Spring 1997

On document filing based upon predicates

Zhijian Zhu
New Jersey Institute of Technology

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ON DOCUMENT FILING BASED UPON PREDICATES

by
Zhijian Zhu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1997

APPROVAL PAGE

ON DOCUMENT FILING BASED UPON PREDICATES

Zhijian Zhu

Dr. James A. McHugh, Dissertation Advisor Date
Associate Chairperson of CIS Department
Full Professor of Computer Science, NJIT

Dr. Peter A. Ng, Dissertation Advisor Date
Chairperson of CIS Department
Full Professor of Computer Science, NJIT

Dr. Jason T.L. Wang, Dissertation Co-Advisor Date
Associate Professor of Computer Science, NJIT

Dr. Michael Bieber, Committee Member Date
Assistant Professor of Computer Science, NJIT

Dr. Qianhong Liu, Committee Member Date
Assistant Professor of Computer Science, NJIT

Dr. Ronald S. Curtis, Committee Member Date
Assistant Professor of Computer Science,
William Paterson College

ABSTRACT

ON DOCUMENT FILING BASED UPON PREDICATES

by
Zhijian Zhu

This dissertation presents a formal approach to modeling documents in a personal office environment, proposes a heterogeneous algebraic query language to manipulating objects (folders) in the document model, and investigates a predicate-driven document filing system for automatically filing documents.

The document model was initially proposed in [38] which adopts a very natural view for describing the office documents using the relational and object-oriented paradigms. The model employs a dual approach to classifying and categorizing office documents by defining both a *document type hierarchy* and a *folder organization.* This dissertation extends and specifies formally the document model. Documents are partitioned into different classes, each document class being represented by *frame template* which describes the properties of the documents of the class. A particular office document, summarized from the view point of its frame template, yields a synopsis of the document which is called *frame instances.* Frame instances are grouped into a folder on the basis of user-defined criteria, specified as predicates, which determine whether a frame instance belongs to a folder. Folders, each of which is a heterogeneous set of frame instances, can be naturally organized into a folder organization. The folder organization specifying the document filing view is then defined using predicates and a directed graph. However, some operators in the algebraic query language [38] do not support the heterogeneous property. This dissertation proposes an algebra-based query language that gives full support to this heterogeneous property.

We investigate the construction problem of a folder organization: does it allow a user to add a new folder with an arbitrary local predicate? Given a folder organization, creating a new folder with arbitrarily defined predicate may cause two abnormalities: *inapplicable edges* (filing paths) and *redundant folders*. To deal such abnormalities in the process of constructing a folder organization, the concept of predicate consistency is discussed and an algorithm is proposed for determining whether the predicate of a new folder is consistent with the existing folder organization.

The global predicate of a folder governs the content of the folder. However, the predicates of folders (that is, global predicates) do not uniquely specify a folder organization. Then, we investigate the reconstruction problem: under what circumstance can we uniquely recover the folder organization from its global predicates? The problem is solved in terms of graph-theoretic concepts such as associated digraphs, transitive closure, and redundant/non-redundant filing paths. A transitive closure inversion algorithm is then presented which efficiently recovers a folder organization digraph from its associated digraph.

After defining a folder organization, we can file a frame instance into the folder organization. A document filing algorithm describes the procedure of filing a frame instance. However, the critical issue of the algorithm is how to evaluate whether a frame instance satisfies the predicate of a folder in a folder organization. In order to solve this issue, a thesaurus, an association dictionary and a knowledge base are then introduced. The thesaurus specifies the association relationship among the key terms that are actually residing in the system and terms that are used by users. An association dictionary gives the association relationship between an attribute of a predicate and a frame template defined in a folder organization. A knowledge base represents background knowledge in a certain application domain.

BIOGRAPHICAL SKETCH

Author:	Zhijian Zhu
Degree:	Doctor of Philosophy
Date:	May 1997

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, New Jersey, 1997
- Master of Engineering in Mechanical Engineering,
  Hefei Polytechnic University, Hefei, Anhui, China, 1987
- Bachelor of Science in Computer Science,
  Nanjing University of Aeronautics and Astronautics,
  Nanjing, Jiangsu, China, 1984

Major: Computer Science

Publications:

Z. Zhu, J. McHugh, J. Wang, and P. Ng, "A Formal Approach to Modeling Office Information Systems", *Journal of Systems Integration*, Vol. 4, No. 4, Pages: 373–403, December, 1994.

F. Mhlanga, Z. Zhu, J. Wang, and P. Ng, "A New Approach to Modeling Personal Office Documents", *Data and Knowledge Engineering*, Vol. 17, No.2, Pages: 127–158, November, 1995.

Z. Zhu, Q. Liu, J. McHugh, and P. Ng, "A Predicate-Driven Document Filing System", *Journal of Systems Integration*, Vol. 6, No. 3, September, 1996.

This dissertation is dedicated to
my wife
Shanmaio Ma
my son
Alec M. Zhu

ACKNOWLEDGMENT

The author would like to take great pleasure in acknowledging his advisor, Professor Peter A. Ng, for his kindly assistance and remarkable contribution to this dissertation. He spent time and effort to review various drafts of the manuscripts and provided a lot of helpful comments and crucial feedbacks that influenced the final manuscript. His guidance and moral support throughout this research are much appreciated. The author is indebted to his other advisor, Professor James A. McHugh, who devoted effort and provided encouragement in the phase of formalizing the document model. His ability to have a solid mathematics professionals significantly contributed to the contents and organization of this dissertation. The author also thanks his co-advisor, Professor Jason T.L. Wang who provided support, encouragement and constructive criticism on this research.

Specially, the author wants to thank to Doctor Michael Bieber, Doctor Qianhong Liu and Doctor Ronald S. Curtis serving as members of the committee.

This dissertation was supported in part by the Separately Budgeted research grant from New Jersey Institute of Technology and by System Integration Program grant from the AT&T Foundation.

The author wishes to thank all the moral support given to him by colleagues, friends, fellow Ph.D. students and all the members of the TEXPROS research group.

The writing of the dissertation was facilitated by the computing resources and equipments in the Department of Computer and Information Science at New Jersey Institute of Technology.

Finally, the author gratefully acknowledges his debt to the authors of the works that are cited in this dissertation and claims full responsibility for any bugs that the text may contain.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1
INTRODUCTION

In an office environment, information is a resource that is needed to perform office workers' jobs. We use information to make decisions and enhance productivity. Generally, information is exchanged in the form of documents [11, 16]. For document management and retrieval, there is a lack of information technology (in particular, customized to individuals in an office environment) for representing and organizing massive information in the multimedia (such as paper and electronic) environment, for storing information pertaining significantly to the individuals into information repositories, and for easily processing and retrieving information when needed (and thus, the corresponding documents could be referred directly from repositories). There also is a lack of information access technology that allows an efficient search of large distributed information repositories [32].

1.1 TEXPROS

TEXPROS (TEXt PROcessing System) [32, 52] is a personalized, customized office information processing system for processing and retrieving office documents. Basically, it has the following major features:

- Modeling the behaviors of common office activities using the state-of-the-art document model [32, 38, 39, 40, 51, 57, 59].

- Classifying documents into types based on their structures [19, 20, 21, 53, 54, 55]. Each document type is defined in terms of attributes to form a *frame template.*

- Extracting the most significant information from an original document to form a *frame instance* [19, 20, 21, 53], with respect to the frame template of the original document. The frame instance is a *synopsis* of the original document.

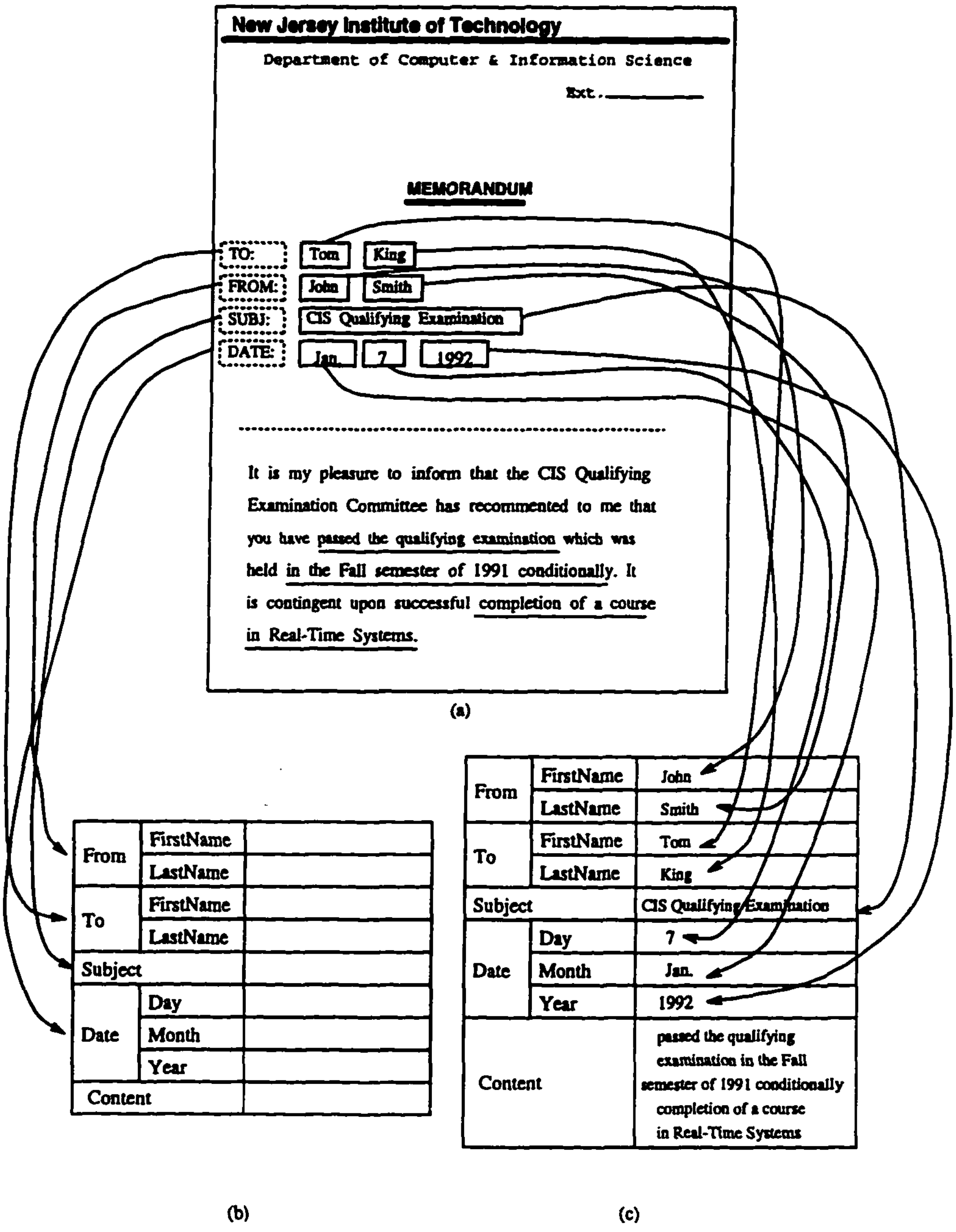
- Filing frame instances into *folders* using a predicate-driven approach [57, 58, 59]. That is, a frame instance is filed in a folder if it satisfies the predicate of the folder.

- Retrieving information from the folder organization [30, 31, 32, 33, 34]. Users retrieve documents or information contained in documents on the basis of the information in their frame instances[1].

In TEXPROS document Model, a document type (*frame template*) is formed by sampling a stream of office documents, abstracting their general attributes, and grouping them into a class. The frame template, filled in by the instances of a particular office document, yields an organized *synopsis* of the original document which we call a *frame instance.* Figure 1.1(a) is an original document (a memorandum). Figure 1.1(b) shows the frame template **Memo** which describes the attributes (or properties) for the class **Memo**. Each memorandum in this class has attributes From (or Sender), To (or Receiver), Subject, Date, Content, etc. The attribute Content represents the non-structured part of the frame template **Memo**. The rest of the attributes represent the structured part of **Memo**. The frame template is instantiated to form a frame instance by assigning values to the attributes of the frame template. Figure 1.1(c) shows the corresponding frame instance for an original memorandum (Figure 1.1(a)) of the type, which is specified by the frame template **Memo** (Figure 1.1(b)).

Frame instances of documents can be grouped into *folders* based on how users organize their information. The *folder organization* represents the user's desired

[1] We keep the original documents in the storage separately from the frame instances. Users can retrieve them as needed. It improves the system performance and reduces cost.



New Jersey Institute of Technology

Department of Computer & Information Science

Ext.________

MEMORANDUM

TO: Tom King

FROM: John Smith

SUBJ: CIS Qualifying Examination

DATE: Jan. 7 1992

It is my pleasure to inform that the CIS Qualifying Examination Committee has recommented to me that you have passed the qualifying examination which was held in the Fall semester of 1991 conditionally. It is contingent upon successful completion of a course in Real-Time Systems.

(a)

From	FirstName	
	LastName	
To	FirstName	
	LastName	
Subject		
Date	Day	
	Month	
	Year	
Content		

(b)

From	FirstName	John
	LastName	Smith
To	FirstName	Tom
	LastName	King
Subject		CIS Qualifying Examination
Date	Day	7
	Month	Jan.
	Year	1992
Content		passed the qualifying examination in the Fall semester of 1991 conditionally completion of a course in Real-Time Systems

(c)

Figure 1.1 (a) An original document (b) Its frame template (c) Its frame instance document filing organization. The *document type hierarchy* provides a means of identifying and organizing structural commonalities among documents, in terms of frame templates, and thus a means of classifying various documents. The overall architecture of TEXPROS is sketched in Figure 1.2. There are four persistent storages: (1) *Document Sample Base* contains sample documents for document classification; (2) *Frame Instance Base* stores frame instances in the folder organization; (3) *Model Base* has definitions of frame templates, folders[2], document type hierarchy and folder organization; and (4) *Knowledge Base* consists of system rule base, fact base, system catalog and association dictionary.

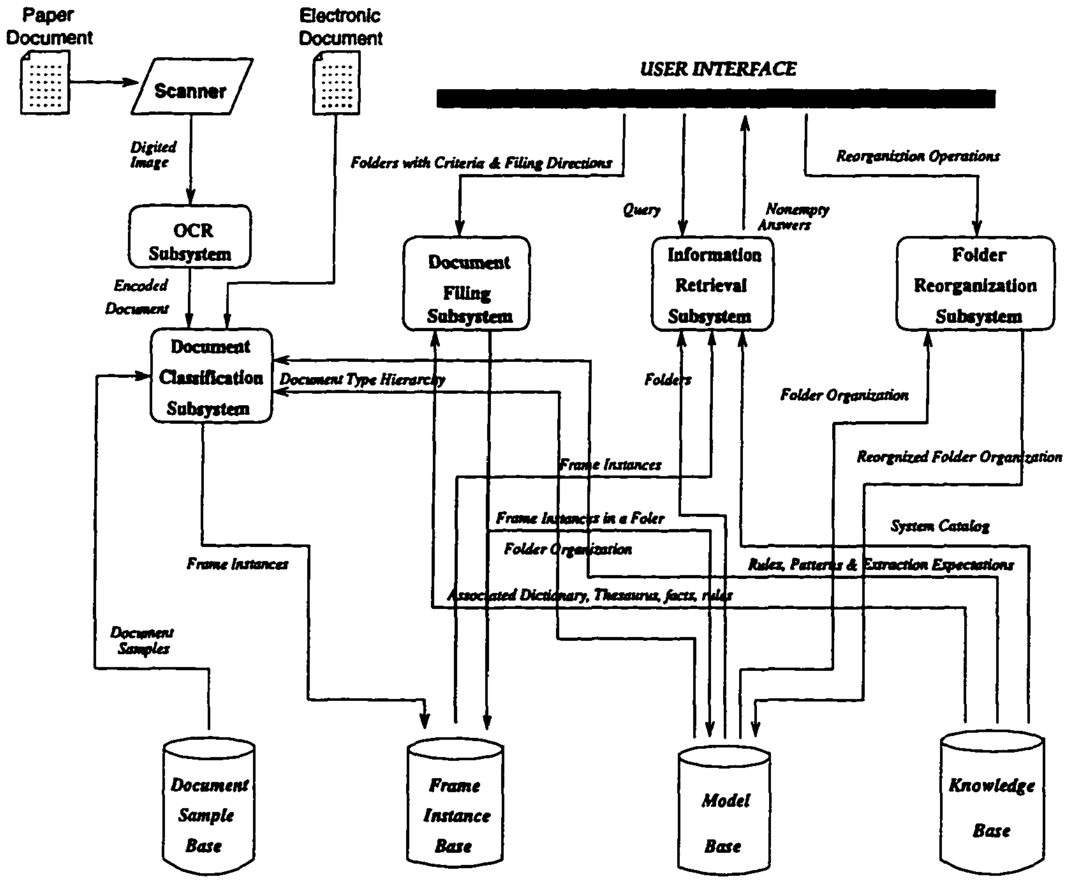


Figure 1.2 Overall architecture of TEXPROS

[2] Note that each frame instance in the frame instance base has a unique identifier associated with it. A folder contains a set of frame instance identifiers which satisfies the criteria of the folder.

- Optical Character Recognition (OCR) Subsystem [7, 6, 19, 47, 53]: A paper document is digitized and thresholded into a binary image by a scanner. In order to encode information from a mixed-mode document which contains text, graphics and pictures, the document image is segmented into textual blocks, graphic blocks and picture blocks. Each block can be further divided into smaller blocks, and all the blocks are encoded.

- Document Classification Subsystem [19, 20, 21, 53, 54]: An encoded document is automatically identified as a document class (frame template) by the sample-base approach. The document type hierarchy is constructed using the concept of specialization and generalization of frame templates. Furthermore, the synopsis of the document is extracted to form a frame instance based upon the structure of the document (i.e. its frame template) and the significant information pertinent to users.

- Document Filing Subsystem [38, 40, 50, 58, 59]: A set of frame instances can be grouped into a folder on the basis of user-defined criteria. TEXPROS allows a user to define a folder organization that mimics his/her filing system. The folder organization is made up of folders which are linked via filing directions. An incoming frame instance can be filed into an appropriate folder if it satisfies the criteria of the folder.

- Information Retrieval Subsystem [30, 31, 33, 34]: This information retrieval subsystem is capable of processing incomplete, imprecise or vague queries and providing meaningful responses to a user. It provides a more flexible and cooperative capability for interpreting and processing queries.

- Folder Reorganization Subsystem [50]: The folder reorganization subsystem provides a set of operations for reorganizing folder organizations, and changing the structure of the organization.

1.2 Scope of the Dissertation

This dissertation mainly focuses on the office information modeling, and the document filing. The scope of this dissertation covers the following aspects:

- Giving an in-depth study on the TEXPROS document model.

  The document model for TEXPROS was proposed in [38, 40, 50, 52, 59]. The model employs a dual approach to classifying and categorizing the office documents by defining both a *document type hierarchy* and a *folder organization* (or *logical filing structure*). The document type hierarchy depicts the structural organization of the document types used in the problem domain. It identifies and organizes the structural commonalities among documents, and facilitates classifying various documents. The folder organization represents the user's view of the document filing organization. A folder is a heterogeneous set of frame instances; that is, a folder contains frame instances over different frame templates. This dissertation gives a formal specification of the TEXPROS document model[3]. A *frame template* (document type) specifies the structure common to different documents or *frame instances* (document instances) of the same kind. The folder organization is defined using predicates and a rooted DAG for specifying the document filing view.

- Proposing an algebraic query language for heterogeneous environment.

  The algebra-based query language in TEXPROS document model, called $\mathcal{D}$_algebra, was proposed in [38, 39]. We observe that some operators in the $\mathcal{D}$_algebra do not support heterogeneous property of the TEXPROS document model. For example, consider the project operator (π). Assume that there are two frame instances, fi_1 = [$\langle$`Title` : A Office Model$\rangle$, $\langle$`Author` : John Smith$\rangle$]

[3] The TEXPROS document model successfully couples a precise mathematical definition with a rooted DAG representation and nested forms.

and fi_2 = [⟨Name : John Smith⟩, ⟨Position : Professor⟩, ⟨DegreeObtained : PhD⟩], in the John_Smith folder. Since the project operator in [38, 39] only allows the projected attributes coming from the same frame template, the query $\pi_{\texttt{Title,Author,DegreeObtained}}$(John_Smith) = [ ]. This dissertation extends $\mathcal{D}$_algebra operators to fit heterogeneous environment [40]. Furthermore, $\mathcal{D}$_algebra only deals with restructuring (*nest* and *unnest*) operators for a single attribute. In the proposed dissertation, two sets of restructuring operators are defined. One is nest (ν) and unnest (μ) operators for a single attribute as in [39]. The other one is nest (ν^*) and unnest (μ^*) operators for multi-attributes. The reason of introducing these two sets of restructuring operators is that ν and μ are not the special case of ν^* and μ^* in TEXPROS document model, respectively.

- Studying the construction and reconstruction problems of a folder organization.

  When a user adds a new folder with arbitrarily defined predicate to a folder organization, it may cause two abnormalities: inapplicable edges (filing paths) and redundant folders. This is called the *Construction Problem.* To resolve this problem, the concept of predicate consistency is discussed and an algorithm is proposed to prevent such abnormalities. The *global predicate* [59] of a folder governs the content of the folder. However, the folder level predicates (that is, global predicate) do not uniquely specify a folder organization. From this arises the *Reconstruction Problem*, namely, under what circumstance it is possible to recover a unique folder organization from its global predicates. The graph-theoretic concepts including associated digraphs, transitive closure, and redundant/non-redundant filing paths are used to investigate the *Reconstruction Problem* and show how a folder organization digraph can be efficiently recovered from its associated digraph.

- Investigating the predicate-driven document filing.

  Document filing is one of the most important components in TEXPROS. Given a folder organization, in which the folders are specified using predicates, how the frame instances all deposited in proper folders is based on the predicates. A filing algorithm is proposed for filing a frame instance in the proper folders. However, the critical issue of the algorithm is how to evaluate whether a frame instance satisfies the predicate of a folder in the folder organization. In order to solve this issue, a thesaurus, an association dictionary and a knowledge base are introduced. The thesaurus associates the key terms that are actually residing in the system and terms that are used by the users. An association dictionary states the association relationship between an attribute of a predicate and a frame template defined in the document type hierarchy. A knowledge base represents background knowledge in a certain application domain.

1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 presents the survey of related work on office document modeling, algebraic query language and document filing. Chapter 3 introduces the preliminary concepts for defining the TEXPROS document model. Chapter 4 formally defines the TEXPROS document model, including frame instances, frame templates, a document type hierarchy, folders, and a folder organization. Chapter 5 extends the existing $\mathcal{D}$_algebra and its properties. Chapter 6 discusses a pair of problems for a folder organization: the construction problem and the reconstruction problem. Chapter 7 investigates the predicate-driven filing problem, namely, given a folder organization, in which the folders are defined using predicates, how do the frame instances deposit in proper folders based on these predicates? Chapter 8 summarizes the dissertation and gives future research directions.

CHAPTER 2

RELATED WORK

In this chapter, an overview of the subjects related to document modeling, algebraic query language and document filing is given.

2.1 Modeling Office Documents

Office documents are one of the basic vehicles for making decisions and taking actions in office work [22]. Office documents exhibit a very broad spectrum of structure, from standardized forms to free text. Basically, three types of structures can be distinguished within a document: the layout structure, the logical structure, and the conceptual structure. The first two structures are referred to as the standard structures of documents in the Office Document Architecture (ODA) [11, 24, 26].

The layout structure is a standard for editing and formatting documents. The logical structure describes the logical components of a document (such as title, section, and paragraphs), and how they are related. The conceptual structure represents the semantic aspects for the document contents. For example, the author or the summary of a technical paper, and the sender of a memorandum are referred to as conceptual components. The aggregation of conceptual components is the document conceptual structure, and documents with analogous conceptual structures are grouped in types [5]. The conceptual level of office documents has been considered widely in the last decade [23, 35, 41, 43, 56, 60].

Modeling is often based on concepts used for semantic data modeling, such as aggregation, association, and specialization [42]. Sometimes, the conceptual structure is blended with ODA layout and physical structures, as far as query formulation is concerned. For example, MULTOS [2, 49] is oriented to multimedia document management. The conceptual components in a document are stored in a database. Documents of the same structure with similar contents are grouped to form a class.

The Kabiria document model [5, 44] is oriented to the classification and retrieval of office documents according to their internal structure and operational meaning. It includes a conceptual document model and a document retrieval model. The conceptual document model concerns the semantic and logical description of documents. The document retrieval model enriches the conceptual model with the explicit description of both the roles of documents in the office and their dependencies from the laws, regulations and habits of the application domain.

The TEXPROS document model was initially proposed in [38]. It adopts a very natural view for describing the office documents. Documents are grouped into classes. Each class is characterized by a frame template, which describes the type for the class of documents. A frame template is instantiated by providing it with values to form a frame instance, representing a synopsis of a particular document associated with the template. Different frame instances can be grouped into a folder based on user defined criteria. The document model describes documents using dual hierarchies: a document type hierarchy (depicting the structural organization of the documents), and a folder organization (representing the user's logical file structure).

2.2 Algebraic Query Language

Mhlanga et al. [38, 39, 51] proposed an algebraic query language (called $\mathcal{D}$_algebra) for manipulating objects in the TEXPROS document model. There are three groups of work that are closely related to the $\mathcal{D}$_algebraic language. The algebra developed by Guting et al. [18] also deals with documents. Following closely the ODA standard, documents are described in terms of schemas, instances and layouts. A schema is represented by an ordered labeled tree, which describes the logical structure and data values contained in a class of documents. In contrast to Guting's algebra, the

$\mathcal{D}$_algebra does not assume any particular (logical or layout) order among attributes of a frame instance. The second group, led by Zdonik of Brown University, developed the algebra for the ENCORE object-oriented data model [46]. While both of the $\mathcal{D}$_model and ENCORE make use of attribute types and object type hierarchy, the former doesn't support object identity and abstract data types for encapsulation of behavior and state. Furthermore, the operators in the $\mathcal{D}$_algebra mainly manipulate heterogeneous objects (i.e., folders) that contain frame instances of different types. This is in contrast to the operators in ENCORE's algebra, whose operands must be collections of objects of the same type. Su et al. [48] proposed an association algebra (called A-algebra) using the pattern-based query formulation for object-oriented databases. The operators of the A-algebra can be used to navigate a network of interconnected object classes along the path of interest to construct a complex pattern as the search condition. In contrast, the highlight operator is introduced in the $\mathcal{D}$_algebra simplifying such navigation. The heterogeneous property of the operators in [48] is totally different from this dissertation in the sense that classes defined in [48] are homogeneous and folders are heterogeneous. In other words, a binary operator is said to be heterogeneous [48] if its two operands are from two different classes, where the objects in each class have the same property (the same set of attributes). However, the objects (i.e. frame instances) in an operand (i.e. a folder) can be over different types (i.e. frame templates) in the $\mathcal{D}$_algebra.

2.3 Document Filing System

A document filing system provides facilities for storing and efficiently retrieving documents. In the Kabiria [5, 44], the general task of the filing system is the acquisition and classification of documents. The filing process is carried out by three modules: the Acquisition module (ACQ), the Classifier module (CLASS), and the Insertion module (INS). ACQ enables the users to define class structures and to insert new document instances into the system in order to file them. In fact, the system can file and then manipulate a document only if it recognizes its conceptual structure. Therefore, as a new document type appears in the office, the system must be provided with its description, comprising both the conceptual structure and the links connecting the document types within the semantic network. The purpose of CLASS is to identify the class a document instance belongs to. INS files classes and instances in both the model base and the document base.

MULTOS [2] divides document filing systems into three categories in terms of retrieval requirements and hardware capabilities: (1) Dynamic document filing systems are used essentially as buffers allowing local storage of documents being manipulated. Generally, a dynamic document filing system is accessed by a single user. (2) Current document filing systems are used for documents that are frequently accessed and so of current interest to the office. (3) Archive document filing systems are used for less frequently accessed documents that have reached a stable state where modification is infrequent. From hardware capacity point of view, archival systems have the greatest capacity, followed by current document filing systems, and finally systems for dynamic document filing. The three filing system categories are also related to the document life cycle. Typically, one would expect a migration from a dynamic filing system toward an archival system.

CHAPTER 3

PRELIMINARIES

The TEXPROS document model uses the concepts of *type*, *instance*, *domain*, and *predicate* to specify information representations.

3.1 Types, Instances and Domains

The *primitive types* are **integer, real, string, text,** and **boolean.** An *enumeration type* is an ordered tuple of finite strings from an alphabet, that is, a finite set of symbols. The primitive and enumeration types are called *basic types.* An *attribute name* (or *attribute*) is a finite string of symbols. An attribute has a corresponding *type.*

Definition 3.1.1 *(Type)* Types are defined recursively as follows:

1. A basic type is a type.

2. Let $\mathtt{A}_i$ be an attribute with its corresponding type T_i, $1 \leq i \leq m$. $T = [\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_m : T_m \rangle]$ is a type, called a *tuple type.* T_1, ..., and T_m are called the *underlying types* of T.

3. $T = \{T_1, ..., T_n\}$ is a type, called a *set type.* T_i, $1 \leq i \leq n$, is an underlying type of T. □

Definition 3.1.2 *(Instance)* Instances are defined recursively as follows:

1. An instance of a basic type is called a *basic instance.*

2. If $\mathtt{A}_1$, ..., and $\mathtt{A}_m$, $m \geq 1$, are distinct attributes of types T_1, ..., T_m and I_1, ..., and I_m are instances of T_1, ..., and T_m, then $I = [\langle \mathtt{A}_1 : I_1 \rangle, ..., \langle \mathtt{A}_m : I_m \rangle]$ is an instance, called a *tuple instance*, of the type $[\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_m : T_m \rangle]$.

3. For $T = \{T_1, ..., T_n\}$, let I_i be an instance of an underlying type T_i. Then, a *set instance* I of the type T is a set of instances of the types T_i. □

Definition 3.1.3 *(Equality of Instances)* Equality between two instances is recursively defined as follows:

1. Two basic instances are equal if and only if they are the same.

2. Let $I_i = [\langle \mathtt{A}_{i_1} : I_{i_1} \rangle, ..., \langle \mathtt{A}_{i_n} : I_{i_n} \rangle]$, and $I_j = [\langle \mathtt{A}_{j_1} : I_{j_1} \rangle, ..., \langle \mathtt{A}_{j_n} : I_{j_n} \rangle]$ be two tuple instances. I_i and I_j are equal if and only if their attribute-instance pairs, $\langle \mathtt{A}_{i_k} : I_{i_k} \rangle$ and $\langle \mathtt{A}_{j_k} : I_{j_k} \rangle$ are equal (i.e. $\mathtt{A}_{i_k} = \mathtt{A}_{j_k}$ and $I_{i_k} = I_{j_k}$) for every k.

3. Two set instances are equal if and only if they have the same instances. □

A tuple type $T = [\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_m : T_m \rangle]$ is called an *aggregation hierarchy* [25] if an underlying type T_i is a non-basic type. We can use a *path-notation*, an attribute followed by a sequence of zero or more attributes, to refer to an instance of a particular component of an aggregation hierarchy. Let $\mathtt{A}$, $\mathtt{B}_1$, ..., $\mathtt{B}_n$ be attributes. The instance referred to by the path notation $\mathtt{A}.\mathtt{B}_1.\cdots.\mathtt{B}_n$ is defined as follows:

1. If $n = 0$, then the instance of the path notation is the instance of $\mathtt{A}$.

2. If $n > 0$, then the instance of the path notation is the instance of attribute $\mathtt{B}_n$ within the instance of $\mathtt{A}.\mathtt{B}_1.\cdots.\mathtt{B}_{n-1}$ if $\mathtt{A}.\mathtt{B}_1.\cdots.\mathtt{B}_{n-1}$ is defined. The path notation $\mathtt{A}.\mathtt{B}_1.\cdots.\mathtt{B}_{n-1}$ is *defined* if there is no set type within $\mathtt{A}.\mathtt{B}_1.\cdots.\mathtt{B}_{n-2}$, and is *undefined* otherwise.

For example, in order to refer once the instance for the attribute year of the frame instance in Figure 1.1(c), the path notation is Date.Year, assuming Date is not a set type.

The set of all possible instances of a type T is called the *domain* of T. For example, the domain of **integer** is the set of integers. We define DOM to be a function mapping a type T to a domain of T as follows:

- If T is a basic type, then $DOM(T)$ is the domain of T.

- If $T = [\langle \mathtt{A}_1 : T_1\rangle, ..., \langle \mathtt{A}_m : T_m\rangle]$, then $DOM(T) = \{[\langle \mathtt{A}_1 : I_1\rangle, ..., \langle \mathtt{A}_m : I_m\rangle] \mid (I_1 \in DOM(T_1)) \wedge ... \wedge (I_m \in DOM(T_m))\}$.

- If $T = \{T_1, ..., T_n\}$, then $DOM(T) = \{\bigcup_{i=1}^{k} I_i \mid (I_i \subseteq DOM(T_1)) \vee ... \vee (I_i \subseteq DOM(T_n))\}$.

Let $T = [\langle \mathtt{A}_1 : T_1\rangle, ..., \langle \mathtt{A}_m : T_m\rangle]$ be a tuple type. Since a tuple instance consists of attribute-instance pairs, $DOM(T) \neq DOM(T_1) \times ... \times DOM(T_m)$. This can be shown by the following example. Consider two tuple types:

- **Employee** = [⟨Name : **string**⟩, ⟨Age : **integer**⟩, ⟨Salary : **real**⟩]

- **Order** = [⟨ProductName : **string**⟩, ⟨Quantity : **integer**⟩, ⟨UnitPrice : **real**⟩]

Employee and **Order** are different tuple types. The domain of a tuple type is the set of all possible attribute-instance pairs. This is not the same as the Cartesian product of the domains of the underlying types (such as, here, **string** × **integer** × **real**).

Let $\tilde{T_1} = [\langle \mathtt{A}_1 : T_1\rangle]$, ..., and $\tilde{T_m} = [\langle \mathtt{A}_m : T_2\rangle]$. The usual Cartesian view of the domain of T is $DOM(\tilde{T_1}) \times ... \times DOM(\tilde{T_m})$, which is too restricted, as shown in the following example. Define the two tuple types:

- **Student** = [⟨ Name: **string** ⟩, ⟨ Major: **string** ⟩,
  ⟨ SBirthday: [⟨ Date: **date** ⟩, ⟨ Month: **month** ⟩, ⟨ Year: **integer** ⟩]⟩]

- **Faculty** = [⟨ Name: **string** ⟩, ⟨ Department: **string** ⟩,
  ⟨ FBirthday: [⟨ Date: **date** ⟩, ⟨ Month: **month** ⟩, ⟨ Year: **integer** ⟩]⟩]

Consider the query: "*Find all the students and faculty who have the same birthday*". Since the type [⟨ SBirthday: [⟨ Date: **date** ⟩, ⟨ Month: **month** ⟩, ⟨ Year: **integer**

$\rangle]\rangle]$ and [$\langle$ FBirthday: [$\langle$ Date: **date** $\rangle$, $\langle$ Month: **month** $\rangle$, $\langle$ Year: **integer** $\rangle]\rangle]$ are different, the instances from these two types cannot be compared to each other. Thus this query cannot be answered using the standard Cartesian product approach. However, our approach can handle this query since the underlying types of both SBirthday and FBirthday are the same.

