDMS)", IEEE Trans. Pattern Analysis and Machine Intelligence, vol. PAMI-6, no. 4, pp. 484-492, July 1984.
20. W.I. Grosky, "Toward a Data Model for Integrated Pictorial Databases", Comp. Vision, Graph. and Image Processing, vol. 25, no. 3, pp. 371-382, March 1984.
21. T.L. Kunii, S. Weyl, and J.M. Tenenbaum, "A Relational Database Schema for Describing Complex Pictures", in Proc. 2nd. Int. Joint Conf. Pattern Recognition, pp. 310-316, Aug 1974.
22. G.Y. Tang, "A Management System for an Integrated Database of Pictures and Alphanumeric Data", Comp. Graph. and Image Process., vol. 16, pp. 270-286, 1981.
23. A. van Dam, "Some Implementation Issues Relating to Data Structures for Interactive Graphics", Int. J. Comp. and Info. Sciences, vol. 1, no. 4, 1972.

24. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", Comms. ACM, vol. 13, no. 6, Jun 1970.
25. D. Weller and R. Williams, "Graphic and Relational Database Support for Problem Solving", Comp. Graphics, vol. 10, no. 2, pp. 183-189, Summer 1976.
26. R. Williams, "On The Application of Relational Data Structures in Computer Graphics", in Proceedings of the 1974 IFIP Congress, pp. 722-726, North-Holland Publishing.
27. J.B. Crampes, C.Y. Chrisment, and G. Zurfluh, "The BIG Project", in ICOD 2, 1983.
28. D.L. Spooner, "Database Support for Interactive Computer Graphics", SIGMOD Rec., vol. 14, no. 2, pp. 90-99, June 1984.
29. D.W. Shipman, "The Functional Data Model and The Data Language DAPLEX", ACM TODS, vol. 6, no. 1, pp. 140-173, Mar 1981.
30. M. Hardwick and G. Sinha, "A Data Management System for Graphical Objects", in Proc. Int. Conf. Data Engineering, pp. 447-455, Feb 1986.
31. L.G. Shapiro, J.D. Moriarty, P.G. Mulgaonkar, and R.M. Haralick, "A Generalised Blob Model for 3-Dimensional Object Representation", in Proceedings of the IEEE Workshop on Picture Data Description and Management, pp. 109-116, Aug 1980.
32. M.T. Garrett and J.D. Foley, "Graphics Programming using a Database System with Dependency Declarations", ACM Transactions on Graphics, vol. 1, no. 2, pp. 109-128, 1982.
33. J.A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System", SIGMOD Record, vol. 15, no. 2, pp. 326-336, June 1986. Proc. SIGMOD '86 - International Conf. on Management of Data
34. H. Ehrig, H-J. Kreowski, and H. Weber, "Algebraic Specification Schemes for Database Systems", in Proc. 4th. VLDB Conf., pp. 427-440, Berlin, 1978.
35. P.C. Lockemann, H.C. Mayr, W.G. Weil, and W.H. Wohlleber, "Data Abstractions for Database Systems", ACM TODS, vol. 14, pp. 60-75, March 1979.

36. F.W. Tompa, "A Practical Example of the Specification of Abstract Data Types", Acta. Inf., vol. 13, pp. 205-244, 1980.
37. M.L. Brodie, "Axiomatic Definitions for Data Model Semantics", Inform. Sys., vol. 7, no. 2, pp. 183-197, 1982.
38. F. Golshani, T.S.E. Maibaum, and M.R. Sadler, A Modal System of Algebras for Database Specification and Query/Update Language Support, pp. 331-339.
39. E.J. Neuhold and T. Olnhoff, "The Vienna Development Method (VDM) and its Use for the Specification of a Relational Database System", in Information Processing 80, ed. S.H. Lavington, pp. 3-16, IFIP/North Holland, 1980.
40. M.L. Brodie, "Research Issues in Database Specification", ACM SIGMOD Record, vol. 13, no. 3, pp. 42-45, 1983.
41. A.L. Furtado and E.J. Neuhold, in Formal Techniques for Database Design, Springer-Verlag, 1986.
42. T. Niemi, "Specification of a Query Language by the Attribute Method", BIT, vol. 24, no. 2, pp. 171-186, 1984.
43. D. Stemple and T. Sheard, "Database Theory for Supporting Specification-Based Database System Development", in Proc. 8th. Int. Conf. on Software Engineering, pp. 34-49, IEEE, 1985.
44. D. Stemple, T. Sheard, and R. Bunker, "Abstract Data Types in Databases - Specification, Manipulation and Access", in Proc. Int. Conf. Data Engineering, pp. 590-597, Feb 1986.
45. W.R. Mallgren, "Formal Specification of Graphic Data Types", ACM Trans. Prog. Lang., vol. 4, no. 4, pp. 687-710, 1982.
46. G.S. Carson, "An Approach to the Formal Specification of Computer Graphics Systems", Computers & Graphics, vol. 8, no. 1, pp. 51-57, 1984.
47. D.A. Duce, E.V.C. Fielding, and L.S. Marshall, "Formal Specification and Graphics Software", RAL-84-068, 1984.
48. K. Ayra, "A Functional Approach to Picture Manipulation", Computer Graphics Forum, vol. 3, no. 1, pp. 35-

46, Mar 1984.

