Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS–93–2–2

Michael Hugh O'Docherty

A Multimedia Information System with Automatic Content Retrieval

A Multimedia Information System with Automatic Content Retrieval[1,2]

Michael Hugh O'Docherty

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
mod@cs.man.ac.uk

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from m1.cs.man.ac.uk (130.88.13.4) in the directory /pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

[2]This work was supported both by the Science and Engineering Research Council and by ICL Computers Limited.

Abstract

Traditional databases use simple data types — chiefly numbers and strings — to represent information such as payroll records or scientific data. Queries are commonly expressed in relational calculus or algebra and the results require little more than a textual terminal for presentation. The natural successor to the traditional database is the Multimedia Information System which also uses richer data types — images, text, sound and so on — to describe some application domain. This calls for more advanced technologies such as graphical workstations for data creation and presentation, high-bandwidth networks for fast access to distributed data and large repositories for the storage of objects. These technologies are now widely available.

Ideally, queries put to a Multimedia Information System should refer to the content of stored objects — *content retrieval* — and results should contain complex as well as simple data types. Manual entry of content descriptions is time-consuming, error-prone and subjective; a better approach is to draw on the emerging fields of image interpretation, text interpretation, speech interpretation and knowledge representation to provide *automatic content retrieval*.

This thesis serves as an introduction to Multimedia Information Systems and related fields and describes in detail a pilot system with automatic content retrieval built by the Multimedia Group at Manchester University; the author has been instrumental in the design and implementation of this system, especially those components that cater for the generation and querying of content descriptions.

To Sam ...

Chapter 1

Introduction

The fashionable term 'multimedia' tends to be applied to any hardware or software system that processes more than one form of electronic medium, no matter what form that processing takes. For the purposes of this thesis, the author proposes a definition for the more specific term *Multimedia Information System*:

> "A Multimedia Information System, or MMIS, is a repository for objects of more than one electronic medium that permits retrieval according to the content of those objects."

MMISs have been a popular research topic in recent years for two reasons. First, because the technologies needed to implement them are only now becoming widely available — eg. optical disks for mass storage, high bandwidth networks for efficient resource sharing and sophisticated workstations for processing and presentation.

Secondly, many applications are multimedia in nature or would be greatly enhanced by multimedia techniques. Some of these applications are:

Education Presentations typically contain words, pictures, video and voice; books contain words and pictures, but there is no reason why an 'electronic book' should not also contain narrations and animations.

Offices Information may arrive in an office in many different forms, each being processed and filed in a different manner. For example, documents, memos, telephone messages, faxes. Since the output of offices is moving towards 'electronic publishing', why not collect and process the information in the same way?

Medical Records A patient's medical record may consist of case histories, X-rays, notes or sketches from consultations, test results, etc. In addition, several new imaging techniques generate digital or video data directly — computer tomography, magnetic resonance and ultrasound.

Libraries and Museums These contain vast amounts of data ranging from literary works to archæological artifacts, all currently accessed by manually generated indexes.

Computer Aided Design (CAD) For example blueprints, simulation, or interactive design with 3D rotation and scaling of objects.

Geographic Databases Maps, satellite images, demographics, even tourist information.

The above applications generate six main types of data, all of which can be represented directly or indirectly by a MMIS:

1. **Facts** Author, date, unique identifier — the types conveniently handled by existing databases.

2. **Statistics** Data that requires graphical (pie charts or bar charts) or tabular (spread sheet) representation.

3. **Text** As may be contained in a book or magazine article.

4. **Sound** Notably the human voice — narrations, comments, lectures — but also music, sounds from nature.

5. **Images**

   - **Raster** Derived from photographs, painting programs, video-discs.
   - **Vector** Line drawings derived from engineering drawings, CAD systems, vector graphics.
   - **Moving** A sequence of the above with extra clues as to three dimensional relationships, history, narrative. Examples are animation or (digitised) video.

6. **Composites** Mixed collections of other media, eg. electronic books.

So what characteristics should 'The Ideal MMIS' have? Well, firstly it should be able to store all those media listed above. Storage requirements will be huge, running to hundreds or thousands of kilobytes for raster images for example. Therefore, designers will have to consider the following:

State-of-the-art storage technology Ideally a 'juke-box' of read/write optical disks.

Multi-user access Since it will be expensive and sought-after, mass storage should be provided per-network rather than per-user.

Distributed architecture To allow more than one site or organisation to provide storage.

High bandwidth networks Local-area and wide-area networks for efficient implementation of multi-user access and distributed architectures.

Sharing An item appearing in more than one composite item could be stored only once while the composite items each received a reference to it. This facility may sometimes need to be over-ridden — for example, an item might be replicated at a remote site to decrease access time.

Data compression techniques These can be used to reduce storage requirements and speed transmission through networks.

Users, rather than just system administrators, should be able to import items from outside the system, using page scanners, sound samplers and the like. Editors should be provided to allow users to modify imported or existing items or to create new ones; although these editors will have to be medium-specific they should all have the same 'look and feel' — eg. all window based, with a similar menu hierarchy and similar high-level options. Editors for composite media could call on the appropriate editor for each sub-medium as required. It should be possible to present items both interactively (eg. images displayed on a terminal) or as hard-copy (eg. laser printer documents or video tape, as appropriate).

Queries[1] should be mixed-media, using the same language to describe all items, but possibly with sub-clauses specific to each medium [CT90]. If the ideal MMIS is to be more than just a filing system then content retrieval must be provided. The simplest form of content retrieval is based on manually generated *tags* — i.e. descriptions entered from the keyboard by an operator. The other extreme is automatic semantic analysis — i.e. descriptions inferred by the system. A middle-ground is retrieval relating to data type and structure, without any sophisticated analysis, eg.

"Find me all documents with a colour image on page five."

Obviously, tagging makes things easy for the implementor whereas automatic analysis is of most benefit to the user. Although automatic analysis will probably never be better than human analysis, the latter is too time consuming to be feasible in most cases, as well as being subjective and inconsistent.

From the user's viewpoint, there are several ways in which queries could be expressed:

Directly The user knows exactly what the system stores in relation to each item, e.g. the tags or indices it uses, the primitives it extracts during semantic analysis, or the way it structures composite media. Direct queries can then be expressed using some query language, as in conventional databases [Dat86].

By Sketch A 'shorthand' version of an item is given on-line by the user and the system tries to stretch and compress this *sketch* to fit stored items. The form of a sketch would vary from one medium to another, and would probably be meaningless in some. An image sketch might comprise a rough outline of a particular style of house

[1] For convenience, the term 'query' in this thesis refers (loosely) to anything that specifies a subset of the items stored in the MMIS. Examples are SQL, logical expressions or a candidate item provided by the user.

with salient features included such as rectangular Georgian windows with multiple panes. A voice sketch might be a sentence spoken by the user in a particular accent, dialogue or language. Some of these techniques won't appear for a long time, but they are still valid from the user's viewpoint. A simple image example that already exists is the Ledeen on-line symbol recogniser [NS81].

By Example Ideally, this would allow queries of the form

> "Retrieve images that are similar to this one."

but in general these are too vague to be soluble. Query by example is only possible if a constrained view of what is meant by 'similar' is taken. For example, for text, an approximate measure of word content can be found using a hashing function to produce *signature files* [FC84]: an example's signature is compared with those of other text items in the system to produce an initial set of candidate matches. Another, more powerful technique for query-by-example is provided by the notion of *classification* of an item or its parts; here, each item is put into one or more classes according to chosen characteristics.

By Association If the system allowed items to be linked together then the user could retrieve an item by following a link from one that had already been retrieved. Links could be generated automatically — eg. by decomposing a document and linking it to each of its pages as separate items — or by users, as in 'hypermedia' (see below).

Often, the items retrieved by the system will not match the query exactly — because sketching, query-by-example and automatic analysis are inherently imprecise. If a score could be given to the closeness of the match then the system could *rank* the items and the user would be able to examine the most interesting first (especially important for large systems). Another possible source of scores would be to associate an importance with different parts of the query — those items that match high scoring parts of the query would receive a high score. The scores generated could be discrete (eg. bad, reasonable, good, very good) or continuous (normalised to the range [0, 1] for instance).

As in any interactive system, the ideal MMIS would have to have a reasonable response time for queries. Automatic analysis can take a long time, sometimes measured in hours. Therefore, content retrieval should be based on analysis done *pre-query* — i.e. before the item is involved in a query — rather than *post-query*. 'Lazy evaluation' could be used as a compromise, where the analysis is performed when first needed and the results retained. Post-query analysis, lazy or otherwise, can greatly increase storage requirements.

Users may wish to generate arbitrary annotated connections between items in a 'hypermedia' fashion. The system should manage sets of connections as 'webs', providing queries such as

> "Get me webs containing image 10."

for informal access to the system.

Another interface technique that should be provided is *browsing*. Here, the user has a set of items (usually a subset of the system's contents obtained by query) and is allowed to scan over them. Various browsing techniques are examined in [CG88, INJ+88]. In order to fit more items on the screen and to avoid getting the full versions from the system, they could be displayed as:

Icons Small pictorial representations.

Miniatures The items themselves but reduced greatly in size and resolution.

Descriptions One-line comments about content — for example, 'headers' entered by operators or system-generated type names.

Many media require modern workstations for display but in a multi-user, distributed system they will vary in resolution, number of bits per pixel and so on. Therefore a workstation would have to assess incoming items and either

- reject them as undisplayable,
- alter the items to fit its own screen by resolution transformation, converting colour to grey-level, reducing the number of bits per pixel and so on,
- display what it can and replace the rest with iconic versions. This would allow a text-only screen to replace images in a page with a box containing the word IMAGE or even a description, while retaining the text.

Having identified the desirable features of a MMIS, the next step is to decide how to build one. Traditional database thinking is inadequate for the implementation of a MMIS; this stems from the fact that traditional databases — eg. network, hierarchical and relational — were motivated by the desire to computerise customer and employee records, scientific data and the like, namely facts as specified earlier. The data objects stored and manipulated are strings and numbers but the system knows little of their semantics other than how to make lexical or arithmetic comparisons. In contrast, MMISs have to cope with data differing widely in structure, size and meaning. Accordingly, other technologies and research fields need to be taken into account, for example databases designed for text, for images, for the object-oriented paradigm or for Artificial Intelligence. These and other fields are explored in Chapter 2.

The cornerstone of the ideal MMIS will be automatic content retrieval, based on pre-query interpretation. In order to address the problem of generating, representing and managing content descriptions, it is necessary to look to the field of Artificial Intelligence. To make conclusions about the content of items, the system must have an understanding of the world it is dealing with which can be expressed in terms of its entities, their possible inter-relationships and how to reason about them. As MMISs develop the semantic analysis of media will become more profound, so inferencing techniques should be general and extensible.

Programming a computer to make useful inferences is extremely difficult for complex media such as image, speech and text, (when the latter is considered as natural language rather than just a collection of keywords). It is tempting to write ad hoc programs that draw the desired inferences within an application domain. Such programs lack generality — a prerequisite for a MMIS — and extensibility, so that they cannot easily adapt to changes in people's understanding of a domain nor can their conclusions be improved. The problem is that the knowledge incorporated into ad hoc programs is hard to pinpoint. The aim of Artificial Intelligence has been stated as

> "to write down descriptions of the world in such a way that an intelligent machine can come to new conclusions about its environment by manipulating these descriptions" [BL85]

In fact, for an intelligent machine to be useful its conclusions only need to be new in the sense that they are not supplied by an operator. To achieve automatic inferencing without 'hard-wiring' knowledge into algorithms and code, three things are required [Ric83]:

- a semantic representation formalism;
- some means of making conclusions;
- a way of gathering knowledge.

Unfortunately, the third requirement usually receives the least attention as it requires expert programmers to sit down with experts in the appropriate field to thrash out a model. The best solution is probably naïve interaction between the application expert and the computer but this is difficult to achieve without more agreement on representations, inferencing techniques and human-computer interaction.

A number of criteria can be used for evaluation of different semantic representations [RD88, Woo75]: it should be possible to express anything we like — *expressive adequacy*; if a semantic representation permits all inferences that could have been made from the original data then it has *logical adequacy*; if the knowledge is incomplete, as is generally the case, we should still be able to make useful conclusions[2]; our selection of *primitives* should reflect the entities and relationships of the world we are interested in.

For the most part, semantic representations fall into four categories — logic, frames, rules and semantic nets — with much correspondence between the different approaches (see Section 4 in [OD91]).

The aim of the Multimedia Group at Manchester University is to advance the techniques used for the management of multimedia information by drawing together the disciplines of image analysis, database design and Artificial Intelligence. A large part of this thesis is devoted to the description and discussion of our second prototype, *MMIS-2*. The aim in building MMIS-2 has been to investigate the management of simple types, text, raster images and documents containing all three within a sound extensible framework for future research. MMIS-2 exhibits automatic content retrieval for images, direct querying, query-by-example and query-by-association (or hypermedia).

[2]Note that traditional databases often find incompleteness difficult to manage.

Chapter 2 of this thesis describes the major fields which should be examined before attempting to design a MMIS.

Chapter 3 outlines the requirements for MMIS-2 and the tools and techniques which were chosen to meet them.

Chapter 4 gives an overview of MMIS-2's operation.

Chapter 5 describes the semantic data extracted from images during interpretation, how that data is represented using semantic nets and how the semantic nets are stored.

Chapter 6 introduces the reader to the querying facilities of MMIS-2, which centre on Prolog, and presents detailed examples of queries.

Chapter 7 discusses the results of the initial implementation of MMIS-2.

Finally, Chapter 8 contains concluding remarks and proposals for further work.

Chapter 2

Related fields

The management of multimedia information is a multi-disciplined task which therefore needs to be tackled by a multi-disciplined team. When considering the storage of items in MMISs, designers and implementors need to draw on the fields of traditional databases which handle simple types and more advanced databases that deal with text, images, composites and object-orientation. When generating content descriptions for automatic content retrieval then the problems of eliciting, representing and storing knowledge become relevant. Finally, querying requirements lead to the studies of mixed-media querying, uncertainty, hypermedia, browsing and deductive databases. This chapter introduces the reader to the major fields touched on by MMIS research, especially those which are relevant to the chosen requirements of MMIS-2, which are in turn described in the next chapter.

2.1 Traditional Databases

> "A database system is essentially nothing more than a *computerized record-keeping system*." [Dat86]

Traditional databases were motivated by the desire to computerise customer and employee records, scientific data and so on. The intention was to improve paper record-keeping by:

- allowing the storage of a greater volume of data;
- speeding access to the data via query languages that provide automatic indexing;
- improving data integrity and consistency;
- reducing redundancy;
- allowing wider, shared access subject to security controls;
- simplifying the insertion and deletion of data;
- making data formats easier to change.

Data describing a particular application is typically entered manually, fact by fact. The data objects stored and manipulated are strings and numbers and the system knows a little of their semantics, such as how to make lexical or arithmetic comparisons.

The majority of databases used in industry are still based on the network or hierarchical models— but then the most popular languages in industry are Cobol and Fortran — whereas most new products are now relational and come with programmatic and interactive query languages. Some of these new products offer storage of manually tagged 'Binary Large OBjects' (BLOBs [She90]) — arbitrary byte-streams such as images treated as black boxes by the system.

Traditional databases do however usually come with high-level design tools (eg. Entity Relationship modelling) and data definition languages (eg. data dictionaries). In addition, relational databases have a sound and complete mathematical basis — algebra, calculus and functional dependencies [KS86].

Traditional database technology is inadequate for the implementation of all components of the ideal MMIS but may be appropriate for certain tasks within the system such as storage and simple querying of semantic representations.

2.2 Text Databases

Of course, not all of the data from different applications takes the form of formatted strings and numbers; much of it comes as paper documents. Further, documents contain a mixture of text, real images and diagrams [Sie82]. The simplest of these document components to computerise is text, hence the proliferation of word-processors, leading to text databases. These range from tagged storage (tags such as author, creation date, etc.) to automatic content retrieval based on word occurrence and position [Aal89] or pattern matching [FC84]. The text is usually stored without formatting information such as the control characters or logical and layout directives used by word processors and document formatters.

2.3 Image Databases

Image databases have received much attention in their own right because of their relevance to image processing, remote sensing, medical imaging, *et al.* The simpler systems are interesting as extensions to traditional databases [ZWW87, Kun], while the more complex ones offer insights into content retrieval of images.

REDI (RElational Database for Images [CF80]) was designed for Landsat images to allow simple query-by-example. Roads, rivers, cities and meadows are identified and stored as relations — eg. roads and rivers are represented by line segments, one per tuple. Queries take the form "Find me the name of the river I just pointed to" or "Find me pictures of cities with a similar road pattern to this one" and are entered in tabular fashion. However, much information is already contained in image tags — names of roads and cities, world coordinates, etc. — which makes interpretation easier.

The remote sensing image database described in [Nag84], uses five-digit chain codes[1] to encode line segments of rivers, roads, etc. These are stored in a relational database — with chains grouped into arcs and arcs into features — for use in similarity retrieval. Chain codes are also proposed for similarity retrieval in an image database described in [GL86]. They are concatenated and compressed to yield a string for pattern matching.

The purpose of Picture Archiving and Communication Systems (PACS) is to store large numbers of digital images while minimising access time and allowing standard communication [RWTP86, PCT+86]. The images are stored with tags for the use of experts. PACS evolved in the area of hospital information systems; for example they may be used by a radiology department, where images are produced on film, video and computer (X-ray, ultrasound scan, computer tomography and NMR). At present, X-rays are filed in envelopes and ultrasound scans on videotape, with the obvious potential for loss or damage. Each image type also requires a different display device — light box, TV or monitor — whereas PACS allow them all to be integrated electronically in one medium.

2.4 Multimedia Information Systems

These are systems which can be seen as contributing to the development of the ideal MMIS.

Minos [CTH+86] was designed for the creation, editing and archiving of hierarchical documents. These are displayed, a page at a time, on a workstation screen, successive pages being displayed by clicking a mouse button. Pages may also be flicked automatically to allow simple animation, or they may be overlaid as transparencies. All multimedia data types are provided; notably, voice may be linked to a section, (comments), or to the turning of a page, (narrations). A document is implemented as a directory hierarchy with one file for each part. These are archived on a file server as concatenated, compressed files, along with a 'structure descriptor' containing the details necessary for reconstruction.

The Esprit project 'Multos' [Tha90] is similar in scope to Minos, with the addition of content retrieval for images [CR87, RS87] and the 'Conceptual Structure Definition' (CSD) [BRT88]. Image segments are classified according to their properties and the images themselves are then classified according to the classes of their segments. Image segments are extracted from the output of a drawing package rather than real images, side-stepping the problem of image analysis in favour of exercising higher-level techniques. The CSD [Lut89] attempts to enforce a semantic structure on composite items so that these too can be classified. It is likely, however, that this technique of classifying all the way from image segments to composite items makes the system too tightly constrained by application, which in this case is the office document. Queries are probabilistic, eg. "Find documents containing a picture of a car with certainty $> 75\%$". Members of the project team have also investigated query optimisation [BRG88, BGR88].

[1] Chain codes are concise descriptions of a string of pixels — each digit representing the movement from one pixel to one of its eight neighbours.

The Orion object-oriented database has been used as a foundation for a MMIS that caters for bit-mapped images and voice [WK87, WLK87, WKL86]. The system also uses application-specific knowledge based on semantic nets.

The aim of ts1 [Cap87] is "to provide transparent access to all of a community's on-line information". Here, 'Information Units', (IU's), are the data carriers; these multimedia objects may refer to each other, or may contain other IU's to form composite items. The main interface technique provided is browsing.

2.5 Object-Oriented Databases

Object-oriented programming [OD91] is now firmly established in the software world; it has evolved from structured programming (Fortran, Pascal, Algol, *et al*) via 'data abstraction' (Simula, Modula, Ada, *et al*). The central notion is to model the real-world entities of an application directly in a program as *objects* which have similar attributes and capabilities. The intuitive programming style can improve software engineering by bringing design and development closer together, encouraging code reuse and making programs more robust.

When an object-oriented application calls for the storage of information, it is natural to carry the object-oriented paradigm over to the database; the database becomes a warehouse of objects which can be pulled into a program and used immediately. The simplest object-oriented database, which is nonetheless very useful, just provides object persistence via the assignment of a unique identifier to each object for direct access (as in Eiffel, see Chapter 30 in [Mey92]). Another option is to allow queries about object attributes to be issued from within a program (eg. via SQL strings in [AHS89]); here, object caching, indexing and query optimisation need to be carefully controlled to avoid retrieving and interrogating each object in turn. Another style of object-oriented database imposes the new paradigm on traditional textual query languages such as SQL, as in IRIS [FBC+87].

A good compromise between retrieving program objects and imposing the object-oriented paradigm on traditional data is to separate most of the data which is used for query resolution from the objects themselves; this allows the designer to concentrate on efficient query resolution without having to retrieve any complex objects. However, this will usually mean that attributes which are calculated dynamically, by object methods for instance, cannot be included in the optimisation. As will be seen in subsequent chapters, the separation of objects from the data describing them is the philosophy behind the design of MMIS-2.

As yet, there are no standards for object-oriented programming or databases, nor is there any agreed mathematical foundation. Instead, general guidance is provided for programmers [Mey88, DS89] and database designers [Hug91].

2.6 Deductive Databases

When the body of knowledge within an Artificial Intelligence system grows large, there is an inevitable intersection with database techniques [Gra84]. The resulting systems are often referred to as *deductive databases* and are characterised by the storage of semantic representations and the provision of query languages that are more expressive than relational algebra.

The most popular approach to building a deductive database uses some form of Prolog at the top level. There are a number of reasons for this; firstly, pure Prolog already has the notion of an internal database of knowledge, although this is manipulated in RAM rather than on disc; since Prolog is a programming language it allows developers to add layers of code directly to the database which hide the details of data management from the user; Prolog comes complete with both an interactive, textual query language and a programmatic query language; finally, there is a natural mapping between a relational tuple and a Prolog 'ground fact' (the head of a Prolog clause that contains atomic arguments only) — for example, consider a tuple representing a football match as part of a 'Match' relation

date	venue	team1	team2	score1	score2
850829	Highbury	Arsenal	Everton	2	1

and the corresponding Prolog fact

match(850829, highbury, arsenal, everton, 2, 1).

Note that Prolog normally manipulates single tuples rather than sets of tuples as in most databases; this is due to the assumption that one solution to a query is as good as any other and can be found far more quickly than all solutions. Relational databases, in contrast, are optimised for the retrieval of all solutions to a query — this can make it difficult to use them directly for the kinds of queries common in Artificial Intelligence applications which tend to involve complex navigation through a body of data.

Draxler [Dra90] identifies four characteristics of Prolog deductive databases which will be used to describe the systems below:

Loose physical coupling Prolog and the database are two distinct processes that communicate over some channel.

Tight physical coupling Prolog and the database are integrated into a single process.

Loose logical coupling The database language and the Prolog language are different.

Tight logical coupling Prolog is used for both database access and application program.

Notable examples of Prolog-based deductive databases include:

Datalog [CGT89] This relies heavily on the mapping of Prolog fact to relational tuple described above, resulting in a loose logical coupling; this coupling is provided by a restricted Prolog that allows queries to be recursive while preventing infinite recursion (relational algebra does not allow recursion). However, general Prolog terms cannot be stored and the user must use set-based retrieval.

Draxler's research This offers tight logical coupling using the common Prolog predicates for finding sets of solutions (`findall, bagof` and `setof`) and is being implemented to allow loose physical coupling to any relational database. The Prolog queries are pre-processed by reformulating those parts which refer to tuples in the database in the more appropriate relational algebra. Users must use set-based retrieval for database tuples and such tuples may only contain atomic arguments.

Quintus [Qui] and BIM-prolog [BIM89] Offer loose logical and loose physical coupling to relational databases.

Megalog [HBD90] This offers tight logical and tight physical coupling so that database tasks can be completely hidden from the user, who just sees Prolog. Tuples may contain any Prolog term and when this term is a rule, it may be fired within the database. There is also a relational mode for use when efficiency is paramount; this permits the storage of Prolog terms although these cannot be keyed nor can they be executed within the database.

2.7 Hypermedia

Hypermedia has applications mainly in education and is the field in which the term 'multimedia' is most often coined. The most common mechanism is to collect sets of multimedia objects together using some form of named or un-named link to form a 'web' of objects. Hypermedia is a development of 'hypertext' [Con87], and both owe their origins to the tendency of humans to perform tasks such as information access or discussion in a disordered fashion — hopping between sources and ideas as necessary. Unfortunately, for a large database of items, it is very easy for a user to become disorientated — or 'lost in hyperspace' — and data management becomes very difficult. Therefore, it is suggested in [INJ+88] that webs should only be used for browsing a small number of items that have been retrieved using other methods, especially since authoring tools and query facilities are usually not provided

One large hypermedia system, Intermedia [Mey86, SZ87], contains basic items such as text, images and lists of historical events linked into webs which are stored as Ingres relations. The user sees a web of icons representing the stored items and may browse around it using a mouse, zooming in on interesting items. Other web-based systems are described in [LYU91, Kir91].

Apple UK's 'Interactive Multimedia' [App88, Pri88] is a hybrid system of video discs, CD-ROM's and other peripherals controlled by a Macintosh running Hypercard, (a hypertext development comprising stacks of ‘cards’ containing text, images and process control commands).

Other approaches to hypermedia include the ‘electronic book’ [And91] and ‘interactive video’ [Pen91].

2.8 Summary

The seamless integration of all the technologies and research which have an impact on MMISs is still some way off but offers great rewards for users seeking access to multimedia information. All such technologies need to be examined by designers seeking to build an ideal MMIS, otherwise the goals of generality and flexibility are unlikely to be met and the full potential of MMISs will be missed. MMIS research requires close cooperation between people from varied disciplines and backgrounds, despite the steep learning curve which will usually result!

Chapter 3

Requirements of MMIS-2 and their Solution

The aim of the Multimedia Group at Manchester University is to investigate the management of raster images, text, simple types and documents (composite items containing all three) with an initial emphasis on the automatic content retrieval of raster images. This has resulted in two prototypes, MMIS-1 and MMIS-2. MMIS-2 is the main subject of the practical parts of this thesis; MMIS-1 is described in [GOC+91, O'D90]. The group proposes that the core of a sound and extensible MMIS can safely be separated from the provision of tools for the novice user, and have therefore chosen not to provide such tools.

The detailed requirements chosen for MMIS-2 are introduced below followed by a discussion of the tools and techniques chosen for implementation. The discussion is based on the generic architecture shown in Figure 3.1. Here, the term *medium instance* is used to refer to an item which is destined for automatic interpretation.

In the generic system, the user interacts with a user interface that gives access to an input part for entering medium instances, a query part for formulating queries and an output part for presentation of query results and production of hard copy output. Medium instances are stored in the database for access by the interpreters and retrieval by the user interface's output part. An interpreter is provided for each medium and interpretation results are stored in the database. The query engine resolves users' queries according to the interpretation results. The query may be translated into a form more suitable for fast resolution.