3.2 Operations and Predicates

The *intersection* and *union* operations between tuple types (instances) are defined as follows. Later on we will use these operations to define an *IS-A* relationship between frame templates, and algebra operations. Let $X = [\langle \mathtt{A}_1 : X_1\rangle, ..., \langle \mathtt{A}_n : X_n\rangle]$, where $\mathtt{A}_i$ $(1 \leq i \leq n)$ is an attribute. If X_i $(1 \leq i \leq n)$ is a type, then X is a tuple type. If X_i $(1 \leq i \leq n)$ is an instance, then X is a tuple instance. We introduce a predicate *is-a-component-of* (denoted by *is-a-comp*) for tuple types and instances. defined as follows:

$$\textit{is-a-comp}(\langle \mathtt{B} : Y\rangle, X) = \begin{cases} \textit{true} & \text{if } \exists \langle \mathtt{A}_i : X_i\rangle \text{ in } X \\ & \quad \text{such that } (\mathtt{B} = \mathtt{A}_i) \wedge (Y = X_i) \\ \textit{false} & \text{otherwise} \end{cases}$$

where B is an attribute and Y is a type (or instance). That is, *is-a-comp*$(\langle \mathtt{B} : Y\rangle, X)$ is true iff X has a component with the same attribute and type (or instance) as $\langle \mathtt{B} : Y\rangle$.

Definition 3.2.1 *(Intersection of Two Tuple Types (Instances))* Let X and $\tilde{X}$ be two tuple types (instances). The intersection of two tuple types (instances), denoted by $X \cap^a \tilde{X}$, consists of all the attribute-type (attribute-instance) pairs which are common components of both X and $\tilde{X}$. That is,

$$X \cap^a \tilde{X} = [\langle \mathtt{B}_i : X_i\rangle \mid (\textit{is-a-comp}(\langle \mathtt{B}_i : X_i\rangle, X) \wedge \textit{is-a-comp}(\langle \mathtt{B}_i : X_i\rangle, \tilde{X}))]$$

where $\mathtt{B}_i$ is an attribute, and X_i is a type (instance). □

Definition 3.2.2 *(Union of Two Tuple Types (Instances))* Let X and $\tilde{X}$ be two tuple types (instances). The union of two tuple types (instances), denoted by $X \cup^{\alpha} \tilde{X}$, consists of all the attribute-type (attribute-instance) pairs which are from either X or $\tilde{X}$. That is,

$$X \cup^{\alpha} \tilde{X} = [\langle \mathtt{B}_i : X_i \rangle \mid (\textit{is-a-comp}(\langle \mathtt{B}_i : X_i \rangle, X) \vee \textit{is-a-comp}(\langle \mathtt{B}_i : X_i \rangle, \tilde{X}))]$$

where $\mathtt{B}_i$ is an attribute, and X_i is a type (instance). □

The operators "$\cap^{\alpha}$" and "$\cup^{\alpha}$" are associative and commutative.

Since the emphasis of the proposed dissertation is on tuple instances, it will be convenient to introduce the following notation. Let $\hat{I}$ be a tuple instance and let A be an attribute or path notation. If the tuple type of $\hat{I}$ includes A as an attribute or a path notation, then $\hat{I}$[A] denotes the instance of A. If A is not in $\hat{I}$, then $\hat{I}$[A] is an empty instance [ ]. For example, consider the following tuple instance,

$$\begin{aligned} \hat{I} = [&\langle \texttt{Name: } [\langle \texttt{FName: } \textit{John} \rangle, \langle \texttt{LName: } \textit{Smith} \rangle] \rangle, \\ &\quad \langle \texttt{QEAppl: } [\langle \texttt{SemesterTaken: } [\langle \texttt{Semester: } \textit{Fall} \rangle, \langle \texttt{Year: } \textit{1991} \rangle] \rangle, \\ &\qquad\qquad \langle \texttt{1stChoice: } \textit{Software Engineering} \rangle, \\ &\qquad\qquad \langle \texttt{2ndChoice: } \textit{Compiler} \rangle] \rangle]. \end{aligned}$$

Then, for the attribute Name, $\hat{I}$[Name] = [⟨ FName: *John* ⟩, ⟨ LName: *Smith* ⟩]. Similarly, for the path notation QEAppl.SemesterTaken.Semester, $\hat{I}$[QEAppl.SemesterTaken.Semester] = *Fall.*

We define predicates as follows. In the case where $\hat{I}$ is a tuple instance and I is an instance, the *atomic predicates* have the following interpretations:

- Equality Predicate: If $\hat{I}$[A] and I are over the same type, then the equality predicate is $\hat{I}[\mathtt{A}] = I$.
- Comparison Predicates: If $\hat{I}$[A] and I are over ordered types, then $\hat{I}[\mathtt{A}] > I$, $\hat{I}[\mathtt{A}] \geq I$, $\hat{I}[\mathtt{A}] < I$ and $\hat{I}[\mathtt{A}] \leq I$ are the comparison predicates.

- Component Predicate: If $\mathtt{A}$ is an attribute, then $\textit{is-a-comp}(\langle \mathtt{A} : I\rangle, \hat{I})$ is the component predicate. Note that a component predicate can be represented by an equality predicate. That is, $\textit{is-a-comp}(\langle \mathtt{A} : I\rangle, \hat{I})$ is identical to $\hat{I}[\mathtt{A}] = I$.

- Membership Predicates: If $\hat{I}[\mathtt{A}]$ is of type T and I is of type $\{T\}$, then $\hat{I}[\mathtt{A}] \in I$ is a membership predicate. If I is of type T and $\hat{I}[\mathtt{A}]$ is of type $\{T\}$, then $I \in \hat{I}[\mathtt{A}]$ is a membership predicate.

- Inclusion Predicates: If $\hat{I}[\mathtt{A}]$ and I are of the same set type, then $\hat{I}[\mathtt{A}] \subset I$, $\hat{I}[\mathtt{A}] \subseteq I$, $\hat{I}[\mathtt{A}] \supset I$ and $\hat{I}[\mathtt{A}] \supseteq I$ are the inclusion predicates.

- Substring Predicates: If $\hat{I}[\mathtt{A}]$ and I are strings, then $\hat{I}[\mathtt{A}] \sqsubset I$ and $I \sqsubset \hat{I}[\mathtt{A}]$ are substring predicates.

A *predicate* is then defined as follows: (1) An atomic predicate is a predicate. (2) If P is a predicate, then (P) and $\neg P$ are predicates. (3) If P_1 and P_2 are predicates, then $P_1 \wedge P_2$ and $P_1 \vee P_2$ are predicates.

CHAPTER 4

TEXPROS DOCUMENT MODEL

The basic elements of the TEXPROS Document Model are frame templates (and their associated frame instances) and folder organizations (and their constituent folders). The attributes (or properties) of frame instances are specified as frame templates. The frame templates form a document type hierarchy whose members are related by an *IS-A* relationship. The frame templates, and therefore the document type hierarchy, are driven by the types of document in the office environment and are relatively stable over time. Folders are defined by the user as heterogeneous sets of frame instances of different frame template types. Frame instances may be added to folders over time. A folder organization is defined by a user corresponding to the user's view of the document organization.

4.1 Document Type Hierarchy

Let $\mathcal{O}$ denote the set of original documents in a user's office environment. Consider these documents of different classes. Each document class is represented by its attributes to form a *frame template.* Information on a particular office document is extracted according to its frame template by filling in attributes with instances, to form a synopsis of the document which is called a *frame instance.* The relationship among office documents, frame templates, and frame instances is shown in Figure 4.1.

In TEXPROS, a classifier creates frame templates for the office documents in an office environment by sampling a stream of office documents, abstracting their general attributes, and grouping them into classes. Formally,

Definition 4.1.1 *(Frame Template)* A frame template F is a tuple type $\mathbf{F} = [\langle \mathtt{A}_1 : T_1\rangle, ..., \langle \mathtt{A}_m : T_m\rangle]$, where $\mathtt{A}_i$ $(1 \leq i \leq m)$ is an attribute over the attribute type T_i. F describes the information structure of a document class in $\mathcal{O}$. □

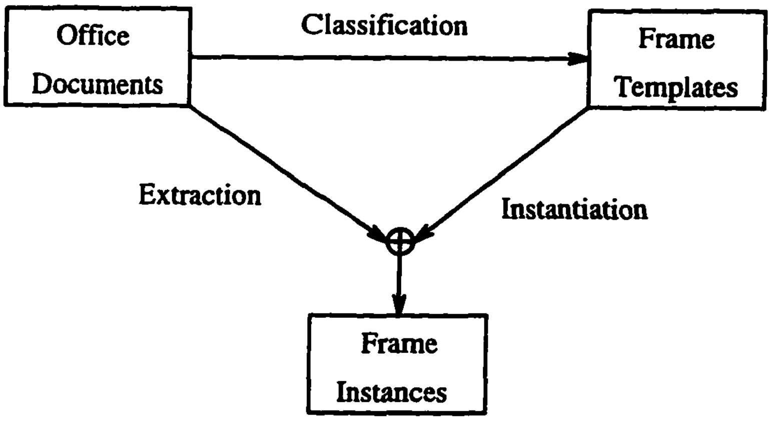


Figure 4.1 Relationship among office documents, frame templates and frame instances

Definition 4.1.2 *(Frame Instance)* Let a document $o \in \mathcal{O}$ belong to a document class $\mathbf{F} = [\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_m : T_m \rangle]$, where $\mathtt{A}_i$ is an attribute, and T_i is an attribute type. Then, a frame instance fi of a document $o \in \mathcal{O}$ is a tuple instance of $\mathbf{F}$, $fi = [\langle \mathtt{A}_1 : I_1 \rangle, ..., \langle \mathtt{A}_m : I_m \rangle]$, where I_i is an instance of attribute type T_i extracted from the document o. □

Given a frame template $\mathbf{F} = [\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_m : T_m \rangle]$, the attributes $\mathtt{A}_1$, ..., $\mathtt{A}_m$ are called the *top level-attributes* of $\mathbf{F}$. We use $< \mathbf{F} >$ to denote all the top level attributes of $\mathbf{F}$. Let A be a top-level attribute and $\mathtt{A.B}_1.\cdots.\mathtt{B}_k$ be a path notation for some attribute $\mathtt{B}_k$. We will simply use attributes to refer to top-level attributes or path notations when the context is clear. Let $\Upsilon(\mathbf{F})$ denote all the possible attributes of $\mathbf{F}$. Let $\mathcal{S} \subseteq \Upsilon(\mathbf{F})$. We define the $\mathcal{S}$-instance of a frame instance fi, denoted $fi(\mathcal{S})$, to be the tuple instance of $\langle \mathtt{A}_j : I_j \rangle$ where $\mathtt{A}_j \in \mathcal{S}$. If $\mathcal{S} \not\subseteq \Upsilon(\mathbf{F})$, then $fi(\mathcal{S}) = [\,]$. For example, let fi be the frame instance shown in Figure 1.1(c) and let $\mathcal{S}$ be {From, To, Subject, Date.Year}. Then $fi(\mathcal{S})$ is the tuple instance [⟨From: [⟨FirstName: "John"⟩, ⟨LastName: "Smith"⟩]⟩, ⟨To: [⟨FirstName: "Tom"⟩, ⟨LastName: "King"⟩]⟩, ⟨Subject: "CIS Qualifying Examination"⟩, ⟨Date.Year: "1992"⟩]. If $\mathcal{S}$ consists of a single attribute, say A, then $fi(\mathcal{S})$ is simply written as fi[A]. For example in Figure 1.1(c), fi[Date.Month] = "Jan.".

Frame templates are related by specialization and generalization [3, 29]. They naturally form a hierarchy which helps to classify documents. An illustration of such a hierarchy is shown in Figure 4.2, where the relationship between frame templates is specified by an *IS-A* relationship. Formally,

Definition 4.1.3 *(IS-A Relationship)* Given two frame templates $\mathbf{F}_1$ and $\mathbf{F}_2$, $\mathbf{F}_1$ *IS-A* $\mathbf{F}_2$ if and only if the attribute-type pairs of $\mathbf{F}_2$ are a subset of the attribute-type pairs of $\mathbf{F}_1$, or equivalently $\mathbf{F}_1 \cap^{\alpha} \mathbf{F}_2 = \mathbf{F}_2$. □

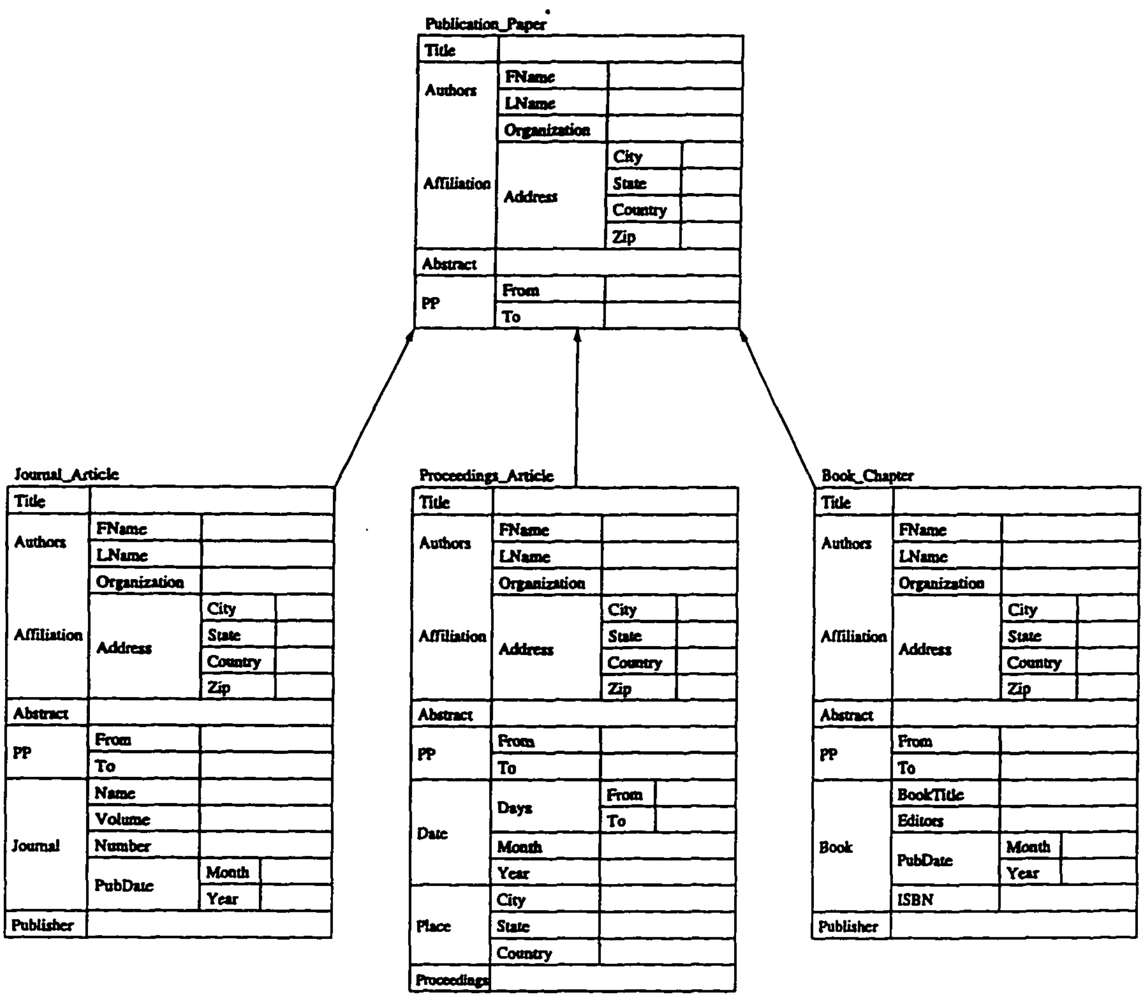


Figure 4.2 *IS-A* relationship among the frame templates

Figure 4.2 shows the *IS-A* relationships among four frame templates: **Paper**, **Journal_Article**, **Proceedings_Article** and **Book_Chapter**. For example, **Journal_Article** *IS-A* (is a specialization of) **Paper**. Whereas, **Paper** can be viewed as a generalization of **Journal_Article**, **Proceedings_Article** and **Book_Chapter**.

Theorem 4.1.1 *The* IS-A *relationship among frame templates is a partial order.*

Proof: Obviously, the IS-A relationship is reflexive, transitive and antisymmetric. □

The *IS-A* relationship is transitive, so it is convenient to define an *immediate-IS-A* relationship as follows.

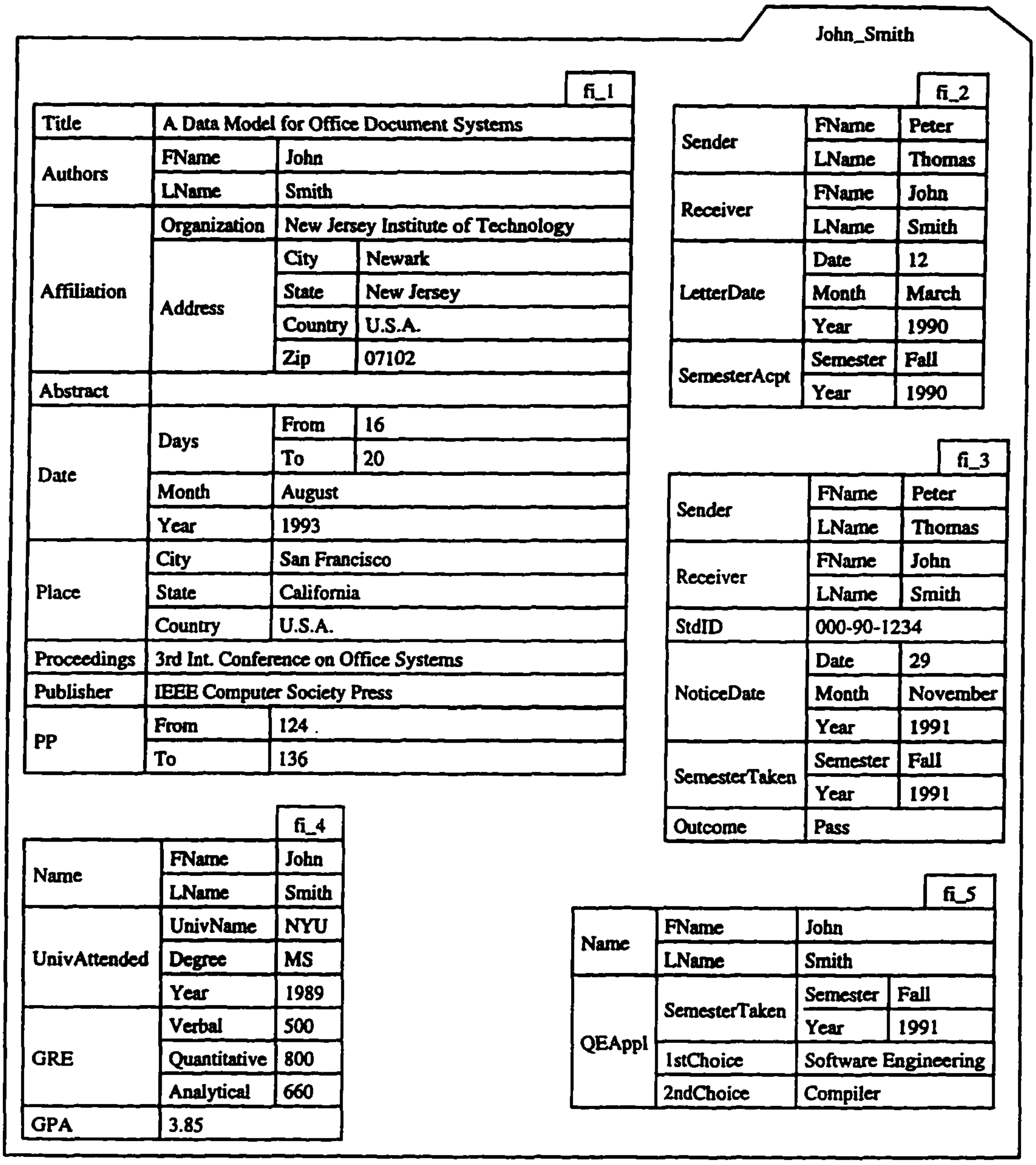
Definition 4.1.4 *(Immediate-IS-A Relationship)* Let $\mathbf{F}_1$ and $\mathbf{F}_2$ be two frame templates. Assume $\mathbf{F}_1$ *IS-A* $\mathbf{F}_2$. We define $\mathbf{F}_1$ *immediately-IS-A* $\mathbf{F}_2$ (denoted *iIS-A*) if and only if there exists no frame template $\mathbf{F}$ ($\neq \mathbf{F}_1$ or $\mathbf{F}_2$) such that $\mathbf{F}_1$ *IS-A* $\mathbf{F}$ and $\mathbf{F}$ *IS-A* $\mathbf{F}_2$. □

Given an *iIS-A* relationship, we define a *document type hierarchy* $\mathcal{DH}(V, E)$ as follows. Each vertex in $V(\mathcal{DH})$ corresponds to a frame template. The root vertex $\mathbf{F}_r$ of $\mathcal{DH}$ is the generic document type (i.e., $\mathbf{F}$ *IS-A* $\mathbf{F}_r$, $\forall \mathbf{F} \in V(\mathcal{DH})$). Given two frame templates $\mathbf{F}_i \in V(\mathcal{DH})$ and $\mathbf{F}_j \in V(\mathcal{DH})$ $(i \neq j)$, $(\mathbf{F}_i, \mathbf{F}_j) \in E(\mathcal{DH})$ if and only if $\mathbf{F}_i$ *iIS-A* $\mathbf{F}_j$. If we impose the additional restriction that whenever x *iIS-A* y and x *iIS-A* z, then $y = z$, then we obtain a *tree document type hierarchy.*

4.2 Folder Organization

A *folder* can be considered as a finite set of frame instances over different frame templates. That is, the folder can be *homogeneous* or *heterogeneous*. Consider frame instances to be grouped into folders on the basis of user-defined criteria, specified as *predicates*, which determine whether a frame instance belongs to a folder. A formal definition of a folder follows.

Definition 4.2.1 *(Folder)* Let Ω denote the set of all the potential frame instances for a user's office environment. A folder f is a set of frame instances in Ω which satisfy a given predicate P. That is $\mathsf{f} = \{fi \mid (fi \in \Omega) \wedge P(fi)\}$, where $P(fi)$ asserts that the frame instance fi satisfies the predicate P. We say P is the predicate associated with the folder f. □



John_Smith

fi_1

Title	A Data Model for Office Document Systems		
Authors	FName	John	
	LName	Smith	
Affiliation	Organization	New Jersey Institute of Technology	
	Address	City	Newark
		State	New Jersey
		Country	U.S.A.
		Zip	07102
Abstract			
Date	Days	From	16
		To	20
	Month	August	
	Year	1993	
Place	City	San Francisco	
	State	California	
	Country	U.S.A.	
Proceedings	3rd Int. Conference on Office Systems		
Publisher	IEEE Computer Society Press		
PP	From	124 .	
	To	136	

fi_2

Sender	FName	Peter
	LName	Thomas
Receiver	FName	John
	LName	Smith
LetterDate	Date	12
	Month	March
	Year	1990
SemesterAcpt	Semester	Fall
	Year	1990

fi_3

Sender	FName	Peter
	LName	Thomas
Receiver	FName	John
	LName	Smith
StdID	000-90-1234	
NoticeDate	Date	29
	Month	November
	Year	1991
SemesterTaken	Semester	Fall
	Year	1991
Outcome	Pass	

fi_4

Name	FName	John
	LName	Smith
UnivAttended	UnivName	NYU
	Degree	MS
	Year	1989
GRE	Verbal	500
	Quantitative	800
	Analytical	660
GPA	3.85	

fi_5

Name	FName	John	
	LName	Smith	
QEAppl	SemesterTaken	Semester	Fall
		Year	1991
	1stChoice	Software Engineering	
	2ndChoice	Compiler	

Figure 4.3 A folder for the Ph.D. student John Smith

Thus a folder is a repository of frame instances which satisfy the folder's predicate. For example, in Figure 4.3, five frame instances relevant to John Smith are grouped into the John_Smith folder. The predicate associated with the John_Smith folder might be specified as follows:

$$\begin{aligned} P(\mathit{fi}) = {} & (\mathit{fi}[\texttt{Name}] = [\langle \texttt{FName} : \mathit{John}\rangle,\ \langle \texttt{LName} : \mathit{Smith}\rangle]) \\ & \vee\ ([\langle \texttt{FName} : \mathit{John}\rangle,\ \langle \texttt{LName} : \mathit{Smith}\rangle] \in \mathit{fi}[\texttt{Authors}] \\ & \vee\ (\mathit{fi}[\texttt{Receiver}] = [\langle \texttt{FName} : \mathit{John}\rangle,\ \langle \texttt{LName} : \mathit{Smith}\rangle]) \end{aligned}$$

If f contains frame instances over frame template **F**, then we say f is associated with **F**. We use f(**F**) to represent all the frame instances in f that are over the frame template **F**. If there is no frame instance in f that is over **F**, then f(**F**) $= \phi$. We use <f> to represent all the frame templates associated with f. Consider Figure 4.3, for instance, <John_Smith> = {**Publication, PhDAcceptLetter, PhDQEResult, UnivTranscript, QEApplication**}. Then, John_Smith(**Publication**) = $\{fi\_1\}$, John_Smith(**PhDAcceptLetter**) = $\{fi\_2\}$, John_Smith(**PhDQEResult**) = $\{fi\_3\}$, John_Smith(**UnivTranscript**) = $\{fi\_4\}$, and John_Smith(**QEApplication**}) = $\{fi\_5\}$.

Folders can be naturally organized into a *folder organization*, where there is an edge from folder (vertex) f_i to folder (vertex) f_j if folder f_j is a subfolder of folder f_i (i.e. every frame instance of f_j is in f_i). For example, Figure 4.4 shows a folder organization represented as a directed tree with seven folders, where the edges are directed from a folder to its subfolders. We will *assume* that the predicate for a child folder f is obtained by imposing an additional restriction or predicate on the uniquely defined predicate of its parent folder f. That is, if f_j is a child of f_i, then $P_{\mathsf{f}_j} = P_{\mathsf{f}_i} \wedge \delta_j$, where δ_j is the additional predicate imposed on f_j, over that imposed on f_i, and P_{f_i} and P_{f_j} are the predicates associated with f_i and f_j. We call this additional predicate δ_j a *local predicate*. In contrast, we call the folder predicates P_{f_j} and P_{f_i} the *global predicates* of folders f_i and f_j, respectively. Thus a frame instance is in a folder f_i if it satisfies the global predicate for f_i while it is also in a child f_j of

f_i if it satisfies the additional requirement represented by δ_j. In set terminology, $\mathsf{f}_i = \{fi \mid fi \in \Omega \wedge P_{\mathsf{f}_i}(fi)\}$ and $\mathsf{f}_j = \{fi \mid fi \in \Omega \wedge P_{\mathsf{f}_j}(fi)\}$. Since $P_{\mathsf{f}_j} = P_{\mathsf{f}_i} \wedge \delta_j$, then $\mathsf{f}_j \subseteq \mathsf{f}_i$.

The paths in a tree folder organization correspond to ***filing paths***. A directed edge $(\mathsf{f}_i, \mathsf{f}_j)$ on a filing path indicates that frame instances in folder f_i are filed into folder f_j if, in addition to the global predicate for f_i, they also satisfy the local predicate for f_j. The filing path for a folder f_i in a tree folder organization is the unique path from the root of the tree to f_i. For example, in Figure 4.4, the filing path for the folder f_4 is $\mathsf{f}_1 \rightarrow \mathsf{f}_2 \rightarrow \mathsf{f}_4$.

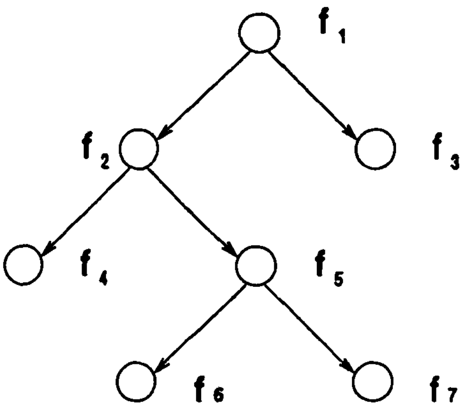


Figure 4.4 A tree folder organization

The child folder f_j of a parent folder f_i is called a *subfolder* (or *immediate subfolder*) of f_i. In the more general situation where there is a nontrivial filing path from f_i to f_j, we refer to f_j as a *remote subfolder* of f_i. For example, in Figure 4.4, every folder in the tree is a remote subfolder of the root folder f_1.

The tree model for a folder organization generalizes naturally to a *DAG (Directed Acyclic Graph) Folder Organization*, where the underlying modeling graph is a rooted DAG whose vertices correspond to folders specified as usual by global predicates, and the root folder is the starting point of document filing. In a DAG folder organization, just like in a tree folder organization, the frame instances belonging to any folder f are obtained by imposing an additional *local predicate*, associated with f, on the global predicates associated with the immediate predecessor vertices (or folders) of f. However, unlike in the tree organization *there may be more than one immediate predecessor.* That is, the global predicate for f is obtained by imposing an additional requirement, represented by the local predicate for f, on the global predicate of each immediate predecessor folder of f. Let f_1, ..., f_k denote all the immediate predecessor folders of the folder f, and let P_i $(1 \leq i \leq k)$ be the global predicates for f_i. The global predicate for f is then just $\delta \wedge (P_1 \vee ... \vee P_k)$, where δ is the local predicate associated with f, or equivalently $\delta(P_1 + ... + P_k)$.

Example 4.2.1 An example is shown in Figure 4.5, where the local predicates are Dept = CIS, Status = PhD, Status = Special Lecturer, Name = John Smith, Name = James Davis, and Name = Kevin Johnson. Thus the frame instances in the James Davis folder satisfy the global predicate:

(Name=James Davis) ∧ [((Dept=CIS) ∧ (Status=PhD)) ∨
((Dept=CIS) ∧ (Status=Special Lecturer))]

□

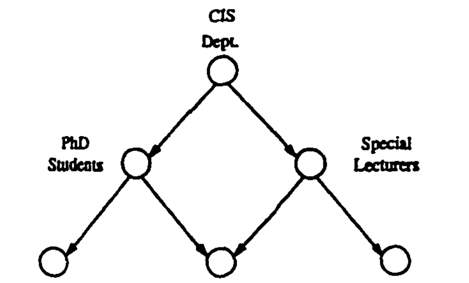


Figure 4.5 An example of DAG folder organization

A folder organization may be formally defined as follows.

Definition 4.2.2 *(Folder Organization)* A folder organization is a two-tuple, $\mathcal{FO}(G, \Delta) = [G(V, E), \Delta]$, where:

1. $G(V, E)$ (also denoted $G(\mathcal{FO})$) is a rooted DAG, with every vertex reachable from the root, and

   - Each vertex in $V(G)$ corresponds to a folder; the root vertex denotes the generic folder of $\mathcal{FO}$.
   - A directed edge $(\mathsf{f}_i, \mathsf{f}_j) \in E(G)$ indicates that frame instances in f_i that additionally satisfy the local predicate for f_j also belong to f_j.

2. $\Delta = \{\delta_i \mid 1 \leq i \leq |V(G)|\}$ is a set of local predicates, δ_i being the local predicate for f_i. □

Thus, a filing path from folder f_i to folder f_j in a $\mathcal{FO}$ is just a path from f_i to f_j in $G(\mathcal{FO})$. Note that there may be more than one filing path from folder f_i to folder f_j.

Each filing path q of a folder f has an *associated predicate* p equal to $\prod_{v \in V(q)} \delta_v$. The global predicate P for each folder $\mathsf{f} \in V(G(\mathcal{FO}))$ can then be represented as:

$$P = \sum_{q \in paths(\mathsf{f})} \Big(\prod_{v \in V(q)} \delta_v \Big),$$

where $paths(\mathsf{f})$ is the set of all filing paths from the root to f and δ_v is the local predicate of $v \in V(q)$.

If two predicates P_1 and P_2 are equivalent, it is denoted by $P_1 \sim P_2$. The equivalence of folder organizations, which we will use it to discuss the optimization problem of folder organizations, is defined as follows.

Definition 4.2.3 *(Equivalence of Two Folder Organizations)* Give any two folder organizations $\mathcal{FO}(G(V, E), \Delta)$ and $\mathcal{FO}(G'(V', E'), \Delta')$, $\mathcal{FO}(G, \Delta)$ is equivalent to $\mathcal{FO}(G', \Delta')$ if and only if $V(G) = V'(G')$ and for $\forall \mathsf{f} \in V(G)$, $\exists \mathsf{f}' \in V'(G')$ such that their global predicates (P_{f}, $P_{\mathsf{f}'}$) are equivalent, that is, $P_{\mathsf{f}} \sim P_{\mathsf{f}'}$. □

A *depends-on* relationship between folders was introduced in [52]. Here, we define a depends-on relationship in terms of a deletion operation *Del*. $Del(\mathcal{FO}(G,\Delta), \mathsf{f})$ indicates that a folder f is deleted from a folder organization $\mathcal{FO}(G,\Delta)$. The folder deletion operation *Del* may be defined as follows.

Definition 4.2.4 *(Folder Deletion Operation (Del))* Given a folder organization $\mathcal{FO}(G,\Delta)$, $Del(\mathcal{FO}(G,\Delta), \mathsf{f}) = \mathcal{FO}(G'(V',E'),\Delta')$ where G' is the induced subgraph [37] on the set of vertices $V' \subseteq V(G) - \{\mathsf{f}\}$ which are reachable from the root of G, and Δ' is the set of local predicates for V'. □

Consider the folder organization $\mathcal{FO}(G(V,E),\Delta)$ shown in Figure 4.6, where $V = \{\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3, \mathsf{f}_4, \mathsf{f}_5, \mathsf{f}_6, \mathsf{f}_7, \mathsf{f}_8\}$. $Del(\mathcal{FO}(G,\Delta), \mathsf{f}_2) = \mathcal{FO}(G'(V',E'),\Delta')$ where $V' = \{\mathsf{f}_1, \mathsf{f}_3, \mathsf{f}_6, \mathsf{f}_8\}$.

Various depends-on relationships between different folders may then be defined as follows.

Definition 4.2.5 *(Depends-On Relationships)* Let $\mathcal{FO} = [G(V,E), \Delta]$ be a folder organization.

1. A folder $\mathsf{f}' \in V(G(\mathcal{FO}))$ is said to ***totally depend-on*** a folder f if $\mathsf{f}' \notin V(G'(Del(\mathcal{FO}(G,\Delta),\mathsf{f})))$.