49. D.J. Abel and J.L. Smith, "A Relational GIS Database Accommodating Independent Partitionings of the Region", in Proc. 2nd. Int. Symposium on Spatial Data Handling, pp. 213-224, Int. Geographical Union, 1986.
50. G.T. Herman, "Surfaces of Objects in Discrete Three Dimensional Space", in Computer Graphics Conf., 1980.
51. E. Wong and W.B. Samson, "The Specification of a Relational Database (PRECI) as an Abstract Data Type and its Realisation in Hope", Computer Journal, vol. 29, no. 3, pp. 261-268, 1986.
52. W.R. Mallgren, Formal Specification of Interactive Graphics Programming Languages, MIT Press, 1983.
53. R.M. Burstall, D.B. MacQueen, and D.T. Sannella, "HOPE. An Experimental Applicative Language", Proc. 1980 LISP Conference, pp. 136-143, 1980.
54. S.M. Deen, D. Nikodem, and A. Vashishta, "The Design of a Canonical Database System (PRECI)", Comp. J., vol. 24, no. 3, Aug. 1981.
55. D. Sannella, "Formal Specification of ML Programs", LFCS, 1987.
56. S.M. Deen, "DEAL : A Relational Language with Deductions, Functions and Recursion", Data and Knowledge Engineering, vol. 1, 1985.
57. R. Sadeghi, A Database Query Language for Operations on Historical Data, Dundee College of Technology, 1987. Ph.D. Thesis
58. A.U. Frank, "Requirements for Database Systems Suitable to Manage Large Spatial Databases", in Proc. Int. Symp. on Spatial Data Handling, pp. 38-60, Geographisches Institut, 1984.
59. K. Ho-Le, "Finite Element Mesh Generation Methods : A Review and Classification", CAD, vol. 20, no. 1, pp. 27-37, Jan/Feb 1988.
60. L. Zhao and S.A. Roberts, "An Object Oriented Data Model for Database Modelling, Implementation and Access", Comp. J., vol. 31, no. 2, pp. 116-124, April 1988.

61. M.R. Blaha, W.J. Premerlani, and J.E. Rumbaugh, "Relational Database Design Using an Object-Oriented Methodology", CACM, vol. 31, no. 4, pp. 414-427, April 1988.
62. R. Sadeghi and W.B. Samson, "HQL - A Historical Query Language", in Proceedings of BNCOD-6, CUP, 1988.
63. R. Milner, "The Standard ML Core Language", Polymorphism, vol. 2, no. 2, pp. 1-28, Oct 1985.
64. L. Cardelli, ML Under UNIX, Bell Labs..
65. A. Wikstrom, "Functional Programming Using Standard ML", in Functional Programming Using Standard ML, Prentice Hall, 1987.
66. J. Guttag, "Notes on Type Abstraction", in Proceedings: Specification of Reliable Software, 1979.
67. J.V. Guttag and J.J Horning, "The Algebraic Specification of Abstract Data Types", Acta Informatica, no. 10, pp. 27-52, 1978.
68. J.V Guttag, E. Horowitz, and D.R Musser, "Abstract Data Types and Software Validation", Comms ACM, vol. 21, pp. 1048-1064, 1978.
69. J.V. Guttag, "Abstract Data Types and the development of Data structures.", Comms. ACM, vol. 20, no. 6, pp. 396-404, June 1977.
70. J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, "Abstract Data Types as Initial Algebras and Correctness of Data Representations", in ACM Conference on Computer Graphics, pp. 89-93, May 1975.
71. D.A. Duce and E.V.C. Fielding, "Better Understanding Through Formal Specification", RAL-84-128, Rutherford Appleton Laboratory, Dec 1984.
72. J.A. Goguen, "Some Design Principles and Theory for OBJ-O, A language to Express and Execute Algebraic Specifications of Programs", in Proc. Int. Conf. Mathematical Studies of Inf. Processing, pp. 425-473, Springer-Verlag, 1978. LNCS 75
73. D.A. Duce and E.V.C. Fielding, "Formal Specification - A Comparison of Two Techniques", RAL-85-051, July 1985.
74. C.J. Date, An Introduction to Database Systems, Volume II, Addison-Wesley, Reading, Mass., 1982.