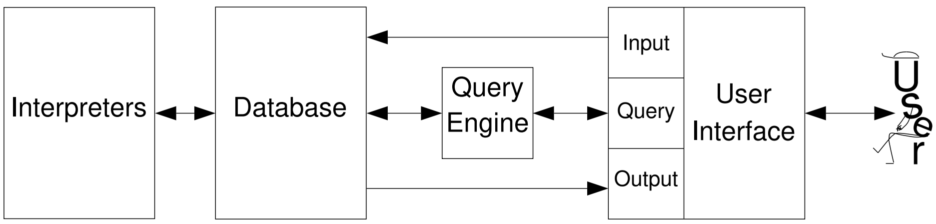


Figure 3.1: An architecture for a generic MMIS

3.1 Requirements

The user requirements for MMIS-2 are a subset of those for the ideal MMIS described in Chapter 1. They are:

- entry of medium instances, initially 8-bit monochrome images and text;
- automatic content retrieval of images and their parts;
- mechanisms for query-by-association, query-by-example and direct querying;
- a textual query language which can refer to the content of all media;
- query results containing medium instances and simple types;
- ranking of items in query results;
- entry of hypermedia links;
- techniques for browsing items returned by query and hypermedia links;
- retrieval of medium instances for display on demand rather than *en masse*;
- reasonable but not necessarily tiny query response times.

From the architectural point of view, the system must be general and extensible so that it is not constrained to a specific application; it must be modular to allow parts of the system to be prototyped without established parts being affected and it should be possible to run system modules on separate machines so that more than one processor can be utilised.

It was thought likely that the interpretation of medium instances would be too time-consuming to be performed entirely in response to queries (post-query); accordingly, semantic representations should be generated largely pre-query and stored in the database. The representation formalism used should ideally be application independent, medium independent, storage efficient and access efficient; it should be possible to express any information about a medium instance (expressive adequacy); it should be possible to deduce anything from the semantic representation that could have been deduced from the original data (logical adequacy); finally, such formalisms should be easy to generate. No formalism will satisfy all these ideals. In particular, complete logical adequacy is not feasible for complex instances such as images (consider the task of asking a person to describe a picture of a room exhaustively, concisely and without repetition); therefore the semantic representations need only provide a sufficient body of information for rich querying.

The interpreters used to generate semantic representations should be broad-based to allow the system to be tailored to new applications. Interpretation may be hierarchical — for example, document interpreters can extract images and text and image interpreters can extract their parts — so this must be accounted for in the architecture.

Finally, the system needs to be tested with an application that offers easily accessible data and knowledge, images which are sufficiently complex and variable to pose an interesting problem for automatic interpretation and from which rich semantic representations can be derived, and potential for interesting documents and text.

3.2 Solutions

The module distribution, storage and display requirements led to the use of a network of Sun workstations with a central file server and the use of the server-client paradigm for remote communication between modules. The need for generality, extensibility and modularity led to the choice of C++[Str86], a *de facto* standard object-oriented system language, as the main enabling language. Note that the architecture is concerned more with functionality than speed.

The semantic net was chosen to describe images because:

- it is one of the principle knowledge representations identified in Artificial Intelligence research;
- it is highly expressive;
- it can be stored in a simple fashion (as a set of tuples);
- it lends itself to querying based on logical resolution, relational algebra and browsing;
- it allows the storage of annotated hypermedia links;
- it can be used to store hierarchical data such as documents.

The requirements of medium instance storage and semantic representation storage would be difficult to satisfy efficiently in a single-module database — semantic representations are complex structures of facts, medium instances are long byte-streams. If the management of semantic representations can be separated from medium instances then the latter do not have to be retrieved by the Query Engine, the Query Interface need only retrieve them when they are explicitly selected by the user and interpreters need only retrieve them pre-query — this all reduces data traffic and speeds querying. Therefore MMIS-2 has two database modules: the *Large Object Server* and the *Semantic Server*.

The Large Object Server caters for the insertion, deletion and retrieval of medium instances as arbitrary length byte-streams indexed by unique *instance identifier*; it is implemented as purpose-built C++ code which delivers whole byte-streams over communication channels, in preference to Sun's NFS which chops data into 8Kb packets [Cro90]. Each medium instance is manipulated in C++ as an object, while the byte-stream in the Large Object Server represents the object's encapsulated data.

The Semantic Server stores semantic nets describing images and their parts for access by interpreters and the Query Engine, and hypermedia links for access by the Query

Interface. Arbitrary sized semantic nets can easily be stored as a set of fixed-length tuples (see Section 5.2.2) and queried by tuple-matching. This suggests the use of Prolog for the Semantic Server, with the advantage of a built-in query language and Query Engine. This, in fact was the approach used in MMIS-1 [O'D90]; however most Prologs (and Prolog-based deductive databases) have problems when used as database as well as query language and Query Engine:

- Most Prologs can call *into* foreign languages but cannot be called *from* them; this leaves Prolog controlling a lot of the system — interpreters for instance — which is inflexible.
- Although Prologs such as SICStus [CW89] have internal databases, they are not disc-based, which is inappropriate for large databases. Further, in order to provide persistence, the database must be written out to a file and re-read for further processing which means that only one Prolog process can change the database at a time; if, instead, a single Prolog process is used to handle all database requests to the persistent file, those requests must be made via a purpose-built inter-process protocol which is likely to be inefficient and would take time to implement.
- Even Prologs such as Megalog that provide full disc-based database facilities still suffer from the problems of wanting to control the system because its predicates cannot be called from other processes.
- BIM-Prolog can be called from foreign languages but the version that was available during the implementation of MMIS-2 had shortcomings:
  - For efficiency, it used fixed size data tables whose sizes were allocated at start-up. Unfortunately this meant that it came to a halt when trying to load large programs or data sets, or executing complex predicates that generated a lot of stack and heap data. The only way round this was to quit and restart with larger tables.
  - Its default operation used non-standard syntax, as did all the documentation.
  - It had an unacceptable number of bugs.

The *Intelligent File Store* (*IFS* [Lav88]) is a hardware tuple database designed for very fast access to large sets of semantic net style structures, represented as sets of tuples and a software simulator is available free to academic institutions [LWW91]. IFS also offers a programmatic query language called from C, a tuple-matching paradigm similar to Prolog's and a commitment to the provision of relational algebra operations, which can be generated from Prolog queries to provide efficient set-based retrieval [Dra90]. For these reasons, the IFS software simulator was chosen as the core of the Semantic Server in MMIS-2.

As for the system's interpreters, the time taken to interpret medium instances is not critical because it happens pre-query and the user only inputs new instances occasionally.

The orderly management of all interpretation, including hierarchical interpretation, is provided by the *Interpretation Server* described in Section 4.2.

Despite the problems associated with using Prolog as database management system, it still offers:

- an interactive query language that is more expressive than relational algebra because of its use of recursion [CGT89];
- a tuple-matching paradigm which is ideal for implementing the Query Engine on top of the Semantic Server;
- a declarative programming language for additional post-query interpretation;
- querying of hypermedia links;
- querying in a set-based and a tuple-based manner to allow traditional database queries and the more navigational style of query common in Artificial Intelligence applications.

Therefore, MMIS-2 's Query Engine is written in Prolog, specifically SICStus 0.6 [CW89] which is available cheaply to academic institutions, is fast, reliable and can call into C (and therefore C++). The resulting Query Engine is a Prolog and C++ hybrid with loose physical coupling to the Semantic Server and tight logical coupling but which is not optimised for set-based queries; the intention is that the optimisation will in future be implemented using Draxler's techniques described in [Dra90].

The query interface is provided by a Sunview program — *Query Tool* — and provides:

- textual querying;
- presentation of mixed-media results;
- entry of hypermedia links;
- browsing of existing hypermedia links;
- ranking of results;
- retrieval of medium instances on demand;
- fast connection to the system.

The implementation of Query Tool is described in [Cro90] and examples of its use are given in Section 6.

Two editors are provided in MMIS-2: the Text Editor allows entry of text and the Image Editor allows the user to capture images from a live video display on-screen.

The application chosen to exercise the system is the description of deciduous trees; the images are of a set of leaves from a particular tree while the text describes name, collection site, collection date and so on. Section 8.3 describes how the system may be extended to contain ODA-like documents that describe a species in terms of nomenclature, textual description, and example images of leaves, fruit, bark and so on.

3.3 Summary

MMIS-2 has been designed with an extensible architecture for research, exercised initially by the management of simple types, text and raster images, with automatic content retrieval of the latter. Querying is by-association, by-example and by direct access to the semantic representation. Query results are a mixed collection of all types, ranked according to users' criteria.

Medium instances and semantic representations are managed separately, by the Large Object Server and Semantic Server respectively, with the semantic representations providing all the information for querying.

The system is used to describe the domain of deciduous trees.

Principle enablers of the system are:

- a network of Sun workstations;
- the client-server paradigm;
- the object-oriented language C++;
- SICStus Prolog;
- Sunview;
- the Intelligent File Store or IFS;
- the semantic net.

Chapter 4

A Description of the System

This chapter describes the operation of and the communication between the major modules of MMIS-2, as illustrated in Figure 4.1. All modules are written in C++ except for the Query Engine which is a C++/Prolog hybrid. The User Interface runs under Sunview. The description of the User Interface is from the point of view of a single user; however, multiple Query Tools, Editors and Input Clients may run simultaneously.

4.1 User Interface

The Text Editor is used to generate new text instances and takes the form of a text window activated from the Query Tool (Figure 4.2) by pressing the 'Create Text' button, as in Figure 4.3. The user edits the contents of this subwindow in the normal way; when the 'Create' button is pressed, the subwindow's contents are sent to the Large Object Server as a text instance and the Text Editor is replaced with a read-only text window representing the stored instance (Figure 4.4). The new subwindow's name stripe contains a *label* which is a symbol representing an instance's type and value. Labels are explained fully in Section 5.2.1.

The Image Editor is started from a shell window and displays the contents of a 24-bit frame buffer linked to a colour video camera. When the user selects the 'store' button from this window, the image is grabbed and an 8-bit monochrome version is sent to the Large Object Server. The returned image identifier is displayed for the user in the shell window. The Image Editor stays running for the grabbing of further images.

Once instances have been entered they must be interpreted and their semantic nets stored in the Semantic Server for subsequent querying. Currently, this interpretation is not activated by the editors, but by the user with the command

`input_client` $<$ *label_1* $>$ $<$ *label_2* $>$ ...

The Input Client sends the given labels to the Interpretation Server, waits until the Interpretation Server confirms that the interpretations are complete and exits, upon which the new semantic nets are available for querying. The detailed operation of the Interpretation Server is described later.

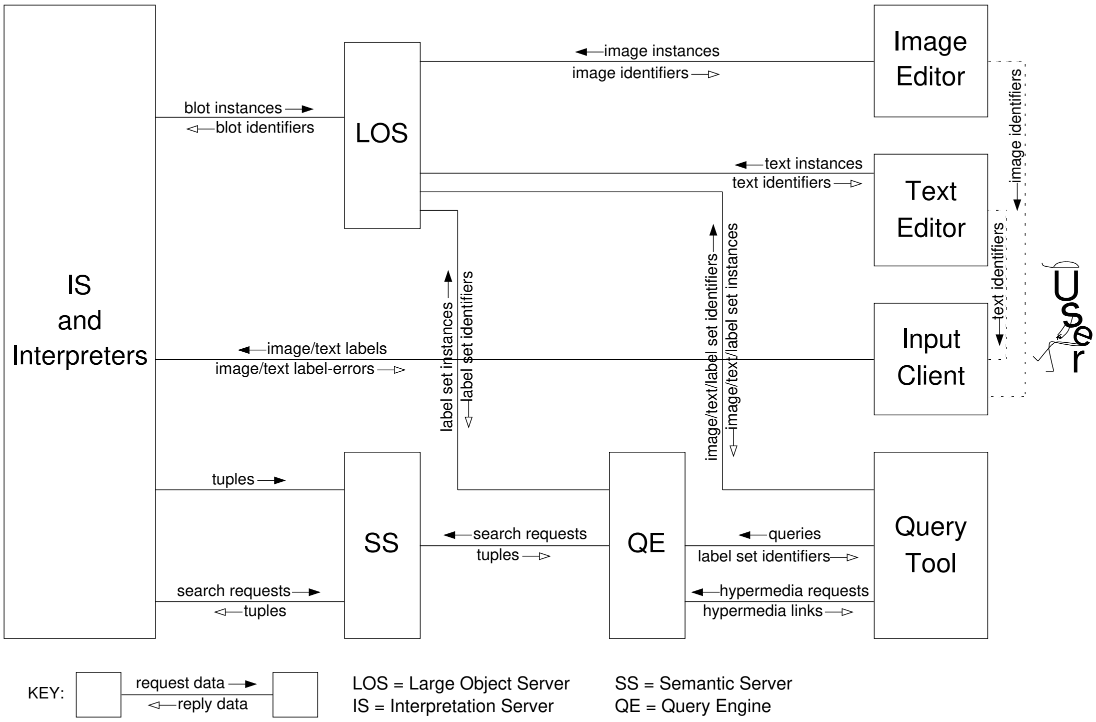


Figure 4.1: The architecture of MMIS-2

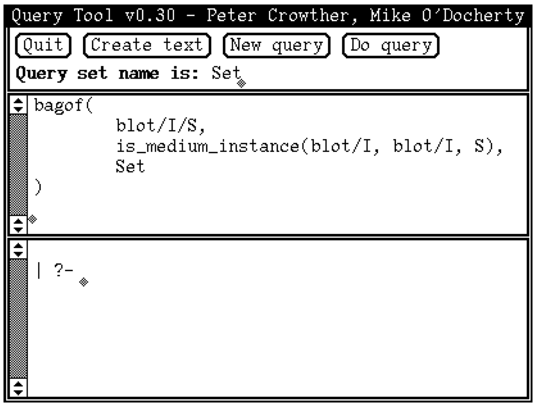


Figure 4.2: The Query Tool

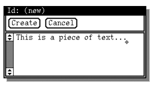


Figure 4.3: A Text Editor

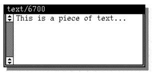


Figure 4.4: A text instance

Since Sunview and SICStus Prolog both insist on controlling program flow, running the Query Tool and Query Engine as a single process is impossible. Instead, they communicate via a textual protocol described in [Cro90]. What the user sees is a Query Tool with two text windows — the upper window is used for editing queries while Prolog diagnostics and error messages appear in the lower window. In fact, any Prolog query can be issued in the lower window, but the Query Tool will not capture or display the results.

When the 'Do Query' button is pressed, the query is sent to Prolog which parses it into Semantic Server search requests and formats the results into a *label set*. A label set is a ranked set of labels (see Section 6.2). The label set is stored in the Large Object Server and its identifier is sent back to the Query Tool — this mechanism is more efficient than passing the label set (which can be a large complex structure) back to the Query Tool using the textual protocol mentioned above. Finally, the Query Tool retrieves the label set from the Large Object Server and displays it for the user, retrieving the medium instances it contains on demand.

Users may also retrieve hypermedia links via the Query Tool and insert new ones; links are stored in the Semantic Server in a format similar to the semantic net (see Section 5.2.2). Querying and hypermedia examples are given in Chapter 6.

4.2 Interpreters and the Interpretation Server

The detailed operation of the Interpretation Server is illustrated in Figure 4.5. Overall, the Interpretation Server receives interpretation requests from Input Clients, spawns interpreters and passes back any errors.

An Input Client makes an interpretation request by sending a sequence of labels to the Interpretation Server, each label representing a medium instance to be interpreted. Any labels that do not specify a medium instance are returned immediately with a 'No such interpreter' error attached. The Input Client waits for a period and asks the Interpretation Server which labels have been finished; the Interpretation Server replies with a sequence of `label_errors` — a `label_error` is a label with an associated (possibly NULL) error. The Input Client keeps making `whats_finished` requests until all its labels have been returned. An Input Client can make multiple `interpret` requests and collect results at any time.

Interpreters are processes which are created and destroyed by the Interpretation Server as needed. When an Interpreter receives an `interpret` request from the Interpretation

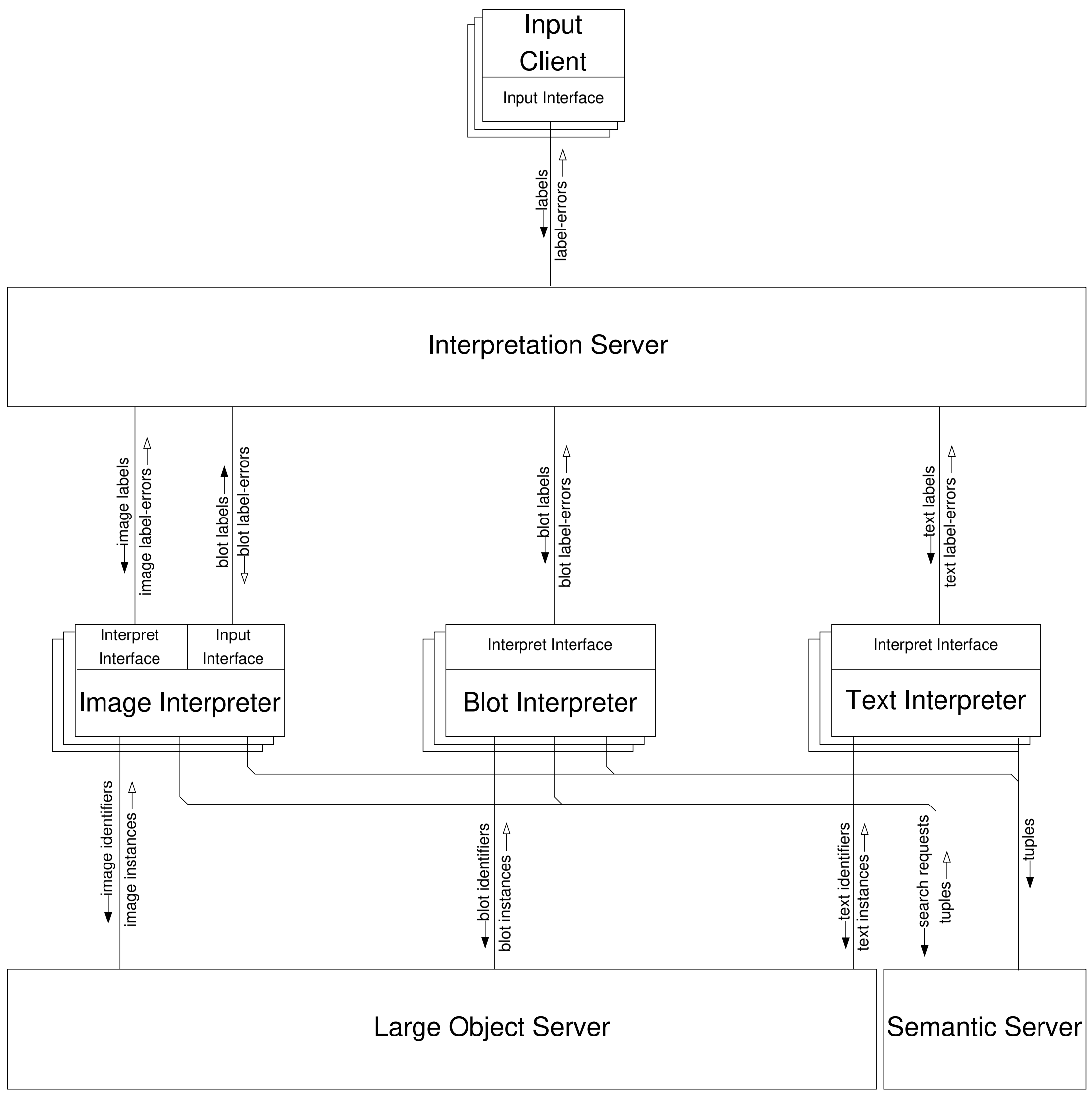


Figure 4.5: The Interpretation Server

Server, it responds with a number specifying how many labels it has accepted and then proceeds with the interpretation of the corresponding instances; when it's finished it sends the resulting `label_errors` back to the Interpretation Server via a `finished` call and waits for more labels or a `die` request — in the latter case, it cleans up, unregisters with the server and exits.

For each label an Interpreter receives, it checks whether the corresponding medium instance already has a semantic net in the Semantic Server; if it does, then no further action is taken; otherwise, the medium instance itself is retrieved from the Large Object Server, interpreted and its semantic net stored in the Semantic Server. The Image Interpreter is hierarchical: it extracts *blots* (image segments) from the retrieved image, stores them in the Large Object Server and dispatches the resulting labels to the Interpretation Server via its input interface; once the `label_errors` are returned (see the Input Client description above), the Image Interpreter stores the image-specific portions of the semantic net in the Semantic Server and proceeds to the next image. Note here the idea of a medium instance (a blot) being created by an interpreter rather than a user.

In order to prevent deadlock, the Interpretation Server will not create more than a maximum number of interpreters (fixed at compile-time), unless some labels of a given medium are awaiting interpretation and no interpreter currently exists for that medium — if this were not done then an Image Interpreter could wait forever for a blot to be interpreted when no Blot Interpreter was running and the maximum number of interpreters had already been spawned. Also, in order to allocate resources fairly, the Interpretation Server will not create more than a maximum number of interpreters for a given medium.

The Document Interpreter currently has no effect. The Text Interpreter stores a minimal semantic net recording only the fact that a particular text instance exists.

Controlling interpretation with the Interpretation Server has a number of advantages when compared to a set of distinct, non-cooperating interpreters:

1. Incremental development of interpreters — eg. develop the blot interpreter, then the text and image interpreters, then the document interpreter.

2. Separate development of interpreters — changing the derivation of some part of a blot's semantic net requires the re-interpretation of all blots, but not necessarily all images.

3. Interpreters can be run on different machines, thereby making better use of resources. Currently this is not done — all interpreters run on the same machine as the Interpretation Server — but could be accomplished with the same protocol.

4. The authors of editor/input programs do not have to worry about how medium instances will receive interpretation, they simply have to dispatch labels to the Interpretation Server from anywhere on the network and wait for the response. An input interface is a C++ object that registers with the Interpretation Server when it is created, unregisters when it goes out of scope, and has two methods: `interpret` and `whats_finished`.

5. The author of an interpreter program simply needs to declare an interpret interface for itself and an input interface if it is hierarchical. The interpret interface registers with the Interpretation Server when created and unregisters when it goes out of scope. In addition, three functions must be filled in: the `number_to_accept` function is used to ensure that an interpreter doesn't take too many labels when there might be idle interpreters elsewhere and reduces the effect of an interpreter crash (in which case the input client gets the labels back with 'Interpreter bombed' errors attached); the `interpret` function is passed a sequence of labels, performs the interpretations and returns a sequence of `label_errors`; the `author_clean_up` function is called when the interpreter is told to die by the server so that it can perform housekeeping — such as unregistering with the Large Object Server and Semantic Server — before exiting.

6. Network communication details are hidden from the authors of editor/input programs and interpreters.

7. While editor/input programs can come and go, interpreters persist until killed by the Interpretation Server; this is particularly useful if an interpreter takes a long time to start up.

8. Making child interpreters independent of their parents means that they can be shared. For example, blots don't have to be generated by an 8-bit-monochrome-image interpreter, they could come from a colour image interpreter, a user's sketch interpreter, and so on.

The Interpretation Server provides the above benefits with virtually no effect on system performance.

4.3 Database

The calls which can be made to the Semantic Server are client-server versions of the calls that can be made to the IFS simulator [LWW91], except that each set of stored tuples (or IFS 'class') has an associated descriptor specifying the contents of each tuple in terms of labels, floats and integers — see Section 5.2.2.

If a medium instance is repeatedly sent to the Large Object Server then the same identifier is returned to the Input Client each time, while the instance's data is stored only once[1]. This approach makes the semantic net used to store interpretation results somewhat more complex (see Section 5.2), but has the advantages that:

- the amount of data in the Large Object Server is reduced;

[1] Conceptually this requires that an incoming medium instance is compared to all other instances in the Large Object Server; in fact, a hash code is quickly generated from the instance and byte-by-byte comparison is only needed when this clashes with a previously encountered code.

- if a blot occurs in more than one image then it only needs to be interpreted once and the resulting semantic net is shared between the images — a similar argument could be applied to the inclusion of an image in more than one document;
- if a query issued to the Query Tool generates a label set that has already been stored, then it is not re-stored.

4.4 Summary

MMIS-2's Text Editor and Image Editor cater for the creation of new text and image instances which are sent to the Large Object Server in return for an instance identifier for each. If a medium instance is sent to the Large Object Server on more than one occasion then the same instance identifier is returned each time while the medium instance is only stored once.

Input Clients send labels representing medium instances to the Interpretation Server which queues them for interpretation. The Interpretation Server organises the spawning of interpreters which in turn generate semantic nets and store them in the Semantic Server. The Interpretation Server sends `label_errors` back to Input Clients to indicate success or failure of the requested interpretations. Image Interpreters are hierarchical in that they extract new blot instances from images and send their labels to the Interpretation Server via their own Input Clients.

As soon as a medium instance has been interpreted its semantic net is available for querying. Queries are entered in a Query Tool which calls on the Query Engine to resolve them. Ranked results are returned to the Query Tool as label sets for display. Medium instances in label sets are displayed symbolically and are retrieved only when selected using the mouse. Hypermedia links may also be stored and retrieved via the Query Tool.

Chapter 5

Semantic Data and its Representation

Deciding what facts to store in knowledge representations is a difficult task. It involves knowledge of the minimum requirements for describing a medium, what sort of descriptions are meaningful to users and what it is reasonable to try and deduce. This is exacerbated by the problem of whether deduction should be pre-query, post-query or lazy. There is also the question of whether knowledge is application-dependent — eg. horse chestnut leaves are palmate — or application-independent — eg. how to calculate the area of an image segment from its boundary. The end result has to be a compromise, but there is a general underlying principle: low-level data from medium instances should be refined as much as possible to provide data that is more helpful to users. Note that the aim of such refinement is not to produce a canonical description [Woo75]. For example, given an image of a tree, it might seem worthwhile to try and collapse its semantic representation to the single comment "Tree". There are two reasons why this should not be attempted:

1. it is generally impossible;
2. it loses all lower-level information such as the shape and colour of the leaves, the tree's species, etc.

Therefore all levels of data should be retained, which calls for a flexible representation which in turn is easy to store and quick to access: the semantic net.

This chapter gives a general description of the semantic data derived from images and their parts and then goes on to describe the representation of that data as semantic nets; finally, it describes how the nets are stored in the Semantic Server.

5.1 The Data

A typical leaf image is shown in Figure 5.1; it contains a number of leaves on a plain background, taken from a single species of deciduous tree, along with a *scale marker*.