2. A folder $\mathsf{f}' \in V(G(\mathcal{FO}))$ is said to ***partially depend-on*** a folder f if some, but not all the (filing) paths from the root of $\mathcal{FO}(G,\Delta)$ to f' are disconnected in $Del(\mathcal{FO}(G,\Delta),\mathsf{f})$.

3. A folder $\mathsf{f}' \in V(G(\mathcal{FO}))$ is said to be *independent-of* a folder f if none of the filing paths to f' is disconnected in $Del(\mathcal{FO}(G,\Delta),\mathsf{f})$. □

We denote these relations as follows: for f' totally dependent-on f: $\mathsf{f}' \prec\!\prec \mathsf{f}$; for f' partially dependent-on f: $\mathsf{f}' \prec \mathsf{f}$; for f' independent-of f: $\mathsf{f}' \prec\!\succ \mathsf{f}$.

These relations are complete and mutually exclusive in the sense that for any pair of folders f' and f ($\mathsf{f}' \neq \mathsf{f}$), exactly one of the relations ($\mathsf{f}' \prec\!\prec \mathsf{f}$, $\mathsf{f}' \prec \mathsf{f}$, $\mathsf{f}' \prec\succ \mathsf{f}$) holds. There are also obvious relationships between these relations. For example, if $\mathsf{f}' \prec\!\prec \mathsf{f}$, then $\mathsf{f} \prec\succ \mathsf{f}'$, because $\mathsf{f}' \prec\!\prec \mathsf{f}$ implies every path from the root to f' passes through f, whence deleting f' from $\mathcal{FO}$ affects no path from the root to f. It is also true that if $\mathsf{f}' \prec \mathsf{f}$, then $\mathsf{f} \prec\succ \mathsf{f}'$, since $\mathsf{f}' \prec \mathsf{f}$ implies there exists some path to f' from f, whence, by the acyclic nature of $G(\mathcal{FO})$, there exists no path to f from f'. We can similarly establish, for example, transitivity, such as if $\mathsf{f}_1 \prec\!\prec \mathsf{f}_2$ and $\mathsf{f}_2 \prec\!\prec \mathsf{f}_3$, then $\mathsf{f}_1 \prec\!\prec \mathsf{f}_3$, and so on.

There is *no* partially depends-on relationship in a Tree Folder Organization because of the uniqueness of paths in a tree. For example, in Figure 4.4, $\mathsf{f}_3 \prec\!\prec \mathsf{f}_1$, and $\mathsf{f}_7 \prec\succ \mathsf{f}_4$, but no folder partially depends-on any other folder. In a DAG folder organization, however, all the depends-on relationships are possible. For example, consider Figure 4.6, where $\mathsf{f}_2 \prec\!\prec \mathsf{f}_1$, $\mathsf{f}_8 \prec \mathsf{f}_5$, and $\mathsf{f}_7 \prec\succ \mathsf{f}_4$.

We extend the *totally-depends-on* relationship to a set of folders as follows. Let F be a set of folders in a DAG folder organization $\mathcal{FO}(G, \Delta)$. We say a folder f' *totally depends-on* the set F (denoted $\mathsf{f}' \prec\!\prec \mathsf{F}$) if f' partially depends-on every folder $\mathsf{f} \in \mathsf{F}$ and $\mathsf{f}' \notin V'(G'(Del(\mathcal{FO}(G, \Delta), \mathsf{F})))$. For example, in Figure 4.6, folder $\mathsf{f}_8 \prec\!\prec \{\mathsf{f}_2, \mathsf{f}_3\}$. The relationship is, of course, not necessarily unique. Thus, in Figure 4.6, we also have: $\mathsf{f}_8 \prec\!\prec \{\mathsf{f}_2, \mathsf{f}_6\}$, $\mathsf{f}_8 \prec\!\prec \{\mathsf{f}_3, \mathsf{f}_5\}$, and $\mathsf{f}_8 \prec\!\prec \{\mathsf{f}_5, \mathsf{f}_6\}$.

Theorem 4.2.1 *If* $\mathsf{F} = \{\mathsf{f}_1, ..., \mathsf{f}_k\}$ *is a set of folders that the folder* f *totally-depends-on, then* $\mathsf{f} \subseteq \bigcup_{i=1}^{k}(\mathsf{f}_i)$.

Proof: By the definition of totally-depends-on a set of folders, every filing path from the root to f passes through at least one vertex (folder) $\mathsf{f}_i \in \mathsf{F}$. Thus, every

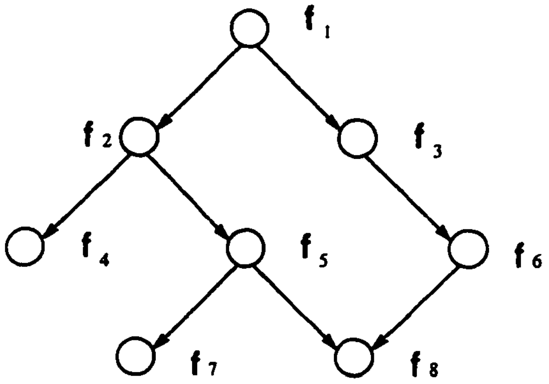


Figure 4.6 A DAG folder organization

instance in f must be contained in at least one f_i, whence it follows that f itself must be contained in the union of the f_i's. □

CHAPTER 5

EXTENDED $\mathcal{D}$_ALGEBRA

Table 5.1 lists the extended $\mathcal{D}$_algebra operators; they are categorized into nine classes. Each class of operators will be discussed in turn in the following sections.

Table 5.1 Operators of the $\mathcal{D}$_Algebra

Class	Operators	Type	Operands	Result
1	$\cup, \cap, -$	binary	folders	folder
2	$\bullet$ $\times$	binary binary	fr. instances folders	fr. instances folder
3	π	unary	folder	folder
4	σ	unary	folder	folder
5	$\bowtie$	binary	folders	folder
6	ρ	unary	folder	folder
7	ν, ν^*, μ, μ^*	unary	folder	folder
8	cont, sum, avg, min, max	unary	folder	NUM
9	γ_{A_β} (β is a subset of the descendant attributes of a top-level attribute A)	unary	folder	folder

Figure 5.1 shows a partial folder organization that a departmental chairperson of a university may use in keeping track of the status of his/her faculty members and Ph.D. students. We illustrate some of the operators using examples drawn from a part of the folder organization shown in Figure 5.1.

5.1 Class 1: Set Theoretic Operators

The first class of operators consists of the binary set theoretic operators for folders. These include the *union* ($\cup$), *intersection* ($\cap$), and *difference* ($-$).

Definition 5.1.1 Let f_1 and f_2 be two folders.

- The *union* of f_1 and f_2, denoted $f_1 \cup f_2$, is the set of frame instances that belong to either f_1 or f_2 or both, i.e., $f_1 \cup f_2 = \{fi | (fi \in f_1) \vee (fi \in f_2)\}$.

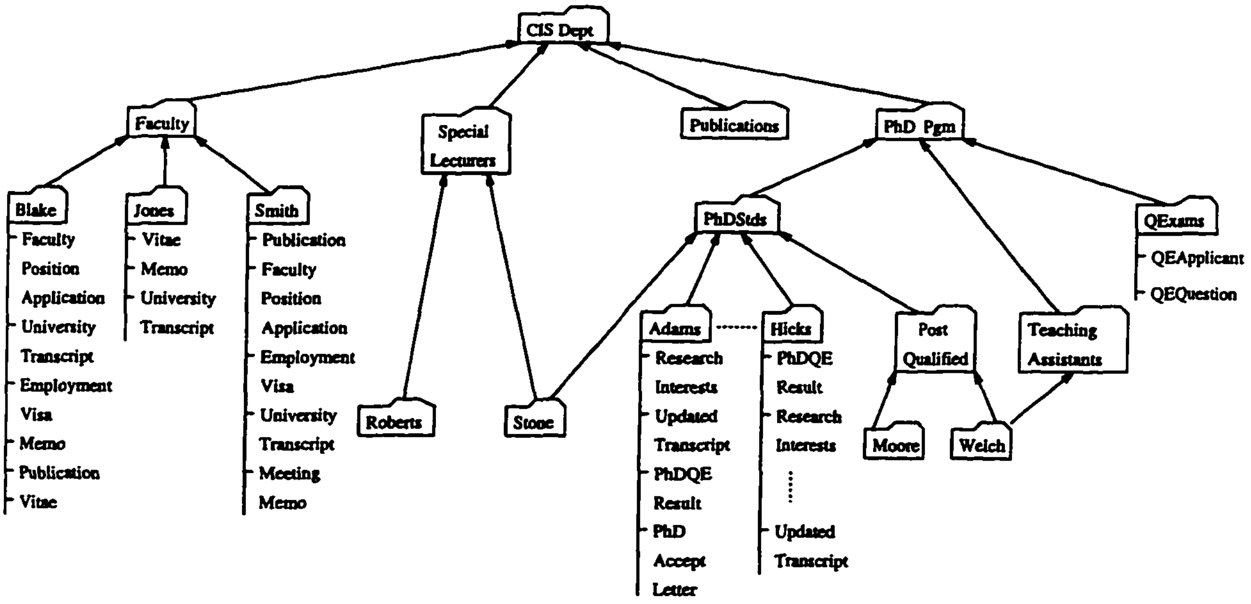


Figure 5.1 A partial folder organization

- The *intersection* of f_1 and f_2, denoted $f_1 \cap f_2$, is the set of frame instances that are in both f_1 and f_2, i.e., $f_1 \cap f_2 = \{fi | (fi \in f_1) \wedge (fi \in f_2)\}$.

- The *difference* of f_1 and f_2, denoted $f_1 - f_2$, is the set of frame instances that are in f_1 but not in f_2, i.e., $f_1 - f_2 = \{fi | (fi \in f_1) \wedge (fi \notin f_2)\}$.

Theorem 5.1.1 *Both the union and the intersection operations are commutative and associative. The difference operation is neither commutative nor associative (i.e., there exist folders* f_1 *and* f_2 *such that* $f_1 - f_2 \neq f_2 - f_1$ *and* $f_1 - (f_2 - f_3) \neq (f_1 - f_2) - f_3$*, respectively).*

5.2 Class 2: Concatenation and Cartesian Product

The second class consists of the *concatenation* and *Cartesian product* operators.

Definition 5.2.1 Let fi_1 and fi_2 be two frame instances over frame templates F_1 and F_2, respectively. Then the concatenation of fi_1 and fi_2, denoted $fi_1 \bullet fi_2$, is:

$$fi_1 \bullet fi_2 = \begin{cases} [\,] & \text{if } \exists A \in \Upsilon(F_1) \cap \Upsilon(F_2) \text{ such that } fi_1[A] \neq fi_2[A] \\ fi & \text{otherwise,} \end{cases}$$

where fi is a frame instance over $\mathbf{F}_1 \cup^\alpha \mathbf{F}_2$ and for each $\langle \mathtt{A}_i : V_i \rangle$ in fi, either $\langle \mathtt{A}_i : V_i \rangle$ is in fi_1 or $\langle \mathtt{A}_i : V_i \rangle$ is in fi_2.

Definition 5.2.2 Let f_1 and f_2 be two folders. Then, the Cartesian product of f_1 and f_2, denoted $\mathsf{f}_1 \times \mathsf{f}_2$, is the folder $\{\mathit{fi}_1 \bullet \mathit{fi}_2 \mid (\mathit{fi}_1 \in \mathsf{f}_1) \wedge (\mathit{fi}_2 \in \mathsf{f}_2)\}$.

We define $\{[\ ]\} = \phi$. Thus, $\{[\ ], \mathit{fi}\} = \{[\ ]\} \cup \{\mathit{fi}\} = \{\mathit{fi}\}$. Intuitively, the Cartesian product of two folders f_1 and f_2 is a set of frame instances which are formed as a result of the concatenation of *every* frame instance of f_1 with *every* frame instance of f_2.

5.3 Class 3: Project Operator

The third class consists of the unary restrictive operator *project* (π) for folders. Informally, given a folder f, the projection of f onto a set of attributes $\mathcal{S}$, denoted $\pi_{\mathcal{S}}(\mathsf{f})$, yields a new folder which is a restriction of f to the attributes in $\mathcal{S}$.

Definition 5.3.1 Let f be a folder, and $\mathcal{S} = \{\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_k\}$ where $\mathtt{A}_j, 1 \leq j \leq k$, is an attribute. The project operation is defined as follows:

$$\pi_{\mathcal{S}}(\mathsf{f}) = \begin{cases} \bigcup_{\mathbf{F} \in <\mathsf{f}>} (\pi_{\mathcal{S}}(\mathsf{f}(\mathbf{F}))) & \text{if } \forall \mathbf{F} \in <\mathsf{f}>, \text{either } \mathcal{S} \cap \Upsilon(\mathbf{F}) = \phi \\ & \text{or } \mathcal{S} \subseteq \Upsilon(\mathbf{F}) \\ \bigcup_{\hat{\mathbf{F}} \in <\hat{\mathsf{f}}>} (\pi_{\mathcal{S}}(\hat{\mathsf{f}}(\hat{\mathbf{F}}))) & \text{otherwise,} \end{cases}$$

where

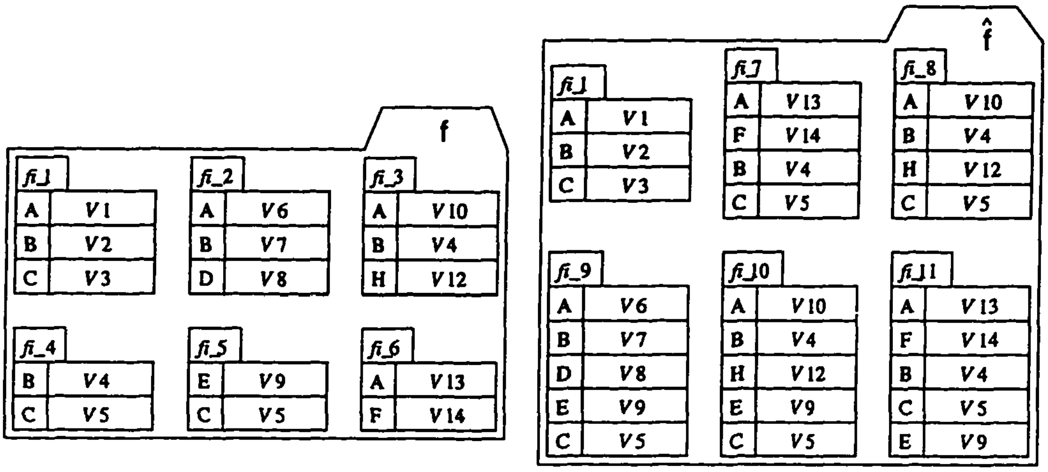
$$\pi_{\mathcal{S}}(\mathsf{f}(\mathbf{F})) = \begin{cases} \phi & \text{if } \mathcal{S} \cap \Upsilon(\mathbf{F}) = \phi \\ \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in \mathsf{f}(\mathbf{F})\} & \text{if } \mathcal{S} \subseteq \Upsilon(\mathbf{F}), \end{cases}$$

and $\pi_{\mathcal{S}}(\hat{\mathsf{f}}(\hat{\mathbf{F}})) = \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in \hat{\mathsf{f}}(\hat{\mathbf{F}})\}$ where

$$< \hat{\mathsf{f}} >= \bigcup_{\mathbf{F} \in \mathcal{A}} \{\mathbf{F}\} \cup \bigcup_{\{\mathbf{F}_{i_1}, \mathbf{F}_{i_2}, \ldots, \mathbf{F}_{i_l}\} \in \mathcal{B}} (\{\mathbf{F}_{i_1} \cup^\alpha \mathbf{F}_{i_2} \cup^\alpha \ldots \cup^\alpha \mathbf{F}_{i_l}\}),$$

$$\hat{f} = \bigcup_{F\in\mathcal{A}} (f(F)) \cup \bigcup_{\{F_{i_1},F_{i_2},...,F_{i_l}\}\in\mathcal{B}} (f(F_{i_1}) \times f(F_{i_2}) \times ... \times f(F_{i_l})),$$

where $\mathcal{A}$ contains all the frame templates $F \in <$ f $>$ such that $\mathcal{S} \subseteq \Upsilon(F)$ and $\mathcal{B}$ is a collection of sets of frame templates $\{F_{i_1}, F_{i_2}, ..., F_{i_l}\} \subseteq <$ f $>$ such that $\mathcal{S} \subseteq \bigcup_{m=i_1}^{i_l} \Upsilon(F_m)$.



(a) (b)

Figure 5.2 Illustration of the project operation

We define $\pi_{\mathcal{S}}(\phi) = \phi$, for all $\mathcal{S}$. Figure 5.2 gives an example to illustrate how the project operator works. Initially, we have a set of frame instances in the folder f (Figure 5.2(a)). That is, f = {*fi_1*, *fi_2*, *fi_3*, *fi_4*, *fi_5*, *fi_6*}. Each frame instance *fi_i*, $1 \leq i \leq 6$, is over the frame template F_i. Let $\mathcal{S}$ = {A, B, C}. By the definition, $< \hat{f} >$ = $\{F_1, F_2 \cup^\alpha F_4, F_2 \cup^\alpha F_5, F_3 \cup^\alpha F_4, F_3 \cup^\alpha F_5, F_4 \cup^\alpha F_6, F_2 \cup^\alpha F_3 \cup^\alpha F_4, F_2 \cup^\alpha F_3 \cup^\alpha F_5, F_2 \cup^\alpha F_4 \cup^\alpha F_5, F_2 \cup^\alpha F_4 \cup^\alpha F_6, F_2 \cup^\alpha F_5 \cup^\alpha F_6, F_3 \cup^\alpha F_4 \cup^\alpha F_5, F_3 \cup^\alpha F_4 \cup^\alpha F_6, F_4 \cup^\alpha F_5 \cup^\alpha F_6, F_2 \cup^\alpha F_3 \cup^\alpha F_4 \cup^\alpha F_5, F_2 \cup^\alpha F_3 \cup^\alpha F_4 \cup^\alpha F_6, F_2 \cup^\alpha F_4 \cup^\alpha F_5 \cup^\alpha F_6, F_2 \cup^\alpha F_3 \cup^\alpha F_5 \cup^\alpha F_6, F_3 \cup^\alpha F_4 \cup^\alpha F_5 \cup^\alpha F_6, F_2 \cup^\alpha F_3 \cup^\alpha F_4 \cup^\alpha F_5 \cup^\alpha F_6, ... \}$. In terms of the definition of Cartesian product, $\hat{f}$ = {*fi_1*} $\cup$ (f(F_2) $\times$ f(F_5)) $\cup$ (f(F_3) $\times$ f(F_4)) $\cup$ (f(F_3) $\times$ f(F_5)) $\cup$ (f(F_4) $\times$ f(F_6)) $\cup$ (f(F_3) $\times$ f(F_4)) $\times$ f(F_5)) $\cup$ (f(F_4) $\times$ (f(F_5) $\times$ f(F_6)) = {*fi_1*, *fi_7*, *fi_8*, *fi_9*, *fi_10*, *fi_11*} (Figure 5.2(b)).

Finally, $\pi_S(\mathsf{f}) = \pi_S(\hat{\mathsf{f}}) = \{[\langle \texttt{A} : V1\rangle, \langle \texttt{B} : V2\rangle, \langle \texttt{C} : V3\rangle], [\langle \texttt{A} : V13\rangle, \langle \texttt{B} : V4\rangle, \langle \texttt{C} : V5\rangle], [\langle \texttt{A} : V10\rangle, \langle \texttt{B} : V4\rangle, \langle \texttt{C} : V5\rangle], [\langle \texttt{A} : V6\rangle, \langle \texttt{B} : V7\rangle, \langle \texttt{C} : V5\rangle]\}$.

Example 5.3.1 Consider again the folder Smith in Figure 5.1. Then, the query $\pi_{\{\texttt{Title},\texttt{Authors},\texttt{Date}\}}(\mathsf{Smith})$ returns a folder composed of frame instances having attributes Title, Authors and Date, namely, {[⟨Title: "D_Model: A Data Model for Office Documents"⟩, ⟨Authors:"Steve Smith"⟩, ⟨Date : [⟨Month : "June"⟩, ⟨Year: "1992"⟩]⟩]}. On the other hand, $\pi_{\{\texttt{Title},\texttt{Authors},\texttt{DegreeObtained}\}}(\mathsf{Smith}) = \pi_{\{\texttt{Title},\texttt{Authors},\texttt{DegreeObtained}\}}(\hat{\mathsf{f}})$, where $\hat{\mathsf{f}} :=$ Smith(**Publication**) × Smith(**FacultyPositionApplication**). And the result would be {[⟨Title: "D_Model: A Data Model for Office Documents"⟩, ⟨Authors:"Steve Smith"⟩, ⟨DegreeObtained:"PhD"⟩]}. □

Theorem 5.3.1 *Let* f *be a folder and* S_1 *and* S_2 *be two sets of attributes. Suppose* $\pi_{S_1}(\mathsf{f}) \neq \phi$ *and* $\pi_{S_2}(\mathsf{f}) \neq \phi$.

(i) If $S_1 = S_2$, *then* $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) = \pi_{S_2}(\pi_{S_1}(\mathsf{f}))$.

(ii) If $S_1 \neq S_2$, *then* $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) \neq \pi_{S_2}(\pi_{S_1}(\mathsf{f}))$ *except where both* $\pi_{S_1}(\pi_{S_2}(\mathsf{f}))$ *and* $\pi_{S_2}(\pi_{S_1}(\mathsf{f}))$ *are empty.*

Proof: (i) is straightforward. For (ii), we consider two cases:

Case 1: $S_1 \cap S_2 = \phi$. Thus, $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) = \pi_{S_2}(\pi_{S_1}(\mathsf{f})) = \phi$.

Case 2: $S_1 \cap S_2 \neq \phi$. There are three subcases to examine:

(1) $S_1 \subset S_2$. $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) = \pi_{S_1}(\mathsf{f}) \neq \pi_{S_2}(\pi_{S_1}(\mathsf{f})) = \phi$.

(2) $S_2 \subset S_1$. $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) = \phi \neq \pi_{S_2}(\pi_{S_1}(\mathsf{f})) = \pi_{S_2}(\mathsf{f})$.

(3) $S_1 \not\subset S_2$ and $S_2 \not\subset S_1$. $\pi_{S_1}(\pi_{S_2}(\mathsf{f})) = \pi_{S_2}(\pi_{S_1}(\mathsf{f})) = \phi$. □

Let S be a set of attributes. We say two folders f_1 and f_2 satisfy the *zero-one condition with respect to* S if for all frame templates $\mathbf{F} \in < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$, either $S \subseteq \Upsilon(\mathbf{F})$ or $S \cap \Upsilon(\mathbf{F}) = \phi$.

Theorem 5.3.2 *Let* S *be a set of attributes and* $\theta \in \{\cup, \cap, -\}$.

(i) For any two folders f_1 *and* f_2, $\pi_S(\mathsf{f}_1 \theta \mathsf{f}_2) = \pi_S(\mathsf{f}_1)\theta\pi_S(\mathsf{f}_2)$ *provided that* f_1 *and* f_2

satisfy the zero-one condition with respect to $\mathcal{S}$.

(ii) There exist two folders f_1 *and* f_2 *such that* $\pi_{\mathcal{S}}(\mathsf{f}_1\theta\mathsf{f}_2) \neq \pi_{\mathcal{S}}(\mathsf{f}_1)\theta\pi_{\mathcal{S}}(\mathsf{f}_2)$ *where* f_1 *and* f_2 *do not satisfy the zero-one condition with respect to* $\mathcal{S}$. *(i.e., there exists* $\mathbf{F} \in\ < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$ *such that* $\mathcal{S} \not\subseteq \Upsilon(\mathbf{F})$ *and* $\mathcal{S} \cap \Upsilon(\mathbf{F}) \neq \phi$*).*

Proof: (i) It suffices to consider only the frame templates $\mathbf{F} \in\ < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$ where $\mathcal{S} \subseteq \Upsilon(\mathbf{F})$. Let $\mathcal{F}$ contain all such frame templates. We only prove $\pi_{\mathcal{S}}(\mathsf{f}_1 \cup \mathsf{f}_2) = \pi_{\mathcal{S}}(\mathsf{f}_1) \cup \pi_{\mathcal{S}}(\mathsf{f}_2)$. For the other operators, they can be proved similarly. For any frame template $\mathbf{F} \in \mathcal{F}$, there are two cases to be examined:

Case 1: $\mathbf{F} \in\ < \mathsf{f}_1 > \cap < \mathsf{f}_2 >$. Then,

$$
\begin{aligned}
&\pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F}) \cup \mathsf{f}_2(\mathbf{F})) \\
&= \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in (\mathsf{f}_1(\mathbf{F}) \cup \mathsf{f}_2(\mathbf{F}))\} && \text{(By Definition 5.3.1)} \\
&= \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in \mathsf{f}_1(\mathbf{F}) \vee \mathit{fi} \in \mathsf{f}_2(\mathbf{F})\} \\
&= \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in \mathsf{f}_1(\mathbf{F})\} \cup \{\mathit{fi}(\mathcal{S}) \mid \mathit{fi} \in \mathsf{f}_2(\mathbf{F})\} \\
&= \pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F})) \cup \pi_{\mathcal{S}}(\mathsf{f}_2(\mathbf{F})). && \text{(By Definition 5.3.1)}
\end{aligned}
$$

Case 2: $\mathbf{F} \in\ < \mathsf{f}_1 > - < \mathsf{f}_2 >$.[1] Then,

$$
\begin{aligned}
&\pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F}) \cup \mathsf{f}_2(\mathbf{F})) \\
&= \pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F}) \cup \phi) && \text{(Since } \mathbf{F} \notin\ < \mathsf{f}_2 >,\ \mathsf{f}_2(\mathbf{F}) = \phi\text{)} \\
&= \pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F})) \cup \pi_{\mathcal{S}}(\mathsf{f}_2(\mathbf{F})). && \text{(Definition 5.3.1 and } \pi_{\mathcal{S}}(\phi) = \phi\text{)}
\end{aligned}
$$

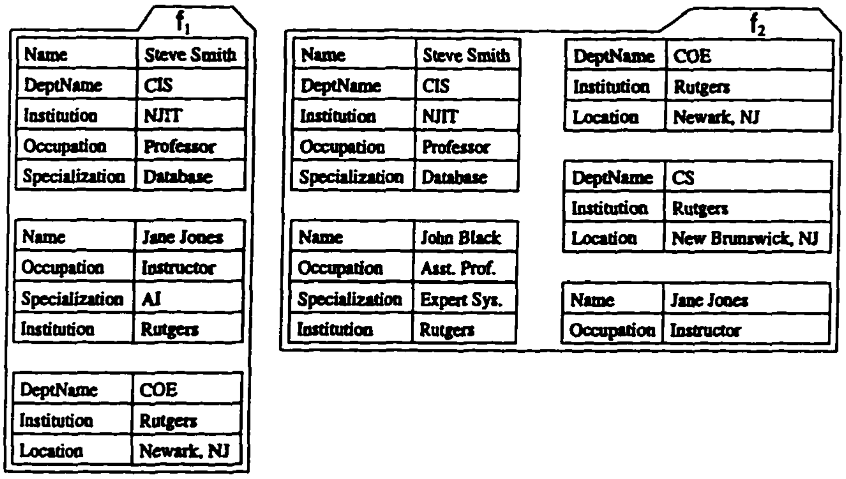
Let $\mathsf{f} = \mathsf{f}_1 \cup \mathsf{f}_2$. Then

$$
\begin{aligned}
&\pi_{\mathcal{S}}(\mathsf{f}_1 \cup \mathsf{f}_2) \\
&= \bigcup_{\mathbf{F} \in <\mathsf{f}_1> \cup <\mathsf{f}_2>} (\pi_{\mathcal{S}}(\mathsf{f}(\mathbf{F}))) && \text{(By Definition 5.3.1)} \\
&= \bigcup_{\mathbf{F} \in \mathcal{F}} (\pi_{\mathcal{S}}(\mathsf{f}(\mathbf{F}))) && \text{(Since } \textstyle\bigcup_{\mathbf{F} \in ((<\mathsf{f}_1> \cup <\mathsf{f}_2>) - \mathcal{F})} (\pi_{\mathcal{S}}(\mathsf{f}(\mathbf{F}))) = \phi\text{)} \\
&= \bigcup_{\mathbf{F} \in \mathcal{F}} (\pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F}) \cup \mathsf{f}_2(\mathbf{F}))) && (\mathsf{f}(\mathbf{F}) = \mathsf{f}_1(\mathbf{F}) \cup \mathsf{f}_2(\mathbf{F})) \\
&= \bigcup_{\mathbf{F} \in \mathcal{F}} (\pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F})) \cup \pi_{\mathcal{S}}(\mathsf{f}_2(\mathbf{F}))) && \text{(In terms of Case 1} \sim \text{2)} \\
&= \bigcup_{\mathbf{F} \in \mathcal{F}} (\pi_{\mathcal{S}}(\mathsf{f}_1(\mathbf{F}))) \cup \bigcup_{\mathbf{F} \in \mathcal{F}} (\pi_{\mathcal{S}}(\mathsf{f}_2(\mathbf{F})))
\end{aligned}
$$

[1] $\mathbf{F} \in\ < \mathsf{f}_2 > - < \mathsf{f}_1 >$ is similar to Case 2.

$= \bigcup_{F\in<f_1>} (\pi_S(f_1(F))) \cup \bigcup_{F\in<f_2>} (\pi_S(f_2(F)))$

$= \pi_S(f_1) \cup \pi_S(f_2).$ (By Definition 5.3.1)



Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor
Specialization	Database

Name	Jane Jones
Occupation	Instructor
Specialization	AI
Institution	Rutgers

DeptName	COE
Institution	Rutgers
Location	Newark, NJ

Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor
Specialization	Database

Name	John Black
Occupation	Asst. Prof.
Specialization	Expert Sys.
Institution	Rutgers

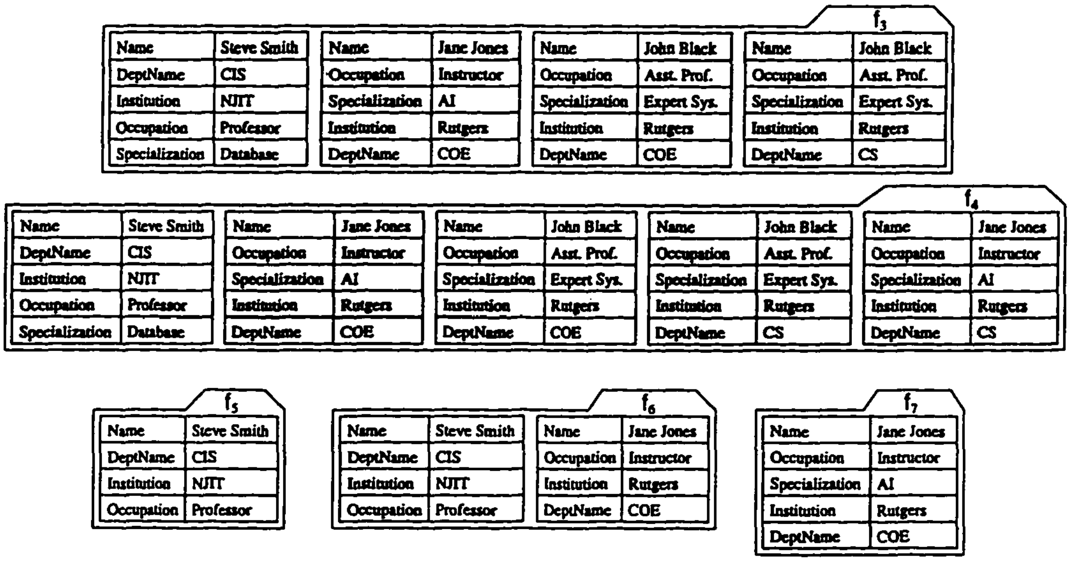
DeptName	COE
Institution	Rutgers
Location	Newark, NJ

DeptName	CS
Institution	Rutgers
Location	New Brunswick, NJ

Name	Jane Jones
Occupation	Instructor

Figure 5.3 Two folders f_1 and f_2

(ii) Consider the folders f_1 and f_2 given in Figure 5.3, and f_3, f_4, f_5, f_6 and f_7 in Figure 5.4. We examine each operator in turn.



Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor
Specialization	Database

Name	Jane Jones
Occupation	Instructor
Specialization	AI
Institution	Rutgers
DeptName	COE

Name	John Black
Occupation	Asst. Prof.
Specialization	Expert Sys.
Institution	Rutgers
DeptName	COE

Name	John Black
Occupation	Asst. Prof.
Specialization	Expert Sys.
Institution	Rutgers
DeptName	CS

Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor
Specialization	Database

Name	Jane Jones
Occupation	Instructor
Specialization	AI
Institution	Rutgers
DeptName	COE

Name	John Black
Occupation	Asst. Prof.
Specialization	Expert Sys.
Institution	Rutgers
DeptName	COE

Name	John Black
Occupation	Asst. Prof.
Specialization	Expert Sys.
Institution	Rutgers
DeptName	CS

Name	Jane Jones
Occupation	Instructor
Specialization	AI
Institution	Rutgers
DeptName	CS

Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor

Name	Steve Smith
DeptName	CIS
Institution	NJIT
Occupation	Professor

Name	Jane Jones
Occupation	Instructor
Institution	Rutgers
DeptName	COE

Name	Jane Jones
Occupation	Instructor
Specialization	AI
Institution	Rutgers
DeptName	COE

Figure 5.4 Five folders f_3, f_4, f_5, f_6 and f_7

(a) For the "$\cup$" operator, let $\mathcal{S} = \{$`Name, DeptName, Institution, Occupation, Specialization`$\}$. $\pi_{\mathcal{S}}(\mathsf{f}_1 \cup \mathsf{f}_2) \neq \pi_{\mathcal{S}}(\mathsf{f}_1) \cup \pi_{\mathcal{S}}(\mathsf{f}_2)$, since $\pi_{\mathcal{S}}(\mathsf{f}_1 \cup \mathsf{f}_2) = \mathsf{f}_4$ whereas $\pi_{\mathcal{S}}(\mathsf{f}_1) \cup \pi_{\mathcal{S}}(\mathsf{f}_2) = \mathsf{f}_3$.

(b) For the "$\cap$" operator, let $\mathcal{S} = \{$`Name, DeptName, Institution, Occupation`$\}$. $\pi_{\mathcal{S}}(\mathsf{f}_1 \cap \mathsf{f}_2) \neq \pi_{\mathcal{S}}(\mathsf{f}_1) \cap \pi_{\mathcal{S}}(\mathsf{f}_2)$, since $\pi_{\mathcal{S}}(\mathsf{f}_1 \cap \mathsf{f}_2) = \mathsf{f}_5$ whereas $\pi_{\mathcal{S}}(\mathsf{f}_1) \cap \pi_{\mathcal{S}}(\mathsf{f}_2) = \mathsf{f}_6$.