75. C. J. Date, An Introduction to Database Systems, Volume I, Fourth Edition, Addison-Wesley, Reading, Mass., 1986.
76. C.J. Date, Relational Database: Selected Writings, Addison-Wesley, 1986.
77. W.M. Newman and R.F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill.

Published Papers

1. W.B. Samson and A.W. Wakelin, "The Representation of Graphics in Databases", Database Technology Information Update, vol. 1, no. 2, pp. 36-46, Pergamon Press, Dec 1987.
2. W.B. Samson, S.M. Deen, A.W. Wakelin, and R. Sadeghi, "Formalising the Relational Algebra - Some Specifications, Observations, Problems and Suggestions.", in Proc. of the Formal Methods Workshop, Middlesborough., July 1988.

The published paper cited below has been removed from the e-thesis due to copyright restrictions:

Samson, W. and Wakelin, A. (n.d.) The representation of graphics in databases. In: Database Technology, pp.36-46.

FORMALISING THE RELATIONAL ALGEBRA - SOME SPECIFICATIONS,
OBSERVATIONS, PROBLEMS AND SUGGESTIONS.

W B Samson*, S M Deen**, A W Wakelin*, R Sadeghi*,

*Department of Mathematics and Computer Studies, Dundee Col-
lege of Technology, Bell Street, Dundee, Scotland.

**DAKE Centre, University of Keele, Keele, Staffordshire,
England.

ABSTRACT

A number of different specifications, with varying closeness to actual implementations of the operators of the relational algebra are proposed. The specifications of individual operators are compared with their implementations in a real query language and it is demonstrated that while a relatively simple specification can encapsulate the fundamental ideas involved in the operators, a much more complex specification is needed if the semantics of a real query language is to be approached.

1. Introduction

A number of authors have used formal specification techniques for various aspects of database query languages [1,2,3,4,5,6,7,8,9,14,15,16,17]. Some of these look only at a limited subset of operators while others have attempted to specify an entire query language [2,5,6,7,8,9,14,15,16,17].

In the course of investigating the design of a database query language for operations on complex objects the authors have attempted to formally specify operators as the design has progressed. There are two reasons for this:

- (1) Formal specification is more likely to lead to a complete, consistent and correct implementation than informal specification.
- (2) Equational specifications may be implemented more or less directly in a functional language, leading to rapid prototyping of the software.

The authors have found, however, that the specification of the traditional relational algebra operators [10] is not trivial and, however it is done, leads sooner or later to problems of expression as a realistic syntax is approached.

In this paper two styles of specification are used -

- (a) Quantified predicates are used to define the properties of operators in a non-executable, but easily understood way, and
- (b) The algebraic, or equational style of specification is used in order to provide executable specifications which are provably correct in terms of (a).

This algebraic method is quite different from a first-order logic specification, as it is expressed in a logic programming language, such as Prolog. The differences are summarised below:

- (1) Algebraic semantics are used to specify abstract data types (or sorts) and operators (or functions) involving these data types in terms of a minimal set of operators (constructors) which may be used to represent any instance of the data type. In this way the operators of relational algebra may be specified. Logic programs, on the other hand, allow specific queries to be specified; the concept of an operator being more or less foreign to the Prolog paradigm. Algebraic semantics is, then, a meta-language in which query language operators and, to a limited extent, constructs may be defined. Prolog, on the other hand, may be viewed as a particularly expressive query language.
- (2) A relation, in algebraic semantics, is considered to be a data type, while it is regarded as a set of predicates in Prolog.
- (3) A query formulated in algebraic terms is a function call which will return a relation as a result. A Prolog query is a predicate, whose value is either true or false - NB it does not return a relation.
- (4) Algebraic semantics is many sorted - ie it is based on a number of distinct data types. Prolog is not many sorted; although there is no reason why a logic programming language should not be many sorted.
- (5) If-then-else is an essential part of algebraic semantics. In Prolog its place is taken by pattern matching and by appropriate conjunctions and disjunctions of predicates.

2. Relations and Tuples

The data structure employed in the relational model is the relation, which is considered to be a set of tuples (or records). This is usually visualised as a table, whose rows are the tuples and whose columns are known as the attributes of the relation. eg

EMPLOYEE

! EMPNO!	NAME !	AGE !	GRADE !	DEPT !
! E1	! SMITH!	26 !	2 !	D1 !
! E2	! JONES!	43 !	5 !	D3 !
! E3	! BROWN!	64 !	8 !	D1 !
! .	! . !	. !	. !	. !
! .	! . !	. !	. !	. !
! .	! . !	. !	. !	. !

etc.