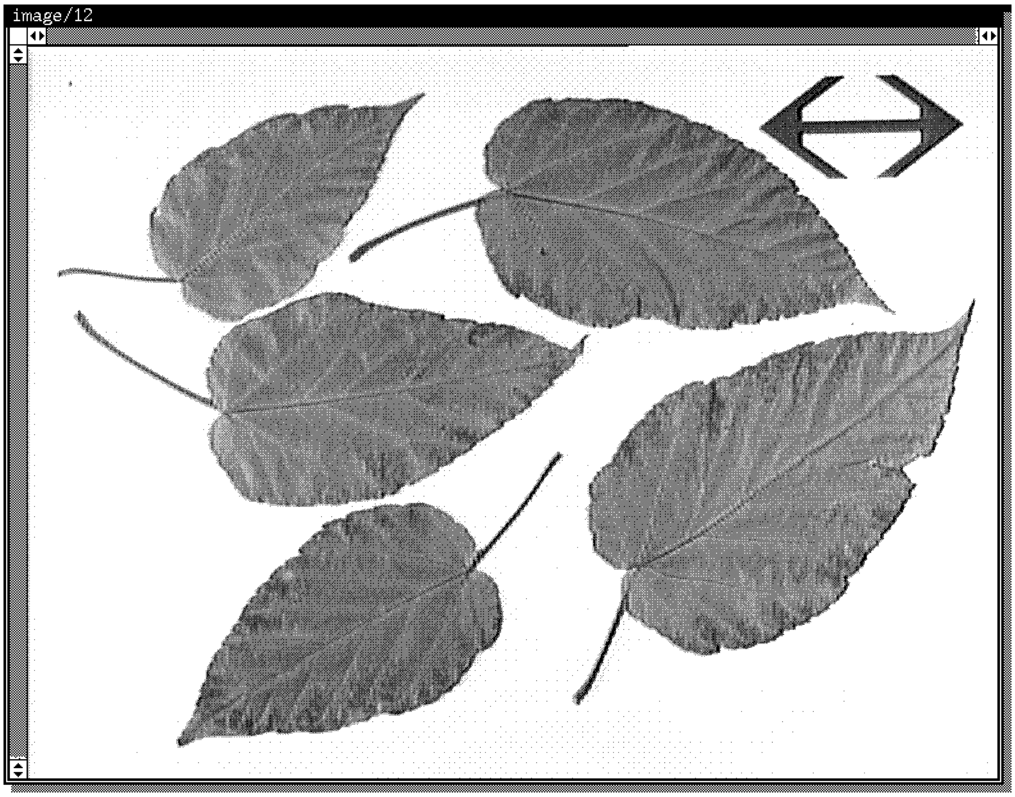


Figure 5.1: An image of deciduous leaves with scale marker

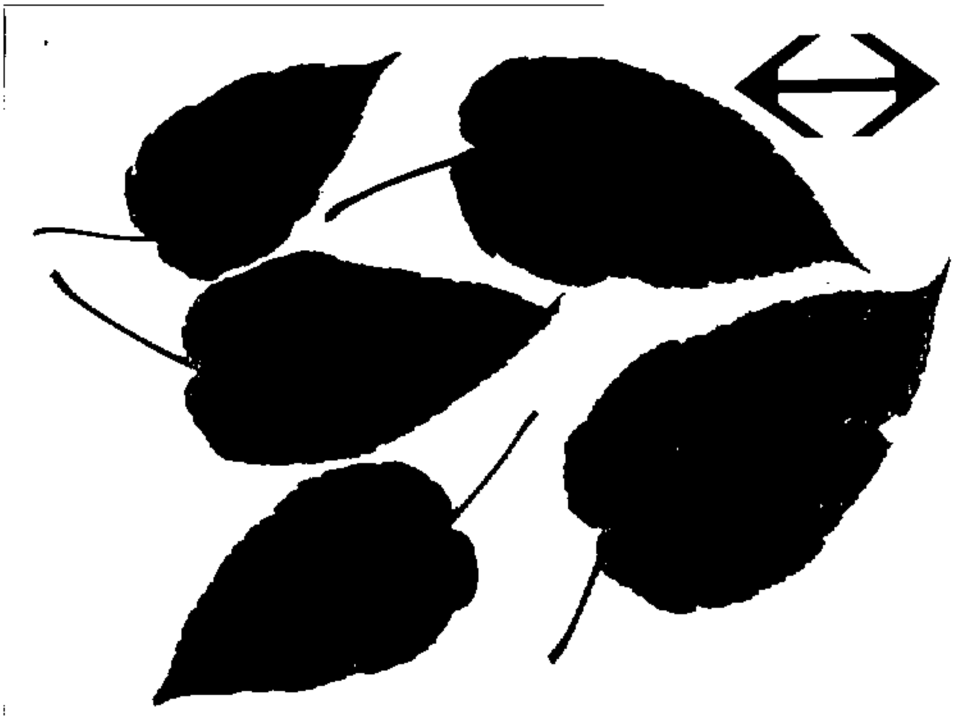


Figure 5.2: A thresholded image

A scale marker is an easily identifiable object of a known size (4 cm long in this case) that is used to standardise scalar measurements across images. Leaves, scale markers and noise are separated from the background of an image using a standard thresholding technique, to yield a binary image such as that in Figure 5.2; this new image is then scanned for connected regions of black pixels, or *blots*, and each blot is stored in the Large Object Server as a new medium instance[1]. The resulting blot labels are dispatched to the Interpretation Server as described in Section 4.2. Each blot is identified as leaf, noise or scale marker by the image interpreter: a blot is considered to be noise if it is smaller than 750 square pixels in area; it is considered to be a scale marker if its aspect ratio, circularity and transparency lie within predefined ranges; any other blot is considered to be a leaf.

Interpreters describe objects (images, blots, leaves, etc.) in terms of *structural relationships* to other objects and *attributes*, which have a name and a value. The attributes and structural relationships derived for the various types of object are described below. No attributes are calculated for scale markers or noise.

[1] In fact, a chain code representing the blot's boundary is stored.

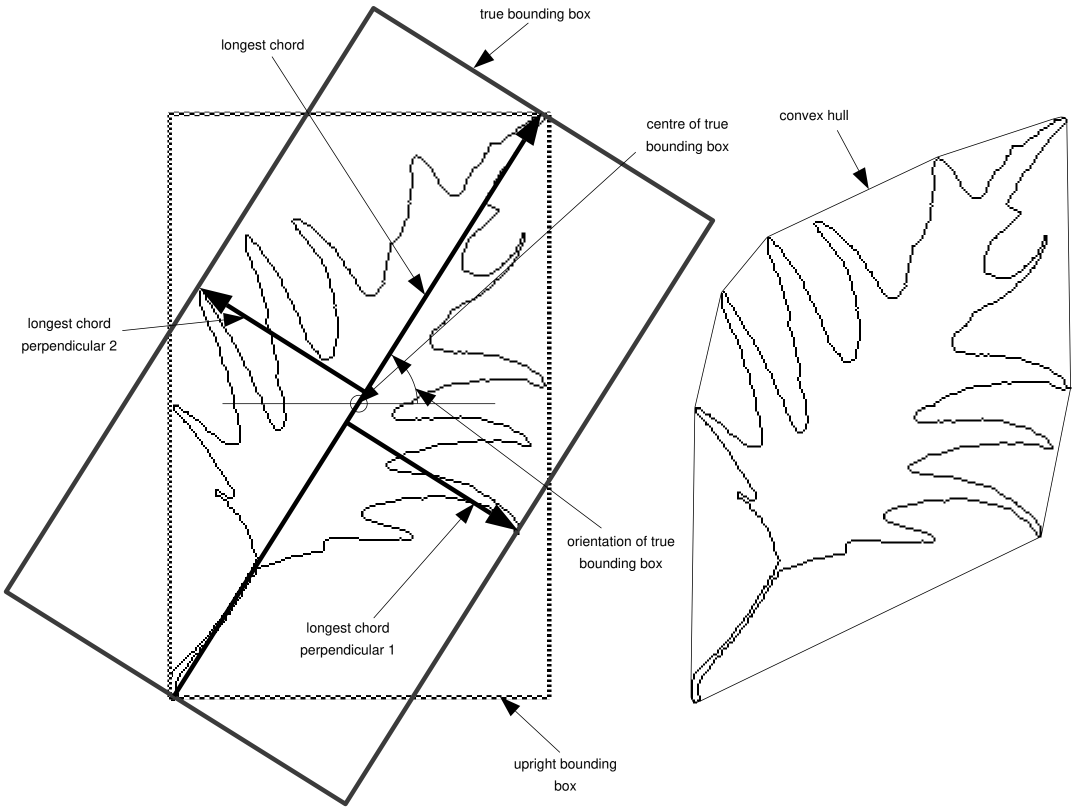


Figure 5.3: Aspects of a blot that are expressed as attributes

5.1.1 Blot Attributes

The following properties of blots are described using attributes and are illustrated in Figure 5.3:

Longest chord The longest line that can be drawn between any two points on the perimeter, defined by its end points.

Longest chord perpendiculars The two longest perpendiculars that can be drawn from the longest chord to the perimeter, one on each side. Defined by their end points.

Upright bounding box The minimum *upright* rectangle that will fit round the blot, defined by its height and width.

True bounding box The minimum rectangle that fits round the blot, defined by its four corners, its centre and its orientation to the x-axis.

Area

Perimeter The length of the perimeter.

Hull area The area of the blot's convex hull (analogous to an elastic band stretched round the perimeter, as in Figure 5.3).

Length The length of the longest chord.

Width The combined length of the longest chord perpendiculars.

Aspect ratio $\frac{\text{Length}}{\text{Width}}$

Transparency The ratio $\frac{\text{Hull area}-\text{Area}}{\text{Hull area}}$, which gives an idea of how incised a blot is; transparency is minimum (0) for a convex shape such as 'O' and maximum (1) for a skeletal shape such as 'T'.

Circularity The ratio $\frac{4\pi\times\text{Area}}{\text{Perimeter}^2}$, which gives an idea of how convex a blot is; it is minimum (0) for a line and maximum (1) for a circle.

Irregularity Irregularity is the quantity

$$\frac{1}{N}\left[\sum_{i=1}^{N}|\Delta a| - 2\pi\right].$$

where N is the number of pixels around the perimeter and Δa is the angle change in moving from one pixel to another; it is used to describe the unevenness of a blot's perimeter.

Convex maxima count The number of major convex extremities on a blot's perimeter [LIX90].

Concave maxima count The number of major concave extremities on a blot's perimeter.

Class Each blot is given a shape classification based on clustering in a multi-dimensional attribute space [IX91]. This is used as an aid for query-by-example. As a simple example of clustering, consider the two-dimensional attribute space illustrated in Figure 5.4. To train the classifier, a large number of blots are plotted and the space is automatically partitioned into a number of disjoint classes. Each class represents a good cluster of data points and each is characterised by its centroid. Now, when a blot is presented to the system it is assigned to the class whose centroid is 'nearest' according to some distance criterion. This technique may be extended to an arbitrary number of dimensions, one per attribute. The set of attributes must be chosen so that each is relevant to a particular characteristic of the blot. In MMIS-2, the classifier uses attributes that describe a blot's shape — namely transparency, circularity, aspect ratio, irregularity, convex maxima count and concave maxima count. A typical shape class is illustrated in Figure 5.5.

5.1.2 Blot Structural Relationships

A blot has one structural relationship to itself, specifying that it is a medium instance.

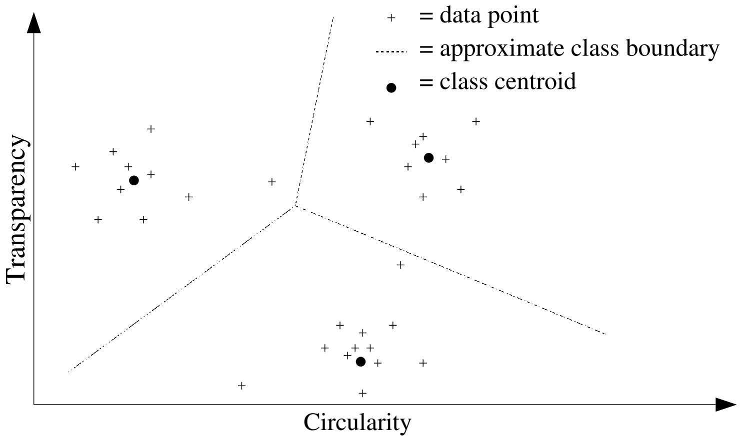


Figure 5.4: A two-dimensional space for circularity and transparency

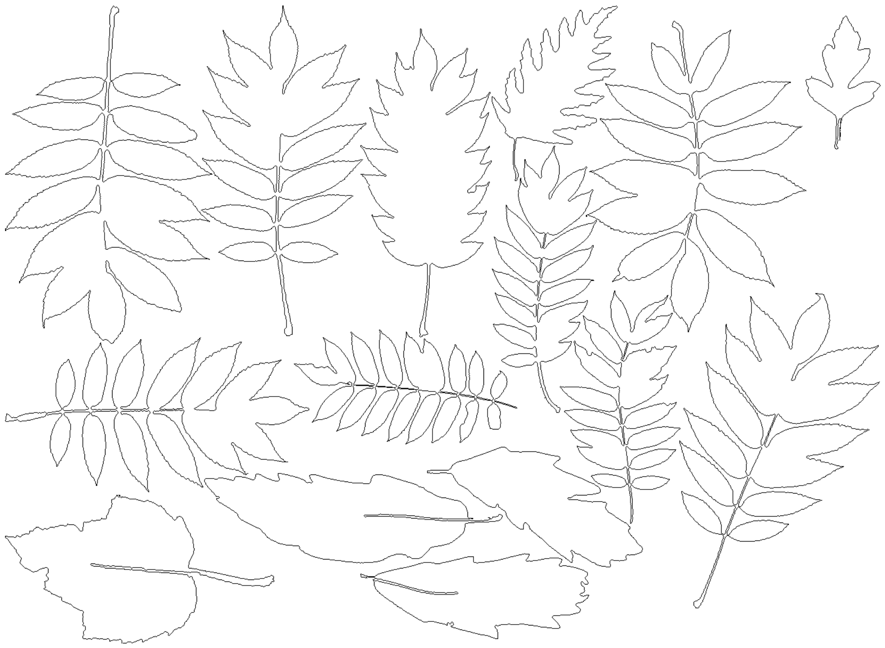


Figure 5.5: A typical shape class

5.1.3 Noise and Scale Marker Structural Relationships

Each scale marker and piece of noise has a structural relationship to its corresponding blot.

5.1.4 Leaf Attributes

Recall that a leaf is a blot that failed to be identified as a scale marker or a piece of noise. Each leaf has an area, hull area, irregularity, perimeter, longest chord, longest chord perpendiculars, length, width, true bounding box (defined by its four corners and centre) and upright bounding box (defined by its height, width and top left corner).

5.1.5 Leaf Structural Relationships

Each leaf has a structural relationship to its corresponding blot. Leaves also have structural relationships to other leaves in the image to describe relative positions in terms of bearing and bounding box overlap. The bearing of one leaf from another is described by compass directions (north, north east, east, etc), between the centres of their true bounding boxes; this is augmented by a test of whether or not their true bounding boxes overlap.

5.1.6 Image Attributes

Images have a length scale ($\frac{\text{Scale marker length}}{0.04}$) for the standardisation of scalar measurements, a height, a width and a class; the class specifies whether an image is 8-bit monochrome, 24-bit colour, etc. — all MMIS-2 images are class 2 (8-bit monochrome).

5.1.7 Image Structural Relationships

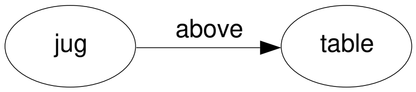
An image has a structural relationship to its scale marker and to each of the leaves and pieces of noise it contains; it also has a structural relationship to itself to specify that it is a medium instance.

5.2 The Representation

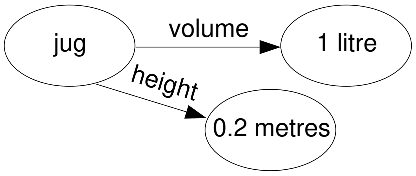
This section describes the style and specific structure of the semantic representation used in MMIS-2 and then goes on to describe the way such representations are stored in the Semantic Server.

5.2.1 Semantic Nets

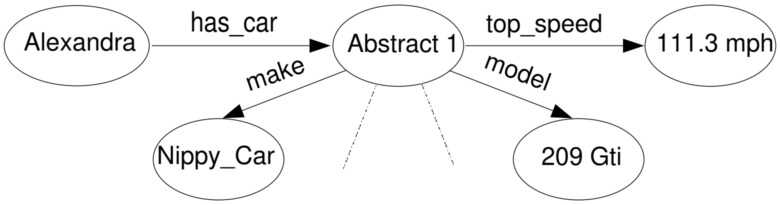
MMIS-2's semantic representation is the semantic net. In its simplest form, a semantic net is a graph of unidirectional annotated arcs that can be used to describe structural relationships such as



and attributes such as



Each arc is a statement of fact; in the case of a structural relationship, both the source and destination of the arc are entities being described with the source having the stated relationship *to* the destination — the entities connected could be images or leaves for example; for an attribute, the source is an entity and the destination is the value of the attribute, with the arc label being the name of the attribute. In MMIS-2 attribute values are simple objects which don't have structural relationships to other objects. In practice, the user formulating queries is unaware of the fine distinction between attributes and structural relationships, especially if arc names are meaningful. The notion of an *abstract node* can be used to denote a concept that needs more description than a single node allows, as in



Here, the abstract node glues Alexandra to the description of her car.

Each arc and node in MMIS-2 is given a *label* of the form

<type> / *<value>*

Using labels to annotate nodes and arcs rather than the usual integers and strings has certain benefits:

1. When a node is used to describe a medium instance, the medium is obvious — this would not be the case if the node was annotated with the instance identifier for example.

2. The meaning of attribute values is more explicit so that they can be more accurately compared.

3. The same arc name can be used for attributes of instances from different media.

4. The same arc name can be used for an attribute expressed in two different units.

Type	Value
image	unsigned long
text	unsigned long
document	unsigned long
blot	unsigned long
iset	unsigned long
int	integer
float	float
atom	string ($\leq$ 120 characters)
abs	unsigned long
'dimension'	unsigned long
'enumeration'	unsigned long
'dynamic'	unsigned long

Table 5.1: Label types and the domains of their values

5. Labels can be easily represented and accessed as distinct type and value components by application programs and query languages, providing more semantic power.

Table 5.1 shows the types that are valid for labels and the domains of their values; examples of each are given below:

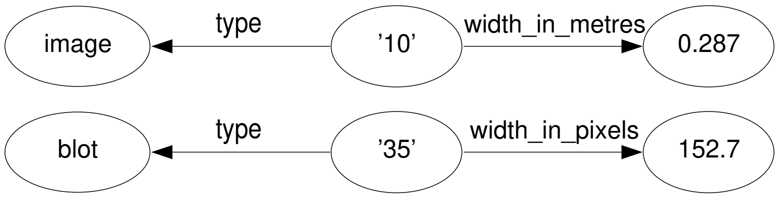
- The values of `int`, `float` and `atom` labels are generated by applications as required. Examples are `int/-19`, `float/0.756E-10` and `atom/freda`.
- Values for the types `image`, `text`, `document`, `blot` and `iset` (used to represent label sets) are the instance identifiers returned by the Large Object Server when medium instances are inserted[2].
  Examples are `text/52` and `iset/8735908`.
- The values of `abs` labels are generated by the Semantic Server when an application wishes to describe a new abstract object.
  Examples are `abs/1` and `abs/1500`.
- The 'enumeration' labels are used to describe a discrete enumerated type, and in this case the new *type* is provided by the Semantic Server.
  Examples are `enum75/3` and `enum948/56`.
- The 'dynamic' labels are again generated by the Semantic Server when an application wants to generate values of some new discrete type.
  Examples are `dyn87/2` and `dyn98432/0`.

[2]Conceptually, these values overlap — eg. `image/1` is distinct from `blot/1` — but for historical reasons they are actually disjoint — eg. `image/1` and `blot/1` cannot exist in the system at the same time. This means that there can be no more than 4×10^9 medium instances in the system at a time (sic).

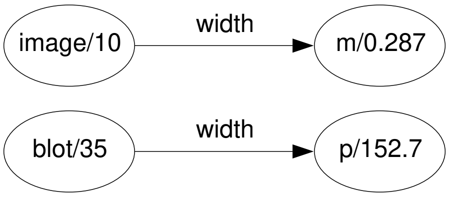
- The type of a ‘dimension’ label is any combination of powers of the SI base units (Ampere (`A`), candela (`cd`), Kelvin (`K`), kilogram (`kg`), metre (`m`), second (`s`)) with the addition of the radian (`rad`) and a new unit, the ‘pixel’ (`p`), but without the mole (`mol`). Powers can range from -8 to $+7$, except for the pixel, which can only take powers from -4 to $+3$. In order to avoid entry and display problems, powers are not super-scripted.
  Examples are `m3/1.73E-7`, `cdm-2/15.9`, and `kgms-2/9.81`.

Since all arc labels are atoms (strings) in MMIS-2, their type (`atom/`) is omitted in figures.

As an example of the extra expressiveness afforded by labels, consider the two net fragments



where quoted integers are instance identifiers, and the equivalent but more concise



Chapter 1 has already introduced the notion of ranking and mentioned that interpreters can help by giving a value to the uncertainty of their results. Now, consider that each arc in a semantic net represents a fact asserted by an interpreter; because facts may be uncertain, each arc can be given an associated float value called its *score* which is constrained to be in the range [0, 1], where 1 represents certainty and 0 represents a complete lack of certainty. Scores are *not* probabilities because it is difficult to generate a distribution to give the probability any mathematical foundation and because it is unclear how to combine probabilities in complex queries when they may or may not be independent. There are situations, however, when a meaningful score can be generated. One example is the use of compass sectors to describe positional relationships. The relationship could be decided on the basis of hard boundaries between compass sectors (see Figure 5.6). This scheme means that if b = 22.5° then S is considered to be south west of D, but if b = 22.499999° then S is south of D although there is clearly little difference. A better approach is to use inexact relationships similar to *fuzzy sets* as illustrated in Figure 5.7(a). Now, for instance, if b = 65° then S is south west of D with score 0.56 and west of D with score 0.44. This makes the boundaries between classes of positional relationships blurred and more intuitive. This technique can be adapted to give positional relationships a stronger influence close to the centre of their class either discretely (Figure 5.7(b)) or continuously (Figure 5.7(c)).

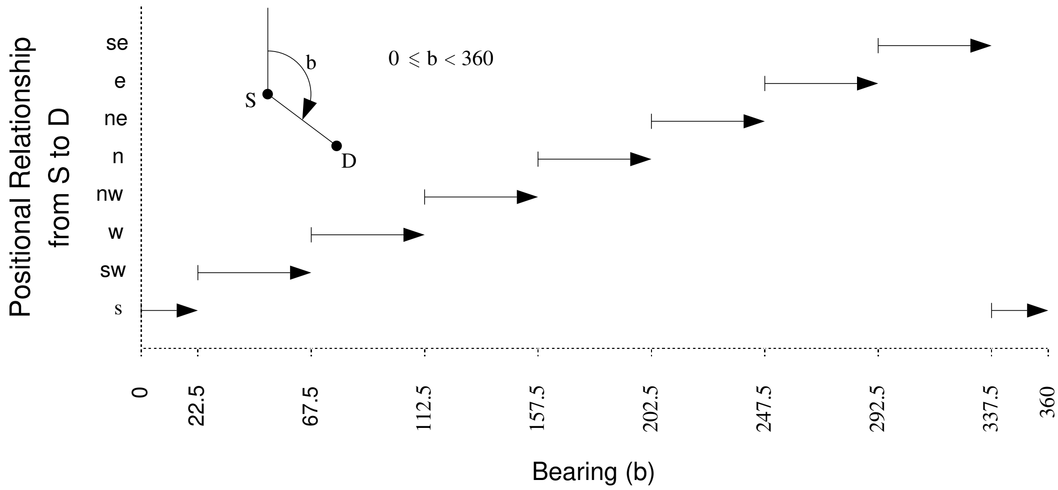


Figure 5.6: Choice of positional relationships

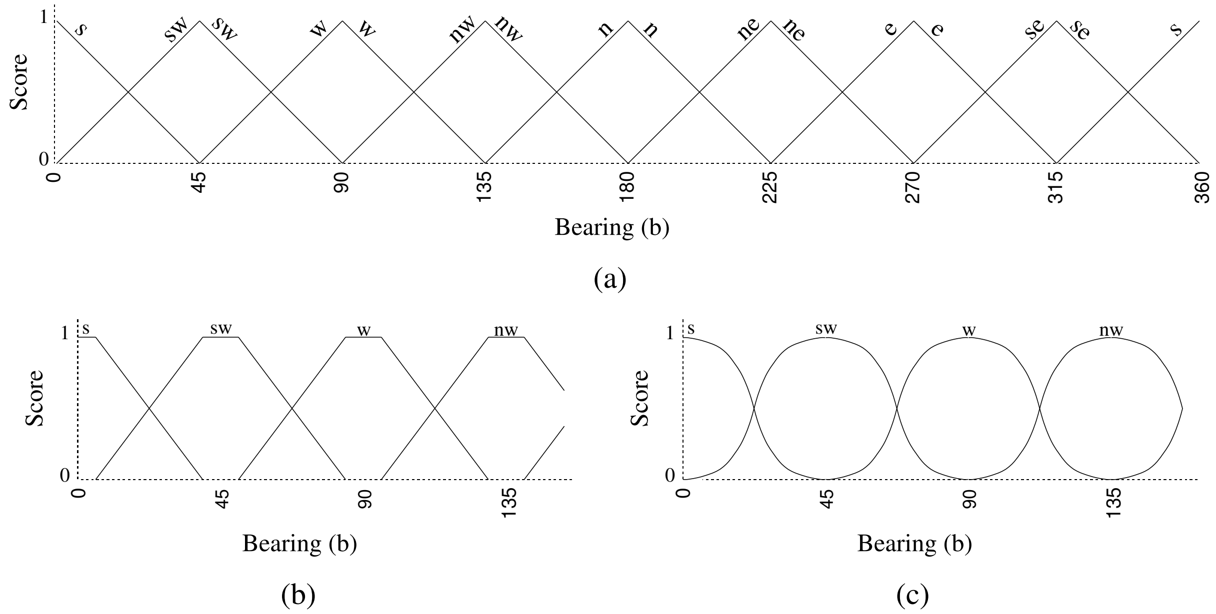


Figure 5.7: Three inexact descriptions of relative position

Another example of scoring is the assignment of a score to a blot's classification according to its distance from the centroid. This distance measure need not be Euclidean [Mah36]. Classes can also be given non-sharp boundaries so that each blot has a degree of membership of several classes, allowing queries such as

> "Find me leaves that are in class 3 with score ≤ 0.75 and in class 8 with score ≥ 0.7."