(c) For the "$-$" operator, let $\mathcal{S} = \{$`Name, DeptName, Institution, Occupation, Specialization`$\}$. $\pi_{\mathcal{S}}(\mathsf{f}_1 - \mathsf{f}_2) \neq \pi_{\mathcal{S}}(\mathsf{f}_1) - \pi_{\mathcal{S}}(\mathsf{f}_2)$, since $\pi_{\mathcal{S}}(\mathsf{f}_1 - \mathsf{f}_2) = \phi$ whereas $\pi_{\mathcal{S}}(\mathsf{f}_1) - \pi_{\mathcal{S}}(\mathsf{f}_2) = \mathsf{f}_7$. □

5.4 Class 4: Select Operator

The fourth class consists of the unary restrictive operator *select* (σ) for folders. The syntax of the selection operation on a folder f is $\sigma_P(\mathsf{f})$, where P is a predicate clause.

Definition 5.4.1 Let f be a folder and P be a predicate clause. Let $\mathcal{S}$ be the set of attributes appearing in P. The select operation is defined as follows:

$$\sigma_P(\mathsf{f}) = \begin{cases} \bigcup_{\mathbf{F} \in < \mathsf{f} >} (\sigma_P(\mathsf{f}(\mathbf{F}))) & \text{if } \forall \mathbf{F} \in < \mathsf{f} >, \text{ either } \mathcal{S} \cap \Upsilon(\mathbf{F}) = \phi \\ & \text{or } \mathcal{S} \subseteq \Upsilon(\mathbf{F}) \\ \bigcup_{\hat{\mathbf{F}} \in < \hat{\mathsf{f}} >} (\sigma_P(\hat{\mathsf{f}}((\hat{\mathbf{F}}))) & \text{otherwise,} \end{cases}$$

where

$$\sigma_P(\mathsf{f}(\mathbf{F})) = \begin{cases} \{fi \mid (fi \in \mathsf{f}(\mathbf{F}) \wedge P(fi))\} & \text{if } \mathcal{S} \subseteq \Upsilon(\mathbf{F}) \\ \phi & \text{if } \mathcal{S} \cap \Upsilon(\mathbf{F}) = \phi, \end{cases}$$

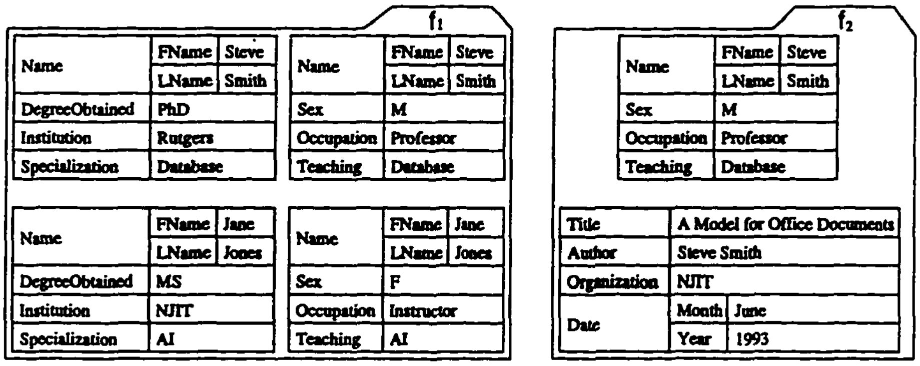
and $\sigma_P(\hat{\mathsf{f}}(\hat{\mathbf{F}})) = \{ fi \mid (fi \in \hat{\mathsf{f}}(\hat{\mathbf{F}}) \wedge P(fi))\}$, where $\hat{\mathsf{f}}$ and $\hat{\mathbf{F}}$ are the same as those in Definition 5.3.1.

Let $\mathcal{S}$ be the set of attributes appearing in a predicate clause P. If $\mathcal{S} \not\subseteq \Upsilon(\mathbf{F})$, then we define $\sigma_P(\mathsf{f}(\mathbf{F})) = \phi$. Furthermore, $\sigma_P(\phi) = \phi$ for any P.

Example 5.4.1 Consider again the folder organization in Figure 5.1 and the query: *List the PhD students who were accepted in the Fall of 1989 and have passed the Qualifying Examination in or before the Spring of 1991.* The algebra expression is as follows: Result $:= \pi_{\{\mathtt{Receiver}\}}(\sigma_P(\mathsf{f})) = \pi_{\{\mathtt{Receiver}\}}(\sigma_P(\hat{\mathsf{f}}))$,
where $P := ((\mathtt{SemTaken} \leq [\langle \mathtt{Season} : \mathit{Spring} \rangle, \langle \mathtt{Year} : 1991 \rangle]) \wedge$
$(\mathtt{SemAccepted} = [\langle \mathtt{Semester} : \mathit{Fall} \rangle, \langle \mathtt{Year} : 1989 \rangle]))$, and
$\hat{\mathsf{f}} :=$ PhDStds(**PhDAcceptLetter**) × PhDStds(**PhDQEResult**). □

In this example, there is no frame template associated with the PhD Students folder PhDStds that contains both attributes SemTaken and SemAccepted (cf. Appendix A). The two attributes are contained in the Cartesian product of PhDStds(**PhDAcceptLetter**) and PhDStds(**PhDQEResult**), in which the frame instances having the same attribute name with different values are eliminated.

The following example shows that selection should usually be performed after applying the Cartesian product to two folders.



Name	FName	Steve
	LName	Smith
DegreeObtained	PhD	
Institution	Rutgers	
Specialization	Database	

Name	FName	Steve
	LName	Smith
Sex	M	
Occupation	Professor	
Teaching	Database	

Name	FName	Jane
	LName	Jones
DegreeObtained	MS	
Institution	NJIT	
Specialization	AI	

Name	FName	Jane
	LName	Jones
Sex	F	
Occupation	Instructor	
Teaching	AI	

Name	FName	Steve
	LName	Smith
Sex	M	
Occupation	Professor	
Teaching	Database	

Title	A Model for Office Documents	
Author	Steve Smith	
Organization	NJIT	
Date	Month	June
	Year	1993

Figure 5.5 Two folders f_1 and f_2

Example 5.4.2 Consider the folders f_1 and f_2 in Figure 5.5. Suppose we are interested in the title, the author of a paper, and the author's degree in the two folders. Let $S = \{$Title, Author, DegreeObtained$\}$. If we simply perform the Cartesian product and projection on f_1 and f_2, we get $\pi_S(\mathsf{f}_1 \times \mathsf{f}_2) = \{[\langle$Title: A Model for Office Document$\rangle$, $\langle$Author: Steve Smith$\rangle$, $\langle$DegreeObtained: PhD$\rangle]$,

[$\langle$`Title`: A Model for Office Document$\rangle$, $\langle$`Author`: Steve Smith$\rangle$, $\langle$`DegreeObtained`: MS$\rangle$]}. This would yield wrong results as it shows inconsistent and extraneous information regarding the degree Smith obtains. To resolve this conflict, we could apply the select operator before projection as follows.

Let $\mathsf{f} := \sigma_{\mathtt{Name}=[\langle \mathtt{FName}:Steve\rangle,\langle \mathtt{LName}:Smith\rangle]}(\mathsf{f}_1 \times \mathsf{f}_2)$.

Then $\pi_S(\mathsf{f})$ yields {[$\langle$`Title`: A Model for Office Document$\rangle$, $\langle$`Author`: Steve Smith$\rangle$, $\langle$`DegreeObtained`: PhD$\rangle$]}. □

Let P_1 and P_2 be two predicate clauses. Let S_1 and S_2 be two sets of attributes appearing in P_1 and P_2, respectively. We say a folder f satisfies the *zero-one condition with respect to P_1 and P_2* if for all frame templates $\mathbf{F} \in <\mathsf{f}>$, either $S_i \subseteq \Upsilon(\mathbf{F})$ or $S_i \cap \Upsilon(\mathbf{F}) = \phi$, $\forall i \in \{1,2\}$.

Theorem 5.4.1 *Let P_1 and P_2 be two predicate clauses.*

(i) For any folder f, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f})) = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}))$ provided that f satisfies the zero-one condition with respect to P_1 and P_2.

(ii) There exists a folder f such that $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f})) \neq \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}))$ where f does not satisfy the zero-one condition with respect to P_1 and P_2 (i.e., there exists $\mathbf{F} \in <\mathsf{f}>$ such that $S_i \not\subseteq \Upsilon(\mathbf{F})$ and $S_i \cap \Upsilon(\mathbf{F}) \neq \phi$, for some $i \in \{1,2\}$, where S_i, $1 \leq i \leq 2$, contains the attributes appearing in P_i).

Proof:

(i) Let S_1 and S_2 be the two sets of attributes appearing in P_1 and P_2, respectively. First, we prove that $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))) = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$, $\forall \mathbf{F} \in <\mathsf{f}>$. There are three cases to be considered:

Case 1: $S_1 \subseteq \Upsilon(\mathbf{F})$ and $S_2 \subseteq \Upsilon(\mathbf{F})$. Thus,

$\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F})))$

$= \sigma_{P_1}(\{fi \mid fi \in \mathsf{f}(\mathbf{F}) \wedge P_2(fi)\})$ (By Definition 5.4.1)

$= \{fi' \mid fi' \in \{fi \mid fi \in \mathsf{f}(\mathbf{F}) \wedge P_2(fi)\} \wedge P_1(fi')\}$

$= \{fi' \mid fi' \in \{fi' \mid fi' \in \mathsf{f}(\mathbf{F}) \wedge P_1(fi')\} \wedge P_2(fi')\}$

$= \sigma_{P_2}\ (\{fi' \mid fi' \in \mathsf{f}(\mathbf{F}) \wedge P_1(fi')\})$

$= \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$ (By Definition 5.4.1)

Case 2: $\mathcal{S}_i \subseteq \Upsilon(\mathbf{F})$ and $\mathcal{S}_j \cap \Upsilon(\mathbf{F}) = \phi$, $i, j \in \{1, 2\}$ and $i \neq j$. There are two subcases: $\mathcal{S}_2 \cap \Upsilon(\mathbf{F}) = \phi$ and $\mathcal{S}_1 \cap \Upsilon(\mathbf{F}) = \phi$. By Definition 5.4.1, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))) = \phi = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$.

Subcase 2.1: $\mathcal{S}_2 \cap \Upsilon(\mathbf{F}) = \phi$. So, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))) = \phi = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$,

Subcase 2.2: $\mathcal{S}_1 \cap \Upsilon(\mathbf{F}) = \phi$. So, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))) = \phi = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$

Case 3: $\mathcal{S}_1 \cap \Upsilon(\mathbf{F}) = \phi$ and $\mathcal{S}_2 \cap \Upsilon(\mathbf{F}) = \phi$. Thus, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))) = \phi = \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F})))$.

Therefore,

$\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}))$

$= \sigma_{P_1}\ (\bigcup_{\mathbf{F} \in <\mathsf{f}>}\ (\sigma_{P_2}(\mathsf{f}(\mathbf{F}))))$ (By Definition 5.4.1)

$= \bigcup_{\mathbf{F} \in <\mathsf{f}>}\ (\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}(\mathbf{F}))))$

$= \bigcup_{\mathbf{F} \in <\mathsf{f}>}\ (\sigma_{P_2}(\sigma_{P_1}(\mathsf{f}(\mathbf{F}))))$ (In terms of Case 1 $\sim$ 3)

$= \sigma_{P_2}\ (\bigcup_{\mathbf{F} \in <\mathsf{f}>}\ (\sigma_{P_1}(\mathsf{f}(\mathbf{F}))))$

$= \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}))$. (By Definition 5.4.1)

(ii) Consider the folder f_1 given in Figure 5.5. Let P_1 be (`Occupation` = *Professor*) $\wedge$ (`DegreeObtained` = *PhD*) and P_2 be (`Specialization` = *Database*).

$\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}_1)) = \phi$.

$\sigma_{P_2}(\sigma_{P_1}(\mathsf{f}_1)) =$ {[⟨`Name` : [⟨`FName` : Steve⟩, ⟨`LName` : Smith⟩]⟩, ⟨`DegreeObtained` : PhD⟩, ⟨`Institution` : Rutgers⟩, ⟨`Specialization` : Database⟩, ⟨`Sex` : M⟩, ⟨`Occupation` : Professor⟩, ⟨`Teaching` : Database⟩]}.

Therefore, $\sigma_{P_1}(\sigma_{P_2}(\mathsf{f}_1)) \neq \sigma_{P_2}(\sigma_{P_1}(\mathsf{f}_1))$. □

Let P be a predicate clause. Let $\mathcal{S}$ be the set of attributes appearing in P. We say two folders f_1 and f_2 satisfy the *zero-one condition with respect to P* if for all frame templates $\mathbf{F} \in < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$, either $\mathcal{S} \subseteq \Upsilon(\mathbf{F})$ or $\mathcal{S} \cap \Upsilon(\mathbf{F}) = \phi$.

Theorem 5.4.2 *Let P be a predicate clause. Let $\theta \in \{\cup, \cap, -\}$.*

(i) For any two folders f_1 and f_2, $\sigma_P(\mathsf{f}_1\theta\mathsf{f}_2) = \sigma_P(\mathsf{f}_1)\theta\sigma_P(\mathsf{f}_2)$ provided that f_1 and f_2 satisfy the zero-one condition with respect to P.

(ii) There exist two folders f_1 and f_2 such that $\sigma_P(\mathsf{f}_1\theta\mathsf{f}_2) \neq \sigma_P(\mathsf{f}_1)\theta\sigma_P(\mathsf{f}_2)$ where f_1 and f_2 do not satisfy the zero-one condition with respect to P (i.e., there exists $\mathbf{F} \in < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$ such that $S \not\subseteq \Upsilon(\mathbf{F})$ and $S \cap \Upsilon(\mathbf{F}) \neq \phi$, where S is the set of attributes appearing in P).

Proof: (i) Let S be the set of attributes appearing in P. It suffices to consider only the frame templates $\mathbf{F} \in < \mathsf{f}_1 > \cup < \mathsf{f}_2 >$ where $S \subseteq \Upsilon(\mathbf{F})$. Let $\mathcal{F}$ contain all such frame templates. We only prove $\sigma_P(\mathsf{f}_1 - \mathsf{f}_2) = \sigma_P(\mathsf{f}_1) - \sigma_P(\mathsf{f}_2)$. For the other operators, they can be proved similarly.

Let $\delta(\mathcal{O})$ be the set of all frame instances. First, we show $\sigma_P(\mathsf{f}_1(\mathbf{F}) - \mathsf{f}_2(\mathbf{F})) = \sigma_P(\mathsf{f}_1(\mathbf{F}))$ - $\sigma_P(\mathsf{f}_2(\mathbf{F}))$, $\forall \mathbf{F} \in \mathcal{F}$. There are two cases to be examined:

Case 1: $\mathbf{F} \in < \mathsf{f}_1 > \cap < \mathsf{f}_2 >$. Thus,

$$
\begin{aligned}
&\sigma_P(\mathsf{f}_1(\mathbf{F}) - \mathsf{f}_2(\mathbf{F})) \\
&= \{fi \mid fi \in (\mathsf{f}_1(\mathbf{F}) - \mathsf{f}_2(\mathbf{F})) \wedge P(fi)\} \\
&= \{fi \mid (fi \in \mathsf{f}_1(\mathbf{F})) \wedge (fi \notin \mathsf{f}_2(\mathbf{F})) \wedge P(fi)\} \\
&= \{fi \mid (fi \in \mathsf{f}_1(\mathbf{F}) \wedge P(fi)) \wedge (fi \notin \mathsf{f}_2(\mathbf{F})) \wedge P(fi)\} \\
&= \{fi \mid fi \in \mathsf{f}_1(\mathbf{F}) \wedge P(fi)\} \cap \{fi \mid fi \notin \mathsf{f}_2(\mathbf{F}) \wedge P(fi)\} \\
&= \sigma_P(\mathsf{f}_1(\mathbf{F})) \cap \{fi \mid fi \in (\delta(\mathcal{O}) - \mathsf{f}_2(\mathbf{F}) \wedge P(fi)\} \\
&= \sigma_P(\mathsf{f}_1(\mathbf{F})) \cap \sigma_P(\delta(\mathcal{O}) - \mathsf{f}_2(\mathbf{F})) \\
&= \sigma_P(\mathsf{f}_1(\mathbf{F})) - \sigma_P(\mathsf{f}_2(\mathbf{F})).
\end{aligned}
$$

Case 2: $\mathbf{F} \in < \mathsf{f}_1 > - < \mathsf{f}_2 >$.[2] Thus,

$$
\begin{aligned}
&\sigma_P(\mathsf{f}_1(\mathbf{F}) - \mathsf{f}_2(\mathbf{F})) \\
&= \sigma_P(\mathsf{f}_1(\mathbf{F}) - \phi) && (\text{Since } \mathbf{F} \notin < \mathsf{f}_2 >, \mathsf{f}_2(\mathbf{F}) = \phi) \\
&= \sigma_P(\mathsf{f}_1(\mathbf{F})) - \sigma_P(\mathsf{f}_2(\mathbf{F})). && (\text{Since } \mathsf{f}_2(\mathbf{F}) = \phi, \sigma_P(\mathsf{f}_2(\mathbf{F})) = \phi)
\end{aligned}
$$

[2] $\mathbf{F} \in < \mathsf{f}_2 > - < \mathsf{f}_1 >$ is similar to Case 2.

Let $f = f_1 - f_2$. Then,

$\sigma_P(f_1 - f_2)$

$= \bigcup_{F\in\langle f_1\rangle\cup\langle f_2\rangle} (\sigma_P(f(F)))$ (By Definition 5.4.1)

$= \bigcup_{F\in\mathcal{F}}(\sigma_P(f(F)))$ (Since $\bigcup_{F\in(\langle f_1\rangle\cup\langle f_2\rangle)-\mathcal{F}} (\sigma_P(f(F))) = \phi$)

$= \bigcup_{F\in\mathcal{F}} (\sigma_P(f_1(F) - f_2(F)))$ ($f(F) = f_1(F) - f_2(F)$)

$= \bigcup_{F\in\mathcal{F}} (\sigma_P(f_1(F))$ - $\sigma_P(f_2(F)))$ (In terms of Case 1 $\sim$ 2)

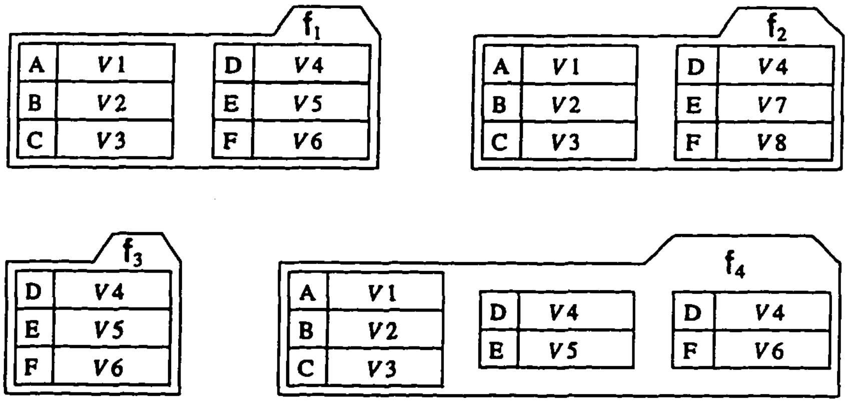
$= \bigcup_{F\in\mathcal{F}} (\sigma_P(f_1(F)))$ - $\bigcup_{F\in\mathcal{F}} (\sigma_P(f_2(F)))$

$= \bigcup_{F\in\langle f_1\rangle} (\sigma_P(f_1(F)))$ - $\bigcup_{F\in\langle f_2\rangle} (\sigma_P(f_2(F)))$

$= \sigma_P(f_1) - \sigma_P(f_2)$ (By Definition 5.4.1).

(ii) Let P be $(A = V1) \wedge (D = V4)$.

Consider the folders f_1, f_2, f_3 and f_4 in Figure 5.6. We examine each operator in turn.



f_1

A	V1		D	V4
B	V2		E	V5
C	V3		F	V6

f_2

A	V1		D	V4
B	V2		E	V7
C	V3		F	V8

f_3

D	V4
E	V5
F	V6

f_4

A	V1		D	V4		D	V4
B	V2		E	V5		F	V6
C	V3						

Figure 5.6 Four folders f_1, f_2, f_3 and f_4

(a) For the "$\cup$" operator, $\sigma_P(f_2 \cup f_3) \neq \sigma_P(f_2) \cup \sigma_P(f_3)$, since

$\sigma_P(f_2 \cup f_3) = \{[\langle A : V1\rangle, \langle B : V2\rangle, \langle C : V3\rangle, \langle D : V4\rangle, \langle E : V7\rangle, \langle F : V8\rangle], [\langle A : V1\rangle, \langle B : V2\rangle, \langle C : V3\rangle, \langle D : V4\rangle, \langle E : V5\rangle, \langle F : V6\rangle]\}$, whereas

$\sigma_P(f_2) \cup \sigma_P(f_3) = \{[\langle A : V1\rangle, \langle B : V2\rangle, \langle C : V3\rangle, \langle D : V4\rangle, \langle E : V7\rangle, \langle F : V8\rangle]\}$.

(b) For the "$\cap$" operator, $\sigma_P(f_1 \cap f_4) \neq \sigma_P(f_1) \cap \sigma_P(f_4)$, since

$\sigma_P(f_1 \cap f_4) = \phi$, whereas

$$\sigma_P(\mathsf{f}_1) \cap \sigma_P(\mathsf{f}_4) = \{[\langle \mathtt{A} : V1\rangle,\ \langle \mathtt{B} : V2\rangle,\ \langle \mathtt{C} : V3\rangle,\ \langle \mathtt{D} : V4\rangle,\ \langle \mathtt{E} : V5\rangle,\ \langle \mathtt{F} : V6\rangle]\}.$$

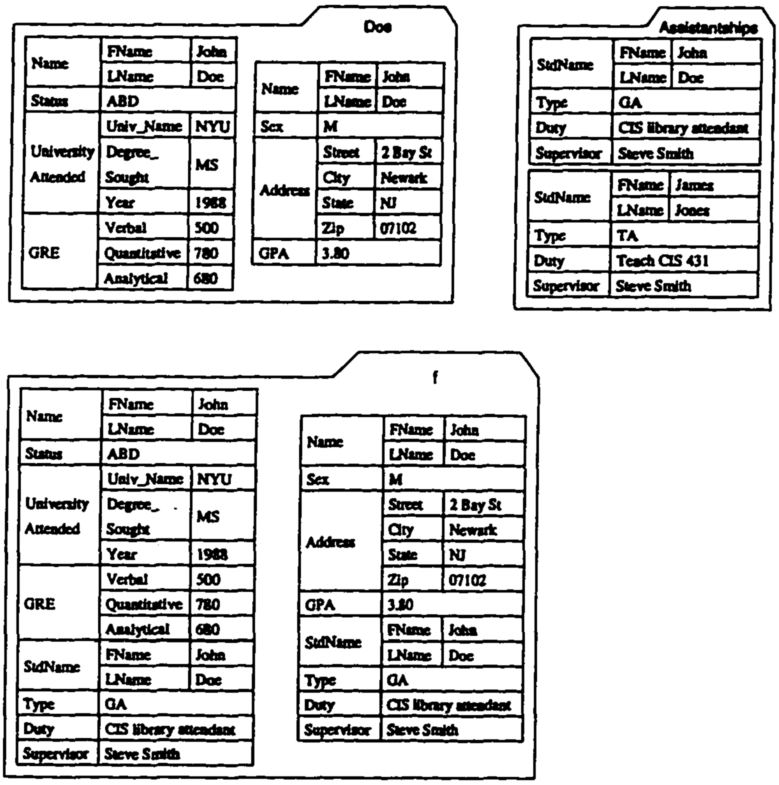
(c) For the "−" operator, $\sigma_P(\mathsf{f}_1 - \mathsf{f}_3) \neq \sigma_P(\mathsf{f}_1) - \sigma_P(\mathsf{f}_3)$, since

$$\sigma_P(\mathsf{f}_1) - \sigma_P(\mathsf{f}_3) = \{[\langle \mathtt{A} : V1\rangle,\ \langle \mathtt{B} : V2\rangle,\ \langle \mathtt{C} : V3\rangle,\ \langle \mathtt{D} : V4\rangle,\ \langle \mathtt{E} : V5\rangle,\ \langle \mathtt{F} : V6\rangle]\},$$

whereas

$$\sigma_P(\mathsf{f}_1 - \mathsf{f}_3) = \phi. \qquad \square$$

5.5 Class 5: Join Operator



Doe

Name	FName	John
	LName	Doe
Status	ABD	
University Attended	Univ_Name	NYU
	Degree_ Sought	MS
	Year	1988
GRE	Verbal	500
	Quantitative	780
	Analytical	680

Name	FName	John
	LName	Doe
Sex	M	
Address	Street	2 Bay St
	City	Newark
	State	NJ
	Zip	07102
GPA	3.80	

Assistantships

StdName	FName	John
	LName	Doe
Type	GA	
Duty	CIS library attendant	
Supervisor	Steve Smith	

StdName	FName	James
	LName	Jones
Type	TA	
Duty	Teach CIS 431	
Supervisor	Steve Smith	

f

Name	FName	John
	LName	Doe
Status	ABD	
University Attended	Univ_Name	NYU
	Degree_ Sought	MS
	Year	1988
GRE	Verbal	500
	Quantitative	780
	Analytical	680
StdName	FName	John
	LName	Doe
Type	GA	
Duty	CIS library attendant	
Supervisor	Steve Smith	

Name	FName	John
	LName	Doe
Sex	M	
Address	Street	2 Bay St
	City	Newark
	State	NJ
	Zip	07102
GPA	3.80	
StdName	FName	John
	LName	Doe
Type	GA	
Duty	CIS library attendant	
Supervisor	Steve Smith	

Figure 5.7 Three folders Doe, Assistantships and f

The join operator, which is applied to two folders, is defined in terms of the Cartesian product and select operators. Intuitively, the join of folders f_1 and f_2 based on a predicate clause P, denoted $\mathsf{f}_1 \bowtie_P \mathsf{f}_2$, is the set of frame instances in the Cartesian product of f_1 and f_2 that satisfy P. Formally,

Definition 5.5.1 Let f_1 and f_2 be two folders and let P be a predicate clause. Then $\mathsf{f}_1 \bowtie_P \mathsf{f}_2 = \sigma_P(\mathsf{f}_1 \times \mathsf{f}_2)$.

Example 5.5.1 Consider the two folders Doe and Assistantships in Figure 5.7. Then,

$$\mathsf{Doe} \bowtie_{\mathtt{StdName}=\mathtt{Name}} \mathsf{Assistantships} = \mathsf{f}. \qquad \Box$$

5.6 Class 6: Renaming Operator

An important operation in dealing with self-join [45] is *renaming*. This operator helps to avoid the ambiguity when referring to an attribute in the corresponding frame templates. The syntax and semantics of the operator are given below:

Definition 5.6.1 Let $\mathtt{A}_{r_1}$, $\mathtt{A}_{r_2}$, ..., $\mathtt{A}_{r_m}$, $\mathtt{A}_{j_1}$, $\mathtt{A}_{j_2}$, ..., and $\mathtt{A}_{j_m}$ be distinct attributes. Suppose that for each $\mathbf{F} \in < \mathsf{f} >$, $\mathtt{A}_{r_i} \not\in \Upsilon(\mathbf{F})$, $1 \leq i \leq m$. Define

$$\rho_{\mathtt{A}_{r_1},\mathtt{A}_{r_2},...,\mathtt{A}_{r_m} \leftarrow \mathtt{A}_{j_1},\mathtt{A}_{j_2},...,\mathtt{A}_{j_m}}(\mathsf{f}) = \bigcup_{\mathbf{F} \in <\mathsf{f}>} (\rho_{\mathtt{A}_{r_1},\mathtt{A}_{r_2},...,\mathtt{A}_{r_m} \leftarrow \mathtt{A}_{j_1},\mathtt{A}_{j_2},...,\mathtt{A}_{j_m}}(\mathsf{f}(\mathbf{F}))),$$

where

$$\rho_{\mathtt{A}_{r_1},...,\mathtt{A}_{r_m} \leftarrow \mathtt{A}_{j_1},...,\mathtt{A}_{j_m}}(\mathsf{f}(\mathbf{F})) = \begin{cases} \{\rho_{\mathtt{A}_{r_1},...,\mathtt{A}_{r_m} \leftarrow \mathtt{A}_{j_1},...,\mathtt{A}_{j_m}}(\mathit{fi}) \mid \mathit{fi} \in (\mathsf{f}(\mathbf{F}))\} & \text{if } \{\mathtt{A}_{j_1}, ..., \mathtt{A}_{j_m}\} \subseteq \Upsilon(\mathbf{F}) \\ \mathsf{f}(\mathbf{F}) & \text{otherwise,} \end{cases}$$

and for a given $\mathit{fi} = [\langle \mathtt{A}_1 : V_1 \rangle, \langle \mathtt{A}_2 : V_2 \rangle, ..., \langle \mathtt{A}_{j_1} : V_{j_1} \rangle, ..., \langle \mathtt{A}_{j_m} : V_{j_m} \rangle, ..., \langle \mathtt{A}_k : V_k \rangle] \in \mathsf{f}(\mathbf{F})$, $\rho_{\mathtt{A}_{r_1},...,\mathtt{A}_{r_m} \leftarrow \mathtt{A}_{j_1},...,\mathtt{A}_{j_m}}(\mathit{fi}) = [\langle \mathtt{A}_1 : V_1 \rangle, ..., \langle \mathtt{A}_{r_1} : V_{j_1} \rangle, ..., \langle \mathtt{A}_{r_m} : V_{j_m} \rangle, ..., \langle \mathtt{A}_k : V_k \rangle]$.

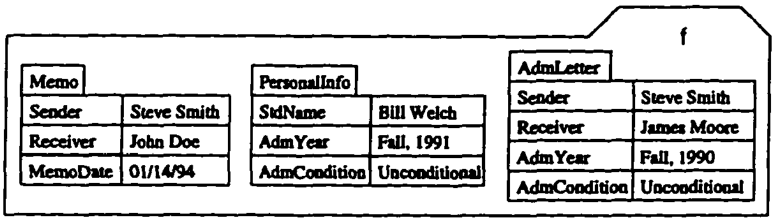
Example 5.6.1 Consider again the folder organization in Figure 5.1 and the query: *List all the PhD students who applied to take the Qualifying Examination in the same semester that Mary Jones applied.* The algebra expression is as follows:

Let $P := (\mathtt{StdName2} = [\langle \mathtt{FName} : Mary \rangle, \langle \mathtt{LName} : Jones \rangle])$.

$\mathsf{QExams2} := \sigma_P(\rho_{\{\mathtt{StdName2},\mathtt{ExamTime2} \leftarrow \mathtt{StdName},\mathtt{ExamTime}\}}(\mathsf{QExams}))$.

$\mathsf{Result} := \pi_{\mathtt{StdName}}((\mathsf{QExams}) \bowtie_{(\mathtt{ExamTime}=\mathtt{ExamTime2})}(\mathsf{QExams2}))$. $\Box$

This example illustrates the use of the *renaming* (ρ), *project* (π), and *join* ($\bowtie$) operators. We perform a join of the QExams folder with itself. The join is accomplished by first generating a folder QExams2 which is a copy of a portion of QExams containing only Mary Jones' applications where StdName is *renamed* to StdName2 and ExamTime is *renamed* to ExamTime2. Then a join operation is performed on the two folders QExams and QExams2 to find all the PhD students from QExams whose ExamTime is the same as ExamTime2 of QExams2.



Memo	
Sender	Steve Smith
Receiver	John Doe
MemoDate	01/14/94

PersonalInfo	
StdName	Bill Welch
AdmYear	Fall, 1991
AdmCondition	Unconditional

AdmLetter	
Sender	Steve Smith
Receiver	James Moore
AdmYear	Fall, 1990
AdmCondition	Unconditional

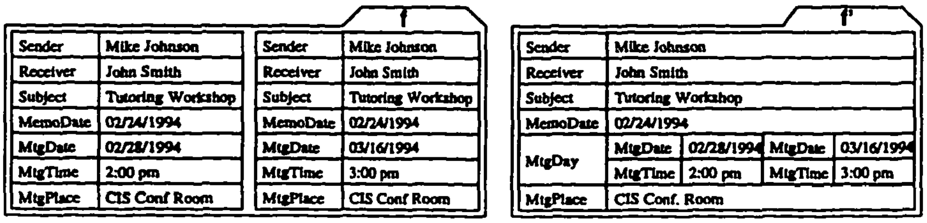
Figure 5.8 A folder f used to illustrate the renaming operator

The renaming operator also helps to get specific frame instances from a folder. For example, consider the folder f in Figure 5.8 and the query: "*List all the admission letters in the folder* f." Simply projecting the attributes on the admission letter does not work, since $\pi_{\{\mathtt{Sender},\mathtt{Receiver},\mathtt{AdmYear},\mathtt{AdmCondition}\}}(\mathsf{f})$ = {[⟨Sender: Steve Smith⟩, ⟨Receiver: James Moore⟩, ⟨AdmYear: Fall, 1990⟩, ⟨AdmCondition: Unconditional⟩], [⟨Sender: Steve Smith⟩, ⟨Receiver: John Doe⟩, ⟨AdmYear: Fall, 1991⟩, ⟨AdmCondition: Unconditional⟩]}, which produces an extraneous and incorrect frame instance [⟨Sender: Steve Smith⟩, ⟨Receiver: John Doe⟩, ⟨AdmYear: Fall, 1991⟩, ⟨AdmCondition: Unconditional⟩]. To solve this problem, we can rename the attributes on admission letters by changing Sender, Receiver, AdmYear, and AdmCondition to From, To, AdmY, and AdmCond, respectively. Then, we can get the desired result by projecting onto the renamed attributes, i.e., $\pi_{\{\mathtt{From},\mathtt{To},\mathtt{AdmY},\mathtt{AdmCond}\}}\ (\rho_{\mathtt{From},\mathtt{To},\mathtt{AdmY},\mathtt{AdmCond}\leftarrow\mathtt{Sender},\mathtt{Receiver},\mathtt{AdmYear},\mathtt{AdmCondition}}(\mathsf{f}))$ = [⟨From: Steve Smith⟩, ⟨To: James Moore⟩, ⟨AdmY: Fall, 1990⟩, ⟨AdmCond: Unconditional⟩].

The renaming operator also helps to establish some implicit relationship between two folders. Consider, for example, two folders Faculty = {[⟨ Name: Steve Smith ⟩, ⟨ Occupation: Professor ⟩, ⟨ Area: Database ⟩]} and PhDStudent = {[⟨ Name: John Doe ⟩, ⟨ Advisor: Steve Smith ⟩]}. Since Faculty × PhDStudent = ϕ, there is no way to know that Steve Smith is the advisor of John Doe. However, such a relationship can be established by applying the renaming operator. Specifically, $\rho_{\mathtt{Advisor}\leftarrow\mathtt{Name}}$(Faculty) × $\rho_{\mathtt{StdName}\leftarrow\mathtt{Name}}$(PhDstudent) = {[⟨ StdName: John Doe ⟩, ⟨ Advisor: Steve Smith ⟩, ⟨ Occupation: Professor ⟩, ⟨ Area: Database ⟩]}, which shows Steve Smith supervises John Doe.