NB Each row comprises an ordered n-tuple of values with various sorts (data types).

3. The Set Operators

The traditional set operations of union, intersection, difference and Cartesian product are defined for relations as sets of tuples. Normally, however, union, intersection and difference will only be applied to relations with the same kinds of tuples, although extended operators such as "outer union" etc have been proposed to overcome this problem.

The following symbols will be used in our specifications:

\forall	- the universal quantifier
\exists	- the existential quantifier
\in	- set membership
\wedge	- logical AND
\vee	- logical OR
\neg	- logical NOT

Assuming that set membership, \in , has been defined, we may formally specify union, difference and Cartesian product with the following predicates:

Union:

$$(\forall R1,R2:\text{relation}, t:\text{tuple}) \\ t \in R1 \vee t \in R2 \leftrightarrow t \in \text{Union}(R1,R2)$$

Difference:

$$(\forall R1,R2:\text{relation}, t:\text{tuple}) \\ t \in R1 \wedge t \notin R2 \leftrightarrow t \in \text{Difference}(R1,R2)$$

Product:

$$(\forall R1,R2:\text{relation}, t,s:\text{tuple}) \\ t \in R1 \wedge s \in R2 \leftrightarrow (t,s) \in \text{Product}(R1,R2)$$

Where (t,s) is the concatenation of the tuples t and s .

Intersection may, of course, be defined in terms of Union and Difference.

The signature of each of the set operators is

$$\text{relation } X \text{ relation } \rightarrow \text{relation}$$

and so each of the set operators involves only one sort - relation.

4. Other operators of relational algebra.

As well as the set operators defined above, there are two other fundamental relational algebra operators, Select (sometimes called Restrict) and Project which, although they return relations as results, require also to operate on objects of sort tuple. No author has, as far as we know, yet produced a simple, clear specification for either of these apparently straightforward operators. Maibaum [14] has not considered Select, and has dealt with Project in a slightly limited specification.

4.1. Select

The Select operator returns a subset of a relation containing only those tuples which satisfy some condition. Such a condition is itself an operator which takes a tuple as its argument and returns a truth value (true or false) as its result - ie its signature is:

$$\text{tuple } \rightarrow \text{truval.}$$

This means that the Select operator itself has two arguments:

- (i) a relation, and
- (ii) a condition.

The signature of Select is therefore:

relation X (tuple \rightarrow truval) \rightarrow relation.

This has a number of serious implications:

- (1) Select is a "higher-order" operator which takes an operator as an argument.
- (2) As a result, it is not possible to specify Select using first-order predicates, as we did for the set operators in section 3.

There are two ways of specifying select - the first is written using only first-order predicates (by assuming that the selection condition is predefined)- the second uses a higher-order specification.

4.1.1.

We assume that Select is a family of operators, each of which uses a different "global" condition which is predefined.

For example, suppose that we have already defined a set of conditions, C_1, C_2, \dots, C_n with identical signatures:

C_k : tuple \rightarrow truval, $k = 1, \dots, n$

A corresponding family of Select operators have signatures:

Select $_k$: relation \rightarrow relation, $k = 1, \dots, n$

We may then specify Select $_k$, say, as follows:

($\forall R$:relation, t :tuple, $k:1..n$)
 $t \in R \wedge C_k(t) \leftrightarrow t \in \text{Select}_k(R)$.

This method, although precise, and easily handled in terms of correctness proofs, does not mirror the actual operation of Select in a query language, where the condition is defined "on the fly".

eg in DEAL [11] or SQL we have

```
SELECT * FROM EMPLOYEE WHERE AGE > 27.
```

4.1.2.

The other way in which the more general select operator may be defined uses higher-order logic:

```
Select: relation X (tuple->truval) -> relation
(∀ R:relation, t:tuple, C:(tuple->truval))
  t ∈ R ∧ C(t) <-> t ∈ Select(R,C).
```

4.2. Project

Projection is intended to return only a subset of the columns of its argument relation. The required columns are usually presented, in real query languages, as a list of column names:

eg in SQL or DEAL

```
SELECT [NAME, DEPT] FROM EMPLOYEE
```

will return the two columns, NAME and DEPT from the EMPLOYEE relation, as its result relation, with any duplicate rows in the result eliminated in order to preserve the idea of a relation as a set of tuples.

There are a number of ways in which it is possible to express a specification of the Project operator:

4.2.1.