As a result of these arguments, all scores in MMIS-2's semantic net are 1 (certain) except in the following cases where the score is in the range [0, 1]:

- A blot's class is given a score to reflect its distance (D) from the centroid of that class:
$$1 - \left(\frac{\arctan(D)}{\frac{\pi}{4}}\right)$$
- The fact that a particular blot is noise is given a score according to how far its area (A) is from the threshold area for noise identification ($750p^2$):
$$\frac{750 - A}{A}$$
- A blot is identified as a scale marker if its aspect ratio (A), transparency (T) and circularity (C) all fall within specified ranges ($[A_l,\ A_h]$, $[T_l,\ T_h]$, $[C_l,\ C_h]$ respectively):
$$\left|\frac{(A_h + A_l - 2A)(T_h + T_l - 2T)(C_h + C_l - 2C)}{(A_h - A_l)(T_h - T_l)(C_h - C_l)}\right|$$
- If a scale marker is identified within an image, then all pixel measurements for the image and its leaves are stored in metres as well; all such metre measurements are given the scale marker identification score shown above.
- All compass bearings between leaves are scored according to Figure 5.7(b).

Having chosen the semantic net primitives — node, arc, label and score — the next step is to define a way of using the nets to describe images and blots. Consider Figure 5.8 which shows one possible semantic net of a document containing an image with two blots, and a second document containing two images, each with two blots, (scores have been omitted for the purposes of this discussion). There is a problem with this net when medium instances are shared between other instances — eg. `image/3` appears in both documents (despite the fact that the documents were created separately, they are both forced to refer to the same image by the Large Object Server's one-to-one mapping from medium instance to instance identifier). The problem occurs when arcs are connected to shared instances: when examining `document/1`, the Document Interpreter has determined that `image/3` is below `image/4` on some page; this is clearly nonsense with respect to `document/1`, in which `image/3` doesn't even exist. Therefore, this version of the net can

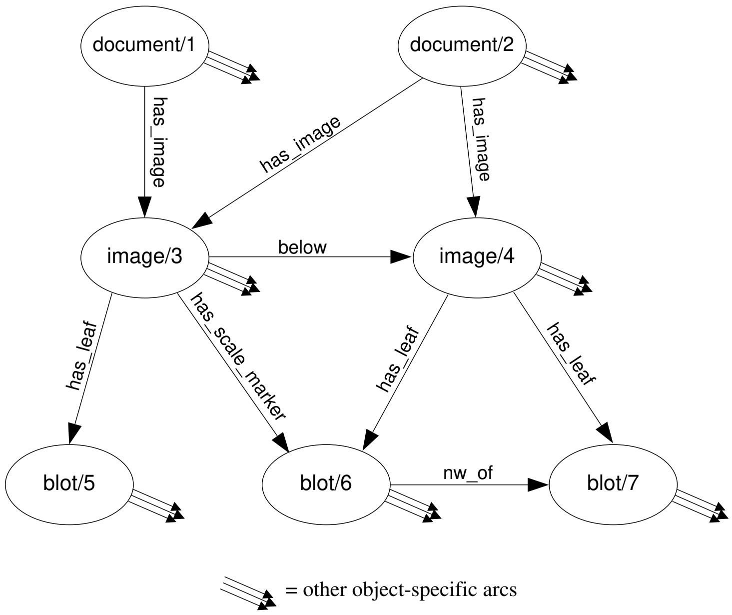


Figure 5.8: All arcs start from medium instances

easily contain incorrect structural relationships. This net contains other anomalies too: `blot/6`, which is in `image/3`, is '`nw_of`' `blot/7` which is in `image/4`; the Image Interpreter has decided that within `image/3`, `blot/6` is a scale marker, but within `image/4` there are better candidates, so `blot/6` ends up as a leaf; now, is `blot/6` a leaf or a scale marker?

As explained above, abstract nodes can be used to represent a context-specific occurrence of a medium instance, while the instance itself is referenced via a '`displayed_as`' arc. This idea could lead to the semantic net of Figure 5.9. This version no longer has any incorrect arcs, although it does have a lot more of them and a lot more nodes too. Further, whenever `blot/6` occurs within an image, it must be re-interpreted in order to attach its attributes to the new parent node; worse, if image interpreters are to be developed separately from blot interpreters, they can no longer be independent because the blot interpreter would either have to be sent "`blot/6` considered as `abs/9`" as an interpretation request or would have to return all the calculated attributes to the Image Interpreter; similar arguments apply to all the other shared instances. This makes the Interpretation Server operation described in Section 4.2, with its attendant benefits, impossible.

As a result, MMIS-2's semantic nets take a third form illustrated in Figure 5.10. This permits the sharing of medium instances and the independence of interpreters while preventing incorrect facts, too many arcs and nodes, and the re-interpretation of medium instances. In summary:

- the Blot Interpreter derives all attributes and the single structural relationship '`is_medium_instance`' for blots;

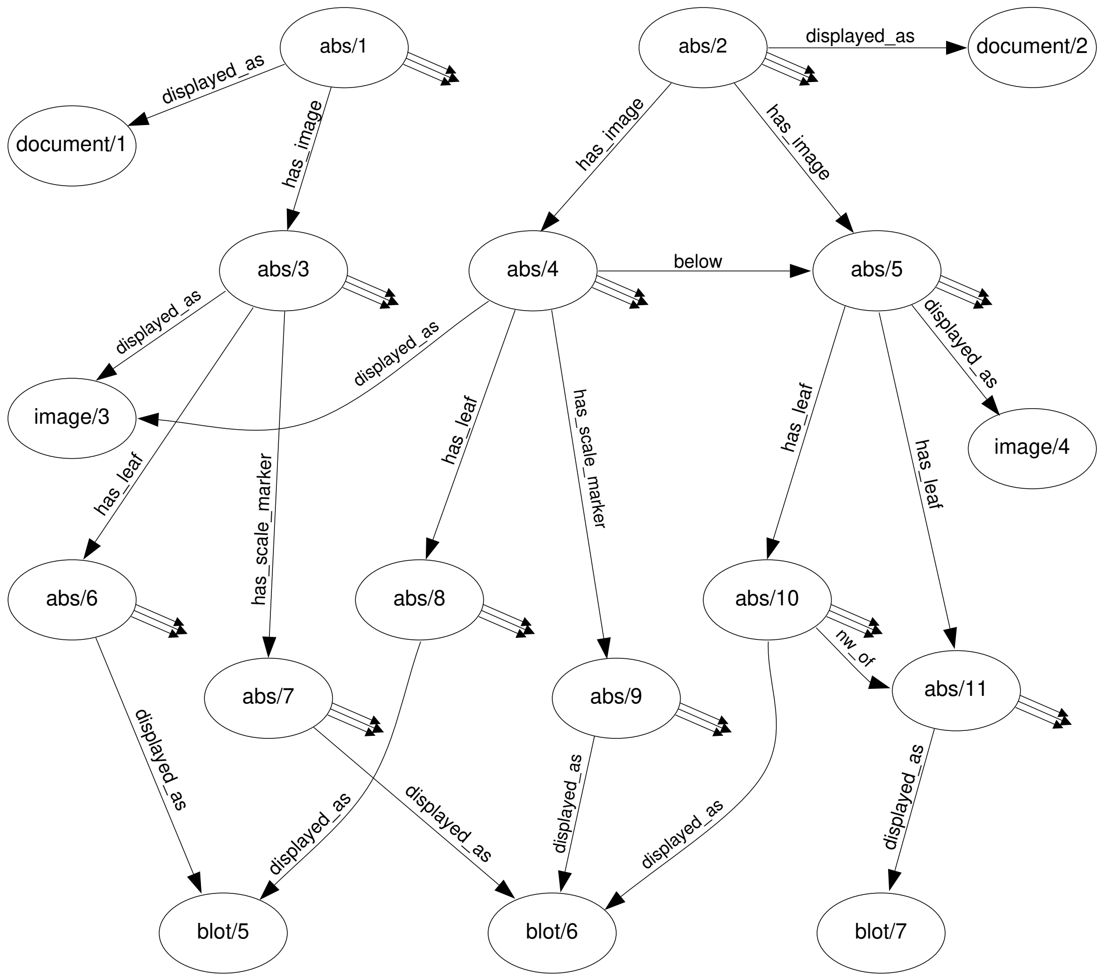


Figure 5.9: All arcs start from abstract nodes

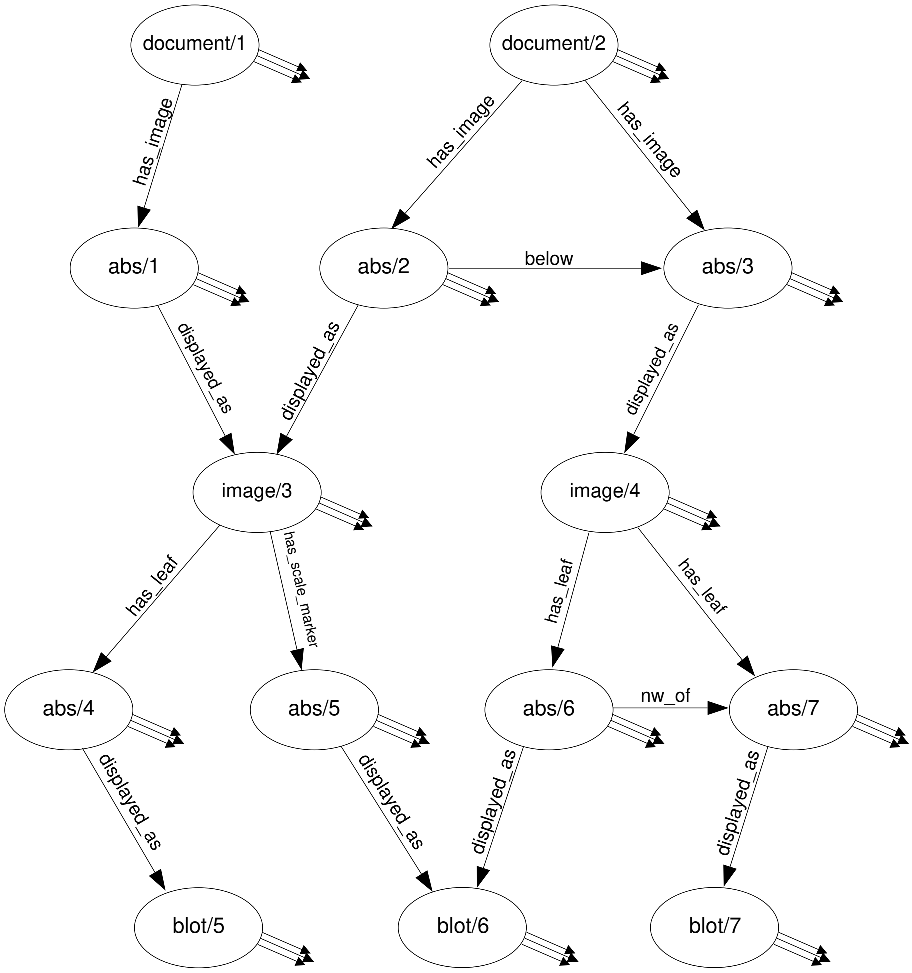


Figure 5.10: Arcs start from medium instances and abstract nodes

- the Image Interpreter derives attributes and structural relationships for leaves, scale markers and noise (represented as abstract nodes) and for images;
- the Document Interpreter (which is not yet implemented) derives document attributes and structural relationships, as well as facts about the abstract nodes that represent occurrences of images and text within each document.

Note that the chosen style of net contains redundant information because of the successive derivation of arcs from other arcs — eg. a leaf's area in m^2 is derived from the corresponding blot's area in p^2 and the image's length scale, which is in turn derived from the scale marker's length in p, and so on. This is not a problem because: successive derivation of higher-level arcs is time-consuming and therefore best performed pre-query and explicitly stored — therefore we can't just store the lowest-level arcs; on the other hand, one might envisage a system that reliably recognised a leaf's species based on all available data and only recorded that fact, but then such a system could not answer a query such as "What is the average transparency of oak leaves?" — therefore we can't just store the highest-level arcs; interpreters are deterministic and objective — they always produce the same set of arcs for a given medium instance — so inconsistency does not occur.

The semantic net provides elegant handling of incompleteness — if a particular fact cannot be deduced because of incomplete information, then it is simply not stored. In order to specify when incompleteness is appropriate, eg. for checking integrity of stored data, an object's arcs can be described as optional or mandatory and as single or multiple:

Mandatory single arcs must exist exactly once for the object — eg. an image's width in p.

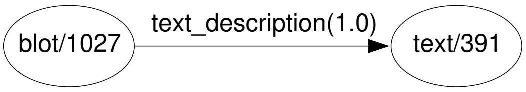
Mandatory multiple arcs must exist at least once for the object — eg. a blot belonging to multiple shape classes, with a different score for each;

Optional single arcs can appear at most once for the object — eg. an image only has a '`has_scale_marker`' arc if a scale marker has been found, but it cannot have more than one;

Optional multiple arcs can occur any number of times for the object — eg. a leaf can be `nw_of` any number of leaves in the same image.

The query language (Prolog) uses the closed world assumption so that the absence of an arc does not cause problems — the related fact is simply "not provable" — nor does the presence of multiple arcs, which are treated as a set of candidate solutions.

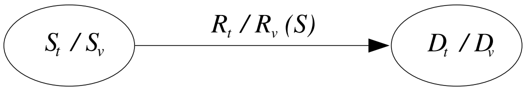
The semantic net that has been described can also be used to represent arbitrary hypermedia links entered by users, such as one linking a blot to a textual description



Hypermedia arcs are kept separate from the automatically derived nets, so that they cannot conflict with the data model — this is done by using a different IFS class for hypermedia links (see next section).

5.2.2 Semantic Net Storage

The semantic net described in the previous section contains a set of facts of the form



where S_t, S_v, R_t, R_v, D_t, D_v, S are source-node type, source-node value, relationship type, relationship value, destination-node type, destination-node value and score respectively, (recall that the relationship type is currently always `atom`). Each of these can be represented as the quadruple

$$< S_t/S_v >< R_t/R_v >< D_t/D_v >< S >$$

so that storing a net becomes a matter of storing a set of quadruples.

Storing a set of quadruples is achieved by implementing the Semantic Server using the IFS simulator [Lav88]. The IFS formalism caters for the storage of sets of tuples as *classes*, where each class has the same number of fields in every tuple and each field is a string of up to 120 characters. The strings themselves are not stored in the tuple; instead, when a tuple is inserted, each field is hashed to a 32-bit token by the *Lexical Token Converter* or *LTC*, which maintains a map of string-to-token conversions. This greatly reduces the storage requirements, while still allowing 30,720 strings to coexist. A query to the IFS takes the form of a prototype tuple with some fields filled in with strings and others left as wild-card (or "don't care") values; the string fields are tokenised and the tuple is compared with every tuple in the nominated class; a stored tuple matches the prototype tuple if its string fields are the same; the set of stored tuples which match are returned as the query result and may then be de-tokenised or further manipulated. The query also specifies which fields are to be returned within each tuple — for instance the user may only want the wild-card fields to come back because the content of the others is known. Obviously, comparing fields as 32-bit tokens is far quicker than comparing them as strings.

In MMIS-2, the IFS is enhanced to allow application programmers to create, decompose and compare labels rather than strings. To this end, a layer of label code is provided to convert labels from external formats (C++ objects and Prolog terms) into their internal IFS representation. Within the IFS, each label is stored as two 32-bit fields that respectively represent its type and value. Numeric values are actually stored in the tuple, while the string values of `atom` labels are tokenised — this is because numeric values can be represented as a 32-bit field (exponent and mantissa, two's complement and so on) and therefore stored and retrieved more quickly. In addition, each field in a Semantic Server tuple may be a 32-bit float (eg. a score) or a 32-bit integer. When a class is created, the format of its tuples is specified as an arbitrary sequence of label, float and integer fields

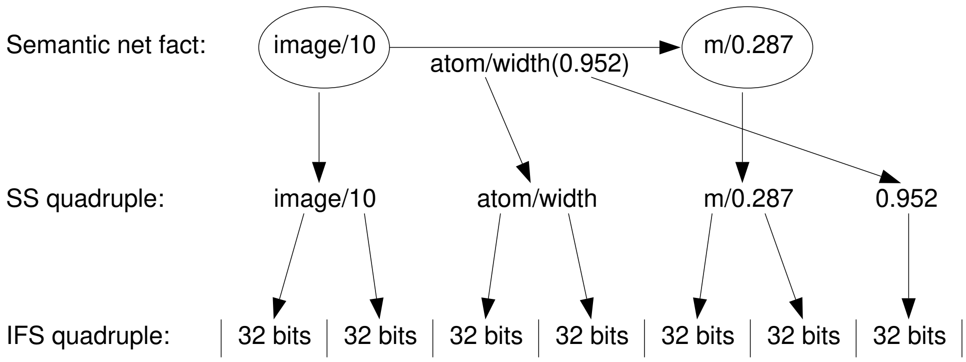


Figure 5.11: Mapping a fact from a semantic net into the IFS

and all subsequent operations on the class must conform to that specification. Figure 5.11 shows an example of the mapping of facts between different contexts.

5.3 Summary

The data extracted during interpretation is used to describe images in terms of the blots they contain, which in turn are identified as leaves, scale markers or noise. The data ranges from base facts such as the length of a blot's perimeter to derived facts such as positional relationships between leaves.

The structure of the semantic nets used allows: the sharing of medium instances between parents; the description of a medium instance as a context-specific entity (eg. an image within a particular document) or as a context-free entity (eg. a blot); the expression of attributes and structural relationships. Nodes and arcs in the semantic nets are annotated with labels, which exhibit richer semantics and more concise expression than the usual string or number variety. Each arc also has an associated score as an aid to ranking.

Semantic nets are stored as sets of quadruples in the Semantic Server, an enhanced version of the IFS which maps a semantic net arc to an IFS tuple and back again. Hypermedia links are represented using the same semantic net formalism but are stored in a separate IFS class from the interpretation results to avoid clashes.

Chapter 6

Querying

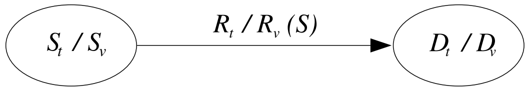
MMIS-2's Query Tool is a Sunview process (see Figure 4.2) whose upper window is used for entering MMIS-2 queries and whose lower window is used for direct interaction with Prolog and the display of diagnostics. The Query Engine is a separate Prolog process containing Prolog predicates and C++ functions. The Query Engine performs a number of tasks: converting queries and hypermedia requests into Semantic Server search requests; inserting hypermedia links into the Semantic Server; inserting label sets into the Large Object Server; returning label set identifiers and hypermedia links to the Sunview process for display. The higher levels of the Query Engine code are presented in Appendix A.

This chapter describes three modes of querying: the general style of Prolog query that can be issued in a Query Tool's lower window; the restricted style of query entered in the upper window that generates label sets; the hypermedia facility. For every occurrence of 'Prolog' in this chapter, read 'SICStus Prolog'; all the examples use SICStus syntax, which is close to the *de facto* standard DEC-10 syntax.

6.1 General Queries

This section concentrates on semantic net style queries, for a good introduction to all Prolog facilities see [Bra86].

As illustrated in Chapter 5, a semantic net is a set of facts where each is a named arc from one node to another



Such facts are represented in the Semantic Server as the quadruple

$$< S_t/S_v >< R_t/R_v >< D_t/D_v >< S >$$

Prolog can manipulate such facts as *predicates* (from First Order Logic), as in the quaternary predicate

```
quadruple(< S_t/S_v >, < R_t/R_v >, < D_t/D_v >, < S >).
```

Since arc labels are always atoms, the arc's value can be used as the predicate name:

```
< R_v >(< S_t/S_v >, < D_t/D_v >, < S >).
```

For example, a leaf's area can be expressed in either of the following forms:

```
quadruple(abs/39, atom/area, m2/0.0027, 0.97).
area(abs/39, m2/0.0027, 0.97).
```

The ternary predicates used in MMIS-2 are listed in Appendix B along with their meaning, derivation and permitted arguments.

A general Prolog query is a *goal* stated by the user for which Prolog must try to find solutions. A simple goal is

```
class(blot/19, int/3, 0.75).
```

This can be regarded as a *query* of the form "Is it true that blot 19's class is the integer 3, with a score of 0.75?". Prolog responds with `yes` if the fact is in the Semantic Server and `no` if it isn't. Expressive power is increased by the use of *variables*; these are objects which can take on any value in the search for a solution. The assignment of a specific value to a variable is called *instantiation* — if solutions to the query can be found, then the user is given a summary of the corresponding instantiations. Variables in Prolog are strings beginning with an upper case letter. For example,

```
has_leaf(image/15, Leaf, Score).
```

is analogous to "If there is an object in image 15 identified as a leaf, then what is it and what score is associated with its identification?", and will produce a succession of instantiations for `Leaf` and `Score` — the user types $\boxed{;}$ after each one to get the next solution or $\boxed{\text{Return}}$ to stop; Prolog prints `no` if there are no more solutions or if there were none at all:

```
Leaf = abs/172, Score = 1.0 ? ;
Leaf = abs/173, Score = 1.0 ? ;
no
```

The under-score character is a special "don't care" variable whose instantiations will not be printed, for example,

```
has_leaf(image/15, _, _).
yes
```

is equivalent to "Are there any leaves in image 15?".

Prolog insists that predicate names are instantiated, so ternary predicates cannot be used for queries such as "What relationships are there ...?". Instead, the `quadruple` predicate must be used:

```
quadruple(image/15, atom/Relationship, abs/172, _).
Relationship = has_leaf ? ;
no
```

Thus far, labels have been shown fully instantiated (eg. `image/15`) or wholly uninstantiated (eg. `Leaf`) in semantic net predicates; as an alternative, the label's type may be instantiated while its value is uninstantiated. For example, if the user is interested in the pixel-width of images but not the metre-width or the width of blots, he or she could issue the query

```
width(image/Image, p/Width).
Image = 1, Width = 760.0 ?
yes
```

Also, a label's type and value may be assigned to separate variables:

```
width(image/Image, Width_type/Width_value, _).
Image = 1, Width_type = p, Width_value = 760.0 ?;
Image = 1, Width_type = m, Width_value = 0.211 ?
yes.
```

However, a label's value may *not* be instantiated if its type is uninstantiated, so that

```
width(image/Image, Width_type/760.0, _).
```

will cause an error. This is due to the way in which labels are converted to IFS words: there is only one way to map a type into a 32-bit field and back again, but the value's representation is dependent on the type — eg. floats are represented as exponent and mantissa, integers are represented in two's complement format, strings are represented as 32-bit references to the real string, and so on; therefore a label's type must be known before its value can be encoded or decoded.

The next extensions to querying are the Boolean operators starting with 'and', written in Prolog as a comma. For instance

```
height(image/69, p/575.0, _), width(image/69, p/760.0, _).
yes
```

means "Is image 69's height 575p *and* is its width 760p?". The scope of a variable is the whole query, not just a sub-clause, so that

```
has_leaf(image/10, Leaf, _), displayed_as(Leaf, blot/3949, _).
no
```

means "Is there a leaf in image 10 such that the *same* leaf is displayed as blot 3949?'. There is also an 'or' operator, written as a semi-colon, as in

```
class(blot/Blot, int/15, _); class(blot/Blot, int/17, _).
no
```

which means "Is there a blot such that it's in class 15 *or* class 17?". This expression is in fact *two* queries, one each side of the semi-colon; only when all solutions to the first query have been exhausted will the second query be tried. Finally, there is the 'not' operator `\+`, as in

```
\+ has_noise(image/25, _, _).
no
```

which means "Are there no pieces of noise in image 25?". Without an understanding of Prolog's internal workings, the `\+` operator can produce unexpected results, so its use is discouraged. For example, the query "Find two leaves such that their true bounding boxes do not overlap and the first is north east of the second" is correctly formulated as

```
ne_of(Leaf1, Leaf2, _), \+ tbb_overlaps(Leaf1, Leaf2, _).
Leaf1 = abs/3150, Leaf2 = abs/3148 ?
yes
```

However, if the sub-clauses are swapped, Prolog will first try to prove that no leaf in the database overlaps any other leaf and, not surprisingly, fail:

```
\+ tbb_overlaps(Leaf1, Leaf2, _), ne_of(Leaf1, Leaf2, _).
no
```

Floating point values are rarely specified exactly because, for instance

```
has_noise(image/110, abs/17, 0.9).
```

will fail if the score is 0.899999. Such values are better specified by a range using Prolog's comparison operators:

```
has_noise(image/110, abs/17, Score), Score >= 0.85, Score =< 0.95.
Score = 0.93008 ?
yes
```

In a database context, it is more common to collect all solutions to a query and present them as a single result to the user. For this purpose, Prolog provides three collection predicates: `findall`, `bagof` and `setof`; of these, the `findall` predicate is the easiest to use and has the form

```
findall(
        < Template >,
        < Query >,
        < Solutions >
).
```

This finds all solutions to $< Query >$, putting each one into $< Solutions >$, formatted according to $< Template >$. $< Query >$ is any Prolog query, as described previously; $< Solutions >$ is a Prolog list of the form

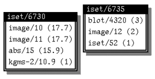


Figure 6.1: Two label sets as they appear on screen

[$< Solution1 >$, $< Solution2 >$, ..., $< SolutionN >$]

$< Template >$ specifies how each solution is presented. For example, to collect all leaves in image 10

```
findall(
    Leaf,
    has_leaf(image/10, Leaf, _),
    Solutions
).
Leaf = _30,
Solutions = [abs/3236,abs/3237,abs/3238,abs/3240,abs/3241] ? ;
no
```

The differences between `findall`, `bagof` and `setof` are explained in [CW89].

6.2 Label Set Queries

The query language used in Query Tool's upper window is designed to retrieve objects according to user-specified criteria and to rank them according to how well each one matches the criteria. This query language allows the user to issue any Prolog query whose result is a label set (because such sets can be browsed using mouse selection only).

A label set is a sequence of $< Label >$ / $< Score >$ terms. Examples are:

```
[abs/15/15.9, 'kgms-2'/10.9/1, image/11/17.7, image/10/17.7]
[image/12/2.0, iset/52/1.0, blot/4320/3.0]
```

The Query Tool displays label sets as a scrollable list of terms within a subwindow headed by the label set's own label, and with each term presented as

$$< Label > (< Score >).$$

Ranking is achieved by listing the terms in descending order of score. The two label sets introduced above are displayed as shown in Figure 6.1. If the user uses the mouse to select a term whose label represents a medium instance, then that instance is retrieved from the Large Object Server and displayed, as in Figure 6.2. Selecting a term that does not represent a medium instance — eg. '`abs/15 (15.9)`' — has no effect.