5.7 Class 7: Restructuring Operators

Intuitively, the nest operator (ν) produces frame instances over frame templates from flatter ones (not necessary flat). Given a frame template and a subset of its attributes, it aggregates a set type that agrees on those attributes. Before giving the formal definition, we need some discussion. For simplicity, we only consider a folder containing frame instances over the same frame template. Suppose the frame template $\mathbf{F} = [\langle \mathtt{A}_1 : T_1\rangle, ..., \langle \mathtt{A}_h : T_h\rangle, \langle \mathtt{A}_{h+1} : T_{h+1}\rangle, ..., \langle \mathtt{A}_k : T_k\rangle]$ associated with the folder f, where $0 \leq h \leq k$. $\nu_{\mathtt{A}=(\mathtt{A}_{h+1},\ldots,\mathtt{A}_k)}$(f) yields a set of frame instances over the frame template $[\langle \mathtt{A}_1 : T_1\rangle, ..., \langle \mathtt{A}_h : T_h\rangle, \langle \mathtt{A} : T\rangle]$, where $T = \{[\langle \mathtt{A}_{h+1} : T_{h+1}\rangle, ..., \langle \mathtt{A}_k : T_k\rangle]\}$. As an example, consider Figure 5.9, f′ = $\nu_{\mathtt{MtgDay}=(\mathtt{MtgTime},\mathtt{MtgDate})}$(f).



Sender	Mike Johnson
Receiver	John Smith
Subject	Tutoring Workshop
MemoDate	02/24/1994
MtgDate	02/28/1994
MtgTime	2:00 pm
MtgPlace	CIS Conf Room

Sender	Mike Johnson
Receiver	John Smith
Subject	Tutoring Workshop
MemoDate	02/24/1994
MtgDate	03/16/1994
MtgTime	3:00 pm
MtgPlace	CIS Conf Room

Sender	Mike Johnson			
Receiver	John Smith			
Subject	Tutoring Workshop			
MemoDate	02/24/1994			
MtgDay	MtgDate	02/28/1994	MtgDate	03/16/1994
	MtgTime	2:00 pm	MtgTime	3:00 pm
MtgPlace	CIS Conf. Room			

Figure 5.9 An example to illustrate ν

However, consider Figure 5.10, the folder f$_2$ is obtained by applying the operator ν. That is, f$_2$ = $\nu_{\mathtt{Authors}=(\mathtt{Author})}$(f$_1$). The question is "*Can we get the folder* f$_3$ *from* f$_1$

using the nest operator ν?' The answer is negative. That is why we introduce two nest operations (ν, ν^*) as follows.

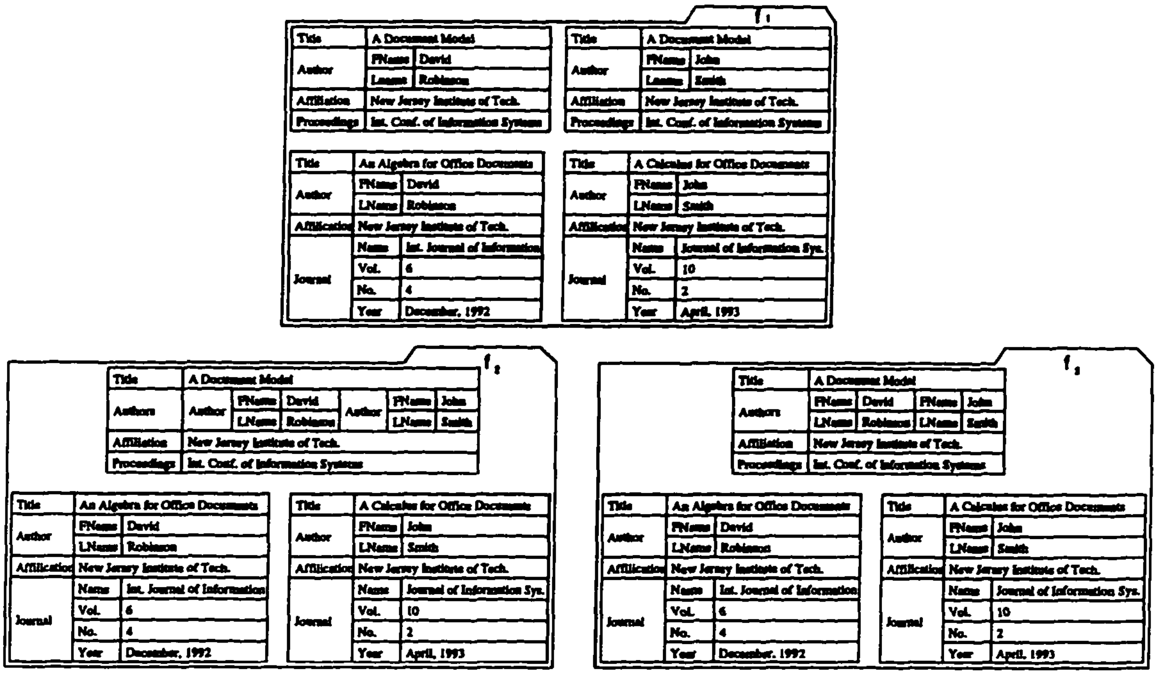


Figure 5.10 An example to illustrate the need of ν^*

Definition 5.7.1 *(Nest Operator (ν))* Let f be a folder and A be an attribute. ν is defined as follows:

$$\nu_{\mathtt{A}=\{\mathtt{A}_1,\ldots,\mathtt{A}_k\}}(\mathtt{f}) = \bigcup_{\mathbf{F}\in<\mathtt{f}>} (\nu_{\mathtt{A}=\{\mathtt{A}_1,\ldots,\mathtt{A}_k\}}(\mathtt{f}(\mathbf{F}))),$$

where

$$\nu_{\mathtt{A}=\{\mathtt{A}_1,\ldots,\mathtt{A}_k\}}(\mathtt{f}(\mathbf{F})) = \begin{cases} S & \text{if } \{\mathtt{A}_1, \ldots, \mathtt{A}_k\} \subset< \mathbf{F} > \\ \mathtt{f}(\mathbf{F}) & \text{if } \{\mathtt{A}_1, \ldots, \mathtt{A}_k\} \not\subset< \mathbf{F} > \end{cases}$$

where $\ell = < \mathbf{F} >$ - $\{\mathtt{A}_1, \ldots, \mathtt{A}_k\}$, $S = \{t \mid (\forall fi)(fi \in \mathtt{f}(\mathbf{F}) \wedge t(\ell) = fi(\ell) \wedge t[\mathtt{A}] = \pi_{\{\mathtt{A}_1,\ldots,\mathtt{A}_k\}}(\mathtt{f}(\mathbf{F}))\}$. □

Definition 5.7.2 *(Nest Operator (ν^*))* Let f be a folder and A be an attribute. ν^* is defined as follows.

$$\nu^*_{\mathtt{A}}(\mathsf{f}) = \bigcup_{\mathbf{F}\in<\mathsf{f}>} (\nu^*_{\mathtt{A}}(\mathsf{f}(\mathbf{F}))),$$

where

$$\nu^*_{\mathtt{A}}(\mathsf{f}(\mathbf{F})) = \begin{cases} S & \text{if } \{\mathtt{A}_1, ..., \mathtt{A}_k\} \subset < \mathbf{F} > \\ \mathsf{f}(\mathbf{F}) & \text{if } \{\mathtt{A}_1, ..., \mathtt{A}_k\} \not\subset < \mathbf{F} > \end{cases}$$

where $\ell = < \mathbf{F} > - \{\mathtt{A}_1, ..., \mathtt{A}_k\}$, $S = \{t \mid (\forall \mathit{fi})(\mathit{fi} \in \mathsf{f}(\mathbf{F}) \wedge t(\ell) = \mathit{fi}(\ell) \wedge t[\mathtt{A}] = \{\mathit{fi}'(\mathtt{A}_1, ..., \mathtt{A}_k) \mid (\forall \mathit{fi}')(\mathit{fi}' \in \mathsf{f}(\mathbf{F}) \wedge \mathit{fi}'(\ell) = t(\ell))\})\}$. □

The unnest operators μ and μ^* are sort of inverse of ν and ν^*, respectively.

Definition 5.7.3 *(Unnest Operator (μ))* Let f be a folder and A be an attribute. μ is defined as follows:

$$\mu_{\mathtt{A}=\{\mathtt{A}_1,...,\mathtt{A}_k\}}(\mathsf{f}) = \bigcup_{\mathbf{F}\in<\mathsf{f}>} (\mu_{\mathtt{A}=\{\mathtt{A}_1,...,\mathtt{A}_k\}}(\mathsf{f}(\mathbf{F}))),$$

where

$$\mu_{\mathtt{A}=\{\mathtt{A}_1,...,\mathtt{A}_k\}}(\mathsf{f}(\mathbf{F})) = \begin{cases} \mathsf{f}(\mathbf{F}) & \mathtt{A} \not\in < \mathbf{F} > \\ S & \text{otherwise,} \end{cases}$$

where let T be the type associated with A, and $\tilde{T} = [\langle \mathtt{A}_1 : T_1 \rangle, ..., \langle \mathtt{A}_k : T_k \rangle]$ and $\ell = < \mathbf{F} > - \{\mathtt{A}_1, ..., \mathtt{A}_k\}$,

$$S = \begin{cases} \{t \mid (\forall \mathit{fi})(\mathit{fi} \in \mathsf{f}(\mathbf{F}) \wedge t(\ell) = \mathit{fi}(\ell) \wedge t(\mathtt{A}_1, \mathtt{A}_2, ..., \mathtt{A}_k) \in \mathit{fi}[\mathtt{A}])\} & \text{if } T = \{\tilde{T}\} \\ \{t \mid (\forall \mathit{fi})(\mathit{fi} \in \mathsf{f}(\mathbf{F}) \wedge t(\ell) = \mathit{fi}(\ell) \wedge t(\mathtt{A}_1, \mathtt{A}_2, ..., \mathtt{A}_k) = \mathit{fi}[\mathtt{A}])\} & \text{if } T = \tilde{T} \end{cases}$$

□

Definition 5.7.4 *(Unnest Operator (μ^*))* Let f be a folder and A be an attributes. μ^* is defined as follows.

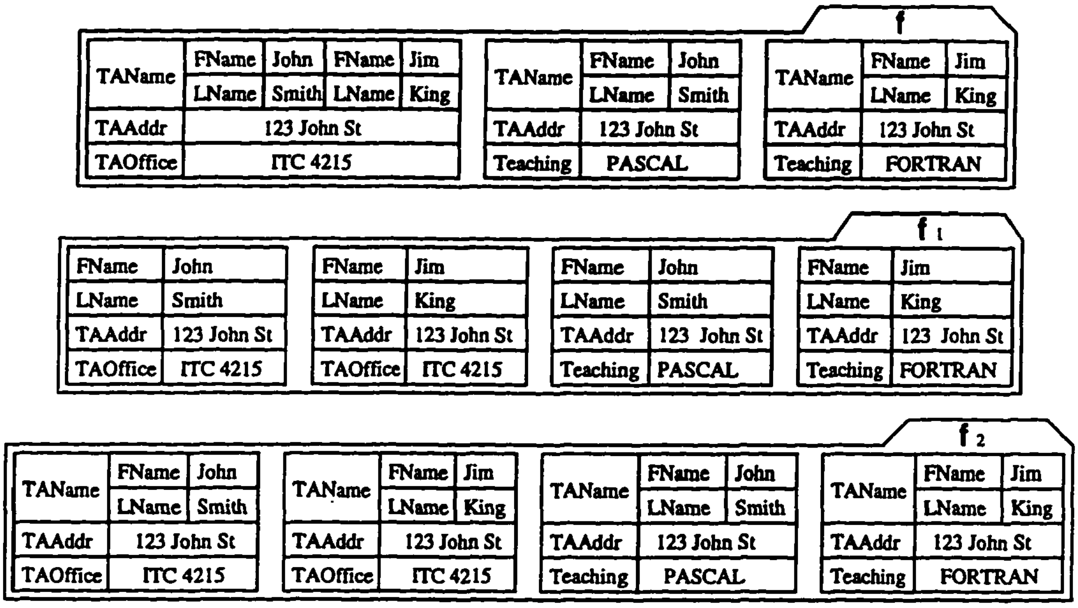
$$\mu^*_{\mathtt{A}}(\mathsf{f}) = \bigcup_{\mathbf{F}\in<\mathsf{f}>} (\mu^*_{\mathtt{A}}(\mathsf{f}(\mathbf{F})),$$

where let T be the type associated with the attribute A and $\ell = < \mathrm{F} > - \{\mathtt{A}\}$,

$$\mu_{\mathtt{A}}^{*}(\mathrm{f}(\mathrm{F})) = \begin{cases} \{t \mid (\forall fi)(fi \in \mathrm{f}(\mathrm{F}) \wedge t(\ell) = fi(\ell) \wedge t[\mathtt{A}] \in fi[\mathtt{A}]\} & \text{if } \mathtt{A} \in < \mathrm{F} > \\ & \text{and } T \text{ is a set type} \\ \mathrm{f}(\mathrm{F}) & \text{otherwise} \end{cases}$$

□

Consider Figure 5.11, $\mathrm{f}_1 = \mu_{\mathtt{TAName}=(\mathtt{FName},\mathtt{LName})}(\mathrm{f})$ and $\mathrm{f}_2 = \mu_{\mathtt{TAName}}^{*}(\mathrm{f})$.



TAName	FName	John	FName	Jim
	LName	Smith	LName	King
TAAddr	123 John St			
TAOffice	ITC 4215			

TAName	FName	John
	LName	Smith
TAAddr	123 John St	
Teaching	PASCAL	

TAName	FName	Jim
	LName	King
TAAddr	123 John St	
Teaching	FORTRAN	

FName	John
LName	Smith
TAAddr	123 John St
TAOffice	ITC 4215

FName	Jim
LName	King
TAAddr	123 John St
TAOffice	ITC 4215

FName	John
LName	Smith
TAAddr	123 John St
Teaching	PASCAL

FName	Jim
LName	King
TAAddr	123 John St
Teaching	FORTRAN

TAName	FName	John
	LName	Smith
TAAddr	123 John St	
TAOffice	ITC 4215	

TAName	FName	Jim
	LName	King
TAAddr	123 John St	
TAOffice	ITC 4215	

TAName	FName	John
	LName	Smith
TAAddr	123 John St	
Teaching	PASCAL	

TAName	FName	Jim
	LName	King
TAAddr	123 John St	
Teaching	FORTRAN	

Figure 5.11 An example to illustrate unnest operators

5.8 Class 8: Aggregate Operators

Class 8 includes five aggregate operators: count, sum, avg, min, and max. These operators take a set of frame instances (a folder) as an argument and produce a single value as a result. Their syntax and semantics are described below.

Definition 5.8.1 Let f be a folder. The syntax for an aggregate operator *op* on an attribute A is $op_{\mathtt{A}}(\mathrm{f})$, where $\mathtt{A} \in \Upsilon(\mathrm{F})$ for some $\mathrm{F} \in < \mathrm{f} >$. Let S contain frame instances in f that have the attribute A. Let $|S|$ represent the cardinality of S. (Recall that $fi[\mathtt{A}]$ represents the value V in the pair $\langle \mathtt{A} : V \rangle$ of fi.) The semantics of the five operators are given below.

1. $\mathsf{count}_{\mathtt{A}}(\mathsf{f}) = |S|$.

2. $\mathsf{sum}_{\mathtt{A}}(\mathsf{f}) = \sum_{fi \in S} fi[\mathtt{A}]$ if $|S| > 0$, and $\mathsf{sum}_{\mathtt{A}}(\mathsf{f})$ is undefined if $|S| = 0$.

3. $\mathsf{avg}_{\mathtt{A}}(\mathsf{f}) = (1/|\mathsf{S}|)\sum_{fi \in S} fi[\mathtt{A}]$ if $|S| > 0$, and $\mathsf{avg}_{\mathtt{A}}(\mathsf{f})$ is undefined if $|S| = 0$.

4. $\mathsf{max}_{\mathtt{A}}(\mathsf{f}) = max_{fi \in S} fi[\mathtt{A}]$ if $|S| > 0$, and $\mathsf{max}_{\mathtt{A}}(\mathsf{f})$ is undefined if $|S| = 0$.

5. $\mathsf{min}_{\mathtt{A}}(\mathsf{f}) = min_{fi \in S} fi[\mathtt{A}]$ if $|S| > 0$, and $\mathsf{min}_{\mathtt{A}}(\mathsf{f})$ is undefined if $|S| = 0$.

In general, one can calculate an aggregate operator independently from the rest of a query and then replace it by its value.

Example 5.8.1 Consider again the folder organization in Figure 5.1 and the query: *How many times has Samantha Adams taken the Qualifying Examination?* The algebra expression is as follows:

$\mathsf{count}_{\mathbf{PhDQEResult.Receiver}}(\sigma_{\mathbf{PhDQEResult.Receiver}=[(\mathtt{FName}:Samantha),(\mathtt{LName}:Adams)]}(\mathsf{PhDStds}))$ □

This example illustrates the use of the count aggregate operator. The number of times Samantha Adams received her own qualifying examination results is returned.

5.9 Class 9: Highlight Operator

A frame template is defined as a tuple type and its underlying types can themselves be bulk types. When this aggregation hierarchy becomes deep, path-notations may become tedious. Here we propose a new operator, called *highlight* (γ), as an alternative to navigate down the hierarchy and take the user to a desired level of aggregation from where the data items can be accessed directly.

Let $fi = [\langle \mathtt{A}_1, V_1 \rangle, \langle \mathtt{A}_2, V_2 \rangle, \ldots, \langle \mathtt{A}_i, V_i \rangle, \ldots, \langle \mathtt{A}_l, V_l \rangle]$ be a frame instance. Let β be a subset of the descendant attributes of $\mathtt{A}_i$. The *minimal cover* of β, denoted by β_{min}, is defined as a subset of β such that:

1. every element in $\beta - \beta_{min}$ is a descendant of an element in β_{min} and,
2. no element of β_{min} is a descendant of any other element in β_{min}.

The β_{min} is well-defined because there exists a unique subset that satisfies the conditions 1 and 2 above. The β-value of fi with respect to the top-level attribute $\mathtt{A}_i$, denoted by $fi_{\mathtt{A}_i}(\beta)$, is the frame instance $\{\langle \mathtt{B}_j, W_j \rangle | \mathtt{B}_j \in \beta_{min}, W_j \subseteq dom(\mathtt{B}_j)$ is the value of $\mathtt{B}_j$ in $fi[\mathtt{A}_i], 1 \leq j \leq |\beta_{min}|\}$.

Definition 5.9.1 Let f be a folder and let $\mathtt{A}$ be a top level attribute of $\mathbf{F} \in < \mathsf{f} >$. Let β contain a subset of the descendant attributes of $\mathtt{A}$. Then,

$$\gamma_{\mathtt{A}_\beta}(\mathsf{f}) = \bigcup_{\mathbf{F} \in <\mathsf{f}>} (\gamma_{\mathtt{A}_\beta}(\mathsf{f}(\mathbf{F}))),$$

where

$$\gamma_{\mathtt{A}_\beta}(\mathsf{f}(\mathbf{F})) = \begin{cases} \{fi_{\mathtt{A}}(\beta) \mid fi \in \mathsf{f}(\mathbf{F})\} & \text{if } \mathtt{A} \in < \mathbf{F} > \\ \phi & \text{otherwise.} \end{cases}$$

Example 5.9.1 Consider again the folder organization in Figure 5.1 and the query: *Display the Database question which was weighted the most during the Fall 1990 Qualifying Examination.* The algebra expression is as follows:

$\mathsf{DBF90QExams} := \pi_{\mathtt{Problems}} (\sigma_{\mathtt{Paper}=Database \wedge \mathtt{ExamTime}=[\langle \mathtt{Semester}:Fall \rangle, \langle \mathtt{Year}:1990 \rangle]}(\mathsf{QExams}))$

$x := \mathsf{max}_{\gamma_{\mathtt{Problems}_{\{\mathtt{Points}\}}}} (\mathsf{DBF90QExams})$

$\text{Result} := \gamma_{\mathtt{Problems}_{\{\mathtt{Quest}\}}} (\sigma_{\gamma_{\mathtt{Problems}_{\{\mathtt{Points}\}}}=x} (\mathsf{DBF90QExams}))$ □

The first selection operation finds the database qualifying exam paper that was given during the Fall of 1990. Then the attribute `Problems` is projected. The max operator returns the maximum value of points for a particular question of this paper. After selecting the problem which has the maximum points, project it *over* the question of the problem.

CHAPTER 6

THE CONSTRUCTION AND RECONSTRUCTION PROBLEMS

This chapter will discuss a pair of problems for a folder organization.

- Construction Problem. When a user adds a new folder with an arbitrarily specified predicate to a folder organization, it may cause two abnormalities: inapplicable edges (filing paths) and redundant folders. This is called the *construction problem.*

- Reconstruction Problem. Given a folder organization $\mathcal{FO}(G, \Delta)$, the global predicate for any folder in $\mathcal{FO}$ can be derived. But the global predicates do not, of course, uniquely specify the folder organization. However, we may ask *under what circumstance can we uniquely recover the folder organization from its global predicates?* We call this problem the *reconstruction problem.*

6.1 The Construction Problem

Initially, a folder organization $\mathcal{FO}(G, \Delta)$ has only one folder f_r (called the *rooted folder* of $\mathcal{FO}$) with the predicate δ_r = *true*, that is, f_r contains all the filed frame instances. Then, $\mathcal{FO}$ can be constructed by applying repeatly the addition operation *Add*. Let the operation $Add(\mathcal{FO}(G, \Delta), \{\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}\}, \mathsf{f})$ denote that a folder f is added into the folder organization $\mathcal{FO}$ as a child of the folders $\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}$ $(n \geq 1)$. Formally, the folder addition operation *Add* can be defined as follows.

Definition 6.1.1 *(Addition Operation (Add))* Given a folder organization $\mathcal{FO}(G, \Delta)$ with folders $\mathsf{f}_i \in V(G)$ $(i \in \{k_1, ..., k_n\})$, and a new folder f with local predicate δ, the operation $Add(\mathcal{FO}(G, \Delta), \{\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}\}, \mathsf{f}) = \mathcal{FO}(G'(V', E'), \Delta')$, where $\Delta' = \Delta \cup \{\delta\}$, $V'(G') = V(G) \cup \{\mathsf{f}\}$, and $E'(G') = E(G) \cup \{(\mathsf{f}_{k_1}, \mathsf{f}), ..., (\mathsf{f}_{k_n}, \mathsf{f})\}$. □

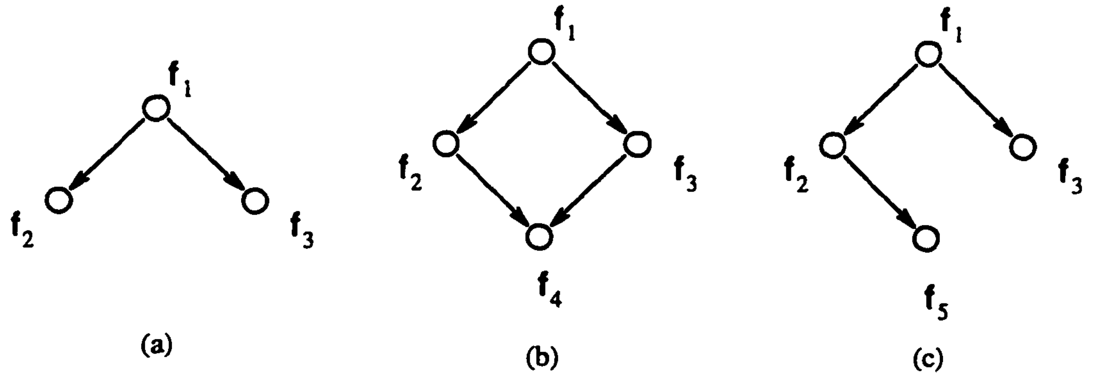


Figure 6.1 An example of inconsistent local predicates

The folder addition operation operation can be used to construct a folder organization, in which the global predication for any folder can be derived by ANDing the local predicates of the folders of a filing path. Then, *does it allow a user to add a new folder with an arbitrary local predicate?* It is called *construction problem of a folder organization.*

Example 6.1.1 In order to illustrate this construction problem, let us consider folder organizations shown in Figure 6.1. Figure 6.1(a) shows an initial folder organization $\mathcal{FO}(G(V,E),\Delta)$, where $V(G) = \{\mathrm{f}_1,\mathrm{f}_2,\mathrm{f}_3\}$, $E(G) = \{(\mathrm{f}_1,\mathrm{f}_2),(\mathrm{f}_1,\mathrm{f}_3)\}$, and $\Delta = \{\delta_1,\delta_2,\delta_3\}$. And their local predicates are:

$$\begin{cases} \delta_1 = (\texttt{Dept} = \mathrm{CIS}) \\ \delta_2 = ((\texttt{Status} = \mathrm{Faculty}) \wedge (\texttt{WorkYear} \geq 5)) \\ \delta_3 = (\texttt{Status} = \mathrm{Staff}) \end{cases}$$

Then the corresponding global predicates of folders f_1, f_2 and f_3 are $P_1 = \delta_1$, $P_2 = \delta_1 \wedge \delta_2$ and $P_3 = \delta_1 \wedge \delta_3$, respectively. Let us consider the following two cases:

- Let $\delta_4 = (\texttt{WorkYear} < 5)$ be the local predicate associated with a folder f_4. The operation $Add(\mathcal{FO}$, $\{\mathrm{f}_2,\mathrm{f}_3\}$, $\mathrm{f}_4)$ yields a folder organization shown in Figure 6.1(b). Then the predicate, $\delta_4 \wedge P_2 = ((\texttt{Dept} = \mathrm{CIS}) \wedge (\texttt{status} = \mathrm{Faculty}) \wedge (\texttt{WorkYear} \geq 5) \wedge (\texttt{WorkYear} < 5))$ is false. It means that the filing edge $(\mathsf{f}_2, \mathsf{f}_4)$ is *inapplicable* since there is no frame instance that can satisfy both P_2 (the global predicate of f_2) and $\delta_4 \wedge P_2$.

- Let δ_5 = (**Status** = Faculty) be the local predicate associated with a folder f_5. The operation *Add*$(\mathcal{FO}, \mathsf{f}_2, \mathsf{f}_5)$ yields a folder organization shown in Figure 6.1(c). Then $\delta_5 \wedge P_2$ is equivalent to P_2. That is, $\mathsf{f}_2 = \mathsf{f}_5$. The new added folder f_5 is *redundant* to the existing folder organization. □

6.2 Consistency of Predicates

When a folder with its local predicate is added into a folder organization, it may create two abnormalities: *inapplicable edges* and *redundant folders.* In order to eliminate such abnormalities, the consistency of a local predicate is defined.

Definition 6.2.1 *(Consistency of Local Predicate)* Given a folder organization $\mathcal{FO}(G(V, E), \Delta)$, P_{k_1}, ..., and P_{k_n} are global predicates associated with folders f_{k_1}, ..., and f_{k_n}, respectively. The local predicate δ of a folder f is *consistent* with respect to the folder organization $\mathcal{FO}$ if and only if none of the following conditions holds in *Add*$(\mathcal{FO}(G(V, E), \Delta)$, $\{\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}\}, \mathsf{f})$:

1. $\exists P_i \in \{P_{k_1}, ..., P_{k_n}\}$, $P_i \wedge \delta$ is false.
2. $\exists P_i \in \{P_{k_1}, ..., P_{k_n}\}$, $\delta \wedge (P_{k_1} \vee ... \vee P_{k_n})$ is logically equivalent to P_i.

Otherwise, it is *inconsistent.* □

The consistency property of the local predicate of a folder ensures that there is no redundant folder or inapplicable edge in the folder organization. Furthermore, the consistency of a global predicate can be defined based on the consistency of the corresponding local predicates.

Definition 6.2.2 *(Consistency of Global Predicate)* Given a folder organization $\mathcal{FO}(G, \Delta)$, the global predicate of a folder is *consistent* with respect to $\mathcal{FO}$ if and only if the corresponding local predicates are consistent. □

Let $\mathcal{FO}$ be a folder organization and δ be the local predicate of a newly added folder f. When the operation *Add*($\mathcal{FO}$, $\{\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}\}, \mathsf{f}$) is invoked, the following procedure can be used to determine whether δ is consistent with the existing folder organization. Let *paths*(f) denote all the possible filing paths from the root folder to the folder f.

```
for each $\mathsf{f}_i \in \{\mathsf{f}_{k_1}, ..., \mathsf{f}_{k_n}\}$ do
begin
    the global predicate $P_i$ := false;
    for each filing path $q \in paths(\mathsf{f}_i)$ do
    begin
        $p$ := true;
        for each folder $\mathsf{f} \in V(q)$ do $p := p \wedge \delta_\mathsf{f}$;
        $P_i := P_i \vee p$;
    end;
end;
for each $P_i \in \{P_{k_1}, ..., P_{k_n}\}$ do
    if $(\delta \wedge P_i)$ is false or $(\delta \wedge (P_{k_1} \vee ... \vee P_{k_n})$ is logically equivalent to $P_i$ then
        return $\delta$ is inconsistent to $\mathcal{FO}$
return $\delta$ is consistent to $\mathcal{FO}$.
```

6.3 The Associated Digraph of a Folder Organization

A folder organization views folders as either subfolders of other folders or restricted subsets of unions of other folders. We can succinctly summarize the possible inclusion relationships among folders by defining an appropriate digraph which we will call an *associated digraph.* This section introduces the concept of an associated digraph and examines some of its properties. The next section uses the associated digraph to characterize when a folder organization can be uniquely reconstructed from its predicates.

The associated digraph is defined in terms of a minimal union of folders that contain a given folder. We require the definition:

Definition 6.3.1 *(Minsum)* Let f, f_1, ..., and f_k be folders. $\mathsf{f}_1 \cup ... \cup \mathsf{f}_k$ is a *minsum* of f (denoted by $\mathsf{f} \subseteq_{min} \mathsf{f}_1 \cup ... \cup \mathsf{f}_k$) if and only if $\mathsf{f} \subseteq \mathsf{f}_1 \cup ... \cup \mathsf{f}_k$ and $\mathsf{f} \not\subseteq \mathsf{f}_{i_1} \cup ... \cup \mathsf{f}_{i_l}$, for any proper subset $\{\mathsf{f}_{i_1}, ..., \mathsf{f}_{i_l}\}$ of $\{\mathsf{f}_1, ..., \mathsf{f}_k\}$. □

Given a folder organization $\mathcal{FO}(G, \Delta)$, we define its *associated digraph* as follows.

Definition 6.3.2 *(Associated Digraph)* Let $\mathcal{FO}(G, \Delta)$ be a folder organization. The associated digraph $\tilde{G}(\tilde{V}, \tilde{E})$ (denoted by $\tilde{G}(\mathcal{FO})$) is defined as follows:

1. $\tilde{V}(\tilde{G}) = V(G)$.

2. If $\mathsf{f} \subseteq_{min} \mathsf{f}_1 \cup ... \cup \mathsf{f}_k$, then $(\mathsf{f}_i, \mathsf{f}) \in \tilde{E}(\tilde{G})$ $(1 \leq i \leq k)$. □

Clearly, the associated digraph $\tilde{G}$ of a folder organization $\mathcal{FO}(G, \Delta)$ satisfies that every vertex is reachable from the root f_{root}. Indeed, for any f ($\neq \mathsf{f}_{root}$) $\in V(G)$, $\mathsf{f} \subseteq_{min} \mathsf{f}_{root}$, whence $(\mathsf{f}_{root}, \mathsf{f}) \in \tilde{E}(\tilde{G})$. Recall the standard definition:

Definition 6.3.3 *(Transitive Closure)* Let $G(V, E)$ be a digraph. The digraph obtained from G by adding an edge (u, v) between any pair of vertices u and v in $V(G)$ whenever v is reachable from u is called the *transitive closure* of G. □

The associated digraph of a folder organization may not be the same as the transitive closure of the folder organization, as the following example shows.

Example 6.3.1 The digraph in Figure 6.2(a), and the local predicates δ_i, $1 \leq i \leq 4$, associated with the folders f_i, define a folder organization.

$$\begin{cases} \delta_1 = (\texttt{Status} = \text{Employee}) \\ \delta_2 = (\texttt{Salary} \leq \text{50K}) \wedge (\texttt{Position} = \text{Professor}) \\ \delta_3 = (\texttt{Salary} > \text{50K}) \\ \delta_4 = (\texttt{Position} = \text{Professor}) \end{cases}$$

The global predicates P_i, $1 \leq i \leq 4$, associated with the f_i are as follows.

$$\begin{cases} P_1 = \delta_1 = (\texttt{Status} = \text{Employee}) \\ P_2 = \delta_2 \wedge P_1 = (\texttt{Status} = \text{Employee}) \wedge (\texttt{Salary} \leq \text{50K}) \wedge (\texttt{Position} = \text{Professor}) \\ P_3 = \delta_3 \wedge P_1 = (\texttt{Status} = \text{Employee}) \wedge (\texttt{Salary} > \text{50K}) \\ P_4 = \delta_4 \wedge (P_2 \vee P_3) = ((\texttt{Position} = \text{Professor}) \wedge (\texttt{Status} = \text{Employee})) \wedge \\ \qquad\qquad\qquad\qquad ((\texttt{Salary} \leq \text{50K}) \vee (\texttt{Salary} > \text{50K})) \\ \qquad\qquad\qquad = (\texttt{Position} = \text{Professor}) \wedge (\texttt{Status} = \text{Employee}) \end{cases}$$

Assume that the atomic predicates ($\texttt{Salary} \leq$ 50K), ($\texttt{Salary} >$ 50K), and ($\texttt{Position} =$ Professor) are logically independent. Observe that, trivially, $\mathsf{f}_2 \subseteq_{min} \mathsf{f}_1$, $\mathsf{f}_3 \subseteq_{min} \mathsf{f}_1$, $\mathsf{f}_4 \subseteq_{min} \mathsf{f}_1$, and $\mathsf{f}_4 \subseteq_{min} \mathsf{f}_2 \cup \mathsf{f}_3$. Furthermore, $\mathsf{f}_2 \subseteq_{min} \mathsf{f}_4$ since P_2 is also a restriction of P_4: $P_2 = ((\texttt{Salary} \leq \text{50K}) \wedge P_4)$. Using these minsum relations, we obtain the associated digraph shown in Figure 6.2(b).