Like Select, it is possible to express Project as a family of operators based on pre-defined mappings from one kind of tuple into another which contains a subset of the attributes of the first.

For example, suppose we have the pre-defined mappings

$$M_k: \text{tuple} \rightarrow \text{tuple}, k = 1, \dots, n$$

and corresponding projections

$$\text{Project}_k: \text{relation} \rightarrow \text{relation}, k = 1, \dots, n$$

We can specify Project_k as follows:

$$\begin{aligned} &(\forall R: \text{relation}, t, s: \text{tuple}, k: 1..n) \\ & (t \in R \rightarrow M_k(t) \in \text{Project}_k(R)) \wedge \\ & (s \in \text{Project}_k(R) \rightarrow (\exists t \in R) M_k(t) = s) \end{aligned}$$

As with `Select`, this definition does not mirror the actual operation of `Project` in a real query language, where the mapping (as a column subset) is defined "on the fly".

Specifications using this simple logic are given in Appendix 1, Specification 1(a) and, in executable form, Appendix 2, Specification 2(a).

4.2.2.

Higher-order logic may again be used in the definition of a more general `Project`:

$$\text{Project}: \text{relation} \times (\text{tuple} \rightarrow \text{tuple}) \rightarrow \text{relation}$$
$$(\forall R: \text{relation}, t, s: \text{tuple}, M: (\text{tuple} \rightarrow \text{tuple}))$$
$$\begin{aligned} & (t \in R \rightarrow M(t) \in \text{Project}(R, M)) \wedge \\ & (s \in \text{Project}(R, M) \rightarrow (\exists t \in R) M(t) = s) \end{aligned}$$

This is closer to a real query operator and is, in fact, a generalisation of `Project`, allowing any mapping from a tuple into a tuple. Such a generalisation allows us, for example, to "Expand" [12] a relation to contain additional columns which contain values which are functions of the values of elements in the other columns.

Specifications for these operators are given in Appendix 1, Specification 1(b), and Appendix 2, Specification 2(b).

Despite the advantage of generalisation, it would also be useful to be able to define `Project` in terms of a relation and a list of attribute names to mirror the operation of such languages as `DEAL` and `SQL`. Such an apparently simple change does, however, lead to complications throughout the algebra, since tuples must now have names associated with each of their attributes, and these must be taken into account when naming the columns of result relations - which is non-trivial when more than one relation is used to produce a result. For example, the choice of names for the columns of a `Union` of two relations is tricky if it is intended that `Union` should be truly commutative.

The problem of naming conventions has no standard, universally recognised solution - although a number of practical solutions have been implemented. In order to avoid being caught up in the naming convention controversy, it is possible to sacrifice readability and "name" the attributes 1, 2, 3,... according to the order in which they appear in the tuples of a relation. Such a strategy avoids ambiguity of names in a relation, as well as the case where there might be a choice of name sets. It also avoids the need to incorporate the names of argument relations in the attributes of result relations, which can lead to very long names indeed, after a few operations.

If this attribute numbering convention is adopted then a function needs to be defined which will return the value of any attribute for any degree of tuple:

value: tuple X num -> alpha

where alpha may be any type [13].

For a unary tuple,

value(a,n) = if n=1 then a else undefined

For a binary tuple,

value((a,b),n) = if n=1 then a
 else if n=2 then b
 else undefined

For a triple,

value((a,b,c),n) = if n=1 then a
 else if n=2 then b
 else if n=3 then c
 else undefined

Such a function definition is potentially infinite and so is not entirely satisfactory. It is, of course, heterogeneous; returning one of a variety of types which appear in the tuple. This feature makes it hard, if not impossible to define generally, in a strongly typed system.

This type of formal specification is also close to practical implementations of database systems and therefore could be used to investigate the properties of data dictionary systems more closely.

4.3. Other operators

The other operators of relational algebra are, of course, easily specified in terms of the five considered above.

There are, however, certain operators which are now generally expected to be in the repertoire of any relational database - these include outer join, outer union and extend. These are easily specified in much the same way as the operators considered above - extend is, in fact, just a special case of the higher-order project defined above. The "outer" operators each involve "null" values and it is beyond the scope of this study to deal with these.

4.4. Tuples as Lists

Languages like LISP have traditionally treated tuples as lists of values, rather than as the Cartesian product of their component domains. In a strongly typed, many sorted formalism, however, there is no analogue of the virtually typeless lists of LISP. Instead, it is only possible to define homogeneous lists of elements with the same sort, if strong typing is to be maintained. The advantage of using list representation for tuples is that a simple, finite definition for the *i*th element of a list is possible. It is possible to define a function which "coerces" any of the sorts of value which might be expected to occur in a tuple into a single sort to effectively allow elements of various sorts.