Label sets can contain other label sets, allowing trees of terms to be constructed. For example, the label sets

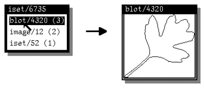


Figure 6.2: Demand-retrieval of an instance

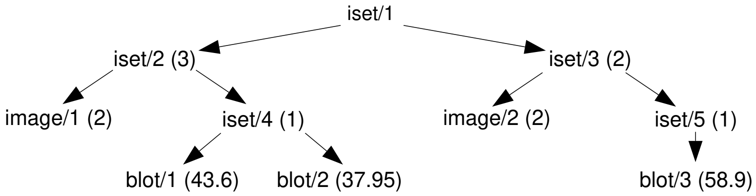


Figure 6.3: A tree-structured label set

iset/1: `[iset/2/3, iset/3/2]`
iset/2: `[image/1/2, iset/4/1]`
iset/3: `[image/2/2, iset/5/1]`
iset/4: `[blot/1/43.6, blot/2/37.95]`
iset/5: `[blot/3/58.9]`

represent the tree in Figure 6.3. In fact, label sets can represent arbitrary directed graphs in which arcs start at label sets and end at any object. When browsing a graphical label set on-screen, selecting a particular label set node displays the *next* level of nodes that can be reached in a forward direction. This means that the Query Tool doesn't get stuck trying to expand recursive label sets such as the one shown in Figure 6.4.

Label sets are normally constructed by Prolog's collection predicates; a simple example is

```
findall(
    Leaf/Score,
    has_leaf(Image, Leaf, Score),
    Set
)
```

which collects all leaves and ranks them according to the accuracy of their identification. The query does not need to sort the results into descending order because this is done automatically by the Query Engine. Note also that the query's trailing '.' is omitted in the upper window. The displayed result is shown in Figure 6.5.

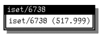


Figure 6.4: A self referencing label set

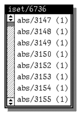


Figure 6.5: All leaves in the system

Tree-structured label sets are usually built by nesting collection predicates inside one another. The query shown in Figure 6.6 collects the (displayable versions of) those leaves whose transparency is in the range [0.35, 0.45], and groups them with their respective source images. The structure of the result is similar to that in Figure 6.3. Figure 6.7 shows the result partially browsed. Points to note are:

1. each member of the top level set is an image followed by the (displayable versions of) its leaves that have the specified transparency;

2. the image is forced to precede the leaves by assigning a higher score to it;

3. the outer `findall` ranks the [image, leaves] sets by the number of qualifying leaves found in the image;

4. the inner `find_all`[1] ranks leaves within each image according to their transparency.

Section C.1 gives more examples of MMIS-2 queries and their execution times.

6.3 Hypermedia

MMIS-2's hypermedia interface allows users to make arbitrary annotated connections between objects. The links are similar to the arcs in a semantic net. Users build *webs* of links to describe some interesting aspect of their application. For example, a user of a historical MMIS may wish to connect instances relevant to his or her interest in Shakespeare, as in Figure 6.8. Other users can browse this web, opening up instances for a closer look.

In MMIS-2, a 'starter' web is automatically generated connecting each image to the displayable versions of its leaves, pieces of noise and scale markers. Some images are also connected to a textual description. Links are examined by moving the mouse cursor over a displayed medium instance and typing $\boxed{\diamond}$ $\boxed{>}$ or $\boxed{\diamond}$ $\boxed{<}$ to retrieve links *from* and links *to* the instance respectively. A snapshot of the hypermedia interface in use is shown in Figure 6.9. Facilities are also provided for entering new links (see figure).

[1]`find_all` is a modified version of `findall` which fails if no solutions are found.

```
findall(

    /*
     * TEMPLATE: Each solution is going to be an image followed
     * by its incised leaves
     */
    [image/Image/2, Leaves/1]/S1,

    /* QUERY: Specify how to find a solution */
    (
    /* Get an image */
    is_medium_instance(image/Image, image/Image, _),

    /* Get its incised leaves, scored by transparency */
    find_all(Blot/T,
        (
            has_leaf(image/Image, Leaf, _),
            displayed_as(Leaf, Blot, _),
            transparency(Blot, float/T, _),
            T >= 0.35, T =< 0.45
        ),
        Leaves),

    /* The image's score is how many incised leaves it has */
    length(Leaves, S1)
    ),

    /* SOLUTIONS: where to put the list of solutions */
    Set
)
```

Figure 6.6: Generating a tree-structured label set

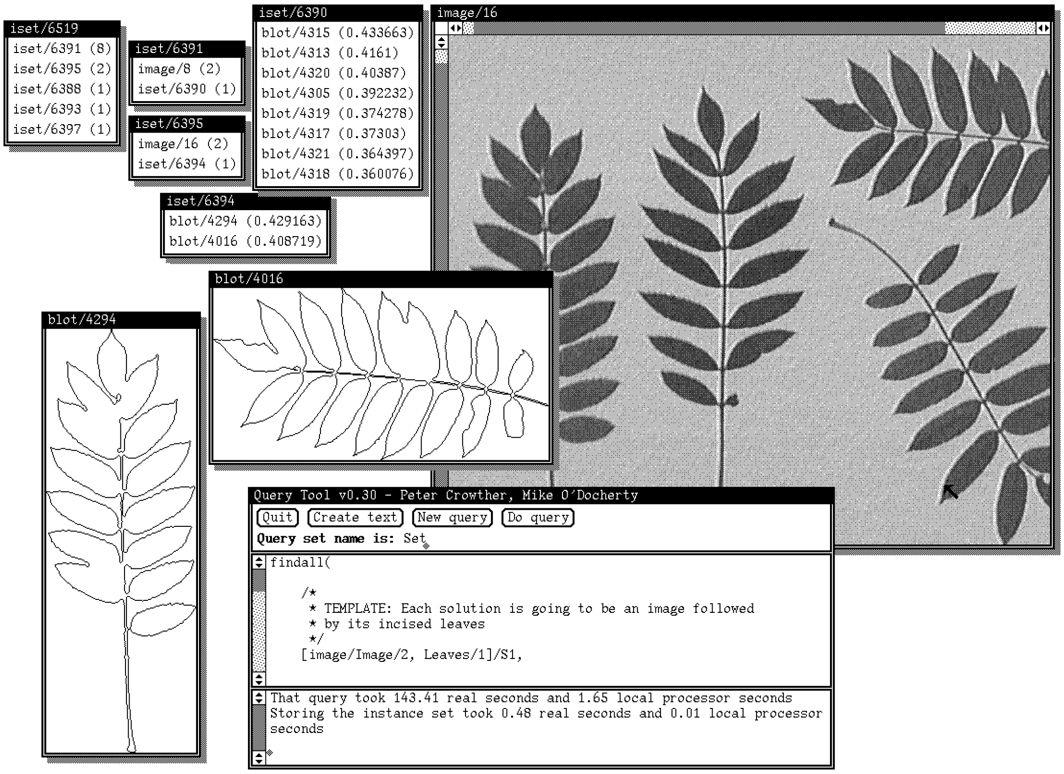


Figure 6.7: Transparent leaves with their images

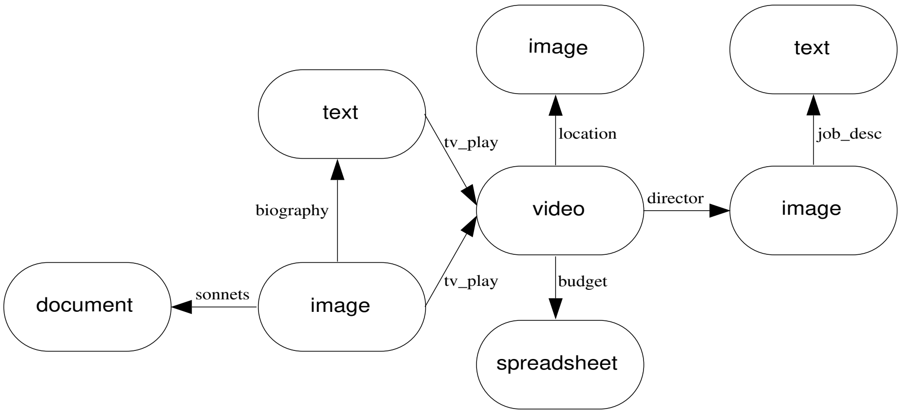


Figure 6.8: A hypermedia web

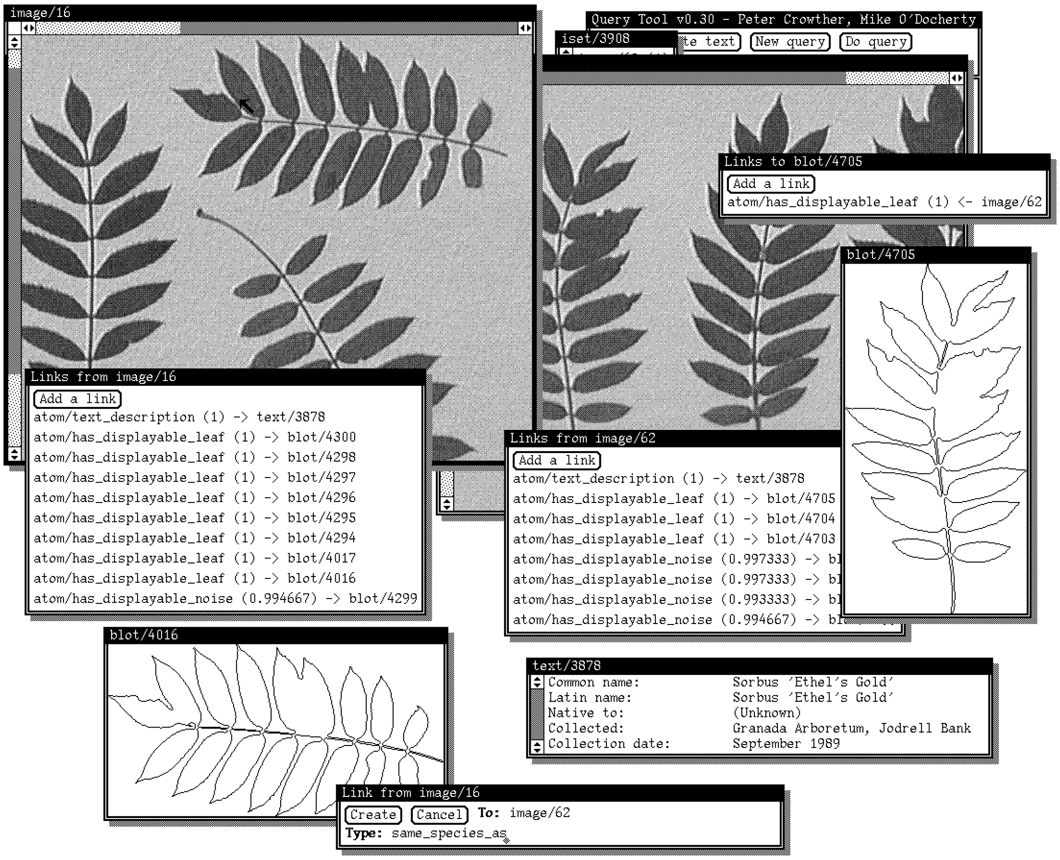


Figure 6.9: MMIS-2's hypermedia interface

Since hypermedia links are arbitrary, they must be kept separate from semantic net arcs; otherwise there would be nothing to stop a user creating a `tbb_overlaps (1)` link from leaf `abs/73` to leaf `abs/9873`, when the system thinks that the two leaves are in different images. To this end, all hypermedia links are stored in a separate IFS class from the semantic nets. The Query Engine's hypermedia code can be found in Appendix A.

6.4 Summary

MMIS-2's Query Tool allows users to enter general queries, queries that generate label sets and hypermedia requests. Label sets and hypermedia links are displayed by the Query Tool for browsing.

The query language (Prolog) relies on the mapping between semantic net arcs and predicates. The arguments of the `quadruple` predicate are 'source node', 'relationship', 'destination node' and 'score'; a better alternative is to use ternary predicates where the predicate name replaces the 'relationship' argument. The power of general queries is provided by tuple matching, variable instantiation, logical operators and arithmetic. Queries that generate label sets use the collection predicates such as `find_all`.

Label sets are displayed in Sunview windows: each level of a label set is displayed as a list of$< Label >$ ($< Score >$) items ranked in descending order; selection of an item that corresponds to a medium instance will cause the instance to be retrieved from the Large Object Server and displayed; since a label set is itself a medium instance, arbitrary directed graphs can be built and browsed.

A user can access a displayed medium instance's hypermedia links and create new ones for it.

Chapter 7

Results

This chapter describes the results from the initial implementation of MMIS-2, with reference to the requirements detailed in Chapter 1 and demonstrates the potential of query optimisation.

7.1 General

MMIS-2's implementation is general because it has few application-specific aspects and because these are confined to the interpreters and those Query Engine rules that generate parts of the semantic nets post-query (namely `leaf_count` and `noise_count` arcs). In fact, the system is applicable to applications other than leaves which generate 8-bit grey-level images except that: the image interpreter identifies leaves, scale markers and noise; the Query Engine derives `leaf_count` and `noise_count` arcs.

The modular design and the use of C++ has permitted rapid prototyping by three researchers, resulting in two prototypes in 18 months. Although C++ has proved to be a usable vehicle for programming with the object-oriented paradigm, the programmer still has all the facilities of C available. Since the object-oriented paradigm is not enforced, C++ programs can easily degenerate into bad C programs — fast but unstructured, and therefore difficult to maintain. It would be better to employ a purer object-oriented language such as Smalltalk[GR85] or Eiffel [Mey92], the latter having the added benefits of static type-checking and executable specification code (pre-conditions, post-conditions and class invariants). The modularity of MMIS-2 means that the implementations of its components with different tools can be compared, while retaining their interfaces. Because of the use of the client-server paradigm, some modules can be distributed — the Large Object Server, Semantic Server and Interpretation Server can all run remotely from the user's machine while the latter runs the user's Query Engine and Query Tool, thereby harnessing the power of four processors simultaneously. Although multi-user access to the database is possible, protection mechanisms to preserve data integrity are incomplete and data security mechanisms are not yet implemented.

It has proved feasible to manage medium instances and their semantic representations separately, so that:

Number of arcs	99185 (250)
Arcs per image	793.5
Number of nodes	25185 (6)
Nodes per image	201.5

Table 7.1: Semantic net size for 125 images

- the query engine does not need to retrieve or examine medium instances;
- the movement of medium instances between system components is minimal;
- medium instances can be interpreted when the system is not busy;
- medium instances do not have to be retrieved en-masse by the Query Tool — instead, the user is quickly provided with a symbolic representation of the result of a query and can retrieve the interesting instances on demand;

The semantic net formalism used is application independent and allows the storage of content descriptions that are rich enough for useful querying, the graphical organisation of results, the assignment of scores to interpretation results to allow ranking, the representation of general hypermedia webs, and the storage of blot classification data. The emphasis on the successive extraction of higher-level data from base facts such as a blot's area means that useful queries can be resolved without having to resort to complicated net traversals or the combination of large numbers of arcs. The combined size of MMIS-2's semantic nets for 125 images is summarised in Table 7.1, where the figures in parentheses indicate the portion of the nets that is generated by the Query Engine.

The benefits expected from using the Interpretation Server to organise interpretation were listed in Section 4.2, and these benefits have been realised. The time taken to interpret a new image depends on many factors, such as:

- the distribution of Interpretation Server, Large Object Server and Semantic Server;
- the number of interpreters running;
- the speed and capacity of the machines running the servers;
- other network and machine activity;
- the number of objects being interpreted (and therefore how many interpreters are busy);
- the number of blots within the image and whether those blots are leaves, scale markers or noise (since the number of structural relationships rises exponentially with the number of blots);

- the current size of the Semantic Server (because searches are linear and insertions are checked for duplication),

and so on. With many different configurations, it has been found that no image interpretation takes longer than five minutes. The Interpretation Server consumes very little CPU time, typically much less than a second for each image.

Prolog has proved to be an expressive query language because of its inherent tuple-matching style — the author has not found it difficult to express any query which it is intuitively possible to solve given the nature of the data derived from images and experience of querying other databases; further, since all data takes the form of medium instance attributes and relationships to other medium instances, users are able to query in a friendly, object-oriented style, but without the system having to retrieve the actual instances. The tuple-matching style also merges well with the IFS so that the user is unaware of any functional difference to pure Prolog.

SICStus Prolog's programming capabilities and its interface to C (and therefore to C++) has proved to be a powerful technique for the implementation of a Query Engine; the interface to C has made it possible to take advantage of Sun's Remote Procedure Call (RPC) library for communication with remote processes; the access to Unix files (and therefore to pipes) has made it possible for a Prolog process to cooperate with a Sunview process, despite the insistence of both on controlling program flow.

The Query Tool allows the entry and browsing of arbitrary hypermedia webs as labelled links between objects, which can then be represented using the same semantic net formalism as is used for interpretation results, and therefore stored as tuples in the Semantic Server. Large webs can be derived automatically without disorienting the user as long as the set of link names is small — MMIS-2 generates the automatic links `has_displayable_leaf`, `has_displayable_noise` and `has_displayable_scale_marker`. The Query Tool also permits the entry of any Prolog query that generates a label set, which is then sorted automatically by score for presentation to the user and subsequent browsing; label sets have been used to represent the results of a wide variety of queries in a convenient manner for browsing.

7.2 A Query Optimisation Example

It has already been stated in Section 3.2 that Prolog is an unrealistic tool for the management of semantic data as well as provider of query language and query processor — the problems relate to bulk storage, unilateral control of the system, multi-user access and slow start up. Unfortunately, in moving some of the tasks which Prolog could perform in RAM to the Semantic Server, there is an inevitable reduction in query speed. This is exacerbated by the fact that the Semantic Server uses the IFS simulator rather than the hardware version and by the fact that query optimisation techniques were not implemented in time to be incorporated into the Query Engine. In order to quantify the effects on performance, MMIS-2 has been compared with a slightly different system — called *MMIS-P* — in which Prolog replaces the the Query Engine and Semantic Server.

```
findall(
    [abs/A1/2, abs/A2/1]/S,
    (
    n_of(abs/A1, abs/X, S1),
    w_of(abs/X, abs/A2, S2),
    S is S1 * S2
    ),
    Set
)
```

Figure 7.1: A query involving a transitive positional relationship

The full results of this comparison are given in Appendix C. Although queries are much faster in MMIS-P, the semantic nets take up more space, the Query Tool takes a lot longer to start, programmatic access is only reasonable for processes (eg. interpreters) written in Prolog, and there is no multi-user access.

For the queries in Appendix C, 78.2–99.5% of the execution time is spent performing linear searches in the Semantic Server and transporting results back to the user's query. The problem here is that the Semantic Server is designed to work in a set-based rather a tuple-based manner, so that using pure Prolog queries directly is inefficient; this problem can be alleviated if queries are pre-processed into set-based analogues using Draxler's techniques.

In order to illustrate the potential of pre-processing queries, consider giving MMIS-2 the query in Figure 7.1 which looks for pairs of leaves such that the first is north of some leaf `X` and `X` is west of the second, ranked according to the accuracy of the relationships; let this be query A. Query A involves one search request to retrieve a sequence of solutions to the `n_of` predicate and then a further search for the solutions to the `w_of` predicate for *each* member of the first sequence. If we assume that all the search requests are satisfied in a similar time (reasonable if most of a search request is taken up by a sequential search of the whole tuple store and returning the result to Prolog), then we have

$$t_e \approx t_s + n_1(t_m + t_s + n_2 t_m) + \alpha t_o \tag{7.1}$$

where t_e is the real execution time, t_s is the time for a search request, t_m is the time to get the next member of a sequence with unification included, t_o is the time to evaluate the expression '`S is S1 * S2`', α is the number of transitive relationships found, n_1 is the number of solutions to the `n_of` predicate and n_2 is the number of solutions to the `w_of` predicate (subject to the solution of the preceding `n_of` predicate, assuming this number is always the same). It is intended that MMIS-2 queries will be pre-processed in a style similar to that demonstrated by Draxler [Dra90], to yield set-based queries. Draxler generates relational queries in SQL, but since the Semantic Server's contents can be considered to be a single quadruple relation, and the Semantic Server search request is analogous to a select-and-project operation, Draxler's techniques should certainly be

```
findall(
    [abs/A1/2, abs/A2/1]/S,
    (
    ssclient_search_and_return([y,n,y,y],[abs/_,atom/n_of,abs/_,_],N),
    ssclient_search_and_return([y,n,y,y],[abs/_,atom/w_of,abs/_,_],W),
    member([abs/A1,abs/X,S1],N),
    member([abs/X,abs/A2,S2],W),
    S is S1 * S2
    ),
    Set
)
```

Figure 7.2: A 'pre-processed' version of the query in Figure 7.1

applicable. Query A is informally equivalent to the relational algebra expressions

$$\begin{aligned} N &= \Pi_{A_1,A_3,A_4}(\sigma_{A_1=\mathrm{abs/\_},A_2=\mathrm{atom/n\_of},A_3=\mathrm{abs/\_}}(Q)) \\ W &= \Pi_{A_1,A_3,A_4}(\sigma_{A_1=\mathrm{abs/\_},A_2=\mathrm{atom/w\_of},A_3=\mathrm{abs/\_}}(Q)) \\ Set &= \Pi_{A_1,A_5,(A_3\times A_6)}(N \bowtie_{N.A_2=W.A_1} W) \end{aligned}$$

where Π_{Θ_1} is a project operation according to the predicate Θ_1, σ_{Θ_2} is a select operation according to the predicate Θ_2, $\bowtie_{\Theta_3}$ is a theta-join operation according to the predicate Θ_3, Q is the quadruple 'relation' stored in the Semantic Server, A_n is the nth attribute of a relation and $R.A_n$ is the nth attribute of relation R. These expressions can be hand-coded using the predicates which import Semantic Server calls into Prolog directly, to produce the query in Figure 7.2 ; let this be query B[1]. In query B, the search requests perform the select-and-project operations, Prolog performs an n^2-efficiency theta-join using the two member predicates, while the template and the arithmetic expression perform the final project. The execution time for query B is given by

$$t_e \approx 2t_s + n_1(t_m + n_1 t_m) + \alpha t_o \tag{7.2}$$

If query B is to execute more quickly than query A, then using equations (7.1) and (7.2), we have

$$\begin{aligned} 2t_s + n_1(t_m + n_1 t_m) + \alpha t_o &< t_s + n_1(t_m + t_s + n_2 t_m) + \alpha t_o \\ \Rightarrow \frac{t_m n_1(n_1 - n_2)}{n_1 - 1} &< t_s \end{aligned} \tag{7.3}$$

If we assume that $n_1 \gg 1$ then equation (7.3) becomes

$$(n_1 - n_2)t_m < t_s \tag{7.4}$$

[1]See Appendix A for a description of the arguments of Semantic Server predicates. The first three arguments of the `ssclient_search_and_return` predicates have been omitted for clarity; here, they specify class 1 (MMIS-2's semantic net class), simple match and logical equality for all fields. This also applies to all subsequent examples using `ssclient_search_and_store` and `ssclient_cardinality`.

Query	CPU time (s)	Real time (s)
A	3.48	196.85
B	13.59	14.00
A (MMIS-P)	3.03	4.97

Table 7.2: Execution times for queries A and B

As stated above, most of a query's execution time is spent outside Prolog, so it is reasonable to assume that multiple calls to `member` are cheaper than a search request and, therefore, that equation (7.4) holds. Table 7.2 shows the actual execution times for the two queries in MMIS-2 and a comparative execution of query A in MMIS-P. All timings in this table are an average of two readings for 65 images, with an error of $\pm 0.01s$. It can be seen that query B is 93% faster than query A and that query B's execution time is of the same order of magnitude as MMIS-P. Therefore pre-processing queries should allow MMIS-2 to approach the query speed of MMIS-P, i.e. of pure Prolog. Further efficiency improvements for the querying of MMIS-2 are proposed in Section 8.2.

7.3 Summary

MMIS-2 has proved a valuable tool for investigating some of the features of the ideal MMIS. It has few application-specific features which makes it an extensible system for future research. The object-oriented and client-server paradigms have encouraged modularity which allows parallel development of system components and comparative implementations.

Separating the management of medium instances and semantic representations has minimised data traffic and allows the Query Engine to concentrate on query resolution on a smaller body of data. SICStus Prolog has been shown to be a powerful query language and a flexible database front-end through its C interface.

The semantic net has worked well as an expressive semantic representation that can handle attribute and structural information, hypermedia and the association of scores with facts.

The principle bottle-neck in MMIS-2 is the Query Engine because it doesn't yet try to optimise queries. However, it has been shown that queries that take account of the set-based storage in the Semantic Server execute far more quickly. The translation to set-based queries can be performed automatically using Draxler's techniques because of the logical correspondence between an IFS class and a relation.

Chapter 8

Conclusions and Future Work

In this thesis, the author has introduced the concept of a Multimedia Information System, described the features of an ideal MMIS, identified the research fields which must be assessed by MMIS researchers and has illustrated the arguments with a detailed description of a working system. By now the reader should be aware of the complexity of MMISs with automatic content retrieval but should be convinced of their worth — consider how vital traditional databases have become in Information Technology and the order of magnitude increase in capability offered by the ideal MMIS.

The Multimedia Group at Manchester University has demonstrated the feasibility of automatic content retrieval of images and their parts and the management of text within a system which forms a sound basis for further research. The ultimate goal of building the ideal MMIS will be achieved through the natural or forced convergence of database technology, knowledge elicitation and representation, automatic interpretation and Human Computer Interaction.

This chapter concludes with a detailed account of the future work on various parts of the system which would greatly enhance its speed, functionality and capacity.

8.1 Comparative Implementations of Database and Query Engine

Parts of MMIS-2's database and Query Engine could be replaced with alternative implementations for efficiency comparison, with little or no loss of functionality and with few changes to interfaces between modules. This is the only way to demonstrate that MMIS-2's implementations are the most suitable.

Appendix C shows how the Semantic Server and half the Query Engine can be replaced with Prolog; the only resulting advantage is faster querying, but the next section describes how the difference in speed can be greatly reduced. Megalog could provide a better replacement for the Semantic Server and the whole Query Engine, but its rule storage and evaluation within the database may hamper its speed as a pure tuple store and non-Prolog interpreters would have no programmatic interface to semantic nets.

It has been demonstrated that there is a mapping from a semantic net to a set of arcs and then to a set of quadruples; these quadruples map directly to Prolog predicates. There is also a direct mapping from Prolog predicates to relations in a relational database when the predicate's arguments are atomic. MMIS-2's arguments are not atomic and representing them as, say, strings in a relational database loses some query functionality, as well as probably increasing storage requirements and query execution times. The lower levels of the Query Engine and the whole Semantic Server could, however, be replaced with a set of five-attribute relations of the form

TBB_OVERLAPS:

St	Sv	Dt	Dv	Sc
abs	20	abs	19	1.0
⋮	⋮	⋮	⋮	⋮

and then Draxler's techniques could be used as they stand to generate relational queries from Prolog. Another possible scheme is to replace the IFS simulator with a set of relations with tokenised attribute values:

TBB_OVERLAPS:

St	Sv	Dt	Dv	Sc
101110...	100010...	101110...	001010...	0101000...
⋮	⋮	⋮	⋮	⋮

while retaining the existing Semantic Server client-code, the label code and the Query Engine. The comparison operations available on fields would depend on the particular database used; some implementations may also permit the storage of 64-bit integers so that the first four columns could be collapsed into two. Unlike MMIS-2, neither of the above relational schemes allows efficient querying of a relation name but such queries have been found to be rare.