Figure 6.2(b) is not the transitive closure of Figure 6.2(a). For example, in Figure 6.2(a), f_2 is not reachable from f_4 while it is in Figure 6.2(b). □

Given a tree folder organization $\mathcal{FO}(G, \Delta)$, the next section shows that, under suitable restrictions, G is the only spanning tree of $\tilde{G}$ whose transitive closure equals $\tilde{G}$. In general, the spanning sub-DAGs of the associated digraph are related to the existence of equivalent, alternative folder organizations. Some spanning sub-DAGs may be equivalent to the original folder organization in the sense that they are DAGs of folder organizations that have the same global predicates as the original

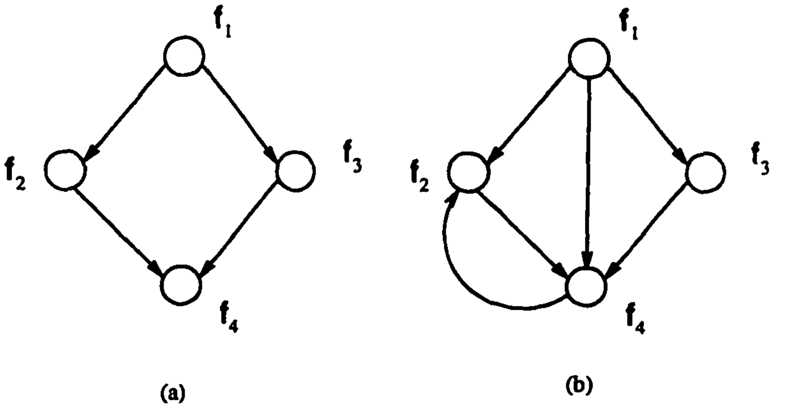


Figure 6.2 (a) A DAG folder organization $G(\mathcal{FO})$; (b) The associated digraph of $\tilde{G}(\mathcal{FO})$

folder organization. The existence of such equivalent folder organizations provides for the possibility of optimization problems. For example, one might seek an equivalent folder organization which is a tree, or which has the least maximum degree, or the minimum height, etc. However, some spanning sub-DAGs may not even correspond to the DAGs of any folder organization. We may also differentiate among spanning sub-DAGs according to whether they have redundant edges or not, defined as follows.

Definition 6.3.4 *(Reducible/Irreducible Folder Organization)* A folder organization $\mathcal{FO}(G(V, E), \Delta)$ is reducible if there exists an edge $(f_i, f_j) \in E(G)$ such that the contents of each folder in $\mathcal{FO}(G(V, E), \Delta)$ are the same as the contents of each folder in $\mathcal{FO}(G(V, E - \{(f_i, f_j)\}), \Delta)$. Otherwise, $\mathcal{FO}(G(V, E), \Delta)$ is irreducible. □

Example 6.3.2 *(Spanning Sub-DAGs of Associated Digraph)* The DAGs shown in Figure 6.3 are all the spanning sub-DAGs of the associated digraph in Figure 6.2(b). The first three sub-DAGs (a), (b) and (c) are DAGs of folder organizations with the same global predicates as the $\mathcal{FO}$ in Figure 6.2(a), though some have different local predicates. The DAGs in Figure 6.3(d) through (i) correspond to DAGs of valid folder organizations, but in each case edges can be omitted without changing the frame instances in each folder. For example, in Figure 6.3(d), (f_2, f_4) can be

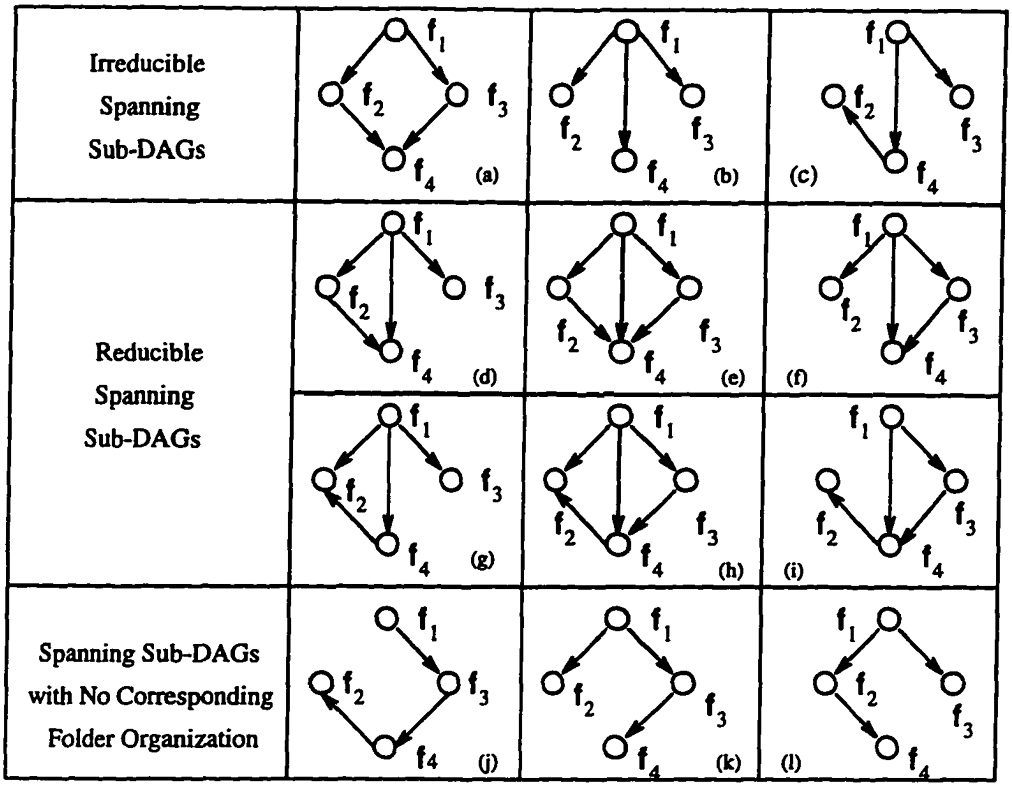

Irreducible Spanning Sub-DAGs
Reducible Spanning Sub-DAGs
Spanning Sub-DAGs with No Corresponding Folder Organization
f1
f2
f3
f4
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)

Figure 6.3 Spanning sub-DAGs of the associated digraph in Figure 8(b)

omitted. In Figure 6.3(e), (f_1, f_4) or, alternatively, both (f_2, f_4) and (f_3, f_4) can be omitted. In Figure 6.3(f), (f_3, f_4) can be omitted. Similarly there are redundant edges in Figures 6.3(g) through (i). In contrast, the DAGs in Figures 6.3(a), (b), (c) are irreducible since none of their edges can be omitted without changing the frame instances that can be in their folders. The last three DAGs (j), (k) and (l) are not DAGs of any valid folder organizations. For example, in Figure 6.3(j), the global predicate of f_4 is not a local predicate based restriction of the global predicate of f_3. Similarly for Figures 6.3(k) and (l). Therefore, these DAGs could not be DAGs of any folder organization based on the global predicates of Example 6.3.1. □

6.4 Reconstructing A Tree Folder Organization

The Reconstruction Problem asks: *under what circumstance can we uniquely recover a folder organization from its global predicates?* We shall show that the following extremely strong property is required to ensure that we can essentially recover an original tree folder organization from its global predicates, or equivalently its associated digraph.

Definition 6.4.1 *(Totally Hierarchical Property)* A DAG folder organization $\mathcal{FO}(G, \Delta)$ is totally hierarchical if and only if for every f, f_1, ..., f_k in $V(G)$, if f $\subseteq_{min}$ $\mathsf{f}_1 \cup ... \cup \mathsf{f}_k$ then f_1, ..., and f_k are ancestors of f in $G(\mathcal{FO})$. □

If $\mathcal{FO}(G, \Delta)$ is a totally hierarchical *tree* folder organization, then f $\subseteq_{min}$ $\mathsf{f}_1 \cup ... \cup \mathsf{f}_k$ implies that $k = 1$ and f_k is an ancestor of f.

The totally hierarchical property is extremely strong and can easily fail to hold. The following example shows this.

Example 6.4.1 Take the global predicates

$$\begin{cases} P_1 = (\texttt{Status} = \text{Employee}) \\ P_2 = (\texttt{Salary} \leq 50\text{K}) \wedge P_4 \\ P_3 = (\texttt{Salary} > 50\text{K}) \wedge P_1 \\ P_4 = (\texttt{Position} = \text{Professor}) \wedge P_1 \end{cases}$$

from Example 6.3.1, and the folder organization digraph shown in Figure 6.4 to specify a new folder organization $\mathcal{FO}_1$. However, P_4 is also identically equal to $\delta_4 \wedge (P_2 \vee P_3)$, thus $\mathsf{f}_4 \subseteq_{min} \mathsf{f}_2 \cup \mathsf{f}_3$. But f_2 is the child of f_4 and f_3 is the sibling of f_4 in $\mathcal{FO}_1$. Therefore, the folder organization $\mathcal{FO}_1$ in Figure 6.4 does not satisfy the totally hierarchical property. □

Even more generally, if for any folder (vertex) f $\in G(\mathcal{FO})$, $P_{\mathsf{f}} = \sum_{x \in children(\mathsf{f})} P_x$, where P_x and P_{f} are the global predicates of x and f respectively, then f $\subseteq_{min}$ $\mathsf{c}_{i_1} \cup \mathsf{c}_{i_2} \cup ... \cup \mathsf{c}_{i_n}$, where $\{\mathsf{c}_{i_1}, \mathsf{c}_{i_2}, ..., \mathsf{c}_{i_n}\}$ is some subset of the children of f. Thus, such a folder organization violates the totally hierarchical requirement that minsums occur only for unions of ancestors of f, and so such a folder organization is not totally

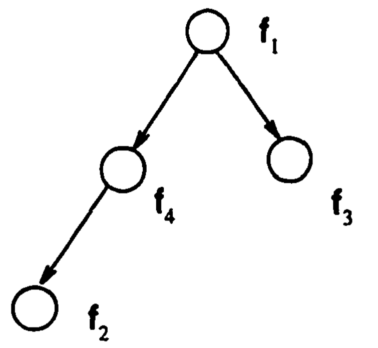


Figure 6.4 A tree folder organization for which totally hierarchical property fails

hierarchical. Since the indicated representation for P_f could easily hold, for example when a folder is contained in a union of its descendants, the totally hierarchical property is clearly very restrictive.

The totally hierarchical property does ensure that the associated digraph of a folder organization and its transitive closure are the same, as shown by the following theorem.

Theorem 6.4.1 *The associated digraph $\tilde{G}(\tilde{V}, \tilde{E})$ constructed from a totally hierarchical tree folder organization $\mathcal{FO}(G, \Delta)$ is the transitive closure of $G(\mathcal{FO})$.*

Proof: By definition, $\tilde{V}(\tilde{G}) = V(G)$. Let f_i and f_j be two folders in $\mathcal{FO}$. We consider two cases:

Case 1: If f_j is reachable from f_i in $G(V, E)$, then f_i is an ancestor of f_j, so that, $f_j \subseteq_{min} f_i$, whence $(f_i, f_j) \in \tilde{E}(\tilde{G})$.

Case 2: If f_j is not reachable from f_i in $G(V, E)$, then $(f_i, f_j) \notin \tilde{E}(\tilde{G})$. The proof is by contradiction. Observe first that if f_j is not reachable from f_i in $G(V, E)$, then f_i is not an ancestor of f_j in G. If at the same time, $(f_i, f_j) \in \tilde{E}(\tilde{G})$, then by the definition of the associated digraph, $f_j \subseteq_{min} f_i \cup \ldots$, whence, by the totally hierarchical property, f_i must be an ancestor of f_j in G, contrary to our observation.

It follows that $\tilde{G}(\tilde{V}, \tilde{E})$ is identical to the transitive closure of $G(V, E)$. □

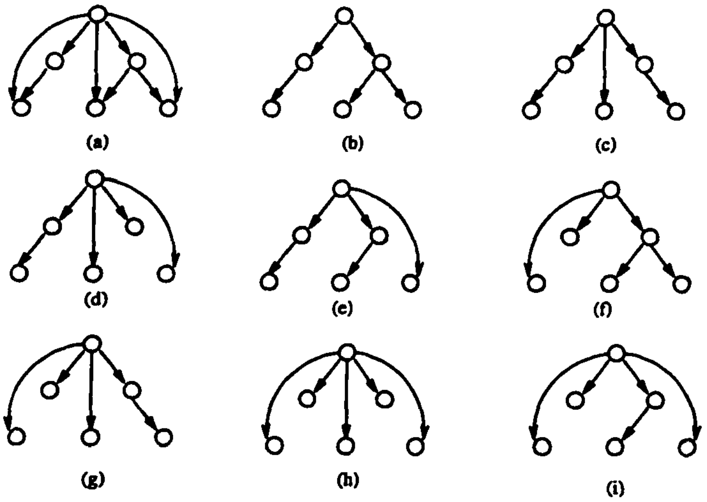


Figure 6.5 (a) A digraph. (b) ~ (i) Spanning trees of (a)

Corollary 6.4.1 *The associated digraph $\tilde{G}(\tilde{V}, \tilde{E})$ of a totally hierarchical tree folder organization $\mathcal{FO}(G, \Delta)$ is a DAG.*

Proof: Suppose $\tilde{G}$ is not a DAG. Then there would be a cycle $v_1, v_2, ..., v_k, v_1$ in $\tilde{G}$. By Theorem 6.4.1, $\tilde{G}$ is the transitive closure of G. Thus, any two vertices in $\{v_1, v_2, ..., v_k\}$ are reachable from each other in G by the definition of the transitive closure of G. This is contrary to the assumption that G is a tree. □

Consider the digraph in Figure 6.5(a). Figures 6.5(b)~(i) are all the spanning trees of Figure 6.5(a). However, Figure 6.5(b) is the only one whose transitive closure is Figure 6.5(a).

If the original tree folder organization is totally hierarchical, then we can recover this tree from the transitive closure as shown by the following theorem.

Theorem 6.4.2 *Let $\mathcal{FO} = [G(V, E), \Delta]$ be a totally hierarchical tree folder organization and let $\tilde{G}(\mathcal{FO})$ be its associated digraph. Then $G(\mathcal{FO})$ is the unique spanning tree of $\tilde{G}(\mathcal{FO})$ whose transitive closure equals $\tilde{G}(\mathcal{FO})$.*

Proof: Suppose some other spanning tree ST of $\tilde{G}(\mathcal{FO})$ also has $\tilde{G}(\mathcal{FO})$ as its transitive closure, and that $ST \neq G$. Then, there exists an edge $(u, v) \in E(G)$ such that $(u, v) \notin E(ST)$. Since the transitive closures of ST and of G are identical, there must exist a path $v_0(=u), v_1, ..., v_n, v_{n+1}(=v)$ in ST from u to v, where $v_i(1 \leq i \leq n)$ are the internal vertices of the path. Since G is a tree, the only edges in $\tilde{G}(\mathcal{FO})$, and therefore in ST, are between vertices x and y where x is an ancestor of y in G. Of course, whenever x is an ancestor of y in G, then there is path from x to y in G. Thus the path P: $v_0, v_1, ..., v_n, v_{n+1}$ in ST can be expanded into a path Q from u to v in G. We merely replace each edge (v_i, v_{i+1}) on the path P by the path from v_i to v_{i+1} in G. All these paths are disjoint because the terminal vertex of any path corresponding to any edge (v_i, v_{i+1}) on P is an ancestor of the starting vertex of the path corresponding to any later edge (v_j, v_{j+1}), where $j \geq i+1$, on P. Since there is a unique path between any pair of connected vertices in a directed tree, the path Q from u to v in G and the edge (u, v) must be the same. It follows that $v_n = u$, so that $(u, v) \in E(ST)$ contrary to the assumption that $(u, v) \notin E(ST)$. □

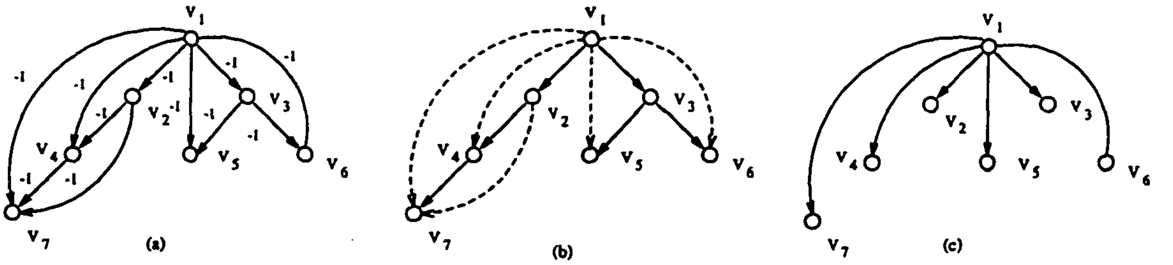


Figure 6.6 (a) The digraph $\tilde{G}$ (b) Spanning tree found by TCI algorithm (c) Spanning tree found by ordinary BFS

Theorem 6.4.2 says that given the transitive closure $\tilde{G}$ of a tree G, we can uniquely invert $\tilde{G}$ to obtain the original tree G that generated $\tilde{G}$. The following algorithm shows how this can be efficiently accomplished using a weighted breadth first search approach, where a weight of -1 is assigned to every edge of the transitive closure $\tilde{G}$. The idea of the algorithm is to identify the unique generating spanning tree established by the previous theorem by removing edges between vertices of distance -2 or less apart. An example is shown in Figure 6.6. The weight of each edge of the digraph in Figure 6.6(a) is -1. After the algorithm is applied, the solid edges remain and the dashed edges are removed, as illustrated in Figure 6.6(b). For example, $Dist(v_1, v_7) = -3$, so the edge (v_1, v_7) is removed. Observe that an ordinary unweighted breadth first search would yield the spanning tree shown in Figure 6.6(c).

Let $r \in \tilde{V}(\tilde{G})$ be the root of $\tilde{G}(\tilde{V}, \tilde{E})$. Assign each edge of $\tilde{G}$ a weight of -1. The following algorithm constructs a digraph H.

```
Transitive Closure Inversion (TCI) Algorithm
$V(H) = \{r\}$;
$E(H) = \phi$;
create(Q);
enqueue(Q,r);
while not empty(Q) do
begin
  $v :=$ dequeue(Q);
  for each vertex $v' \in \tilde{V}(\tilde{G})$ such that the shortest distance
        from $v$ to $v'$ is $-1$ do
  begin
    enqueue(Q,v');
    $V(H) := V(H) \cup \{v'\}$;
    $E(H) := E(H) \cup \{(v, v')\}$;
  end;
end;
```

The following theorem shows the correctness of the above algorithm.

Theorem 6.4.3 *(Correctness of TCI Algorithm) Let $\tilde{G}(\tilde{V}, \tilde{E})$ be the associated digraph of a totally hierarchical tree folder organization. Then the digraph $H(V, E)$ produced by the TCI algorithm is a spanning tree of $\tilde{G}(\tilde{V}, \tilde{E})$ and has $\tilde{G}$ as its transitive closure.*

Proof: The TCI algorithm must include, in the digraph $H(V, E)$ that it produces, all edges (u, v) between vertices u and v in $\tilde{G}(\tilde{V}, \tilde{E})$ when $Dist(u, v)$ equals -1. For if such an edge $(u, v) \notin E(H)$, then the transitive closure of H could not equal $\tilde{G}$. Because for the transitive closure of H to equal $\tilde{G}$, there would then have to be a nontrivial path in H from u to v, not equal to the edge (u, v). But then $Dist(u, v)$ in $\tilde{G}$ would be less than -1, contrary to the assumption. Thus edges (u, v) in $\tilde{G}$ with $Dist(u, v) = -1$ must lie in H, and the algorithm clearly includes them. Conversely, any edge (u, v) with $Dist(u, v) < -1$ should not be in H. Otherwise, there would be a nontrivial shortest path $u_0 (= u), u_1, ..., u_n, u_{n+1} (= v)$ (with more than one edge) from u to v in $\tilde{G}$. Each edge (u_i, u_{i+1}) on that path is a sub-path of that shortest path, and so is itself the shortest between its endpoints u_i and u_{i+1}. So, every edge on the path satisfies that $Dist(u_i, u_{i+1}) = -1$. But by our initial argument, the edges (u_i, u_{i+1}) must be in H. Thus, if the edge (u, v) $(= (u_0, u_{n+1}))$ were also in H, then H would not be a tree. Thus edges (u, v) such that $Dist(u, v) < -1$ should not be in H, and, of course, by design, the algorithm excludes precisely such edges. □

Corollary 6.4.2 *Let $H(V, E)$ be the spanning tree produced by the TCI algorithm from the associated digraph $\tilde{G}$ of a totally hierarchical tree folder organization $\mathcal{FO}(G, \Delta)$. Then $H = G$.*

Proof: In terms of Theorem 6.4.3, H is a spanning tree whose transitive closure is $\tilde{G}$. By Theorem 6.4.2, such a spanning tree is unique. By definition, $\tilde{G}$ is the transitive closure of G. Therefore, H is identical to G. □

6.5 Reconstructing a DAG Folder Organization

This section extends the results of the previous section to a DAG folder organization. Recall that a vertex u is an *ancestor* of a vertex v in a DAG $G(V, E)$ if and only if v is reachable from u in G, while a DAG folder organization $\mathcal{FO}(G, \Delta)$ is said to be *totally hierarchical* if and only if f $\subseteq_{min}$ f$_1$ $\cup \ldots \cup$ f$_k$ (where f, f$_1$, ..., f$_k$ are in $V(G)$) implies f$_1$, ..., and f$_k$ are ancestors of f. As the example in Figure 6.2 shows the associated digraph of a folder organization $\mathcal{FO}(G, \Delta)$ may not be the same as its transitive closure if $G(\mathcal{FO})$ is not totally hierarchical. In order to generalize the results of the previous section, we need to introduce the concept of a *redundant filing path*. Recall that a filing path for a folder f is a path from the root to f. A redundant filing path is defined as follows.

Definition 6.5.1 *(Redundant Filing Path)* Let $p_i(1 \leq i \leq m)$ be the predicates associated with the filing paths $q_i(1 \leq i \leq m)$ of a folder f in a DAG folder organization $\mathcal{FO}(G, \Delta)$ (Thus, the global predicate P_f for f satisfies: $P_f = \sum_{i=1}^{m} p_i$). Let S_i = $\{fi \mid p_i(fi)\}$, where $1 \leq i \leq m$. If $S_i \subseteq S_{i_1} \cup \ldots \cup S_{i_k}$, then the filing path for S_i is redundant with respect to the filing paths for S_{i_1}, ..., S_{i_k}. □

A DAG folder organization is *non-redundant* if there is no redundant filing path. The concept of a redundant filing path is illustrated by the following example.

Example 6.5.1 The digraph G of a DAG folder organization $\mathcal{FO}(G, \Delta)$ is shown in Figure 6.7. Denote the local predicates of the folder f$_i$ by δ_i $(1 \leq i \leq 5)$. The predicates $p_5^{(i)}$ associated with the filing paths $q_5^{(i)}$ for the folder f$_5$ are: $p_5^{(1)} = \delta_1\delta_2\delta_4\delta_5$, $p_5^{(2)} = \delta_1\delta_3\delta_4\delta_5$, and $p_5^{(3)} = \delta_1\delta_3\delta_5$. Let $S_5^{(i)} = \{fi \mid p_5^{(i)}(fi)\}$, where $1 \leq i \leq 3$. Clearly, the filing path $q_5^{(2)}$ is redundant with respect to the filing path $q_5^{(3)}$, since $S_5^{(2)} \subseteq S_5^{(3)}$, because, in this example, the local predicate product defining $q_5^{(3)}$ is a substring of the local predicate product for $q_5^{(2)}$. Observe, however, that none of the edges (f$_1$, f$_3$), (f$_3$, f$_4$) and (f$_4$, f$_5$) on the filing path $q_5^{(2)}$ can be deleted even though $q_5^{(2)}$ is redundant. For example, removing (f$_4$, f$_5$) disconnects the (possibly) non-redundant filing path $q_5^{(1)}$. Of course, it is even possible that the filing paths $q_5^{(1)}$ and $q_5^{(3)}$ are also redundant, but this depends on the local predicates and cannot be determined from the folder organization digraph alone. □

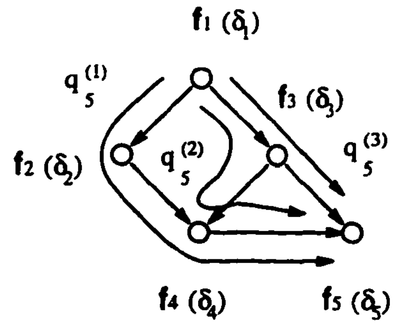


Figure 6.7 Redundant filing paths

We will show that if a DAG folder organization is both non-redundant *and* totally hierarchical, then we can essentially recover the original DAG from the associated digraph.

Theorem 6.5.1 *Let $\mathcal{FO}(G, \Delta)$ be a non-redundant and totally hierarchical DAG folder organization. Then, the associated digraph $\tilde{G}(\tilde{V}, \tilde{E})$ is the transitive closure of $G(\mathcal{FO})$.*

Proof: By definition, $\tilde{V}(\tilde{G}) = V(G)$. Let f_i and f_j be folders in $\mathcal{FO}$. We consider two cases:

Case 1: If f_j is reachable from f_i in $G(\mathcal{FO})$, then f_i is an ancestor of f_j. We show that for any ancestor f_i of f_j, $f_j \subseteq_{min} f_i \cup f_{x_1} \cup \ldots \cup f_{x_k}$, for some, possibly empty, set of folders $\{f_{x_1}, \ldots, f_{x_n}\}$, whence $(f_i, f_j) \in \tilde{E}(\tilde{G})$.

We first consider the case where there exists an edge (f, f_j), where f is the root of $G(\mathcal{FO})$. In this case, there is no other path Q from f to f_j. Otherwise, Q would be redundant with the (one edge) path (f, f_j). Thus the only ancestor f_i of f_j would be f, and so trivially $f_j \subseteq_{min} f_i (= f)$.

We next consider the case where there is no edge (f, f_j). We then argue as follows. Remove all the paths from f to f_j that pass through f_i (i.e., $Del(G, f_i)$). If f is disconnected from f_j in the resulting graph $Del(G, f_i)$, then trivially $f_j \subseteq_{min} f_i$. If f is not disconnected from f_j in $Del(G, f_i)$, then let Q be the set of paths from f to f_j, not passing through f_i. Denote by $\hat{f}_j$ the subset of frame instances in f_j that arrive via Q. By the non-redundancy assumption, $\hat{f}_j \neq f_j$, since any filing path through f_i contains some frame instances not in any union of other filing paths, and so not in the union of any filing paths in Q, and so not in $\hat{f}_j$. Observe that $\hat{f}_j \subseteq f_{x_1} \cup \ldots \cup f_{x_n}$, where $\{f_{x_1}, ..., f_{x_n}\} \subseteq V(Q) - \{f, f_j\}$, which is non-empty in this case, because by assumption f_j is reachable from f but (f, f_j) is not an edge, so there is a nontrivial disconnecting set between f and f_j. Thus, $f_j \subseteq f_i \cup f_{x_1} \cup \ldots \cup f_{x_n}$. Since $f_j \not\subseteq f_{x_1} \cup \ldots \cup f_{x_n}$ alone, then $f_j \subseteq_{min} f_i \cup f_{x_{i_1}} \cup \ldots \cup f_{x_{i_k}}$, for some subset $\{f_{x_{i_1}}, ..., f_{x_{i_k}}\}$ of $\{f_{x_1}, ..., f_{x_n}\}$.

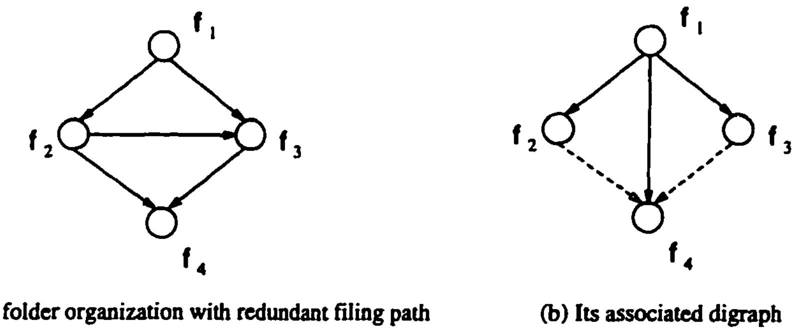
Case 2: If f_j is not reachable from f_i in $G(V, E)$, then $(f_i, f_j) \notin \tilde{E}(\tilde{G})$. The proof is identical to the proof of case 2 in Theorem 6.4.1.

It follows that $\tilde{G}(\tilde{V}, \tilde{E})$ is identical to the transitive closure of $G(\mathcal{FO})$. □

Corollary 6.5.1 *Let a DAG folder organization $\mathcal{FO}(G, \Delta)$ be non-redundant and totally hierarchical. Then, its associated digraph $\tilde{G}(\tilde{V}, \tilde{E})$ is a DAG.*

Proof: The proof is similar to the proof of Corollary 6.4.1. □

Example 6.5.2 Figure 6.8 gives a counterexample that shows how the associated digraph may not be the transitive closure of a DAG if the non-redundancy condition fails. Let $\delta_i (1 \leq i \leq 4)$ be the local predicates of the folders f_i in Figure 6.8(a). Let $S = \{fi \mid p(fi)\}$ and $S' = \{fi \mid p'(fi)\}$, where $p = \delta_1\delta_2\delta_3$ and $p' = \delta_1\delta_3$. Clearly, $S \subseteq S'$, so the filing path f_1, f_2, f_3 is redundant with respect to the filing path f_1, f_3. Thus, $f_3 \not\subseteq_{min} f_1 \cup f_2$. For an appropriate choice of δ_2, $f_3 \not\subseteq f_2$, so the edge (f_2, f_3) is not in the associated digraph. Thus the associated digraph need not even contain all the edges of the original DAG G, and so certainly need not equal the transitive closure of G. Incidentally, the minsum relations in Figure 6.8(a) are: $f_2 \subseteq_{min} f_1$, $f_3 \subseteq_{min} f_1$, $f_4 \subseteq_{min} f_1$, so edges (f_1, f_2), (f_1, f_3), and (f_1, f_4) are in the associated digraph Figure 6.8(b). On the other hand, while $f_4 \subseteq f_2 \cup f_3$, we can not say for certain that $f_4 \subseteq_{min} f_2 \cup f_3$, though at least one of the edges (f_2, f_4) or (f_3, f_4) must be



(a) An DAG folder organization with redundant filing path

(b) Its associated digraph

Figure 6.8 Counterexample to Theorem 3.6 if non redundancy condition fails.

in the associated digraph. The dashed lines in Figure 6.8(b) indicate this uncertainty. □

Theorem 6.5.2 *Let a DAG folder organization $\mathcal{FO} = [G(V, E), \Delta]$ be non-redundant and totally hierarchical and $\tilde{G}(\mathcal{FO})$ be its associated digraph. Then, $G(\mathcal{FO})$ is the unique non-redundant spanning sub-DAG of $\tilde{G}(\mathcal{FO})$ whose transitive closure equals $\tilde{G}(\mathcal{FO})$.*

Proof: Suppose some other non-redundant spanning sub-DAG SD of $\tilde{G}(\mathcal{FO})$ also has $\tilde{G}(\mathcal{FO})$ as its transitive closure, and that $SD \neq G$. Then, there exists an edge $(u, v) \in E(G)$ such that $(u, v) \notin E(SD)$. Since the transitive closure of SD and G are identical, there must exist a path $v_0(= u), v_1, ..., v_n, v_{n+1}(= v)$ in SD from u to v, where $v_i(1 \leq i \leq n)$ are the internal vertices of the path. Since $G(\mathcal{FO})$ is non-redundant, the only edges (x, y) in $\tilde{G}(\mathcal{FO})$ are between vertices x and y where x is an ancestor of y in $G(\mathcal{FO})$. Of course, whenever x is an ancestor of y in $G(\mathcal{FO})$, then there is a path from x to y in $G(\mathcal{FO})$. Thus a path $P : v_0, v_1, ..., v_n, v_{n+1}$ in SD can be expanded into a path Q from u to v in G. We merely replace each edge (v_i, v_{i+1}) on the path P by the path from v_i to v_{i+1} in $G(\mathcal{FO})$. All these paths are disjoint because the terminal vertex of any path corresponding to any edge (v_i, v_{i+1}) on P is an ancestor of the starting vertex of the path corresponding to any later edge (v_j, v_{j+1}) where $j \geq i + 1$ on P. Since there is no redundant filing path in $G(\mathcal{FO})$,

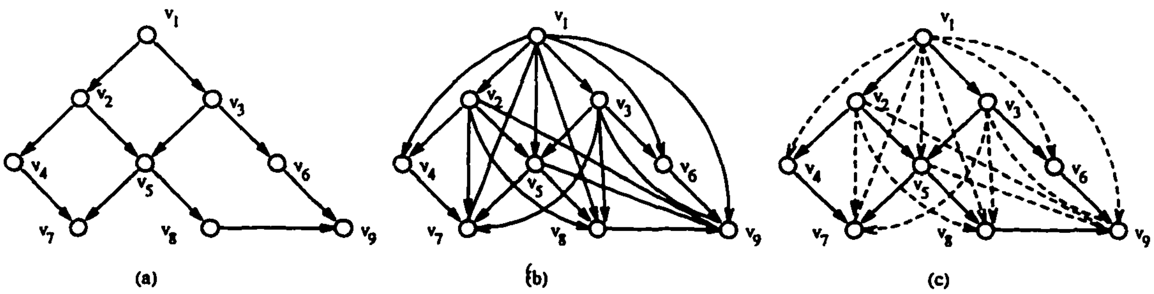


Figure 6.9 (a) A DAG $\mathcal{FO}$ (b) Its associated digraph (c) Digraph resulting from TCI algorithm

the paths P and Q from u to v must be the same. It then follows that $v_n = u$, so that $(u, v) \in E(SD)$ contrary to the assumption that $(u, v) \notin E(SD)$. □

Theorem 6.5.2 says that if an associated digraph $\tilde{G}$ is built from a non-redundant and totally hierarchical DAG folder organization $\mathcal{FO}(G, \Delta)$, we can uniquely invert $\tilde{G}$ to obtain the original DAG $G(\mathcal{FO})$. The TCI algorithm can also be used to accomplish this inversion. An example is shown in Figure 6.9. Figure 6.9(a) is a non-redundant totally hierarchical folder organization, provided there are no logical relations among the local predicates. In terms of Theorem 6.5.2, its associated digraph shown in Figure 6.9(b) is the transitive closure of Figure 6.9(a). The weight of each edge of the digraph in Figure 6.9(b) is -1. After the TCI algorithm is applied, the solid edges remain and the dashed edges are removed, as illustrated in Figure 6.9(c). Obviously, Figure 6.9(a) and Figure 6.9(c) are identical.

The following theorem shows the TCI algorithm also works for totally hierarchical and non-redundant DAG folder organizations.

Theorem 6.5.3 *Let $\tilde{G}(\tilde{V}, \tilde{E})$ be the associated digraph of a totally hierarchical and non-redundant DAG folder organization. Then the digraph $H(V, E)$ produced by the TCI algorithm is a non-redundant spanning sub-DAG of $\tilde{G}(\tilde{V}, \tilde{E})$ and has $\tilde{G}(\tilde{V}, \tilde{E})$ as its transitive closure.*

Proof: The digraph $H(V, E)$ produced by the TCI algorithm includes exactly all edges (u, v) between vertices u and v in $\tilde{G}(\tilde{V}, \tilde{E})$ where $Dist(u, v) = -1$. If there were such an edge $(u, v) \notin E(H)$, then the transitive closure of H could not equal to $\tilde{G}(\tilde{V}, \tilde{E})$. Because the transitive closure of H is $\tilde{G}(\tilde{V}, \tilde{E})$, there would be a nontrivial path in H from u to v. But then $Dist(u, v) < -1$ in $\tilde{G}(\tilde{V}, \tilde{E})$, contrary to the assumption $Dist(u, v) = -1$. Thus edges $(u, v) \in \tilde{E}(\tilde{G})$ with $Dist(u, v) = -1$ must be in H, and the algorithm clearly includes them.