The disadvantage of this representation for tuples is that it now becomes necessary to check the consistency of types of values and lengths of tuples held in a relation.

This strategy was adopted by Wong and Samson [9]. It leads to executable specifications but at a high cost in terms of clarity.

A side-effect of this strategy is that the Cartesian product involves concatenation of lists, rather than the formation of ordered pairs of tuples.

Maibaum [14] deals with projection using a model which is not unlike this one. He models a tuple as a set of attributes and corresponding values. He avoids the problem of naming conventions by insisting that attribute names in the database are all distinct and that no two attributes with different names are union-compatible.

An algebraic specification for an algebra based on tuples modelled as lists is given in Appendix 3.

5. Conclusions

The formal specification of Codd's relational algebra is not as straightforward as one might expect. Problems arise because:

- (1) Select and Project are most naturally expressed as higher-order operators, making theorem proving difficult.
- (2) The naming of attributes needed for a concrete syntax substantially increases the complexity of the specification - tending to bias the specifier towards a particular implementation by, for example, modelling tuples as lists.

The various levels of abstraction might be viewed as steps along a path from a truly abstract specification to something approaching an implementation. Specifications at three levels of abstraction are appended. Specifications 1(a) and 1(b) present predicates which show the important features of the operators of relational algebra - 1(a) uses many sorted first-order logic only - 1(b) uses higher-order logic. Specifications 2(a) and 2(b) are executable equational specifications for 1(a) and 1(b) which are provably correct. Specification 3 is an executable equational specification which includes the concept of tuples as lists.

All of the executable equational specifications are written in a dialect of HOPE, an experimental applicative language [13].

APPENDIX 1

Specification 1(a) - using many-sorted first-order logic:

dec Union: relation X relation -> relation

($\forall R1, R2: \text{relation}, t: \text{tuple}$)
 $t \in R1 \vee t \in R2 \leftrightarrow t \in \text{Union}(R1, R2)$

dec Difference: relation X relation -> relation

($\forall R1, R2: \text{relation}, t: \text{tuple}$)
 $t \in R1 \wedge \neg(t \in R2) \leftrightarrow t \in \text{Difference}(R1, R2)$

dec Product: relation X relation -> relation

($\forall R1, R2: \text{relation}, t, s: \text{tuple}$)
 $t \in R1 \wedge s \in R2 \leftrightarrow (t, s) \in \text{Product}(R1, R2)$

dec Ck: tuple -> truval, k = 1, ..., n

!These may have any convenient specification!

dec Selectk: relation -> relation, k = 1, ..., n

($\forall R: \text{relation}, t: \text{tuple}, k: 1..n$)
 $t \in R \wedge Ck(t) \leftrightarrow t \in \text{Selectk}(R)$.

dec Mk: tuple -> tuple, k = 1, ..., n

dec Projectk: relation -> relation, k = 1, ..., n

($\forall R: \text{relation}, t, s: \text{tuple}, k: 1..n$)
 $(t \in R \rightarrow Mk(t) \in \text{Projectk}(R)) \wedge$
 $(s \in \text{Projectk}(R) \rightarrow (\exists t \in R) Mk(t) = s)$

Specification 1(b) - using many-sorted higher-order logic

[- As for 1(a) except for Select and Project:]

dec Select: relation X (tuple -> truval) -> relation

($\forall R:\text{relation}, t:\text{tuple}, C:(\text{tuple}\rightarrow\text{truval})$)

$t \in R \wedge C(t) \leftrightarrow t \in \text{Select}(R,C).$

dec Project: relation X (tuple -> tuple) -> relation

($\forall R:\text{relation}, t,s:\text{tuple}, M:(\text{tuple} \rightarrow \text{tuple})$)

$(t \in R \rightarrow M(t) \in \text{Project}(R,M)) \wedge$

$(s \in \text{Project}(R,M) \rightarrow (\exists t \in R) M(t) = s)$

APPENDIX 2

Specification 2(a) - an executable model of specification 1(a).