Any number of object-oriented databases now available could be used to replace the whole of MMIS-2's database but this is unlikely to be fruitful because medium instances would have to be loaded and examined post-query, even if an object's attributes and structural relationships were computed pre-query and stored with it or in associated description objects. Alternatively, the querying structures, i.e. semantic nets, could be stored as a single object and loaded by a Semantic Server-like module; such a server could not easily be made to access its data dynamically from disc, thereby restricting the number of semantic nets which could be stored. The IFS simulator is a functional equivalent to a hardware tuple store whose bulk storage is disc-based, so the simulator could be replaced easily by the hardware to give greater capacity and speed with minimal change to the Semantic Server and no change to other MMIS-2 modules.

```
findall(
    [abs/A1/2, abs/A2/1]/S,
    (
    ssclient_search_and_store([y,n,y,y],[abs/_,atom/n_of,abs/_,_],N),
    ssclient_search_and_store([y,n,y,y],[abs/_,atom/w_of,abs/_,_],W),
    ssclient_join_and_return(N,W,[2,1],[y,y,y,n,y,y],NjoinW),
    member([abs/A1,abs/X,S1,abs/A2,S2],NjoinW),
    S is S1 * S2
    ),
    Set
)
```

Figure 8.1: Relational algebra within the Semantic Server

8.2 Improving the Semantic Server and Query Engine

The existing Semantic Server and Query Engine can be improved in a number of ways without replacing their tools:

1. Prolog queries should be pre-processed using techniques derived from those of Draxler, to produce set-based queries which can be further optimised by algebraic manipulation — pushing select before product for instance.

2. A *Relational Algebra Processor (RAP)* should be provided to execute relational operations on tuple classes efficiently. The IFS developers have stated a commitment to providing such a RAP within the IFS.

3. The Semantic Server's `ssclient_search_and_store` can be used to store the result of a select-and-project operation in a new class and return the class identifier rather than the actual result; this request is not yet implemented but in conjunction with a RAP sited within the IFS, it would allow all relational algebra to be performed remotely, with only the final result being returned over the network. For example, query A in Section 7.2 could be preprocessed into something like that shown in Figure 8.1, where the predicate `ssclient_join_and_return` joins the given classes on the specified columns and returns the result projected according to its fourth argument.

4. The IFS simulator code is written with functionality rather than efficiency in mind and could be significantly speeded up. In particular, it performs sequential searches on tuple classes rather than using indexing or hashing techniques.

5. The IFS formalism permits the comparison of a prototype field with a stored field as 'equal', 'not equal', 'greater than' or 'less than'; further, the comparison may be

'arithmetic' (a signed bitwise comparison) or 'logical' (an unsigned bitwise comparison). Although all these comparisons have been exported to Prolog, only 'logical equality' comparisons are used by the predicates which provide functional equivalence to Prolog because they are all that is necessary for Prolog's unification. Query 3 in Appendix C performs a range test in Prolog on the transparency of returned blots; this is an example of a query which could be pre-processed so that one of the comparisons were performed within the IFS, which would be faster and would return less data to Prolog.

6. The Query Engine code that provides functional equivalence to Prolog uses the IFS's 'simple match' search algorithm and any cross comparisons of uninstantiated fields are performed within Prolog; any implied cross references in a query could be detected by the pre-processor so that the IFS's 'one way match 1' algorithm could be used to perform the cross comparisons in-store.

7. It is not possible to distinguish the special '_' variable within a Prolog predicate, so that the corresponding tuple column is returned by the IFS unnecessarily; this problem could be avoided if some atom such as `dont_care` were reserved for this purpose — the slight reduction in the number of allowable values for `atom/` labels would not be serious.

8. The Semantic Server has an `ssclient_cardinality` request which returns the number of matching tuples rather than the tuples themselves; this could be used to implement a faster version of Prolog's `\+` operator so that, for example

   ```
   \+ tbb_overlaps(A, B, C).
   ```

   would become

   ```
   ssclient_cardinality(
       [n,n,n,n],
       [_/_,atom/tbb_overlaps,_/_,_],
       0
   ).
   ```

9. The IFS, and therefore the Semantic Server, will not return a result that is larger than a size specified at compile time; for MMIS-2 this fixed maximum size is 64K IFS words (or 256 Kb) — if a query overflows this result buffer then only the first 64K words are returned. The intention is that if the same query were re-issued with no intervening changes to the class — by backtracking in Prolog for example — then the next chunk of the result would be returned. Although overflow of the result buffer is detected in MMIS-2 and reported to the user, the retrieval of the next chunk of data is not yet implemented. Of the queries in Appendix C, the biggest result set is retrieved by the `n_of` predicate in query 4. The largest number of images that can be in the system before this predicate will fail to retrieve all its

solutions is given by

$$I_{\max} = \frac{\text{Buffer size}}{\text{Size of retrieved tuple} \times \text{Number of n\_of arcs per image}}$$

Given a buffer size of 64K IFS words, a retrieved tuple size of 5 IFS words and 1094 `n_of` arcs for 125 images, we have

$$I_{\max} = \frac{65536}{5 \times \frac{1094}{125}} \approx 1462.5$$

Therefore we can assume that if successive retrieval is not implemented, query 4 will produce incorrect answers for more than 1,462 images.

10. Each set of semantic net arcs with the same name could be assigned to a distinct tuple class, with a mapping maintained by Prolog; this means that for any query involving a named arc the IFS simulator would have to search far fewer tuples (but the `quadruple` predicate would become less efficient).

8.3 Documents and Text

Although text can be stored in and retrieved from MMIS-2 and included in hypermedia webs, it is not interpreted. Such interpretation could range from keyword content and position to natural language understanding to deduce events, actors, objects and so on.

Documents are not yet stored; it is intended that they will take the form of hierarchical aggregates of medium instances and simpler objects in the style of ODA. An appropriate generic logical structure for a tree description document is given in Appendix D. ODA provides a schema definition language and the potential for interfacing to standard formatters and communication to heterogeneous systems. An ODA document could either be stored in the Large Object Server or in the Semantic Server due to its structured composition which is amenable to representation as a semantic net. Storage in the Large Object Server could benefit from related research into *Stored ODA* (*SODA*[Fea87]).

Retrieval of complete structured objects from the Semantic Server is currently inefficient because the latter has no transitive closure operation. For instance, blot classification results are stored and then retrieved by blot interpreters when they start; because there is no transitive closure operation and no pre-processor to transform C++ queries into set-based analogues, this retrieval is slow, taking about 15 minutes. Instead, each tuple could be given an extra 'tag' field to specify that it is part of a larger external structure.

Document interpretation could pertain to relative positions of objects on the formatted page, document category, derivation of information from parts of the document (eg. a tree's classifier and family can be derived from its botanical name) and the interpretation results of contained instances.

8.4 Data Correctness

The correctness of data in the Semantic Server could be ensured by forcing it to conform to some 'meta-data' description; if this enforcement is made when the data is inserted then insertion will become much slower. A meta-data formalism is described in [O'D90] which is used by a process (initiated by an administrator) to check that the stored semantic nets are sensible. This meta-data formalism and its accompanying process could be enhanced to work with MMIS-2.

Data correctness in the Semantic Server is largely ensured by the automatic interpretation of medium instances but the data can be corrupted by process bugs, process crashes, inappropriate deletions of tuples by the user and insertion/deletion clashes between processes. In order to minimise these effects, the contents of the Semantic Server are written to disc every hour.

8.5 Problems with Data Sharing and Resource Sharing

There are no access privileges in MMIS-2 — every process has read and write access to all the data.

The Large Object Server has a locking scheme [Cro90] so that

- a medium instance may be read-locked by multiple processes simultaneously;
- write locks are only granted for instances that are not currently read-locked;
- to avoid deadlock, a process may only lock one set of instances at a time.

The Large Object Server's implementation predates the use of labels so that instances have a unique identifier and an internal type, but only the identifier is exported. Identifiers are assigned in order from the creation of the database so, for instance, `image/11` and `blot/11` cannot co-exist; this behaviour could be modified to use labels properly.

There is no locking of tuples within the Semantic Server — changing a tuple and restoring it actually generates a *new* tuple and duplicate insertions cause an error. Locking should be provided for entire semantic nets so that it would not be possible for a process to change some part of a net while another is reading it. This could be achieved by assigning a tag to tuples from a particular semantic net, as described previously, so that sets of tags could be locked.

In order to prevent re-interpretation of medium instances in MMIS-2, interpreters look in the Semantic Server for an instance's `is_medium_instance` arc before interpreting it — if the arc is already there then interpretation succeeds with no further action; this can cause problems if, for instance, two 'interpret blot 57' requests are sent to the Interpretation Server, then each can end up with a separate interpreter; both interpreters then check the Semantic Server for blot 57's `is_medium_instance` arc, fail to find it and proceed with interpretation; the first interpreter to finish will successfully store the new semantic net, but the second will receive ‘Duplicate tuple insertion’ errors from the Semantic Server. This is a result of the nature of MMIS-2’s interpreters rather than its Interpretation Server and although the Interpretation Server could be modified to check that an incoming instance is not already in transit, it would be better to implement a locking scheme in the Semantic Server as described above.

8.6 Application Evaluation

MMIS-2’s current application, storing images of deciduous tree leaves, is somewhat artificial; the system should be evaluated with real applications to test its generality and extensibility.

8.7 User Interface

The User Interface should be evaluated by a number of independent users; since there has been no commitment to user interface design within the project, such users would have to be expert, or at least coached in the existing tools. The author has demonstrated the system to a large number of visitors and has received favourable comments about its capabilities, but there is a need to develop interfaces for novice users — one example would be a browser for label sets, semantic nets and hypermedia webs that presented the user with a scrollable ‘map’ of the whole structure.

Prolog provides many facilities for the generation of data post-query using rules to rename predicates, combine arcs into new predicates, to generate new representations and so on. The management of such user views of system data should be investigated.

Since MMIS-2’s semantic net arcs each have an associated label rather than just a string, hypermedia links can be annotated with any object; this allows richer webs to be built than is possible with other hypermedia systems. For historical reasons, the Query Tool only allows textual (`atom/`) labels to be assigned to links, but this could easily be changed.

Appendix A

Prolog Code for the Query Engine

This Appendix presents the Prolog code for MMIS-2's query engine. Only the interfaces of the predicates which provide Semantic Server calls are reproduced because much of their work is done by external C++ functions; the bodies of the predicates which use the Semantic Server calls to provide users with functional equivalence to Prolog and hypermedia are given in full. In the comments preceding predicates, an argument which must be instantiated is specified as `+Name`, one which must not be instantiated as `-Name` and one which may be either is specified as `?Name`.

```
/*********************************************************
 *                                                       *
 * This first group of predicates imports the Semantic   *
 * Server client functions from C++                      *
 *                                                       *
 *********************************************************/

losclient_register.
/*
 * Get a connection to the Large Object Server
 *
 * fail: if no server is running on the default machine (spec0) or
 *          on the machine specified by the LOSERVER environment
 *          variable
 * fail: if we are already registered
 */

losclient_unregister.
/*
 * Release our connection to the Large Object Server
 *
 * fail: if we are not already registered
 */
```

```
ssclient_register.
/*
 * Get a connection to the Semantic Server
 *
 * fail: if no server is running on the default machine (spec0) or
 *           on the machine specified by the SSERVER environment
 *           variable
 * fail: if we are already registered
 */

ssclient_unregister.
/*
 * Release our connection to the Semantic Server
 *
 * fail: if we are not already registered
 */


ssclient_get_base_class(?Class).
/* The base class for MMIS-2's semantic nets is Class
 *
 * fail: if we're not registered with the Semantic Server
 */

ssclient_declare(+Mmis_field_types, ?Class).
/*
 * Class is a new IFS class whose format is Mmis_field_types;
 * Mmis_field_types is an atom consisting of any combination of
 * L (label), F (float) and I (IFS word) specifiers.
 *
 * fail: if we're not registered with the Semantic Server
 * fail: if Mmis_field_types is invalid
 */

ssclient_undeclare(+Class).
/*
 * Discard Class and all its tuples
 *
 * fail: if we're not registered with the Semantic Server
 * fail: if Class doesn't exist
 */

ssclient_convert(+Class, ?Mmis_field_types).
/*
 * Mmis_field_types is the format of Class.
 *
```

```
 * fail: if we're not registered with the Semantic Server
 * fail: if Mmis_field_types is not the format of Class
 */

ssclient_insert_tuples(+Class, +Tuples).
/*
 * Insert Tuples into Class. Tuples is a sequence of tuples, each
 * of which is a sequence of label, float or integer fields, eg:-
 *
 *  ssclient_insert_tuples(
 *      1,
 *      [
 *          [text/99,atom/is_medium_instance,text/99,1.0],
 *          [text/3972,atom/is_medium_instance,text/3972,1.0]
 *      ]
 *  ).
 *
 * fail: if we're not registered with the Semantic Server
 * fail: if Class does not exist
 * fail: if any tuple is not in the correct format for Class
 * fail: if a label's value is not appropriate for its type, eg:-
 *           image/10 is legal, image/10.0 is not
 * fail: if a label's type is non-existent, eg:-
 *           video/55 is illegal
 */

ssclient_search_and_return(+Class, +Alg, +Ops, +Rfs, +Query, ?R)
/* R is the sequence of tuples in Class that match Query.
 *
 * Alg is one of:-
 *     s  (simple match)
 *     o1 (simple match)
 *     o2 (simple match)
 *     i  (identical match)
 *
 * Ops is a sequence of comparison operators, one per MMIS-2 field.
 * The allowed operators are:-
 *     =   (logical equality)
 *     \== (logical inequality)
 *     @>  (logical greater than)
 *     @<  (logical smaller than)
 *     =   (arithmetic equality)
 *     =\= (arithmetic inequality)
 *     >   (arithmetic greater than)
 *     <   (arithmetic smaller than)
 *
```

```
* Rfs is a sequence of atoms, one per MMIS-2 field, specifying
* whether the corresponding field should be returned (y) or not
* returned (n).
*
* Query is a sequence of patterns, one per MMIS-2 field, for
* matching with stored fields. A label pattern must have one of
* the following forms:-
*     +Type/+Value
*     +Type/-Value
*     -Type/-Value
* The under-score character generates a new variable name for the
* search, i.e. the query [_/_,_/_,_/_,_] is equivalent to
* [A/B,C/D,E/F,G]. If a field is not completely instantiated then
* the corresponding operator is forced to logical equality (but
* the operator given should still be legal!).
*
* Examples:-
* ssclient_search_and_return(
*     1,
*     s,
*     [=,=,=,=],
*     [n,n,n,y],
*     [abs/876,atom/displayed_as,blot/243,_],
*     [[1.0]]
* ).
* ssclient_search_and_return(
*     1,
*     s,
*     [=,=,=,<],
*     [y,n,y,n],
*     [abs/A,atom/se_of,abs/B,0.9],
*     R
* ).
* ssclient_search_and_return(
*     1,
*     o1,
*     [=,=,=,=],
*     [y,n,y,y],
*     [A/B,atom/is_medium_instance,A/B,_],
*     R
* ).
*
* fail: if we're not registered with the Semantic Server
* fail: if Class does not exist
* fail: if Ops, Rfs or Query is not in the correct format for
*           Class
```

```
 * fail: if a label's value is not appropriate for its type, eg:-
 *           image/10 is legal, image/10.0 is not
 * fail: if a label's type is non-existent, eg:-
 *           video/55 is illegal
 * fail: if any member of Ops or Rfs is illegal
 * fail: if Alg is illegal
 */

ssclient_cardinality(+Class, +Alg, +Ops, +Rfs, +Query, ?C)
/*
 * Identical to ssclient_search_and_return except that C is the
 * number of tuples that matched rather than the tuples
 * themselves
 */

ssclient_delete(+Class, +Alg, +Ops, +Rfs, +Query, ?N)
/*
 * Identical to ssclient_search_and_return except that each
 * matching tuple found is deleted. N is the number of tuples
 * deleted.
 */


/************************************************************
 *                                                          *
 * This next group of predicates provides Prolog            *
 * functional equivalence                                   *
 *                                                          *
 ************************************************************/

/*
 * quadruple(?Source, ?Relationship, ?Score).
 *
 * This is the underlying predicate which provides functional
 * equivalence to Prolog
 */
quadruple(B/C, D/E, F/G, A) :-
    /*
     * If the class hasn't been specified then use the base
     * class
     */
    base_class(H),
    quadruple(H, B/C, D/E, F/G, A).
quadruple(A, C/D, E/F, G/H, B) :-
    /*
     * Decide which fields need returning and build a template
```

```
     * for the returned tuple. All fields which are not wholly
     * instantiated will be returned.
     */
    (   var(B) ->
        I=y,
        J=[B]
    ;   I=n,
        J=[]
    ),
    (   var(H) ->
        K=y,
        L=[G/H|J]
    ;   K=n,
        L=J
    ),
    (   var(F) ->
        M=y,
        N=[E/F|L]
    ;   M=n,
        N=L
    ),
    (   var(D) ->
        O=y,
        P=[C/D|N]
    ;   O=n,
        P=N
    ),
    /*
     * Use simple match with logical equality because Prolog
     * does all the unification and cross-comparisons after
     * retrieval
     */
    ssclient_search_and_return(
        A, s, [=,=,=,=], [O,M,K,I], [C/D,E/F,G/H,B], Q
    ),
    /* Return the next tuple from the result */
    member(P, Q).

/* Provide a ternary predicate front end for each arc */
is_medium_instance(A, C, B) :-
    quadruple(A, atom/is_medium_instance, C, B).

aspect_ratio(A, C, B) :-
    quadruple(A, atom/aspect_ratio, C, B).

circularity(A, C, B) :-
```

```
    quadruple(A, atom/circularity, C, B).

tbb_centre_x(A, C, B) :-
    quadruple(A, atom/tbb_centre_x, C, B).
/* etc... */


/* Derive leaf_count and noise_count post-query */
leaf_count(image/I, int/N, 1.0):-
    is_medium_instance(image/I, image/I, 1.0),
    findall(L, has_leaf(image/I, blot/L, _), F),
    length(F, N).

noise_count(image/I, int/N, 1.0):-
        is_medium_instance(image/I, image/I, 1.0),
    findall(L, has_noise(image/I, blot/L, _), F),
    length(F, N).

/*
 * Called by the Query Tool. This executes Query, stores the
 * resulting label set specied by Resultname in the Large Object
 * Server and sends the label set identifier (via redirection)
 * to the Query Tool
 */
query(+Queryid,-Resultname,+Query).


/***********************************************************
 *                                                         *
 * This last group of predicates provides hypermedia       *
 * facilities                                              *
 *                                                         *
 ***********************************************************/

/*
 * Called by the Query Tool. This finds all matching links by
 * searching the link class and collecting all matching
 * solutions to derived_link
 */
link_request(+Requestid, ?St/?Sv, ?Rt/?Rv, ?Dt/?Dv, ?Score).

/*
 * Generate post-query links between images and leaves, scale
 * markers and noise
 */
derived_link(image/I, atom/has_displayable_leaf, blot/B, S):-
```

```
    has_leaf(image/I, abs/A, S),
    displayed_as(abs/A, blot/B, 1.0).

derived_link(image/I, atom/has_displayable_noise, blot/B, S):-
    has_noise(image/I, abs/A, S),
    displayed_as(abs/A, blot/B, 1.0).

derived_link(image/I, atom/has_displayable_sm, blot/B, S):-
    has_scale_marker(image/I, abs/A, S),
    displayed_as(abs/A, blot/B, 1.0).
```

Appendix B

A Glossary of Predicates

Below is a list of the arcs used in MMIS-2's semantic net, given as ternary predicate templates of the form:

$< R_v >$(< S_t/ >, < D_t/ >, < S >).

The reader is referred to Table 5.1 for an explanation of the types of the types of argument values. The score `scale_score` is the identification error of the parent image's scale marker, `classification_score` is the distance of a blot from its class centroid and `positional_score` is the accuracy of a positional relationship — the derivation of these three scores is explained in Section 5.2; all other scores are `1.0` (certain).

`area(abs/, m2/, scale_score)` The area of a leaf in m^2.

`area(blot/, p2/, 1.0)` The area of a blot in p^2.

`aspect_ratio(blot/, float/, 1.0)` The ratio $\frac{\text{Length}}{\text{Width}}$ for a blot.

`circularity(blot/, float/, 1.0)` The ratio $\frac{4\pi.\text{Area}}{\text{Perimeter}^2}$ for a blot.

`class(blot/, int/, classification_score)` The nearest shape class to the specified blot, calculated from its transparency, aspect ratio, circularity, irregularity and the ratio $\frac{\text{Convex maxima count}}{\text{Concave maxima count}}$.

`class(image/, int/, 1.0)` The class of the specified image, currently always 2 (8-bit monochrome).

`concave_max_count(blot/,int/, 1.0)` The number of major concave extremities on a blot's perimeter.

`convex_max_count(blot/,int/, 1.0)` The number of major convex extremities on a blot's perimeter.

`displayed_as(abs/, blot/, 1.0)` The blot as which a leaf, scale marker or piece of noise can be displayed.

`e_of(abs/, abs/, positional_score)` The first leaf is east of the second.

`has_leaf(image/, abs/, 1.0)` The specified abstract node is a leaf within the specified image.

`has_noise(image/, abs/, noise error)` The specified abstract node is a piece of noise within the specified image.

`has_scale_marker(image/, abs/, scale_score)` The specified abstract node is a scale marker within the specified image.

`height(image/, m/, scale_score)` An image's y-dimension in m.

`height(image/, p/, 1.0)` An image's y-dimension in p.

`hull_area(abs/, m2/, scale_score)` The area of a leaf's convex hull, in m^2.

`hull_area(blot/, p2/, 1.0)` The area of a blot's convex hull, in p^2.

`irregularity(abs/, 'm-1rad'/, scale_score)` A quantity describing the unevenness of a leaf's perimeter, in $rad\, m^{-1}$. Calculated from the corresponding blot and the parent image's length scale.

`irregularity(blot/, 'p-1rad'/, 1.0)` A quantity describing the unevenness of a blot's perimeter, in $rad\,p^{-1}$. Calculated as $\frac{1}{N}\left[\sum_{i=1}^{N}|\Delta a| - 2\pi\right]$, where N is the number of pixels around the blot's perimeter and Δa is the angle change in moving from one pixel to another.

`is_medium_instance(blot/, blot/, 1.0)` Identifies a blot as a medium instance.

`is_medium_instance(image/, image/, 1.0)` Identifies an image as a medium instance.

`lc_end1_x(abs/, m/, scale_score),`

`lc_end1_x(abs/, p/, 1.0),`

`lc_end1_y(abs/, m/, scale_score),`

`lc_end1_y(abs/, p/, 1.0),`

`lc_end2_x(abs/, m/, scale_score),`

`lc_end2_x(abs/, p/, 1.0),`

`lc_end2_y(abs/, m/, scale_score),`

`lc_end2_y(abs/, p/, 1.0)` The coordinates of the ends of a leaf's longest chord, in m and p from the top left corner of the image.

`lc_end1_x(blot/, p/, 1.0),`

`lc_end1_y(blot/, p/, 1.0),`

`lc_end2_x(blot/, p/, 1.0),`

`lc_end2_y(blot/, p/, 1.0)` The coordinates of the ends of a blot's longest chord, in p from the top left corner of its upright bounding box.

`lcp_in1_x(abs/, m/, scale_score),`

`lcp_in1_x(abs/, p/, 1.0),`

lcp_in1_y(abs/, m/, scale_score),

lcp_in1_y(abs/, p/, 1.0),

lcp_in2_x(abs/, m/, scale_score),

lcp_in2_x(abs/, p/, 1.0),

lcp_in2_y(abs/, m/, scale_score),

lcp_in2_y(abs/, p/, 1.0),

lcp_out1_x(abs/, m/, scale_score),

lcp_out1_x(abs/, p/, 1.0),

lcp_out1_y(abs/, m/, scale_score),

lcp_out1_y(abs/, p/, 1.0),

lcp_out2_x(abs/, m/, scale_score),

lcp_out2_x(abs/, p/, 1.0),

lcp_out2_y(abs/, m/, scale_score),

lcp_out2_y(abs/, p/, 1.0) The coordinates of the ends of a leaf's longest chord perpendiculars, in m and p from the top left corner of the image.

lcp_in1_x(blot/, p/, 1.0),

lcp_in1_y(blot/, p/, 1.0),

lcp_in2_x(blot/, p/, 1.0),

lcp_in2_y(blot/, p/, 1.0),

lcp_out1_x(blot/, p/, 1.0),

lcp_out1_y(blot/, p/, 1.0),

`lcp_out2_x(blot/, p/, 1.0)`,

`lcp_out2_y(blot/, p/, 1.0)` The coordinates of the ends of a blot's longest chord perpendiculars, in p from the top left corner of its upright bounding box.

`length(abs/, m/, scale_score)` The length of a leaf's longest chord, in m.

`length(blot/, p/, 1.0)` The length of a blot's longest chord, in p.

`length_scale(image/, 'm-1p'/, scale_score)` Conversion factor between p and m measurements within an image, in pm^{-1}.

`n_of(abs/, abs/, positional_score)` The first leaf is north of the second.

`ne_of(abs/, abs/, positional_score)` The first leaf is north east of the second.

`nw_of(abs/, abs/, positional_score)` The first leaf is north west of the second.

`perimeter(abs/, m/, scale_score)` The length of a leaf's perimeter, in m.

`perimeter(blot/, p/, 1.0)` The length of a blot's perimeter, in p.

`s_of(abs/, abs/, positional_score)` The first leaf is south of the second.

`se_of(abs/, abs/, positional_score)` The first leaf is south east of the second.

`tbb_Ax(abs/, m/, scale_score)`,

`tbb_Ax(abs/, p/, 1.0)`,

`tbb_Ay(abs/, m/, scale_score)`,

`tbb_Ay(abs/, p/, 1.0)`,

`tbb_Bx(abs/, m/, scale_score)`,

`tbb_Bx(abs/, p/, 1.0)`,

`tbb_By(abs/, m/, scale_score)`,

`tbb_By(abs/, p/, 1.0),`

`tbb_Cx(abs/, m/, scale_score),`

`tbb_Cx(abs/, p/, 1.0),`

`tbb_Cy(abs/, m/, scale_score),`

`tbb_Cy(abs/, p/, 1.0),`

`tbb_Dx(abs/, m/, scale_score),`

`tbb_Dx(abs/, p/, 1.0),`

`tbb_Dx(blot/, p/, 1.0),`

`tbb_Dy(abs/, m/, scale_score),`

`tbb_Dy(abs/, p/, 1.0)` The coordinates of the corners of a leaf's true bounding box, in m and p from the top left corner of the image; A, B, C and D are the corners taken clockwise from `end2` of the leaf's longest chord.