On the other hand, any $(u, v) \in \tilde{E}(\tilde{G})$ with $Dist(u, v) < -1$ should not be in H. Otherwise, there would be a nontrivial shortest path $u_0 (= u), u_1, ..., u_n, u_{n+1} (= v)$ from u to v in $\tilde{G}(\tilde{V}, \tilde{E})$. Each edge (u_i, u_{i+1}) path is a sub-path of that shortest path, and so itself is the shortest between u_i and u_{i+1} (i.e., $Dist(u_i, u_{i+1}) = -1$). By the initial argument, the edges (u_i, u_{i+1}) must be in H. Thus, if the edge $(u, v)(= (u_0, u_{n+1}))$ were also in H, then H would not be a non-redundant DAG because the path $u_0, u_1, ..., u_n, u_{n+1}$ is redundant with respect to (u, v). Thus edges (u, v) such that $Dist(u, v) < -1$ should not be in H. Of course, the TCI algorithm excludes precisely such edges. □

Corollary 6.5.2 *Let $H(V, E)$ be the non-redundant spanning sub-DAG produced by TCI algorithm from the associated digraph $\tilde{G}$ of a totally hierarchical and non-redundant DAG folder organization $\mathcal{FO}(G, \Delta)$. Then $H = G$.*

Proof: The proof is similar to Corollary 6.4.2. □

CHAPTER 7

DOCUMENT FILING

A folder organization represents the user's view of the document filing organization. Evaluating whether a frame instance satisfies the global predicate of a folder in a folder organization becomes a central issue of document filing. In this chapter, we will discuss a document filing algorithm and predicate evaluation.

7.1 A Document Filing Algorithm

In TEXPROS, the document filing is a process of filing a frame instance into proper folders in a folder organization based upon a user defined predicates. The global predicate of a folder governs its contents (that is, frame instances in the folder).

For a folder f in a folder organization $\mathcal{FO}(G(V,E),\Delta)$, let p_1, ..., and p_n be the predicates corresponding to n filing paths pa_1, ..., and pa_n of f, respectively. Then, $P = p_1 \vee ... \vee p_n$ is the global predicate of f. For each filing path ps_i (say $\mathsf{f}_{i_1},...,\mathsf{f}_{i_k},\mathsf{f}$), let $\delta_{i_1},...,\delta_{i_k}$ and δ be the local predicates corresponding to the folders $\mathsf{f}_{i_1},...,\mathsf{f}_{i_k}$ and f, respectively. The predicate associated with the filing path pa_i is then $p_i = \delta_{i_1} \wedge ... \wedge \delta_{i_k} \wedge \delta$. A frame instance *fi* can be deposited in a folder f if *fi* satisfies the predicate p_i associated with the filing path pa_i $(1 \leq i \leq n)$.

A frame instance *fi* can be deposited in a folder f if *fi* satisfies the predicate, p_i (that is, the local predicates, $\delta_{i_1},...,\delta_{i_k}$, and δ) $(1 \leq i \leq n)$, associated with the filing path pa_i $(1 \leq i \leq n)$. For instance, in the folder organization of Figure 7.1, a frame instance can be deposited into the folder FACULTY if it satisfies the predicates, *Department = CIS*, *Class = Employee* and *Status = Faculty*.

7.1.1 An Object-Oriented Description of a Folder Organization

We adopt the object-oriented concept to refer to frame instances, folders and a folder organization as objects. That is, frame instances, folders and a folder organi-

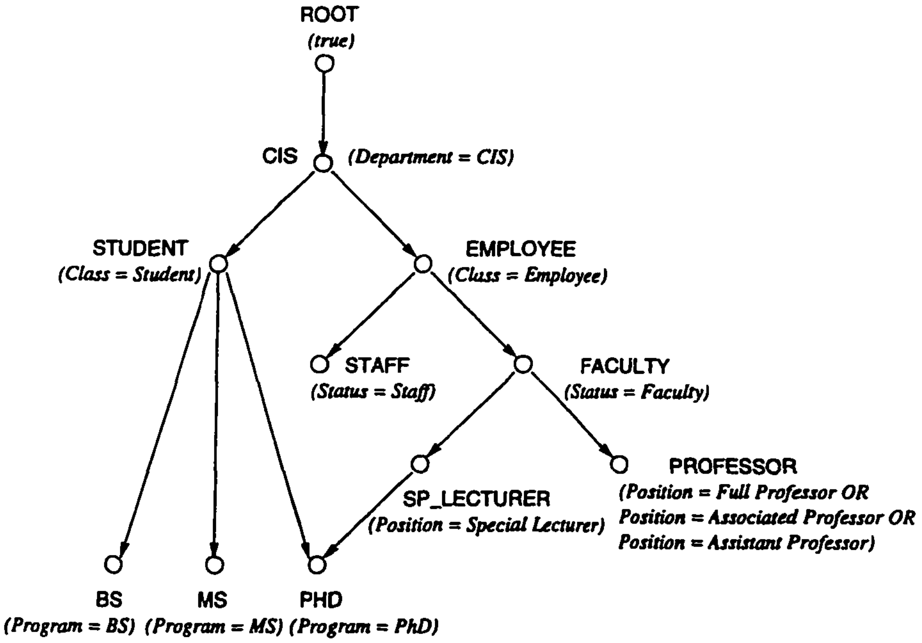


Figure 7.1 A folder organization

zation are defined by `FrameInstance` class, `Folder` class and `FolderOrganization` class, respectively. Each class contains a private data structure (*attributes*) and corresponding *methods* that can be performed on the data structure. Figure 7.2 sketches the class hierarchy of a folder organization [1]. A box in the figure represents a class. The top part of a box consists of class name, the middle part is for class attributes and the bottom part specifies methods. The relationship between the classes are the containing relationship. That is, the `FolderOrganization` class contains `Folder` class, `Thesaurus` class, `AssoDictionary` class and `KnowledgeBase` class; and `FrameInstance` class is contained in a `Folder` class.

As we discussed in the previous chapter, a folder is a heterogeneous set of frame instances. By unifying the data structure of frame instances in a folder, we use a frame instance identifier[2] rather than a frame instance itself stored in a folder.

[1] Note that methods of classes are not shown in the figure due to the size of the page. We will list and discuss methods of each class in the following sections.

[2] When a frame instance comes in the filing system, it is assigned a unique identifier.

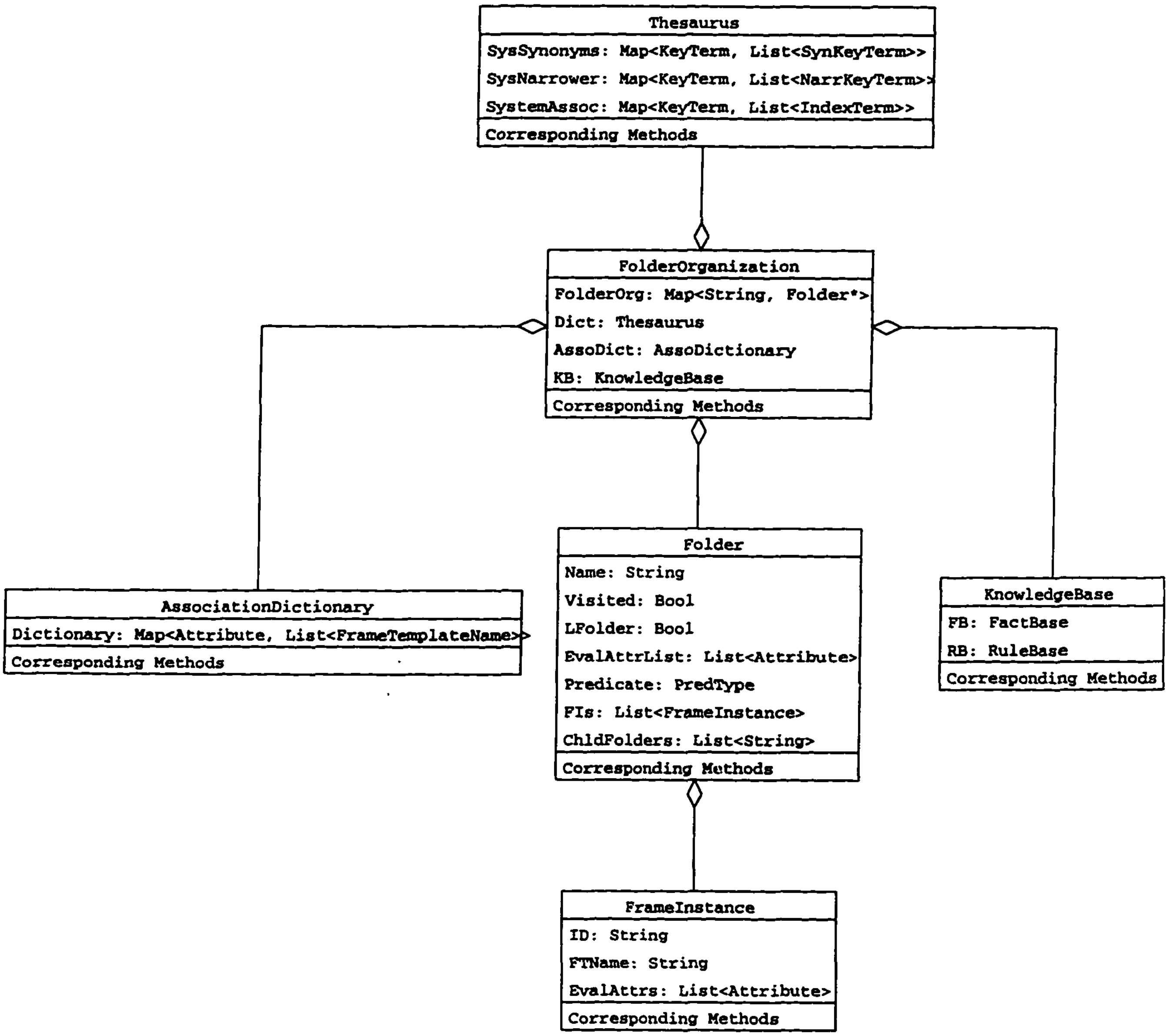


Figure 7.2 Class hierarchy of a folder organization

Note that a frame instance is stored in the instance base and can be retrieved by its identifier.

In FrameInstance class, there is an attribute FTName indicating that a frame instance is over which frame template. Each frame template consists of a list of evaluated attributes that are pre-defined for the filing evaluation. The evaluated attribute list of a frame template is defined when the frame template is constructed. By default, every attribute of a frame template is used for the filing evaluation. The following procedure describes a guideline for a user to define the evaluated attribute list.

- STEP 1: define a frame template (including attributes and types).
- STEP 2: list all the attributes of the frame template.
- STEP 3: ask a user whether he/she uses the default evaluated attribute list or defines an evaluated attribute list.
- STEP 3.1: if the default is selected, then all the attributes of the frame template are added to the evaluated attribute list.
- STEP 3.2: else a user selects attributes and adds them into the evaluated attribute list.

There are three groups of selections [3] for a user to select evaluated attributes. We will use the frame template **Memo**, which contains attributes Sender, Receiver, Date, Subject, Contents and CC, as an example to illustrate it:

- *required attributes* are attributes that must be included in the evaluation. For example, Sender and Receiver are the required attributes for the **Memo**.

[3] Note that attributes in these three groups are defined by a filing system designer/expert.

- *recommended attributes* are attributes that are most likely to be used in the evaluation. For example, Date, Subject and CC are the recommended attributes.

- *not recommended attributes* are attributes that are less likely to be used in the evaluation. For example, Contents is not a recommended attribute.

The complete frame instance class (FrameInstance) is defined as follows.

```
class FrameInstance
{
public:
    FrameInstance(); // constructor
    ~FrameInstance(); // destructor
    // set access methods
    void id(String ID); // set frame instance identifier
    void addFrameTemplate(FrameTemplateName FTName); // add frame template name
    void addEvalAttr(Attribute attr); // add attribute in evaluated attribute list
    // get access methods
    String id(); // get frame instance identifier
    FrameTemplateName getFTName(); // get frame template name
    Attribute firstAttr(); // get first evaluated attribute
    Attribute nextAttr(Attribute attr); // get next evaluated attribute
private:
    // attributes
    String ID; // frame instance identifier
    FrameTemplateName FTName; // frame template name
    List<Attribute> EvalAttrs; // evaluated attribute list
};
```

A folder contains a set of frame instances that satisfy the predicate of the folder. There are two kinds of folders defined in a folder organization: (1) a regular folder, which contains frame instances, each of which satisfies predicates along a filing path; and (2) a L-Folder which refers to a local folder, containing frame instances, each of which satisfies the local predicate of a folder.

The criterion of constructing a folder organization is that L-folders must be defined as the children of regular folders and a regular folder cannot be a child folder of a L-folder.

The reason of introducing the L-folder is to allow a user to re-partition frame instances of a regular folder into various L-folders of frame instances with their local predicates. Consider an example shown in Figure 7.3. There are six regular folders (NJIT, CIS, STUDENT, FACULTY, J. Smith and K. Johnson) and two L-folders (S. Klein* and S. Thomas*). Assume that S. Klein and S. Thomas are not faculty members of the CIS department at NJIT. Suppose that S. Klein sent a letter (let *fi* be the frame instance of the letter) to J. Smith. Since J. Smith is a faculty member of the CIS department at NJIT, the letter can be filed all the way down to the J. Smith folder if *fi*[To] is used for evaluation. If S. Klein* was a regular folder, the letter would not be deposited in it because the letter does not satisfy the predicates along the filing path (NJIT → CIS → FACULTY → J. Smith → S. Klein*). In order to file *fi* in the folder S. Klein*, *fi*[From] will be used for evaluation. However, *fi*[From] does not satisfy the global predicate P: $(\texttt{affiliation} = \mathit{NJIT}) \wedge (\texttt{department} = \mathit{CIS}) \wedge (\texttt{position} = \mathit{faculty}) \wedge (\texttt{name} = \mathit{J.\ Smith}) \wedge (\texttt{name} = \mathit{S.\ Klein})$. After introducing L-Folder, the letter can be filed into the folder S. Klein* by determining a frame instance whether satisfies the local predicate of the folder.

A regular folder also has a list of evaluated attributes. An evaluated attribute list of a regular folder is a list of attributes (of a filed frame instance) that satisfy the global predicate of the folder. The evaluated attribute list of a regular folder

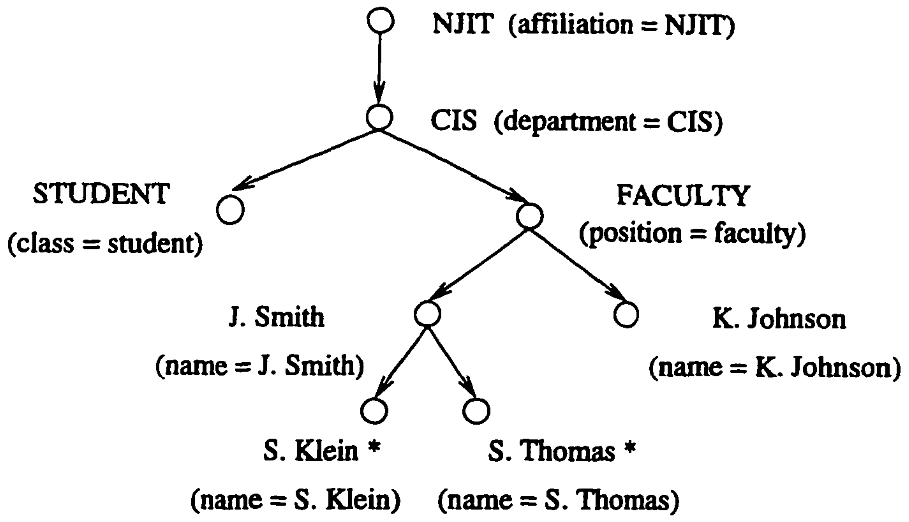


Figure 7.3 An example of a folder organization

is transient. That is, it is generated when a frame instance is filed into the folder and it is removed when another filing begins. A detailed discussion of generating the evaluated attribute list will be given in the Section 7.1.2. Since a folder organization is defined as a rooted DAG, the attribute `ChldFolders` in the class `Folder` is used to represent the folder's children folders. For filing a frame instance, a folder organization is traversed in such way that the predicate of a folder may be evaluated more than once, because a folder organization is a DAG. In the class `Folder`, an attribute (`Visited`) is used for indicating whether a folder has been visited. The complete folder class (`Folder`) is defined as follows.

```
class Folder
{
public:
   Folder(); // constructor
   ~Folder(); // destructor
   // set access methods
   void name(String folderName); // set folder name
   void resetVisited(); // reset visited flag
```

```
    void setVisited(); // set visited flag
    void setLFolder(); // set L-Folder flag
    void makeEvalAttrListEmpty(); // make evaluated attribute list empty
    void addEvalAttrs(List<Attribute> attrs); // add evaluated attribute list
    void addEvalAttr(Attribute attr); // add evaluated attribute
    void predicate(PredType* pred); // add local predicate
    void addFI(FrameInstance fi); // add frame instance
    void addChldFolder(String folderName); // add child folder
    // get access methods
    String name(); // get folder name
    Bool getVisited(); // get visited flag
    Bool getLFolder(); // get LFolder flag
    Attribute firstEvalAttr(); // get first evaluated attribute
    Attribute nextEvalAttr(Attribute attr); // get next evaluated attribute
    PredType* predicate(); // get local predicate
    FrameInstance firstFI(); // get first frame instance
    FrameInstance nextFI(FrameInstance fi); // get next frame instance
    String firstChildFolder(); // get first child folder
    String nextChildFolder(String folderName); // get next child folder
private:
    // attributes
    String Name; // folder name
    Bool Visited; // visited flag
    Bool LFolder; // L-Folder flag
    List<Attribute> EvalAttrList; // a list of evaluated attributes
    PredType Predicate; // local predicate
    List<FrameInstance> FIs; // frame instance list
    List<String> ChldFolders; // a list of children folders
};
```

In the class FolderOrganization, folders are organized in the associative array FolderOrg. The associative array FolderOrg is an array that it is looked up by strings (i.e., folder names which are keys of folders). Internally, the keys are stored in a hash table, so lookups are always very fast regardless of how many entries are in the array. Suppose that there are n folders in a folder organization, a lookup takes $O(1)$ in average case and $O(n)$ in worst case. Besides access methods, there are another three methods in the FolderOrganization. Their complete implementations will be given in the following sections.

- startFiling (public method): initialize a folder organization and start filing process.
- filing (private method): this is a recursive filing algorithm invoked by the startFiling method.
- eval (private method): this is an evaluation function that checks whether a frame instance satisfies the predicate of a folder.

```
class FolderOrganization
{
public:
   FolderOrganization(); // constructor
   ~FolderOrganization(); // destructor
   // set access method
   void addFolder(String folderName, Bool LFold,
      PredType pred, List<FrameInstance> FIs); // add folder in folder organization
   // get accessors
   Folder getFolder(String folderName); // get folder
   // behavior method
   void startFiling(FrameInstance fi); // file frame instance into folder organization
private:
    // methods
    void filing(Folder f, FrameInstance fi);
        // file frame instance into folder organization with root f
    Bool eval(FrameInstance fi, Attribute attr, PredType pred);
        // evaluation function
    // attributes
    Map<String, Folder*> FolderOrg;
    Thesaurus Dict;
    AssociationDictionary AssoDict;
    KnowledgeBase KB;
};
```

Besides the attribute FolderOrg in the class FolderOrganization, there are another three attributes:

- Dict is referred to as a thesaurus which describes synonymous relationship between attributes/values in an application domain;
- AssoDict is referred to as association dictionary which describes the association relationship between attributes appeared in predicates and in frame templates;
- KB is referred to as a knowledge base which contains facts and rules in an application domain.

These attributes will be discussed in the later sections.

7.1.2 A Filing Algorithm

There is a special folder, the root folder (ROOT), in a folder organization. The predicate of ROOT folder is true, that is, it contains all the frame instances in a folder organization. It is the root (starting point) of a folder organization and is pre-defined by the system.

The method `FolderOrganization::startFiling(FrameInstance fi)` is invoked when a frame instance *fi* arrives at the folder organization. The method resets visited flags of folders in a folder organization, initializes evaluated attribute lists to be empty, adds evaluated attribute list of *fi* to the root folder and calls the method `FolderOrganization::filing(Folder f, FrameInstace fi)` to file the frame instance. Suppose that there are n folders in a folder organization and m evaluated attributes corresponding to the filed frame instance *fi*. The complexity of the method `FolderOrganization::startFiling(FrameInstance fi)` is $O(m+n)$.

```
void
FolderOrganization::startFiling(FrameInstance fi)
{
  String fdName = FolderOrg.first();
  while (fdName != NULL)
  { // initialize folder organization
    FolderOrg[fdName]->resetVisite();
    FolderOrg[fdName]->makeEvalAttrListEmpty();
    fdName = FolderOrg.next(fdName);
  };
  Attribute attr = fi.first();
  while (attr != NULL)
  { // add evaluated attribute list of fi to ROOT folder
    FolderOrg["ROOT"]->addEvalAttr(attr);
    attr = fi.next(attr);
  };
  filing(FoderOrg["ROOT"], fi); // start filing
}
```

As mentioned in Section 7.1.1, both a frame template and a folder have an evaluated attribute lists. However, an evaluated attribute list of a frame template is *static* in the sense that it is pre-defined to determine what attributes in a frame template will be used for filing evaluation when the frame template is defined. On the other hand, an evaluated attribute list in a folder is *dynamic*. It is formed during the filing depending on a filed frame instance and the predicate of a folder. An evaluated attribute list of a folder only contains attributes of a frame template that satisfy the predicate of the folder. Figure 7.4 sketches the procedure of forming an evaluated attribute list of a folder and the complete procedure is described the method `FolderOrganization::filing()`.

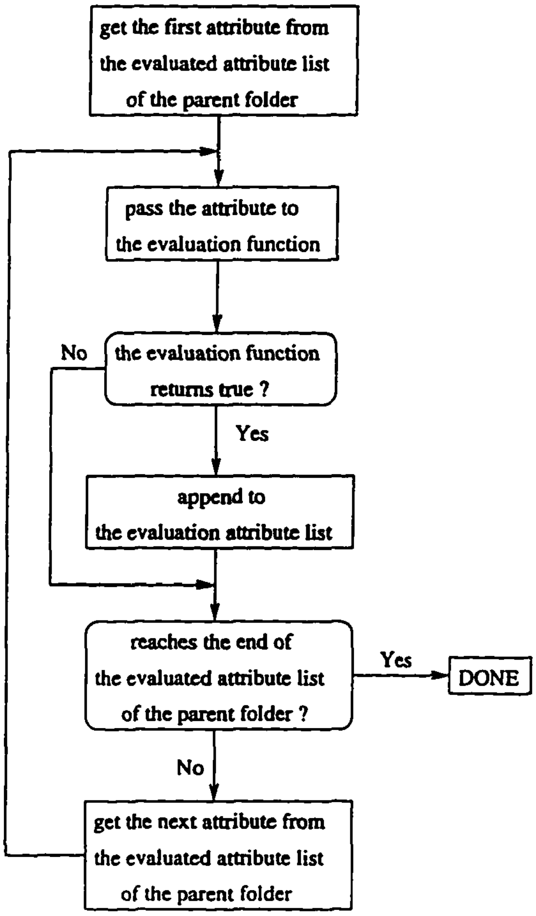


Figure 7.4 Procedure of forming an evaluated attribute list

The following recursive algorithm describes how a frame instance *fi* can be filed into a folder f and its descendant folders using depth first search approach. The idea of the algorithm is to repeatedly extend a filing path as far as possible (if a frame instance satisfies a predicate) into a folder organization, retract it, and then re-extend it in another direction, until all the directions of the folder organization are traversed.

```
void
FolderOrganization::filing(Folder f, FrameInstance fi)
{
  Folder fd;
  String chldFolder = f.firstChildFolder(); // get left most child
  f.addFI(fi); // deposit fi into f
  f.setVisited(); // set f visited
  while (chldFolder) != NULL)
  { f has child folder
    fd = getFolder(chldFolder);
    if ((fd.getVisited() == FALSE) && (fd.getLFolder == FALSE)))
    { // fd is not L-folder
      for (Attribute attr = f.firstEvalAttr();
          attr != NULL; attr = f.nextEvalAttr(attr))
      { // forming evaluated attribute list
        if (eval(fi, attr, fd.predicate()) == TRUE)
          fd.addEvalAttr(attr); // add attr to fd's evaluated attribute list
      };
      if (fd.firstEvalAttr() != NULL)
        filing(fd, fi); // recursive filing
    }   else if (fd.getVisited() == FALSE)
    { // fd is L-folder
```

```
      Bool flag = FALSE;
      for (Attribute attr = fi.firstEvalAttr();
          attr != NULL; attr = fi.nextEvalAttr(attr))
      {
        if (eval(fi, attr, fd.predicate()) == TRUE)
        {
          flag = TRUE;
          break;
        };
      };
      if (flag == TRUE)
        filing(fd, fi); // recursive filing
    };
    chldFolder = f.nextChildFolder(chldFolder); // get right sibling
  };
}
```

The following theorem shows the correctness of the above algorithm.

Theorem 7.1.1 *(Correctness of Filing Algorithm) Let $G(V, E)$ be a folder organization. Give a frame instance fi, all the folders that fi is already in their parent folders are visited and their predicates are evaluated by the filing algorithm.*

Proof: The proof is by induction. By definition, $G(V, E)$ is a rooted DAG and the predicate of the root folder is true. Thus, *fi* is deposited in the root folder. By the depth first search[37], all the child folders of the root folder will be visited and their predicates will be evaluated to see whether *fi* satisfies them. Assume that *fi* satisfies the predicate of the folder f $\in V(G)$ so that it is deposited in f[4]. and there are

[4] Note that in the filing algorithm, the evaluation function is called and returns true if a frame instance satisfies the predicate and is deposited in the folder, returns false and is

folders f_{p_1}, ..., f_{p_m} such that $(\mathsf{f}, \mathsf{f}_{p_1}) \in E(G)$, ..., $(\mathsf{f}, \mathsf{f}_{p_m}) \in E(G)$. Suppose that there is a folder f_{p_q}, where $(\mathsf{f}, \mathsf{f}_{p_q}) \in E(G)$, that it is not visited by *fi*. It is contrary to the depth first search algorithm. And so the result follows by induction. This completes the proof of the theorem. □

Suppose that there are n folders in a folder organization, and the predicate evaluation[5] takes $O(m \times k)$ (where m is number of frame templates in the association dictionary and k is the max level of root trees in the rule base) for the worst case. The filing algorithm takes then $O(m \times k \times n)$ for the worst case.

7.2 Predicate Evaluation

In the filing algorithm `FolderOrganization::filing(f, fi)`, the evaluation function `FolderOrganization::eval(fi, attr, f.predicate)` is true if the frame instance *fi* with the attribute attr satisfies the predicate f.predicate. Note that the evaluated attribute list is transparent to the evaluation function because an evaluated attribute is passed to the evaluation function by the filing procedure. The evaluation function takes the attribute, evaluates it and returns true if it satisfies the predicate, otherwise it returns false. Then, the *evaluation problem* is *how to determine whether a frame instance fi satisfies the predicate of a folder* f*?*

There are two possible cases to be considered:

- Case 1: all the attributes in a predicate appear in *fi*.
- Case 2: some attributes in a predicate do not appear in *fi*.

7.2.1 Case Study: Case 1

For the first case, the evaluation is simpler. For instance, consider the folder PHD with the predicate `Program` = *PhD*. And consider a frame instance, fi_1 =

not deposited in the folder otherwise. The correctness of the evaluation function will be discussed in Section 7.2.

[5] We will give detail discussion and an algorithm for predicate evaluation in Section 7.2.

[⟨StudentName: *Jennifer Wallace*⟩, ⟨Program: *PhD*⟩, ⟨StartDate: *09/04/94*⟩]. Since the attribute Program in the local predicate δ is appearing in the frame instance fi_1, we instantiate the attribute Program from fi_1 (i.e. fi_1[Program] = *PhD*). That is, the attribute Program in δ is replaced by the value *PhD*. Then, we conclude that fi_1 satisfies δ because *PhD* = *PhD* is true. Let us consider another frame instance fi_2 = [⟨StudentName: *John Thompson*⟩, ⟨Program: *Doctorate*⟩, ⟨StartDate: *09/04/95*⟩]. By instantiating the attribute Program from fi_2, we get fi_2[Program] = *Doctorate* which concludes that fi_2 does not satisfy the local predicate (Program = *PhD*) because *Doctorate* = *PhD* is false. However, *PhD* and *Doctorate* have the same semantical meaning.

In order to solve the above problem, a *thesaurus* is consulted. The thesaurus [32] is defined in the system catalog. It is represented by the three components *SysSynonyms*, *SysNarrower* and *SystemAssoc*. The thesaurus class (Thesaurus) is then defined as follows:

```
class Thesaurus
{
public:
  Thesaurus(); // constructor
  ~Thesaurus(); // destructor
  // set access methods
  void addKeyTerm(KeyTerm KT); // add key term
  void addSynKeyTerm(KeyTerm KT, SynKeyTerm SKT); // add synonym key term
  void addNarrKeyTerm(KeyTerm KT, NarrKeyTerm NKT); // add narrow key term
  void addIndexTerm(KeyTerm KT, IndexTerm IT); // add index term
  void delKeyTerm(KeyTerm KT); // delete key term
  void delSynKeyTerm(KeyTerm KT, SynKeyTerm SKT); // delete synonym key term
  void delNarrKeyTerm(KeyTerm KT, NarrKeyTerm NKT); // delete narrow key term
  void delIndexTerm(KeyTerm KT, IndexTerm IT); // delete index term
```

```
   // get access methods
   KeyTerm first(); // first key term in thesaurus
   KeyTerm next(KeyTerm KT); // next key term
   KeyTerm findKeyTermEntry(SynKeyTerm SKT); // find key term for given a synonym term
   KeyTerm findKeyTermEntry(NarrKeyTerm NKT); // find key term for given a narrow key term
   KeyTerm findKeyTermEntry(IndexTerm IT); // find key term for given an index term
   List<SynKeyTerm> getIndexTerms(KeyTerm KT);
     // get a list of synonym terms of a given key term
   List<NarrKeyTerm> getIndexTerms(KeyTerm KT);
     // get a list of narrow key terms of a given key term
   List<IndexTerm> getIndexTerms(KeyTerm KT);
     // get a list of index terms of a given key term
private:
   // attributes
   Map<KeyTerm, List<SynKeyTerm>> SysSynonyms; // system synonyms
   Map<KeyTerm, List<NarrKeyTerm>> SysNarrower; // system narrower
   Map<KeyTerm, List<IndexTerm>> SystemAssoc; // system association
}
```

In the filing evaluation, the system synonyms of the thesaurus are used. It contains a key term part and a synonym key term part. They are one-to-many relationship. That is, a key term may have many synonym key terms and a synonym key term refers to one and only one key term. A synonym key term can refer to a key term if they have the same meaning.

Figure 7.5 shows a portion of the system synonyms in a thesaurus. For instance, the synonym key terms *PhD* and *Doctorate* refer to the key term *PhD*. For the frame instance fi_2 = [⟨StudentName: *John Thompson*⟩, ⟨Program: *Doctorate*⟩, ⟨StartDate: *09/04/95*⟩], after consulting the thesaurus, we know that *PhD* is the key term for *Doctorate*. Then, we conclude that fi_2 satisfies the local predicate Program = *PhD*.

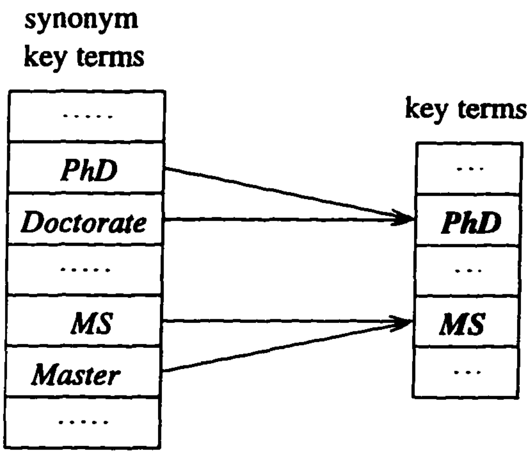


Figure 7.5 A portion of system synonyms in a thesaurus

Consider another scenario in which we have a folder (DOCTOR) with the predicate (Program = *Doctorate*) and we want to file the frame instance fi_1. By instantiating the predicate attribute (Program) and consulting the thesaurus, fi_1 still does not satisfy the predicate of folder DOCTOR. However, as we discussed above, doctorate and PhD have the same meaning. To solve the problem, we take the value (*Doctorate*) in the predicate to consult the thesaurus. Since the key term for *Doctorate* is *PhD*, the frame instance fi_1 satisfies the predicate of DOCTOR folder.

7.2.2 Case Study: Case 2

For the second case, since there are some attributes in the predicate which do not appear in the frame instance, the predicate cannot be directly instantiated by the attribute values from the frame instance. In order to solve this problem, we need (1) to establish a relationship between attributes in predicates and frame templates defined in a folder organization; (2) to have background knowledge in a certain application domain. Then, an *association dictionary* and a *knowledge base* are introduced besides a thesaurus which has been discussed in Section 7.2.1.

7.2.2.1 Association Dictionary: The association dictionary describes association relationships between attributes in predicates defined in the folder organization and

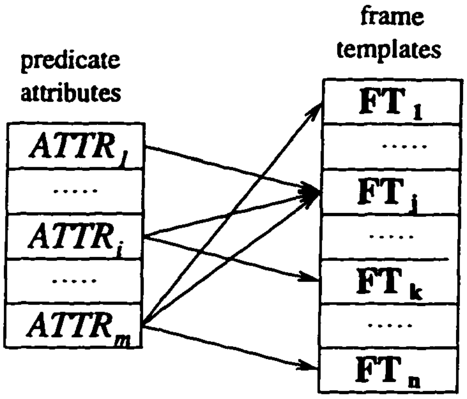


Figure 7.6 An example of an association dictionary

various frame templates. As shown in Figure 7.6, the predicate attribute $ATTR_m$, for example, is associated with frame templates $\mathbf{FT}_1$, $\mathbf{FT}_j$ and $\mathbf{FT}_n$. In the evaluation procedure for the Case 2, the association dictionary is first consulted to check whether a predicate attribute is associated with the frame template of a filed frame instance. If an association relationship is found in the dictionary, then the further evaluation will be processed. Otherwise, the evaluation will be terminated and will return false to the filing program.