 [- without higher-order functions.]

```

typevar tuple, tuple1, tuple2; ! Use polymorphism of Hope!

infix :: :6; ! constructor for relation!

data relation(tuple) == nil ++ tuple :: relation(tuple);

! membership of relation !
infix ∈ :5;

dec ∈: tuple X relation(tuple) -> truval;

--- a ∈ nil <= false;

--- a ∈(b::r) <= if a=b then true else a ∈ r;

!Insertion of tuple, avoiding duplicate tuples !

dec insert: tuple X relation(tuple) -> relation(tuple);

--- insert(t,r) <= if t ∈ r then r else t :: r;

dec Union: relation(tuple) X relation(tuple) -> relation(tuple);

--- Union(nil,r) <= r;
--- Union(t::r1, r2) <= insert(t, Union(r1, r2));

dec Difference: relation(tuple) X relation(tuple) ->
relation(tuple);

--- Difference(nil, r) <= nil;
--- Difference(t::r1, r2) <= if t ∈ r2 then Difference(r1, r2)
else t::Difference(r1, r2);

! Auxiliary operator needed for definition of Product !

dec tprod: tuple1 X relation(tuple2) -> relation(tuple1 X
tuple2);

--- tprod(t, nil) <= nil;
--- tprod(a, b::r) <= insert((a,b), tprod(a, r));

dec Product: relation(tuple1) X relation(tuple2) ->
relation(tuple1 X tuple2);
--- Product(nil, r) <= nil;
--- Product(a::r1, r2) <= Union(tprod(a, r2), Product(r1, r2) );

! Selection condition !

dec cond: tuple -> truval;

```

```

! Define as required !

dec Select: relation(tuple) -> relation(tuple);
--- Select(nil) <= nil;
--- Select(t::r) <= if cond(t) then t::Select(r) else Select(r);

! tuple mapping for projection !

dec f: tuple1 -> tuple2;

! define as required !

dec Project: relation(tuple1) -> relation(tuple2);

--- Project(nil) <= nil;
--- Project(t::r) <= insert(f(t), Project(r) );

```

Specification 2(b) - using higher-order functions.

[as for Specification 2(a) except for Select and Project.]

! Higher-order select !

```

dec Select: relation(tuple) X (tuple->truval) -> relation(tuple);

--- Select(nil,c) <= nil;
--- Select(t::r, c) <= if c(t) then t::Select(r, c) else
Select(r,c);

```

! Higher-order project !

```

dec Project: relation(tuple1) X (tuple1->tuple2) ->
relation(tuple2);

--- Project(nil, f) <= nil;
--- Project(t::r, f) <= insert(f(t), Project(r, f) );

```


APPENDIX 3

Specification 3 - with tuples modelled as lists:

```

typevar alpha; ! Polymorphic type !

! A new sort to coerce existing sorts into a single sort !

data atom == al(num) ++ a2(char) ++ a3(list(char)) ++ a4(alpha);

! Any other sorts may be added to this definition !

data typecode == tnum ++ tchar ++ tlistchar ++ talpha;

! Operator to detect nature of coercion !

dec typeof: atom -> typecode;

--- typeof(al(x)) <= tnum;
--- typeof(a2(x)) <= tchar;
--- typeof(a3(x)) <= tlistchar;
--- typeof(a4(x)) <= talpha;

! Define a tuple to be a list of atoms !

type tuple == list(atom);

! Define the "type" of a tuple to be a list of typecode !

type tuptype == list(typecode);

! Define an operator to return "type" of a tuple !

dec type_of_tup: tuple -> tuptype;

--- type_of_tup(nil) <= nil;
--- type_of_tup(a::t) <= typeof(a)::type_of_tup(t);

! Define relation as a tuptype along with a list of tuples !

type relation == tuptype X list(tuple);

! Membership of relation !

infix ∈ :5;

dec ∈: tuple X relation -> truval;

--- t ∈ (tt,nil) <= false;

--- t ∈ (tt,a::l) <= if t=a then true else t ∈ (tt,l);

! Insertion of tuple into relation without duplication !

```

```

dec Insert: tuple X relation -> relation;

--- insert(t,(tt,l)) <= if type_of_tup(t) = tt then if t ∈
                        if t ∈ (tt,l) then (tt,l)
                        else (tt,t::l) else undefine;

! list membership !

dec elem:alpha X list(alpha) -> truval
--- elem(a,nil) <= false;
--- elem(a,b::c) <= if a=b then true else elem(a,c);

! Auxiliary operation for list concatenation with duplicate
elimination!

dec listunion: list(alpha) X list(alpha) -> list(alpha);

--- listunion(nil,a) <= a;
--- listunion(a::b,c) <= a::listunion(b,c) if not elem(a,c) else
listunion(b,c);

! Relational algebra Union !

dec Union: relation X relation -> relation;

--- Union((tt1,b1),(tt2,b2)) <= if tt1=tt2 then (tt1,
listunion(b1,b2)) else undefine;

! Relational algebra difference !

dec Difference: relation X relation -> relation;

--- Difference((tt1,nil),(tt2,l)) <= if tt1=tt2 then (tt1,nil)
else undefine;
--- Difference((tt1,a::b1),(tt2,b2)) <= if tt1=tt2 then if
elem(a,b2) then Difference((tt1,b1),(tt2,b2)) else
Insert(a,Difference((tt1,b1),(tt2,b2))) else undefine;

! Auxiliary operator !

dec tprod: tuple X relation -> relation;

--- tprod(a,(tt,nil)) <= (concat(ta,tt),nil) where ta ==
type_of_tup(a);
--- tprod(a,(tt,b::c)) <= insert((a,b), tprod(a,(tt,c)));

! Cartesian product !

dec Product: relation X relation -> relation;

--- Product((tt1,nil),(tt2,r)) <= (concat(tt1,tt2),nil);
--- Product((tt1,a::b),r) <=
Union(tprod(a,r),Product((tt1,b),r));