`tbb_Ax(blot/, p/, 1.0),`

`tbb_Ay(blot/, p/, 1.0),`

`tbb_Bx(blot/, p/, 1.0),`

`tbb_By(blot/, p/, 1.0),`

`tbb_Cx(blot/, p/, 1.0),`

`tbb_Cy(blot/, p/, 1.0),`

`tbb_Dy(blot/, p/, 1.0)` The coordinates of the corners of a blot's true bounding box, in p from the top left corner of its upright bounding box; A, B, C and D are the corners taken clockwise from `end2` of the blot's longest chord.

`tbb_centre_x(abs/, m/, scale_score),`

tbb_centre_x(abs/, p/, 1.0),

tbb_centre_y(abs/, m/, scale_score),

tbb_centre_y(abs/, p/, 1.0) The coordinates of the centre of a leaf's true bounding box, in m and p from the top left corner of the image.

tbb_centre_x(blot/, p/, 1.0),

tbb_centre_y(blot/, p/, 1.0) The coordinates of the centre of a blot's true bounding box, in p from the top left corner of its upright bounding box.

tbb_overlaps(abs/, abs/, 1.0) The true bounding box of the first leaf overlaps that of the second.

tbb_theta(blot/, rad/, 1.0) The anti-clockwise angle, in rad, from the x-axis to a blot's longest chord (where the latter is considered to be a vector from its end1 to its end2).

transparency(blot/, float/, 1.0) The ratio $\frac{\text{Hull area}-\text{Area}}{\text{Hull area}}$ for a blot.

ubb_height(abs/, m/, scale_score) The height of a leaf's upright bounding box, in m.

ubb_height(blot/, p/, 1.0) The height of a blot's upright bounding box, in p.

ubb_top_left_x(abs/, m/, scale_score),

ubb_top_left_x(abs/, p/, 1.0),

ubb_top_left_y(abs/, m/, scale_score),

ubb_top_left_y(abs/, p/, 1.0) The coordinates of the top left corner of a leaf's upright bounding box, in m and p from the top left corner of the image.

ubb_width(abs/, m/, scale_score) The width in m of a leaf's upright bounding box.

`ubb_width(blot/, p/, 1.0)` The width in p of a blot's upright bounding box.

`w_of(abs/, abs/, positional_score)` The first leaf is west of the second.

`width(abs/, m/, scale_score)` A leaf's width in m (i.e. the combined length of its longest chord perpendiculars).

`width(blot/, p/, 1.0)` A blot's width in p.

`width(image/, m/, scale_score)` An image's x-dimension in m.

`width(image/, p/, 1.0)` An image's x-dimension in p.

Appendix C

Performance Evaluation

The performance of MMIS-2's various components has been evaluated and the results are presented below. In order to put this evaluation into context, a slightly different system is also evaluated — this modified system is referred to as *MMIS-P*. From the point of view of a user issuing queries, MMIS-2 and MMIS-P are equivalent; however, new images cannot be interpreted in MMIS-P. Other differences that would become apparent if all MMIS-2's capabilities were made available in MMIS-P are described in subsequent sections.

Recall that MMIS-2's Semantic Server is client-server code on top of the IFS simulator and that its Query Engine is a hybrid of Prolog and C++ (Appendix A). In MMIS-P, the Query Engine and Semantic Server are replaced by Prolog itself with the addition of a small amount of code to store label sets, derive post-query arcs and communicate with the Query Tool; semantic nets are represented as a list of ternary predicates, held by Prolog in RAM and loaded at the start of each query session.

The system configuration for the discussion that follows is:

1. a Sun 4/280 with 64 Mb of RAM, 64 Mb of swap space and 2 Gb of disc space;

2. a SPARCstation 2 with 32 Mb of RAM, 32 Mb of swap space and 207 Mb of disc space;

3. 10 Mbit/s Ethernet connection between the SPARCstation and 4/280;

4. all MMIS data files and executables are retrieved from the 4/280's disc;
5. the Large Object Server runs on the 4/280;
6. the Query Tool runs on the SPARCstation;
7. for MMIS-P, the Prolog process providing the query engine and semantic net store runs on the SPARCstation;
8. for MMIS-2, the Query Engine and Semantic Server run on the SPARCstation, so that the Query Engine has local access to the semantic nets, as in MMIS-P;
9. the only other processes running are Sunview and a command tool on the SPARC-station, and SunOS system administration tasks on both machines.

C.1 Querying

Each plot in Figures C.1– C.6 shows the total time taken to execute the corresponding query, for four different sets of images. All times are an average of five readings with an error of $\pm 0.01s$. The only external overheads in executing a query are asking SunOS for elapsed times and clashes with SunOS system administration tasks.

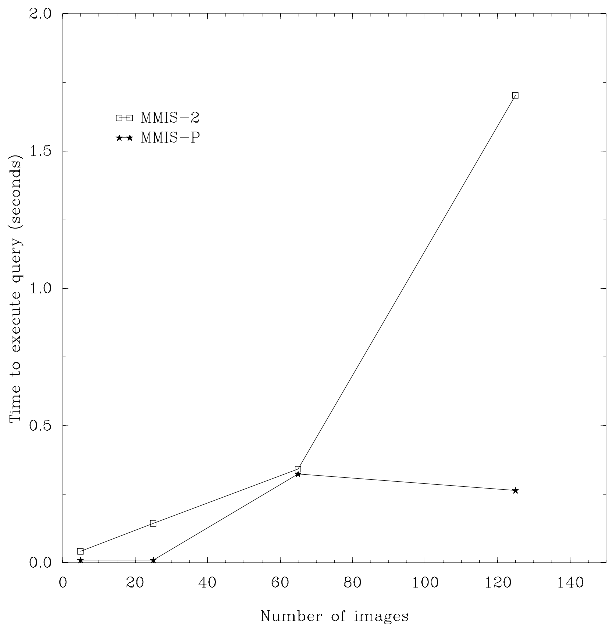


Figure C.1: Execution times for query 1

1. "Find all the images in the system"

```
findall(image/Id/S,
    is_medium_instance(image/Id, image/Id, S),
    Set)
```

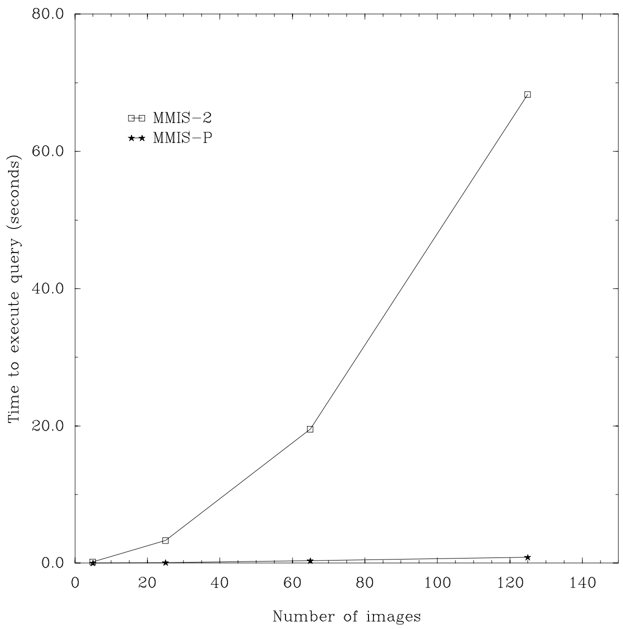


Figure C.2: Execution times for query 2

2. "Find the images with the greatest number of leaves"

```
setof(N/I/S, leaf_count(image/I, int/N, S), NISs),
concatenate(_, [Greatest/_/_], NISs),
findall(image/Id/S, member(Greatest/Id/S, NISs), Set1),
Set = [int/Greatest/2.0 | Set1]
```

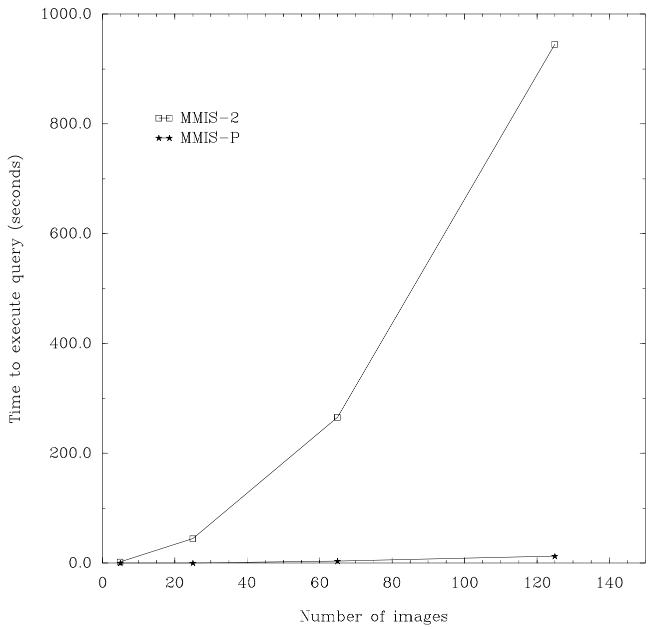


Figure C.3: Execution times for query 3

3. "Find the displayable representations of leaves with transparency in the range [0.35, 0.45] and group them with their images"

```
findall([image/Id/2.0, Leaves/1.0]/S1,
    (
    is_medium_instance(image/Id,image/Id,1.0),
    find_all(
        Blot/T,
        (
            has_leaf(image/Id, Leaf, _),
            displayed_as(Leaf, Blot, _),
            transparency(Blot, float/T, _),
            T >= 0.35, T =< 0.45
        ),
        Leaves),
    length(Leaves, S1)
    ),
    Set
)
```

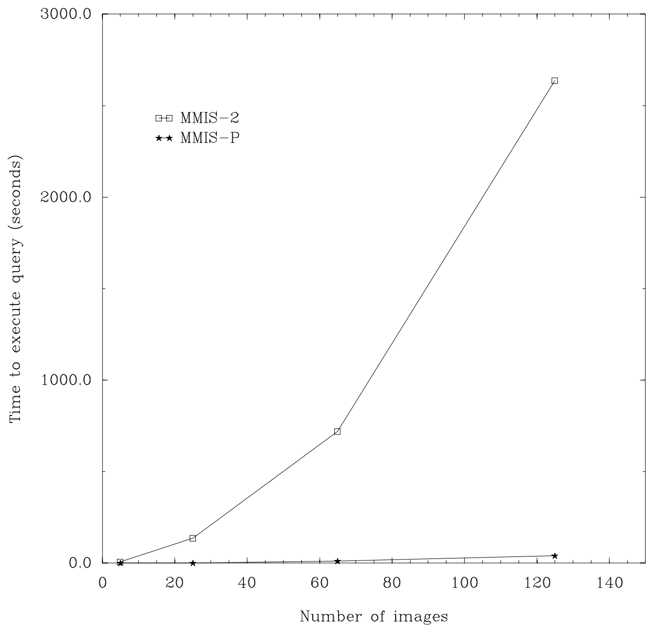


Figure C.4: Execution times for query 4

4. “Find the displayable versions of all pairs of leaves such that the first is north of the second”

```
findall(
    [blot/Blot_id1/3, blot/Blot_id2/2, image/Image_id/1]/S1,
    (
    n_of(abs/Abs_id1, abs/Abs_id2, S1),
    (\+tbb_overlaps(abs/Abs_id1, abs/Abs_id2, _)),
    displayed_as(abs/Abs_id1, blot/Blot_id1, _),
    displayed_as(abs/Abs_id2, blot/Blot_id2, _),
    has_leaf(image/Image_id, abs/Abs_id1, _)
    ),
    Set
)
```

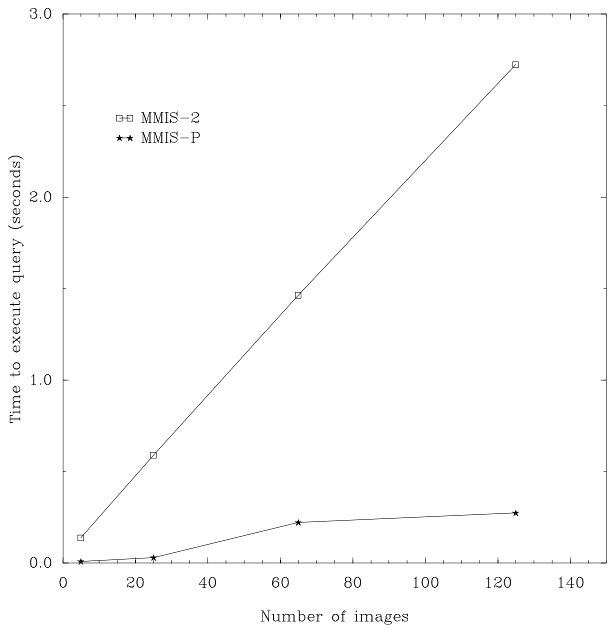


Figure C.5: Execution times for query 5

5. "Find the biggest leaf in the database whose area has been measured in m^2 and return its size and its displayable version"

```
findall(A,area(_,m2/A,S),As),
biggest(As, Biggest),
findall(
    Blot/S,
    (
        area(Leaf, m2/Biggest, S1),
        displayed_as(Leaf, Blot, S2),
        S is S1 * S2
    ),
    Set1
),
Set = [m2/Biggest/2.0 | Set1]
```

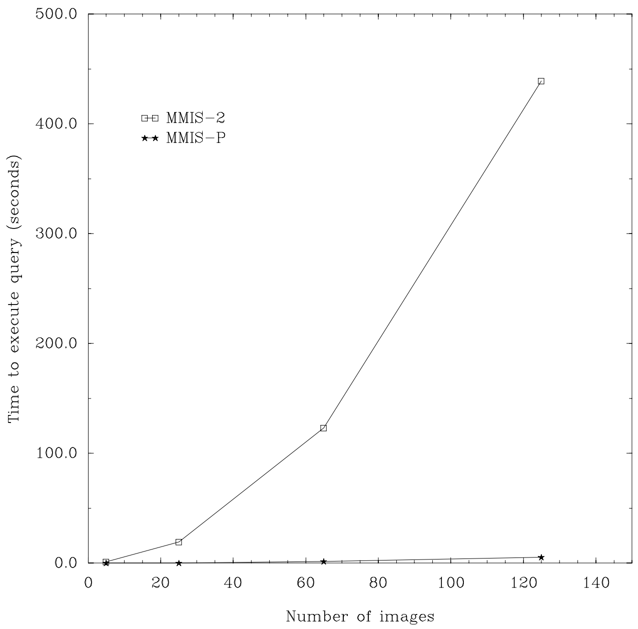


Figure C.6: Execution times for query 6

6. "Find all blots with the same shape class as blot 3951"

```
class(blot/3951, int/SC, _),
findall(Blot/S,
    (
    class(Blot, int/SC, S),
    ((displayed_as(Leaf, Blot, _), has_leaf(_, Leaf, _))->true)
    ),
    Set
)
```

It can be seen that the execution times for MMIS-2 are much greater than those for MMIS-P in all cases; there are a number of reasons for this:

1. The IFS simulator is somewhat inefficient; in particular, it performs a sequential search of all its tuples for each search request, whereas Prolog *indexes* its predicates by name.

2. The representation of semantic nets as a set of ternary predicates rather than a large set of `quadruple` clauses is the natural Prolog implementation and means that a full tuple search is only required in the rare case of an uninstantiated arc name.

3. Once the IFS has collected a sequence of tuples, it is dispatched to C++ via the RPC protocol; the C++ sequence then has to be sent to Prolog tuple by tuple (because SICStus cannot transport sequences directly between C and Prolog); finally, in order to provide functional equivalence to Prolog easily, the sequence is again transported tuple by tuple to the user's query so that Prolog thinks it's accessing a normal list of predicates. In contrast, MMIS-P has fast access to tuples within the same process, avoiding most of the tasks performed by MMIS-2.

C.2 Result Storage and Retrieval

The time taken to store a query's label set result in the Large Object Server is equivalent for MMIS-2 and MMIS-P. Based on an average of ten readings with 125 images, and with an error of $\pm 0.01s$, only query 4's result takes a significant length of time to store — $47.83s$ — while the results from all other queries are stored in less than a second. This is because query 4's result is a set containing 1094 smaller sets, each of which contains three elements — most of the time is spent sending these 1095 sets over the network to the Large Object Server.

Exact timings for the retrieval and display of a label set by the Query Tool were difficult to collect but are broadly similar to the storage times. Demand-retrieval of

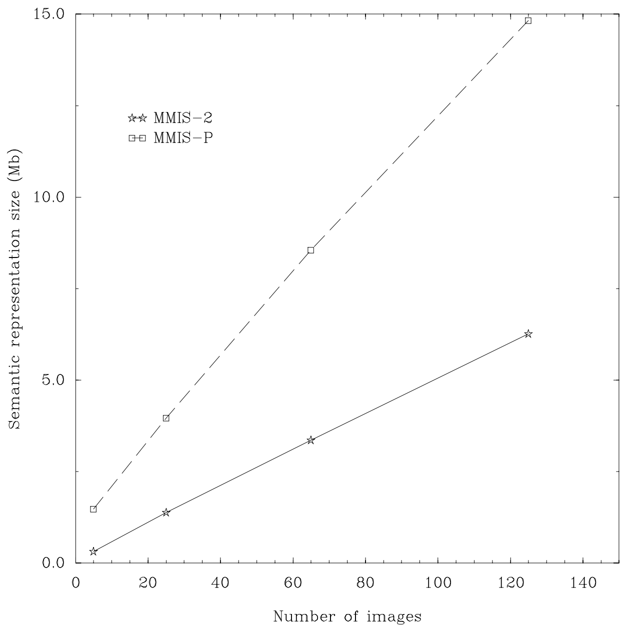


Figure C.7: Memory used to store semantic nets

medium instances is fast, taking no longer than a few seconds for an image, even when the network is heavily loaded.

C.3 Size of the Semantic Net in Memory

Figure C.7 compares the sizes of semantic representation held in RAM by MMIS-2 and by MMIS-P; the size of the Prolog and C++ code is comparable for both systems — approximately 1.5 Mb. It can be seen that MMIS-P's internal representation of the semantic nets is much less concise than the MMIS-2's so that it cannot represent as many images on any given machine. For 125 images, MMIS-P has insufficient memory left to run the Query Tool.

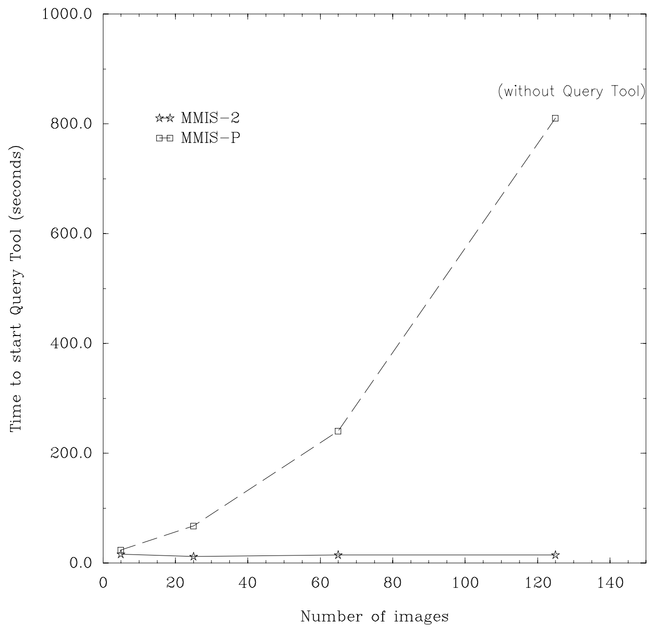


Figure C.8: Query Tool start times

C.4 Starting the Query Tool

When the user begins a query session, it is important that the Query Tool starts quickly. MMIS-2's semantic net storage is persistent, so the Query Tool only has to load the Sunview and Query Engine code; in contrast, for MMIS-P, the Query Tool must load the whole set of semantic nets from disc as well. The real startup times for the two systems ($\pm 1s$) are shown in Figure C.8. MMIS-2's Query Tool starts quickly, regardless of the number of images in the system, whereas the start time of MMIS-P's Query Tool increases rapidly.

C.5 Multi-process Access to Semantic Nets

Because MMIS-P takes a private copy of the semantic nets from a single system-wide file, any changes to this file cannot immediately affect query processes already running; a solution is to write summaries of changes to semantic nets to disc so that users can instruct their query process to take note of the changes as required. Note also that interpreters in MMIS-P would not have programmatic access to semantic nets unless they were written in Prolog. This is a less attractive scheme than that of MMIS-2 where changes to the Semantic Server are immediately visible to all query processes and interpreters.

C.6 Summary

To summarise the comparison of MMIS-2 and MMIS-P:

MMIS-2	MMIS-P
Tuple sets passed Semantic Server–C++–Prolog	Tuples available within the same process
Query language not pre-processed into set-based queries but large speed increase when this is done by hand (see Section 7.2)	Query language optimised for tuple-based queries
Semantic net is ~50 Kb/image for 125 images	Semantic net is ~119 Kb/image for 125 images
Constant, fast Query Tool start	Slow, increasing Query Tool start
Shared access to semantic nets	Single-process access to a private copy of the semantic nets
Semantic net changes available immediately, system-wide	Semantic net changes only available by explicit update
All processes have programmatic access to semantic nets	Only Prolog processes have programmatic access to semantic nets, non-Prolog processes must use an inefficient textual protocol

Appendix D

An ODA Tree Description Document

This appendix contains part of the ODA definition for a proposed tree description document, browse it like any other (upside-down) tree — top-down, depth first is probably best. It represents the *generic logical* description of the document — i.e. the part that says what logical objects a tree description may contain. It corresponds to a regular grammar where each object or node in the tree is either *basic* (mapping to an actual piece of the document) or *composite* (composed of other objects). The *subordinates* or children of composite objects are generated according to one of the following:

SEQuence The subordinates occur in sequence.

CHOice Only one of the subordinates occurs.

AGGregate The subordinates occur in any order.

Additionally, each subordinate may be further specified as:

OPTional The subordinate is optional.

REPeated The subordinate can be repeated.

OPTional, REPeated The subordinate may occur 0, 1 or more times.

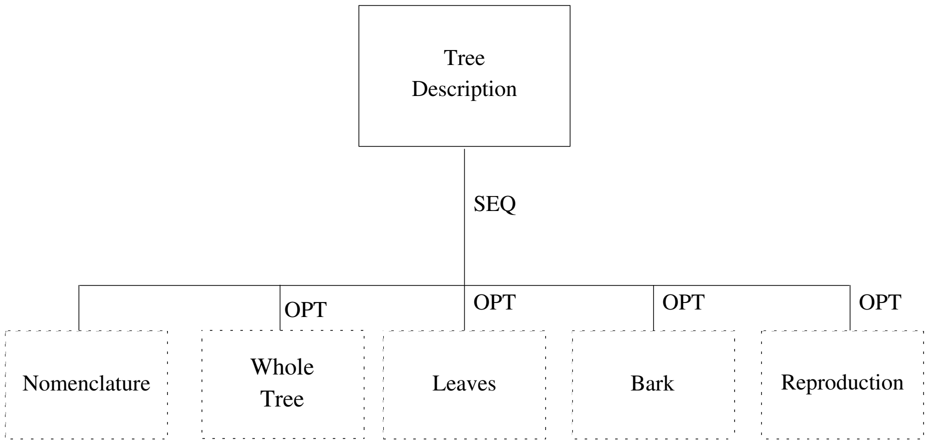
If a subordinate has no specifier then it is compulsory.

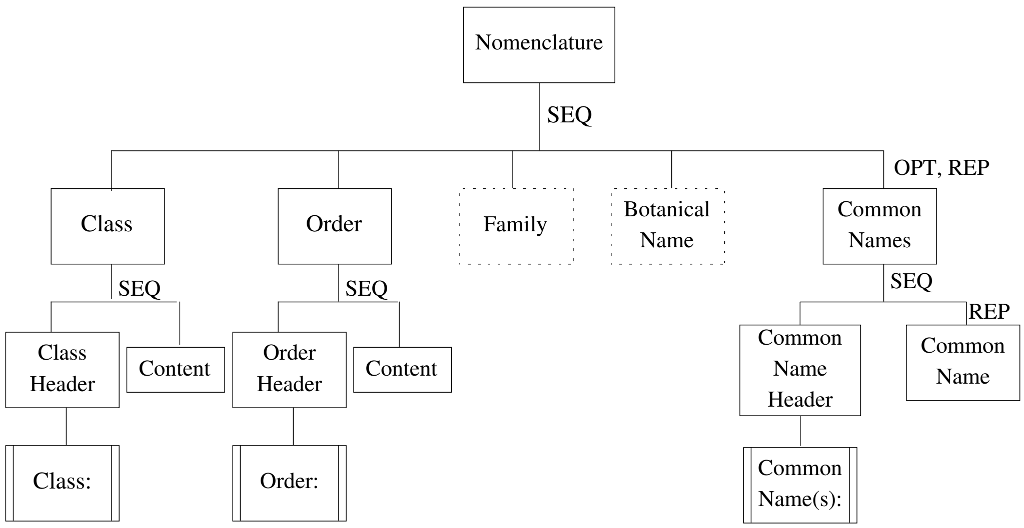
Dashed boxes in the figures are composite objects expanded in a separate figure. Basic objects are mappings to actual, atomic parts of the document that appear on paper or screen. Each atomic part is called a *content portion* and may appear in the generic or specific ODA structures. A *generic content portion* is a piece of the document that always appears when the relevant branch of the tree is instantiated and is indicated by a box with double sides — eg. the string `"BARK"` is automatically included if the optional branch 'Reproduction' is filled in. Conversely, *specific content portions* are added below the generic basic objects during the editing process when, for example, pictures of leaves are included.

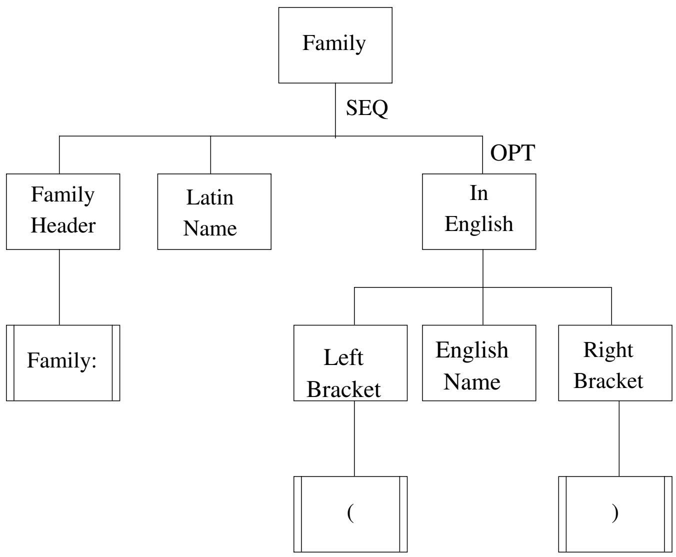
It is informative to consider the leaves of the tree written in sequence as below, where the items in bold face are generic content portions and the items in normal face are the (names of the) specific content portions:

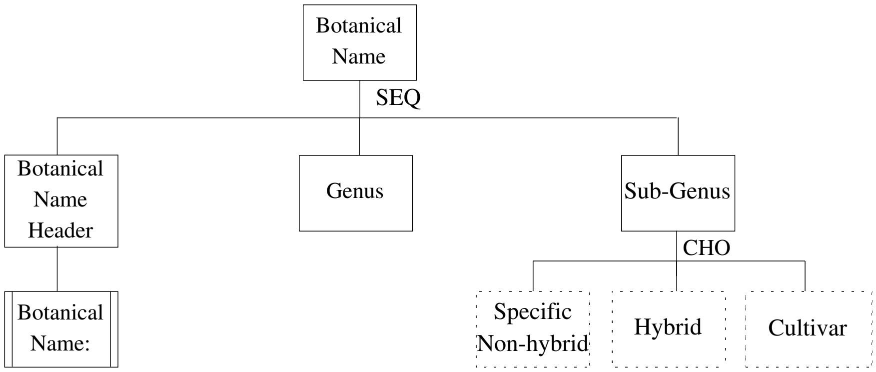
> **Class:** Content **Order:** Content **Family:** Latin Name (English Name)
> **Botanical Name:** Genus, etc...

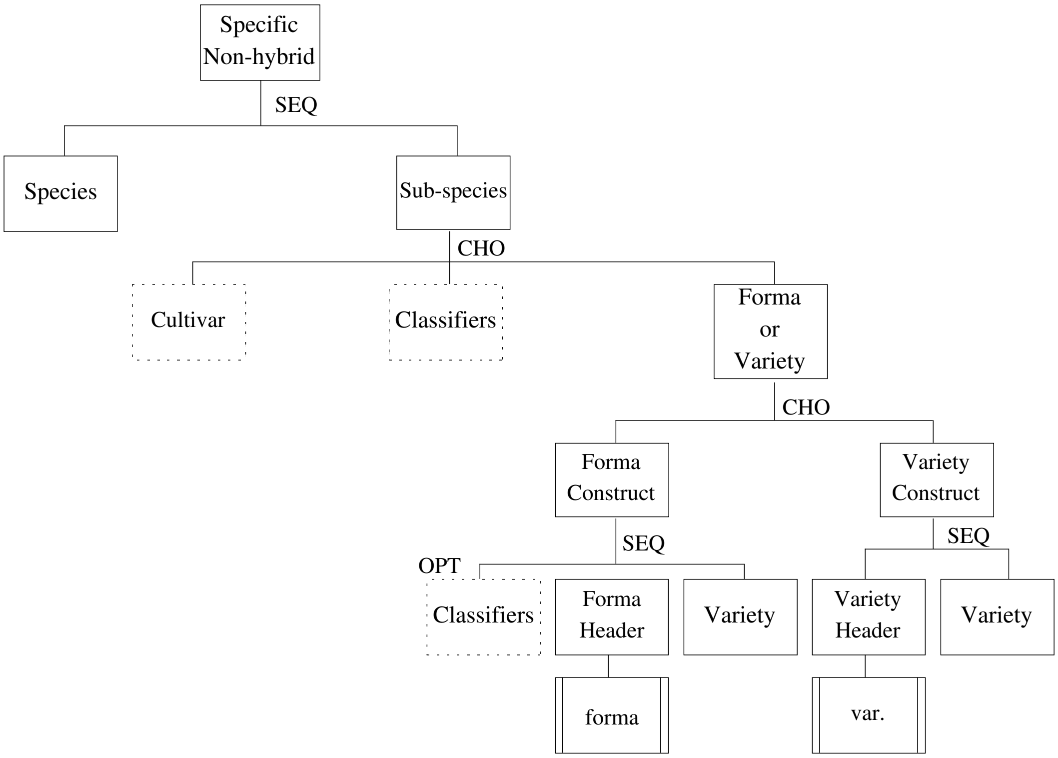
The 'Nomenclature' branch is the most detailed, reflecting the rich semantic content and standard structure of classifications, eg. "Picea Pungens Engelm. forma glauca (Reg.) Beissri"; such a detailed branch allows more precise and meaningful queries of the form "What trees in the system are in the family Rosaceæ and were classified by Linnæus?". By contrast, the reproduction arm is sparse because the only data that it's reasonable to include without further interpreters are a few pictures and a short text description.

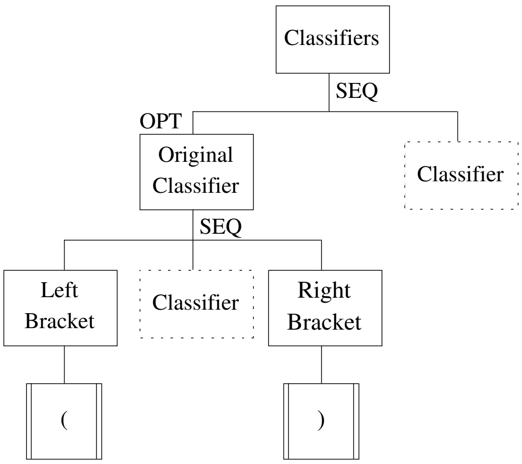

Tree Description
SEQ
OPT
OPT
OPT
OPT
Nomenclature
Whole Tree
Leaves
Bark
Reproduction

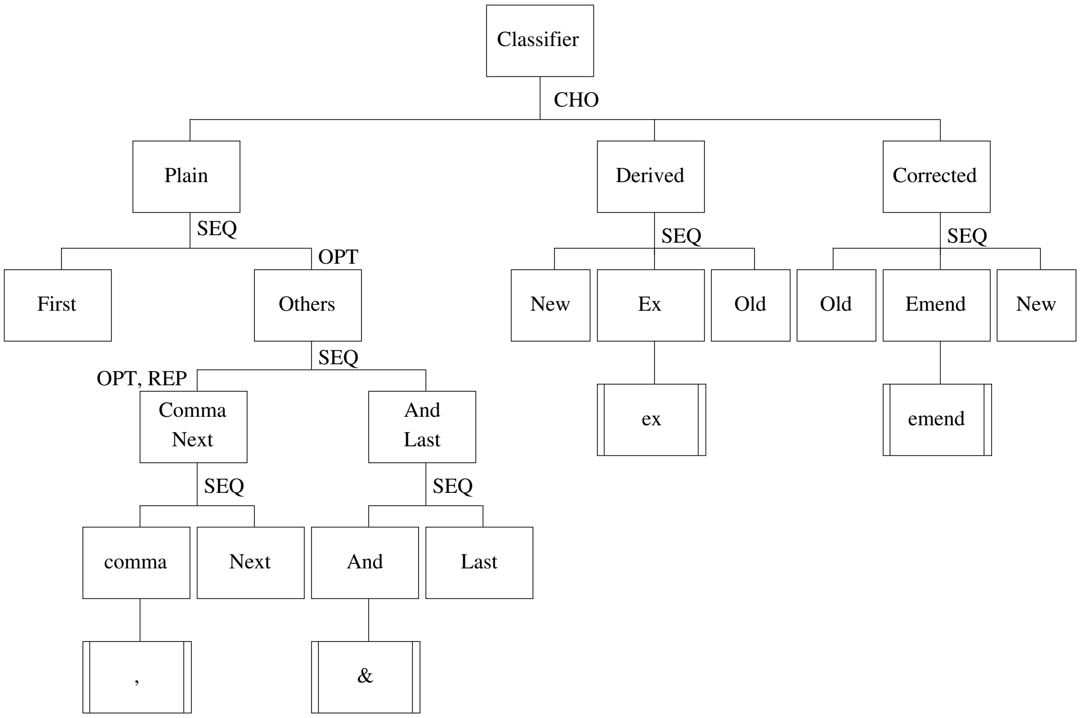

Nomenclature
SEQ
OPT, REP
Class
Order
Family
Botanical Name
Common Names
SEQ
SEQ
SEQ
REP
Class Header
Content
Order Header
Content
Common Name Header
Common Name
Class:
Order:
Common Name(s):

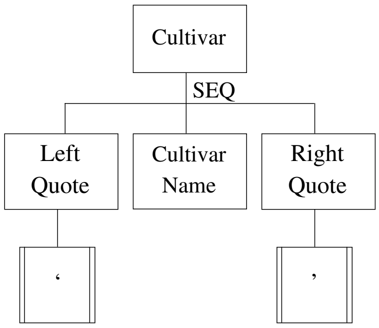

Family
SEQ
OPT
Family Header
Latin Name
In English
Family:
Left Bracket
English Name
Right Bracket
(
)

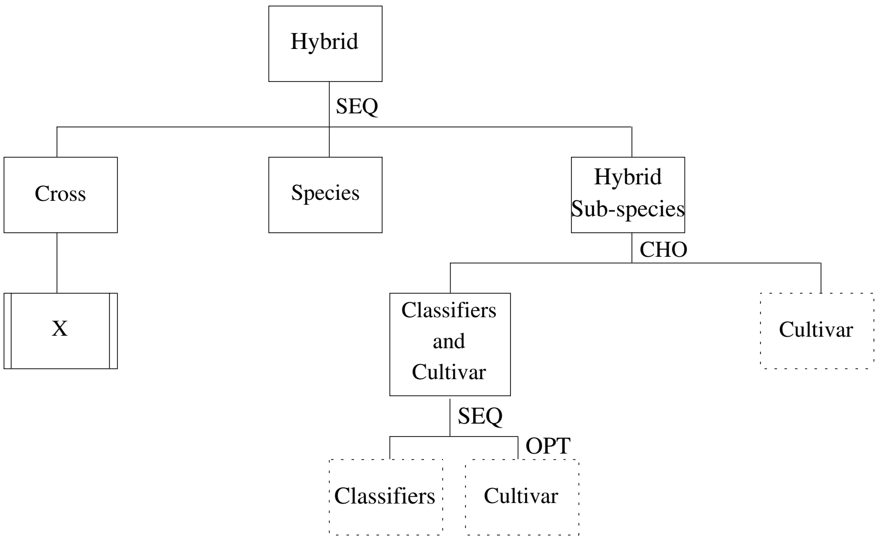

Botanical Name
SEQ
Botanical Name Header
Genus
Sub-Genus
Botanical Name:
CHO
Specific Non-hybrid
Hybrid
Cultivar

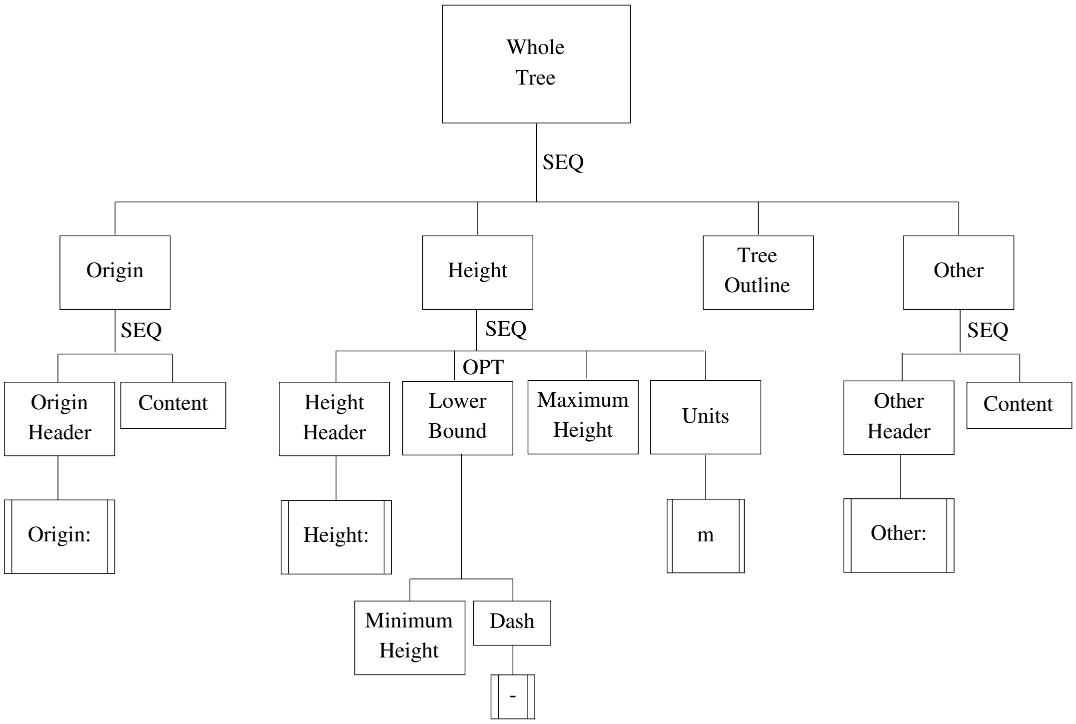

Specific Non-hybrid
SEQ
Species
Sub-species
CHO
Cultivar
Classifiers
Forma or Variety
CHO
Forma Construct
Variety Construct
SEQ
OPT
Classifiers
Forma Header
Variety
SEQ
Variety Header
Variety
forma
var.

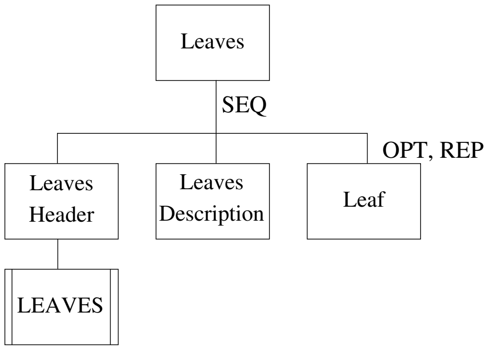

Classifiers
SEQ
OPT
Original Classifier
Classifier
SEQ
Left Bracket
Classifier
Right Bracket
(
)

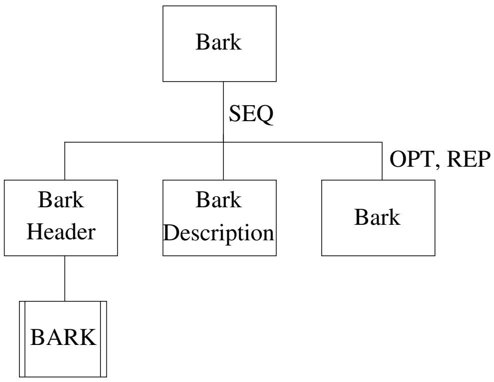

Classifier
CHO
Plain
Derived
Corrected
SEQ
OPT
First
Others
SEQ
New
Ex
Old
ex
SEQ
Old
Emend
New
emend
OPT, REP
Comma Next
And Last
SEQ
comma
Next
,
SEQ
And
Last
&

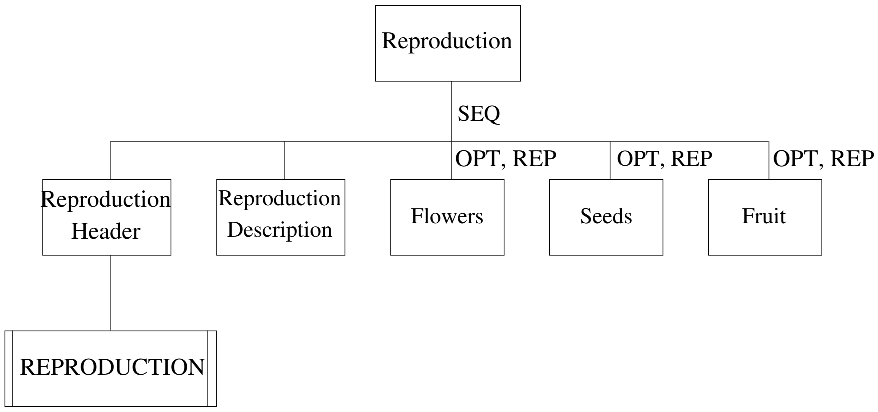

Cultivar
SEQ
Left Quote
Cultivar Name
Right Quote
‘
’


Hybrid
SEQ
Cross
Species
Hybrid Sub-species
X
CHO
Classifiers and Cultivar
Cultivar
SEQ
OPT
Classifiers
Cultivar


Whole Tree
SEQ
Origin
Height
Tree Outline
Other
SEQ
SEQ
OPT
SEQ
Origin Header
Content
Height Header
Lower Bound
Maximum Height
Units
Other Header
Content
Origin:
Height:
m
Other:
Minimum Height
Dash
-


Leaves
SEQ
OPT, REP
Leaves Header
Leaves Description
Leaf
LEAVES


Bark
SEQ
OPT, REP
Bark Header
Bark Description
Bark
BARK


Reproduction
SEQ
OPT, REP
OPT, REP
OPT, REP
Reproduction Header
Reproduction Description
Flowers
Seeds
Fruit
REPRODUCTION

Bibliography

[Aal89] I. J. Aalbersberg. A parallel full-text document retrieval system. In *WOODMAN '89: Workshop on Object-Oriented Document Manipulation*, pages 268–279, Rennes, France, May 1989. AFCET.

[AHS89] T. Andrews, C. Harris, and K. Sinkel. *The ONTOS Object Database*. Ontologic Incorporated, Massachussetts, 1989.

[And91] J. Andersson. A design proposal for a hypermedia abstract machine. In Lars Kjelldahl, editor, *Eurographics Workshop on Multimedia*, Stockholm, April 1991. Springer-Verlag (Berlin).

[App88] Apple Computer UK Limited. *Apple Interactive Multimedia — A Guide to Multimedia Development Tools*. Hertfordshire, 1988.

[BGR88] E. Bertino, S. Gibbs, and F. Rabitti. Document query processing strategies: Cost evaluation and heuristics. *ACM SIGOIS Bulletin*, 9:169–181, 1988.

[BIM89] BIM sa/nv, Belgium. *BIM_Prolog 2.4 Manual*, March 1989.

[BL85] R.J. Brachman and H.J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, California, 1985.

[Bra86] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, 1986.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

[BRT88] E. Bertino, F. Rabitti, and C. Thanos. MULTOS: A document server for distributed office systems. *Lecture Notes in Computer Science*, 303:606–615, March 1988.

[Cap87] M. Caplinger. An information system based on distributed objects. In *OOPSLA '87: Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 126–137. ACM, December 1987.

[CF80] N. S. Chang and K. S. Fu. Query by pictorial example. *IEEE Transactions on Software Engineering*, SE-6(6):519–524, November 1980.

[CG88] S. Christodoulakis and S. Graham. Browsing within time-driven multimedia documents. *ACM SIGOIS Bulletin*, 9:219–227, 1988.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[Con87] Jeff Conklin. Hypertext: An introduction and survey. *IEEE Computer*, pages 17–41, September 1987.

[CR87] P. Conti and F. Rabitti. Retrieval of multi-media document images in MULTOS. In *Fourth Esprit Conference*, pages 1389–1412, Amsterdam, 1987. North-Holland.

[Cro90] P. J. Crowther. Skeletal database model and anatomy of the pilot. Internal Multimedia Group document, September 1990.

[CT90] S. Ceri and L. Tanca. Retrieval of multimedia documents by imprecise query specification. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Ad-*

vances in Database Technology — EDBT '90, pages 203–218, Venice, March 1990. Springer-Verlag. Lecture Notes in Computer Science No. 416.

[CTH+86] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria. Multimedia document presentation, information extraction, and document formation in MINOS: A model and a system. *ACM Transactions on Office Information Systems*, 4(4):345–383, October 1986.

[CW89] M. Carlsson and J. Widen. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, Kista, October 1989.

[Dat86] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, fourth edition, 1986.

[Dra90] C. Draxler. Logic programming and databases. Technical Report 90.09, Institut fur Informatik der Universitat Zurich, September 1990.

[DS89] S. C. Dewhurst and K. T. Stark. *Programming in C++.* Prentice Hall, 1989.

[FBC+87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):49–69, January 1987.

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytic performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[Fea87] S. Feather. Office document architecture storage manager structural design. Esprit project 1024 deliverable, June 1987.

[GL86] W.I. Grosky and Y. Lu. Iconic indexing using generalized pattern matching techniques. *Computer Vision, Graphics and Image Processing*, 35:383–403, 1986.

[GOC+91] C. A. Goble, M. H. O'Docherty, P. J. Crowther, M. A. Ireton, C. N. Daskalakis, J. Oakley, S. Kay, and C. S. Xydeas. The Manchester multimedia information system. In Lars Kjelldahl, editor, *Eurographics Workshop on Multimedia*, Stockholm, April 1991. Springer-Verlag (Berlin).

[GR85] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1985.

[Gra84] P. M. D. Gray. *Logic, Algebra and Databases.* Ellis Horwood, 1984.

[HBD90] T. Horsfield, J. Bocca, and M. Dahmen. *MegaLog User Guide.* ECRC, October 1990.

[Hug91] J.G. Hughes, editor. *Object-Oriented Databases.* Prentice-Hall International, 1991.

[INJ+88] J.H. Irven, M.E. Nilson, T.H. Judd, J.F. Patterson, and Y. Shibata. Multimedia information services: A laboratory study. *IEEE Communications Magazine*, 26(6):27–44, June 1988.

[IX91] M. A. Ireton and C. S. Xydeas. Classification of shape for content retrieval of images in a multimedia information system. In *6th IEE International Conference on Digital Processing of Speech in Communications*, Loughborough, 1991.

[Kir91] Kirste, Hubner. An open hypermedia system for multimedia applications. In Lars Kjelldahl, editor, *Eurographics Workshop on Multimedia*, Stockholm, April 1991. Springer-Verlag, Berlin.

[KS86] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1986.

[Kun] T.L. Kunii. A relational database schema for describing complex pictures with color and texture.

[Lav88] S. H. Lavington. Technical overview of the intelligent file store. *Knowledge Based Systems*, 1(3):166–172, June 1988.

[LIX90] X. Li, M. A. Ireton, and C. S. Xydeas. On the detection of the extreme points of closed contours. To be published in IEE Proceedings I, Communications Speech and Vision, November 1990.

[Lut89] Ernst Lutz. Knowledge based classification of office documents. In *WOODMAN '89: Workshop on Object-Oriented Document Manipulation*, pages 353–362, Rennes, France, May 1989. AFCET.

[LWW91] S. H. Lavington, J. Wang, and M. E. Waite. The external procedural interface (EPI) for the IFS/2, part 1. Technical Report CSM-160, University of Essex, Department of Computer Science, January 1991.

[LYU91] A. Lundeburg, T. Yamamoto, and T. Usuki. SAL — a hypermedia prototype system. In Lars Kjelldahl, editor, *Eurographics Workshop on Multimedia*, Stockholm, April 1991. Springer-Verlag, Berlin.

[Mah36] P. C. Mahalanobis. On the generalised distance in statistics. *Proceedings of the Indian National Institute of Science*, 2:49–55, 1936.

[Mey86] N. Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *OOPSLA '86: Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 186–201. ACM, September 1986.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, 1988.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall International, 1992.

[Nag84] M. Nagata. Retrieval for remote sensing image database with multimedia user interface. In *IGARSS '84: Remote Sensing — From Research Towards Operational Use*, pages 77–82, August 1984.

[NS81] W.M. Neumann and R.F. Sproull. *Principles of Interactive Computer Graphics*, pages 202–209. McGraw-Hill, 2 edition, 1981.

[O'D90] M. H. O'Docherty. The generation, representation and querying of semantic data in a pilot multimedia information system. Internal Multimedia Group document, September 1990.

[OD91] M. H. O'Docherty and C. N. Daskalakis. Multimedia information systems — the management and semantic retrieval of all electronic data types. *The Computer Journal*, 34(3):225–238, June 1991.

[PCT+86] D. M. Parrish, J. L. Creasey, B. G. Thompson, D. C. Rogers, E. R. Johnston, J. R. Perry, and E. V. Staab. Functional requirements for interfacing PACS to RIS. *Proceedings SPIE — International Society of Optical Engineers*, 626:597–602, 1986.

[Pen91] D. Penna. CD-I, a multimedia system. In Lars Kjelldahl, editor, *Eurographics Workshop on Multimedia*, Stockholm, April 1991. Springer-Verlag (Berlin).

[Pri88] Isobel Pring. *Apple Multimedia Developers in Europe.* Knowledge Research Limited, London, 1988.

[Qui] Quintus Computer Systems Inc., Mountain View, Ca. Quintus prolog reference manual version 10.

[RD88] G.A. Ringland and D.A. Duce, editors. *Approaches to Knowledge Representation — An Introduction.* Research Studies Press, 1988.

[Ric83] E. Rich. *Artificial Intelligence.* McGraw-Hill, 1983.

[RS87] F. Rabitti and P. Stanchev. An approach to image retrieval from large image databases. In *Proceedings the Tenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 284–295, June 1987.

[RWTP86] D. C. Rogers, R. Wallace, B. G. Thompson, and D. M. Parrish. Information flow analysis as a tool for PACS development. *Proceedings SPIE — International Society of Optical Engineers*, 626:690–697, 1986.

[She90] T. Shetler. Birth of the blob. *BYTE*, pages 221–226, February 1990. Special Issue on Multimedia.

[Sie82] Siemens AG. *Field Trial: Integrated Office*, 1982. Unpublished report.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Menlo Park, California, 1986.

[SZ87] K.E. Smith and S.B. Zdonik. Intermedia: A case study of the differences between relational and object-oriented database systems. In *OOPSLA '87: Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 452–465. ACM, December 1987.

[Tha90] C. Thanos, editor. *Multimedia Office Filing and Retrieval: The MULTOS Approach.* North-Holland, 1990.

[WK87] D. Woelk and W. Kim. Multimedia information management in an object-oriented database system. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 319–329, September 1987.

[WKL86] D. Woelk, W. Kim, and W. Luther. An object-oriented approach to multimedia databases. In *Proceedings of ACM SIGMOD '86: International Conference on Management of Data*, pages 311–325, May 1986.

[WLK87] D. Woelk, W. Luther, and W. Kim. Multimedia applications and database requirements. In *IEEE Computer Society Office Automation Symposium*, pages 180–189, April 1987.

[Woo75] W.A. Woods. What's in a link: Foundations for semantic networks. In D.G. Bobrow and A.M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, New York, 1975.

[ZWW87] X. Zhang, J.K. Wu, and W.T. Wan. Design of a relational image database system: IMDAT. In *Proceedings of the International Workshop on Industrial Automation Systems: Seiken Symposium*, pages 138–42, February 1987.