```

! Selection !

```
dec Select: relation X (tuple -> truval) -> relation;
```

```
--- Select((tt,nil),f) <= nil;  
--- Select((tt,a::l),f) <= if f(a) then  
Insert(a,Select((tt,l),f)) else Select((tt,l),f);
```

! Auxiliary operator !

```
dec element: list(alpha) X num -> alpha;
```

```
--- element(nil,succ(n)) <= undefine;  
--- element(a::b,n) <= if n=1 then a else element(b,n-1);
```

! Auxiliary operator !

```
dec sublist: list(alpha) X list(num) -> list(alpha);
```

```
--- sublist(a,nil) <= nil;  
--- sublist(a,n::t) <= element(a,n)::sublist(a,t);
```

! Relational algebra Projection !

```
dec Project: relation X list(num) -> relation;
```

```
--- Project((tt,nil),lnum) <= (sublist(tt,lnum),nil);  
--- Project((tt,a::b),lnum) <= Insert(sublist(a,lnum),  
Project((tt,b),lnum));
```

REFERENCES

1. Brodie, M L & Schmidt, J (1978) "What is the use of Abstract Data Types in Databases", Proc VLDB 1978, pp 140-141
2. Date C J (1982) "A Formal Definition of the Relational Model" in "An Introduction to Database Systems" Vol 2, pp 187-196
3. Ehrig, H, Kreowski, H-J & Weber, H (1978) "Algebraic Specification of Schemes for Data Base Systems", Internal publication of Habn-Meitner-Institut, Berlin.
4. Gehani, N (1982) "Specifications: formal and informal - a case study", Software Practice and Experience, Vol 12, pp 433-444
5. Golshani, F, Maibaum, T S, & Sadler, M R (1983) "A Model System of Algebra for Database Specification and Query/Update Language Support" Proc VLDB 1983.
6. Louis, G & Pirotte, A (1982) " A Denotational Definition of the Semantics of DRC, a Domain Relational Calculus", Proc. VLDB 1982, pp 348-356
7. Maibaum, T S (1977) "Mathematical Semantics and a Model for Data Bases", Information Processing 1977, pp 133-137, IFIP.
8. Pirotte, A. (1982), "A Precise Definition of Basic Relational Notions and of the Relational Algebra", ACM Sigmod Record, 1982, pp 30-35.
9. Wong, E Y & Samson W B (1986) "The Specification of a Relational Database (PRECI) and its Realisation in HOPE", the Computer Journal, vol 29, no 3, pp 261-268
10. Codd, E F (1970) "A Relational Model for Large Shared Data Banks" CACM, vol 13, no 6, pp 377-387
11. Deen, S M (1985) "A Relational Language with Deductions, Functions and Recursions", Data and Knowledge Engineering, Vol 1.
12. Chamberlin, D D, Astrahan, M M, Eswaran, K P, Griffiths, P P, Lorie, R A, Mehel, J W, Reisner, P & Wade, B W (1976) "SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control", IBM Jnl of Research and Development, Nov 1976, pp560-575.
13. Burstall, R M, Macqueen, D B, & Sannella, D T (1980) "Hope: An Experimental Applicative Language", Univ. of Edinburgh, Dept. of Computing Science Internal Report CSR-62-80.
14. Maibaum T S (1985) "Database Instances, Abstract Data Types and Database Specification", Comp. J., vol 28, no 2, pp 154-161
15. Turner R & Lowden B G (1985) "An Introduction to the Formal Specification of Relational Query Languages", Comp. J., vol 28, no 2, pp 162-169
16. Stemple D. & Sheard T. (1985) "Database Theory for Supporting Specification-Based Database System Development", Proc. (th Int. Conf on Soft. Eng., pp 34-49

17. Furtado A L & Neuhold E J (1986) "Formal Techniques for Database Design", Springer Verlag.

18. Ullman J (1982) "Principles of Database Systems" , Pitman