In the association dictionary, predicate attributes and frame templates are many-to-many relationship. That is, a predicate attribute is associated with many frame templates and a frame template is associated with many predicate attributes. The complete description of association dictionary class (AssociationDictionary) is given below.

```
class AssociationDictionary
{ public:
  AssociationDictionary(); // constructor
  ~AssociationDictionary(); // destructor
  // set accessors
  void addAttribute(Attribute attr); // add attribute
  void addFrameTemplateName(Attribute attr, FrameTemplateName name);
```

```
        // add frame template name
    void delAttribute(Attribute attr); // delete attribute
    void delFrameTemplateName(Attribute attr, FrameTemplateName name);
        // delete frame template name
    // get accessors
    Bool findFrameTemplateName(Attribute attr, FrameTemplateName ftName);
        // check whether attribute is associated with frame template name
    List<FrameTemplateName> listFrameTemplateName(Attribute attr);
        // list frame template names associated with attribute
private:
    // attributes
    Map<Attribute, List<FrameTemplateName>> Dictionary;
}
```

7.2.2.2 Knowledge Base: The knowledge base [17] consists of two parts, a *fact base* and a *rule base*. In the fact base, each object-attribute-value triple represents the fact that an object has a property which is described by an attribute along with its value. For instance, the triple

[*Jennifer A. Wallace* **Program** *PhD*]

states that Jennifer A. Wallace is in a PhD program.

Consider a fact that James Israel is a staff of EE department and is also in the PhD program of CIS department. Such fact can be represented as

[*James Israel* **Role** [[[**Dept** *EE*] [**Status** *Staff*]] [[**Dept** *CIS*] [**Program** *PhD*]]]]

A fact with a simple value (such as *PhD*) is called a *simple fact*. A fact with a composite value (such as [[[**Dept** *EE*] [**Status** *Staff*]] [[**Dept** *CIS*] [**Program** *PhD*]]]) is called a *composite fact*. The formal description of a fact can be defined as follows:

```
⟨Fact⟩ ::= Object ⟨Attribute⟩ ⟨Value⟩
⟨Attribute⟩ ::= ⟨String⟩
⟨Value⟩ ::= ⟨SimpleValue⟩
          | ⟨CompositeValue⟩
⟨SimpleValue⟩ ::= ⟨Numeric⟩
          | ⟨String⟩
          | ⟨Numeric⟩ {, ⟨Numeric⟩}*}
          | ⟨String⟩ {, ⟨String⟩}*}
⟨CompositeValue⟩ ::= [{[[⟨Attribute⟩ ⟨SimpleValue⟩]]}+]}+]
          | [{[[⟨Attribute⟩ ⟨CompositeValue⟩]]}+]}+]
```

A rule in the rule base is of the form LHS → RHS, where (1) LHS (also called IF Condition) is a conjunction of facts, L_1, ..., L_m, which specifies the conditions of applying the rule, and (2) RHS (also called Conclusion) is either a conjunction of facts, R_1, ..., R_n, or a predicate in the folder organization, which is true only if LHS is true. For instance, the following rule represents the fact that *X* is a faculty if *X* is an assistant professor (where *X* is a variable).

[*X* **Position** *Assistant Professor*] → [*X* **Status** *Faculty*]

The rules in the rule base can be organized into a set of *AND/OR rule trees* (or abbreviated as *rule trees*). For instance, the following rules can be represented by a rule tree shown in Figure 7.7. (Notations of NOT and ¬ are used interchangeably)

$$\begin{cases} O, P \to T \\ M \to O \\ N \to O \\ A, \neg B \to P \end{cases}$$

Each attribute in a knowledge base has a set of legal values. Suppose that the attribute **Program** has the set of legal values {*BS*, *MS*, *PhD*}. If there is a fact [*James Thomas* **Program** *PhD*] in a fact base, then the following rules are true:

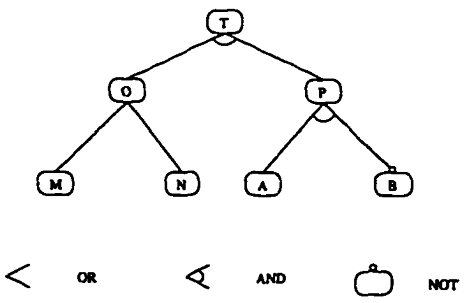


Figure 7.7 An AND/OR rule tree representing a collection of rules

[*James Thomas* **Program** *PhD*] → NOT [*James Thomas* **Program** *BS*]

[*James Thomas* **Program** *PhD*] → NOT [*James Thomas* **Program** *MS*]

Example 7.2.1 Consider the following rules, which are employed to file frame instances into the folder organization of Figure 7.1.

$$\left\{\begin{array}{l}
[X\ \textbf{Class}\ \textit{Employee}] \rightarrow [\textit{Class} = \textit{Employee}] \\
[X\ \textbf{Status}\ \textit{Staff}] \rightarrow [\textit{Status} = \textit{Staff}] \\
[X\ \textbf{Status}\ \textit{Faculty}] \rightarrow [\textit{Status} = \textit{Faculty}] \\
[X\ \textbf{Status}\ \textit{Staff}] \rightarrow [X\ \textbf{Class}\ \textit{Employee}] \\
[X\ \textbf{Status}\ \textit{Faculty}] \rightarrow [X\ \textbf{Class}\ \textit{Employee}] \\
[X\ \textbf{Position}\ \textit{Special Lecturer}] \rightarrow [X\ \textbf{Status}\ \textit{Faculty}] \\
[X\ \textbf{Position}\ \textit{Full Professor}] \rightarrow [X\ \textbf{Status}\ \textit{Faculty}] \\
[X\ \textbf{Position}\ \textit{Associate Professor}] \rightarrow [X\ \textbf{Status}\ \textit{Faculty}] \\
[X\ \textbf{Position}\ \textit{Assistant Professor}] \rightarrow [X\ \textbf{Status}\ \textit{Faculty}] \\
[X\ \textbf{Department}\ \textit{CIS}] \rightarrow [\textit{Department} = \textit{CIS}] \\
[X\ \textbf{Program}\ \textit{BS}] \rightarrow [\textit{Program} = \textit{BS}] \\
[X\ \textbf{Program}\ \textit{MS}] \rightarrow [\textit{Program} = \textit{MS}] \\
[X\ \textbf{Program}\ \textit{PhD}] \rightarrow [\textit{Program} = \textit{PhD}] \\
[X\ \textbf{Position}\ \textit{Special Lecturer}] \rightarrow [\textit{Position} = \textit{Special Lecturer}] \\
[X\ \textbf{Position}\ \textit{Assistant Professor}] \rightarrow [\textit{Position} = \textit{Assistant Professor}] \\
[X\ \textbf{Position}\ \textit{Associate Professor}] \rightarrow [\textit{Position} = \textit{Associate Professor}] \\
[X\ \textbf{Position}\ \textit{Full Professor}] \rightarrow [\textit{Position} = \textit{Full Professor}] \\
[X\ \textbf{Class}\ \textit{Student}] \rightarrow [\textit{Class} = \textit{Student}] \\
[X\ \textbf{Program}\ \textit{BS}] \rightarrow [X\ \textbf{Class}\ \textit{Student}] \\
[X\ \textbf{Program}\ \textit{MS}] \rightarrow [X\ \textbf{Class}\ \textit{Student}] \\
[X\ \textbf{Program}\ \textit{PhD}],\ \mathrm{NOT}[X\ \textbf{Position}\ \textit{Special Lecturer}] \rightarrow [X\ \textbf{Class}\ \textit{Student}]
\end{array}\right.$$

These rules can be organized into a set of rule trees as shown in Figure 7.8. The leaf nodes and the immediate nodes of a rule tree are associated with facts, and the rooted node of a tree is a predicate appeared in the folder organization. □

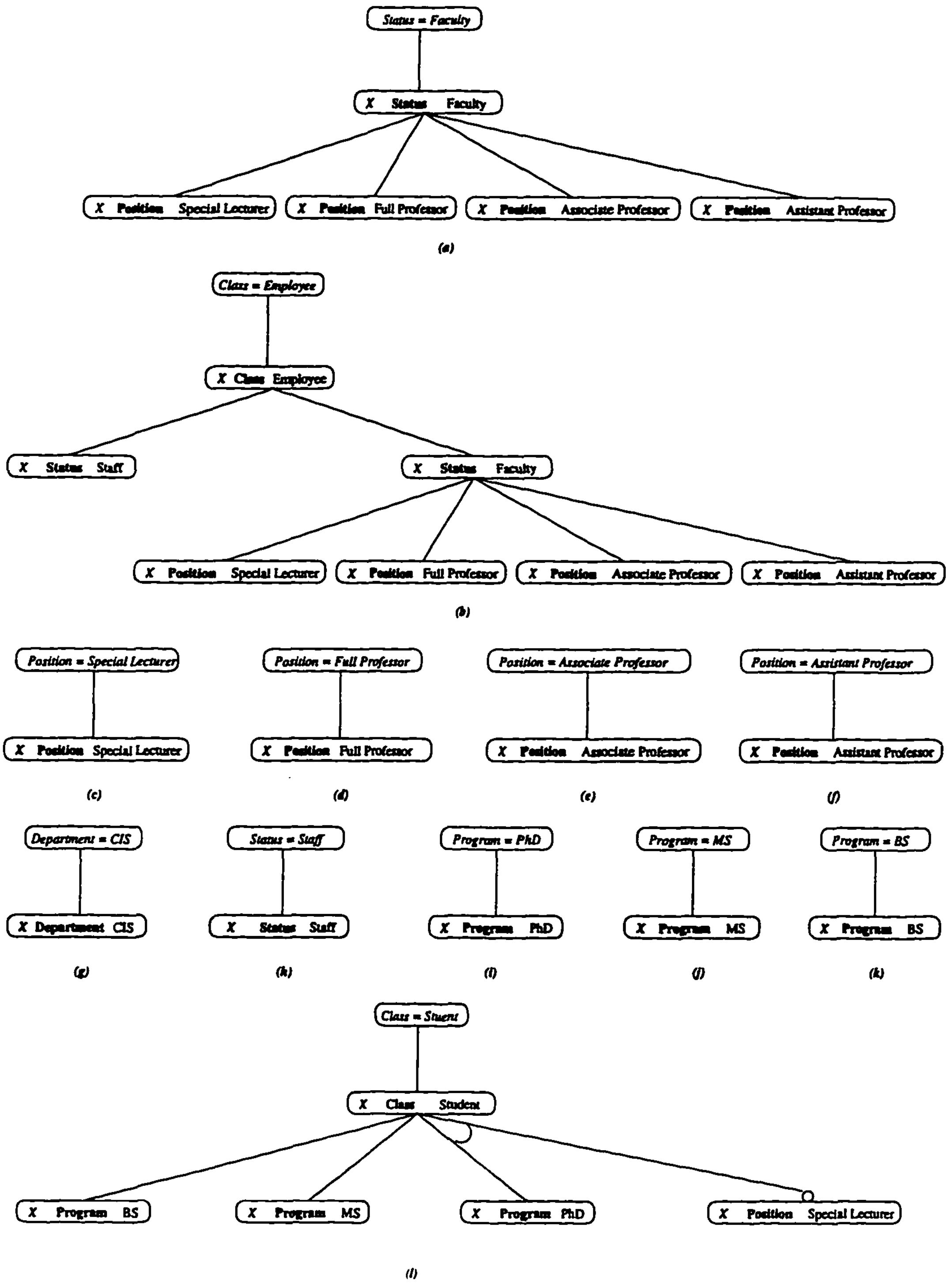

Status = Faculty
X Status Faculty
X Position Special Lecturer
X Position Full Professor
X Position Associate Professor
X Position Assistant Professor
(a)
Class = Employee
X Class Employee
X Status Staff
X Status Faculty
X Position Special Lecturer
X Position Full Professor
X Position Associate Professor
X Position Assistant Professor
(b)
Position = Special Lecturer
X Position Special Lecturer
(c)
Position = Full Professor
X Position Full Professor
(d)
Position = Associate Professor
X Position Associate Professor
(e)
Position = Assistant Professor
X Position Assistant Professor
(f)
Department = CIS
X Department CIS
(g)
Status = Staff
X Status Staff
(h)
Program = PhD
X Program PhD
(i)
Program = MS
X Program MS
(j)
Program = BS
X Program BS
(k)
Class = Stuent
X Class Student
X Program BS
X Program MS
X Program PhD
X Position Special Lecturer
(l)

Figure 7.8 An example of rule trees

Knowledge Base class (KnowledgeBase) is defined as follows.

```
class KnowledgeBase
{
public:
  KnowledgeBase(); // constructor
  ~KnowledgeBase(); // destructor
  // set accessors
  void addFact(Fact fact); // add fact
  void addRule(Rule rule); // add rule
  // get accessors
  Rule findRuleTree(PredType pred); // find rule tree with root "pred"
  // behavior method
  Bool reason(Rule rule); // goal-directed reasoning from "rule"
private:
  // attributes
  FactBase FB; // fact base
  RuleBase RB; // rule base
};
```

The backward (goal-directed) reasoning [10] is used to execute the rules in the rule base and can be described by a *recognize-reduce cycle* [1] where rules are viewed as laws by which a goal can be reduced to a number of subgoals. In our system, since the rules are organized as a set of rule trees, we can have the backward reasoning by traversing these rule trees, each from a rooted node to the leaf nodes, for determining whether a frame instance satisfies the predicates of the folders in the folder organization. For example, let the goal be *Class* = *Employee*. The inference engine selects the rule tree in Figure 7.8(b). By traversing the tree (say, breadth first search [37]), the goal *Class* = *Employee* is reduced to another goal [X Class *Employee*]. That is, if [X Class *Employee*] is true, then *Class* = *Employee* is true.

Otherwise, the goal [*X* **Class** *Employee*] is further reduced to two subgoals [*X* **Status** *Staff*] and [*X* **Status** *Faculty*]. If one of the two subgoals holds, then the goal [*X* **Class** *Employee*] holds. Therefore, the goal *Class* = *Employee* is true. If not, the system will continue going on. Since [*X* **Status** *Staff*] is a leaf of the rule tree, it can not be further reduced. On the other hand, the subgoal [*X* **Status** *Faculty*] is reduced into four sub-subgoals ([*X* **Position** *Special Lecturer*], [*X* **Position** *Full Professor*], [*X* **Position** *Associate Professor*], and [*X* **Position** *Assistant Professor*]). If one of them is true, then the goal *Class* = *Employee* is true.

The method `KnowledgeBase::reason(Rule rule)` is an implementation of goal-directed reasoning.

```
Bool
KnowledgeBase::reason(Rule rule)
{
  List<Fact> queue;
  queue.make_empty(): // empty the queue
  Fact fact = rule.leftMostChild(); get left most child of the root
  queue.enqueue(fact); // add the fact to the queue
  while ((fact = rule.rightSibling(fact)) != NULL)
  { // add children of the root to the queue
    queue.enqueue(fact);
  };
  while ((fact = queue.dequeue()) != NULL)
  {
    if (FB.findFact(fact) == TRUE) return TRUE; // fact is found in fact base
    if (rule.leftMostChild(fact) != NULL)
    {
      while ((fact = rule.rightSibling(fact)) != NULL)
      {
        queue.enqueue(fact);
      };
    };
  };
  return FALSE;
}
```

7.2.2.3 Evaluation Function: The following method `FolderOrganization::eval(FrameInstance fi, Attribute B, PredType "A` θ `V")` describes the evaluation procedure for determining whether a frame instance *fi* with an attribute B satisfies an atomic predicate A θ V.

```
Bool
FolderOrganization::eval(FrameInstance fi, Attribute B, PredType "A $\theta$ V")
{
  if (A == B)
  { // Case 1
    if (fi[B] $\theta$ V) return TRUE;
    for (each token value v in V)
      $V_{KT}$.append(Dict.getKeyTerm(v));
    if (Dict.getKeyTerm(fi[B]) $\theta$ $V_{KT}$) return TRUE;
    return FALSE;
  }
  else
  { // Case 2
    if (Dict.getKeyTerm(A) == Dict.getKeyTerm(B))
    {
      if (fi[Dict.getKeyTerm(B)] $\theta$ V) return TRUE;
      for (each token value v in V)
```

```
          V_KT.append(Dict.getKeyTerm(v));
        if (Dict.getKeyTerm(fi[Dict.getKeyTerm(B)]) θ V_KT) return TRUE;
        return FALSE;
     }
     else
     {
       if (AssoDict.findFrameTemplateName(A, fi.getFTName()) == TRUE)
       { // check if A associates with frame template of fi
         Rule rule = KB.findRuleTree("A θ V");
         if (rule != NULL)
         {
           if (KB.reason(rule) == TRUE) return TRUE;
           return FALSE;
         };
         rule = KB.findRuleTree("Dict.getKeyTerm(A) θ V_KT");
         if (rule != NULL)
         {
           if (KB.reason(rule) == TRUE) return TRUE;
           return FALSE;
         };
      };
      return FALSE;
    };
  };
}
```

The following theorem shows the correctness of the above evaluation function.

Theorem 7.2.1 *(Correctness of the evaluation function) Given an atomic predicate* A θ V *(where* A *is an attribute,* V *is a value and theta is a comparison operator), the* *above evaluation function determines whether a frame instance fi with an attribute* B *satisfies* A θ V.

Proof: We will establish the correctness of the evaluation function by considering the following cases:

- Case 1: the attribute A is an attribute in *fi*. That is, A and B are the same. For this case, the function checks whether *fi*[B] θ V holds. If yes, it returns true. Otherwise, it consults the thesaurus to get corresponding key terms *fi*$[\mathtt{B}]_{\mathtt{KT}}$ and $\mathtt{V}_{\mathtt{KT}}$ for *fi*[B] and V, respectively. If *fi*$[\mathtt{B}]_{\mathtt{KT}}$ θ $\mathtt{V}_{\mathtt{KT}}$ holds, it returns true and returns false otherwise.
- Case 2: the attribute A is not an attribute in *fi*. There are two sub-cases to be considered:
  - Sub-case 2.1: A and B refer to the same key term in the thesaurus. For this case, the proof is similar to the Case 1.
  - Sub-case 2.2: A and B refer to different key terms in the thesaurus. For this case, it checks association dictionary to see if the attribute A associates with the frame template of the frame instance *fi*. If there is no such association relationship, the evaluation function returns false. Otherwise, the evaluation function searches rule trees as defined in Section 7.2.2.2. If there is a rule tree with root fi[B] θ V or *fi*$[\mathtt{B}]_{\mathtt{KT}}$ θ $\mathtt{V}_{\mathtt{KT}}$, then the evaluation function traverses the rule tree using breadth first search. If a subgoal holds, it returns true. Otherwise, it returns false. □

Suppose that there are k frame templates in the association dictionary and the deepest levels of rule tree in the rule base is m. The worst case of the evaluation function takes $O(k \times m)$.

A local predicate δ in the form of boolean expression can be transformed into a disjunctive normal form. That is, $\delta = L_1 \vee ... \vee L_k$, where L_i $(1 \leq i \leq k)$ is the conjunction of atomic predicates (i.e., $L_i = a_{i_1} \wedge ... \wedge a_{i_m}$, a_{i_j} $(1 \leq j \leq m)$ is an atomic predicate). The following method can be used to convert a predicate into a disjunctive normal form.

```
void
Folder::transform(PredType $\delta$)
{
  while (there is a negation sign not immediately before atomic predicate)
  {
    if (there is a form $\neg(\neg S)$ in $\delta$)
    {
      $\neg(\neg S) = S$;
    };
    if (there is a form $\neg(S \wedge T)$ in $\delta$)
    {
      $\neg(S \wedge T) = \neg S \vee \neg T$;
    };
    if (there is a form $\neg(S \vee T)$ in $\delta$)
    {
      $\neg(S \vee T) = \neg S \wedge \neg T$;
    };
  };
  while (there is a form $R \wedge (S \vee T)$ in $\delta$)
  {
    $R \wedge (S \vee T) = (R \wedge S) \vee (R \wedge T)$;
  };
}
```

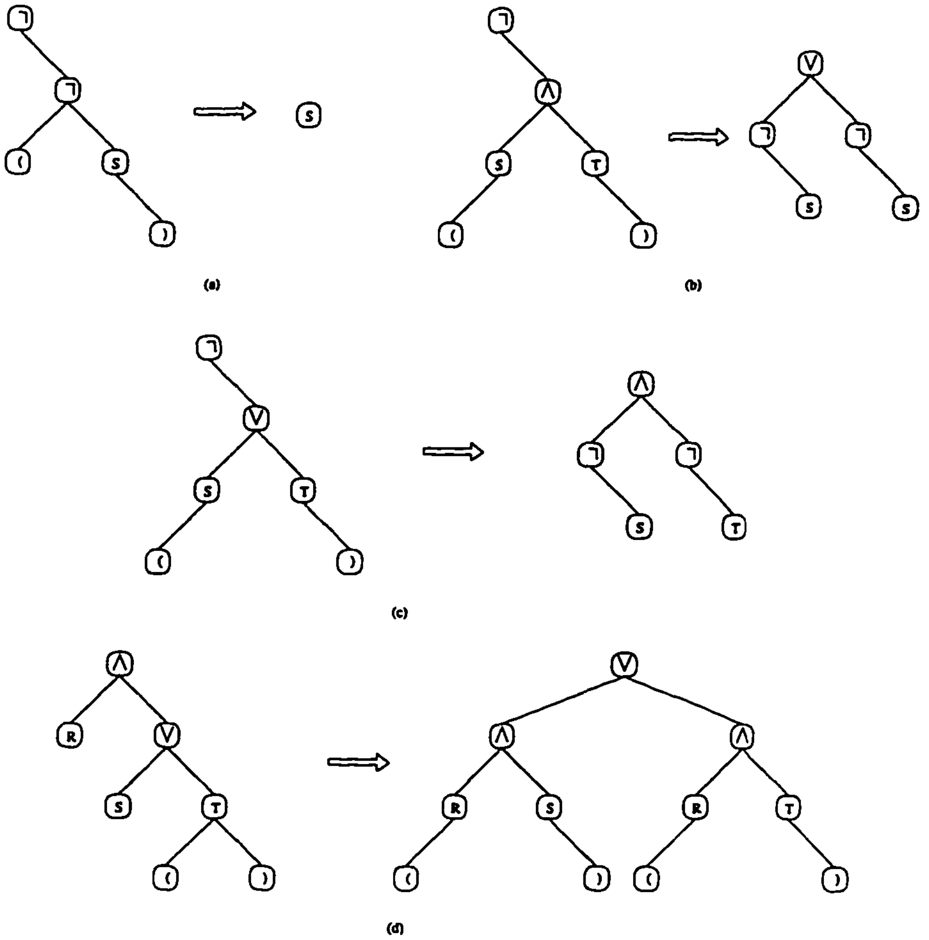


Figure 7.9 Convert a predicate to a disjunctive normal form

Internally, a predicate is represented as an expression tree. Then, tree pattern matching mechanism can be used to recognize the forms $\neg(\neg S)$, $\neg(S \wedge T)$, $\neg(S \vee T)$, and $R\wedge(S\vee T)$. Figure 7.9 shows the tree representation of these forms. For example, if the pattern (in the left side of Figure 7.9(c)) is recognized in an expression tree, then it is replaced by the right side of Figure 7.9(c). That is, $\neg(S \wedge T)$ is converted to be $\neg S \vee \neg T$.

If a frame instance satisfies L_i ($\exists i \in \{1, ..., k\}$), then it satisfies δ. To determine whether a frame instance satisfies L_i, the system evaluates whether it satisfies each of atomic predicates, $a_{i_1}, ..., a_{i_m}$ using the function `FolderOrganization::eval()`.

CHAPTER 8

CONCLUDING REMARKS

This chapter will conclude the work of this dissertation and give potential research directions. Generally, the major contributions of this dissertation include (1) the extension of an existing document model and an algebraic query language, (2) the reconstruction of folder organizations, and (3) the automation of document filing.

8.1 Document Models and Algebraic Query Languages

Previously, a folder organization was defined in terms of depend-on relationship [38], an inclusion relation. That is, a folder f_i depends-on a folder f_j if and only if $f_i \subseteq f_j$. Based upon this definition, a folder organization is a tree structure. This dissertation extends the folder organization from a tree structure into the rooted DAG structure, which represents explicitly document filing directions. There are three kinds of depend-on relationships: totally depend-on, partially depend-on and independent-of. These relationships are complete and mutually exclusive in the sense that for any pair of folders in a folder organization, one and only one of the relationships holds.

The algebraic query language (called $\mathcal{D}$_algebra) defined in [38] only handles homogeneous folders. Whereas, in the reality, a folder can be a heterogeneous set of frame instances. By observing this limitation, this dissertation extends the $\mathcal{D}$_algebra operations to support folders of heterogeneous frame instances. The deposit of frame instances in the folders of a folder organization is governed by the constraints specified for each folder. The constraints are specified terms of predicates.

Although many of the operators in the $\mathcal{D}$_algebra correspond to operators in the relational algebra [36], there is one major difference: the $\mathcal{D}$_algebra operators can manipulate heterogeneous sets (i.e., folders containing frame instances of different types), whereas the relational algebra operators only deal with homogeneous sets (i.e.,

Table 8.1 Differences between $\mathcal{D}$_model and relational models

Components	$\mathcal{D}$_Model	Relational Model
Tuples and sets of tuples (i.e. tables)	√	√
Frame templates and recursively defined bulk data types	√	×
Document type hierarchy and *is-a* relationship between frame templates	√	×
Predicate-based folders containing frame instances of different types	√	×
Folder organization with depends-on relationship between folders	√	×
Path notation and highlight operator	√	×
Algebraic operators for manipulating homogeneous sets	√	√
Algebraic operators for manipulating heterogeneous sets	√	×
Normalization and functional dependencies	×	√
Keys and foreign keys	×	√
Referential integrity	×	√

tables containing tuples of the same type). We have defined a subset of the $\mathcal{D}$_algebra and proved that the subset is at least as expressive as the relational algebra [40]. Table 8.1 summarizes the key differences between the $\mathcal{D}$_model and the relational model, where "√" indicates that the component exists in the corresponding model and "×" indicates that the component does not exist in the corresponding model. Note that since the $\mathcal{D}$_algebraic operators are all defined on heterogeneous sets, as opposed to the homogeneous sets handled by the relational algebraic operators, their semantics are entirely different from those in the relational algebra.

The nest and unnest operators (ν, μ) are first introduced in algebra for NF^2 relational data model. Jaeschke and Schek [28] proposed these two operators only applied to nesting over single attributes defined over atomic attributes. Fischer and Thomas [14] extended this to multiple attributes and multiple level of nesting. However, as we discussed in the previous chapter, if we simply extend the restructuring operators in NF^2 algebra into our document model, it does not fully support our document model. The reason is as follows. In NF^2 relational data model, a database schema is a collection of rules of the form $R_j = (R_{j_1}, R_{j_2}, ..., R_{j_n})$, where R_j and R_{j_i} $(1 \leq i \leq n)$ are relation names. Using this kind of rule, it can not generate a set of relations. That is, nest and unnest operators in NF^2 relational algebra can not express a set of relations. Recall that, in our document model, there is such rule $T = \{\tilde{T}\}$ to generate set of sets since $\tilde{T}$ can be any type. In this sense, our document algebra is more powerful than NF^2 relational algebra.

8.2 Reconstruction of Folder Organizations

Folder Organizations are defined in terms of directed graph. Each node is associated with a folder. For each folder, there is a constraint specifying what should be contained in it. These constraints are specified in terms of *local predicates* and *global predicates.* A user only specifies local predicates for the folders and the global predicates of the folders are derived by ANDing the local predicates of folders of a filing path of the underlying graph of a folder organization. The global predicate of a folder determines the contents of a folder. A *Reconstruction Problem* for folder organizations is then formulated, viz., under what circumstances it is possible to reconstruct a folder organization from its folder level global predicates. The Reconstruction Problem is solved in terms of such graph-theoretic concepts as associated digraphs, transitive closure, and redundant/non-redundant filing paths. A transitive closure inversion algorithm is presented which efficiently recovers a folder organization digraph from its associated digraph. The reconstruction result is as follows. Given a set of folders with their global predicates, we can construct the associated digraph $\tilde{G}(\mathcal{FO})$ of a folder organization $\mathcal{FO}(G, \Delta)$. If $\mathcal{FO}(G, \Delta)$ is a totally hierarchical tree folder organization, then the underlying digraph G of $\mathcal{FO}(G, \Delta)$ is the only spanning tree of $\tilde{G}(\mathcal{FO})$ whose transitive closure is equal to the associated digraph $\tilde{G}(\mathcal{FO})$. If $\mathcal{F}(G, \Delta)$ is a totally hierarchical and non-redundant DAG folder organization, the underlying digraph G is then non-redundant spanning sub-DAG of $\tilde{G}(\mathcal{FO})$ whose transitive closure is also equal to the associated digraph $\tilde{G}(\mathcal{FO})$. Therefore, we can use the Transitive Closure Inversion algorithm to reconstruct the unique folder organization digraph G from its associated digraph $\tilde{G}$.

8.3 Automation of Document Filing

A folder organization represents a users' real world document filing system. For the existing document filing systems [8, 9, 44, 49], they use the same filing criteria (*type-driven*) to organize documents according to their types. That is, homogeneous documents (of the same document types) are grouped together. This dissertation provides a heterogeneous environment of organizing documents using *predicate-driven* filing criteria. Heterogeneous documents (of different document types) can be grouped into a folder if they satisfy the predicate of the folder. Table 8.2 summarizes their differences. In the real office environments, office workers organize their documents in terms of various criteria rather than simply by document types. For example, a department chairperson wants to create folders for individual faculty members. Each folder may contain many document types, such as **Faculty Position Application, University Transcript, Memo, Publication, Vita,** etc. The type-driven filing approach fails to support such office environment. However, as discussed in the previous chapters, the predicate-driven approach can support such environment by defining a proper predicate for a folder. On the other hand, the type-driven filing approach is only a a special case of the predicate-driven filing approach (if a predicate is defined as `frame-template` = *a-document-type*, for example), or we can use the document type hierarchy in TEXPROS document model to mimic the type-driven approach of organizing and filing documents.

Table 8.2 Differences between TEXPROS Document Filing and Other Systems

System	Filing Model	Filing Criteria	Organization	Country	Year
MINOS	Object-Oriented	Type-Driven	U. of Waterloo	Canada	1986
MULTOS	Object-Oriented	Type-Driven	CNR	Italy	1988
AIM	Nested Relational	Type-Driven	IBM	U.S.A.	1989
Kabiria	Semantic Network	Type-Driven	Bull HN	Italy	1993
TEXPROS	Folder Organization	Predicate-Driven	NJIT	U.S.A.	1996

8.4 Future Research Directions

This section presents an overview of some future research directions that emanate from the work described in the dissertation.

8.4.1 Specification of Criteria for the Folders

In this dissertation, a criterion of a folder is specified by a predicate. An atomic predicate is defined as ⟨Attribute⟩ ⟨Comparison Operator⟩ ⟨Value⟩. The comparison operators are pre-defined. A user has to use the restricted specification to specify predicates of folders. Such restricted specification may make a user difficult to map his/her criteria to predicates.

Considerable extension of predicate specification is to use a general first order predicate specification [15, 4]. For example, the predicate $(\forall x) Journal\_Article(x) \land Database(x)$ can be used to define a folder containing all the frame instances which are journal articles in the database area. By using such general first order predicate specification, there are two classes of predicates needed to be supported by the system.

- Pre-defined predicates. The system provides a set of pre-defined predicates so that a user can use to define common predicates of folders. The study is needed to determine what is primary set of pre-defined predicates.

- User-defined predicates. The pre-defined predicates may not meet a user's need to specify predicates of folders. A mechanism should be provided so that users can define additional predicates and their semantics.

8.4.2 Knowledge Discovery and Data Mining

A document filing system gathers and stores a large amount of documents. However, the documents themselves are of little direct value. What is of value is the knowledge that can be inferred from documents and put to use. Knowledge discovery in database (KDD) and data mining [13, 27] have the potential of providing good information and knowledge management support for a document filing system. The potential research issues include:

- Understandability of patterns. In office information systems, it is important to make the discoveries more understandable by humans. Possible solutions include graphical representations, rule structuring, natural language generation, and techniques for visualization of data and knowledge.

- Non-structured and multimedia documents. A significant trend is that a document base contains not just structured documents but large quantities of non-structured and multimedia documents. Non-structured documents contain nonstandard data types, such as non-numeric, non-textual, geometric, and graphical data, as well as non-stationary, temporal, spatial and relational data. Multimedia documents include free-form multi-lingual text as well as digitized images, video, speech and audio data. These data types are largely beyond the scope of current KDD and data mining technology.

8.4.3 Reorganization of a Filing System

A filing system is dynamic and evolving. A user can reorganize his/her filing system such as add a new folder, delete a folder, merge folders to be one folder, move a folder from one spot to the other, etc. As a consequence, some frame instances must be re-filed. There is an ongoing research using an agent-based architecture to cope with file reorganization [50]. Each folder is monitored by an agent. Agents are represented as objects using an object-oriented approach. It encapsulates the internal representation of folders with the operations that manipulate them, thereby enhancing re-usability of code and information hiding.

8.4.4 A Multi-User Environment

Currently, TEXPROS document filing system is a personal (single-user) customizable system. However, for the demand of accessing the shared information, a multi-user filing system is needed. Consider a department document filing system containing the departmental information. Suppose the ResearchReport folder contains research reports of the department and it is shared to anyone. A user then can access and retrieve the abstracts of reports. In order to support a multi-user environment, the following considerations need to be made:

- Security issue. Like any other multi-user system, the security is always the first concern. We may categorize folders in the filing system into three classes: (1) public folders, (2) restricted folders, and (3) personal folders. A public folder contains public information that allows any user to access. A restricted folder only allows certain group of users or a privileged user to access. A personal folder has pure personal information and only the owner of the folder can access. A security mechanism needs then to be defined and built on the top of a multi-user filing system.

- Centralizing the information. Folders in a multi-user filing system may distributed over the network. For example, Professor John Smith creates his personal folder (called John_Smith) on his workstation and it is a child folder of Faculty folder which resides at another machine. In order to keep track the network information, a client-server model can be adopted. A machine is dedicated as a filing server that stores filing system information such as where are folders in the network, and what are the relationships among them. Whenever any change occurs, we only update the filing server. A client sends a request to the filing server to get filing system information. There will be a need of providing a set of protocols that govern the consistency of a filing system.

- Internet availability. The World Wide Web has transformed the online world. Users of the Web have a great deal of choices for selecting and viewing information. Java [12] opens up a new degree of interactivity and customizability of interaction for the Web. The integration of Netscape, Java and TEXPROS will make TEXPROS filing system available on the internet. A user can use a Web browser to file documents, and to retrieve documents.

REFERENCES

1. H. Adeli, *Knowledge Engineering*, McGraw-Hill, New York, NY, 1990.

2. E. Bertino, F. Rabitti, and S. Gibbs, "Query Processing in a Multimedia Document System," *ACM Transactions on Office Information Systems*, vol. 6, no. 1, pp. 1–41, January 1988.

3. G. Booch, *Object Oriented Design with Applications*, The Benjiamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

4. L.F. Boron, *Elements of Mathematical Logic*, Addison-Wesley Publishing Company, Inc., New York, NY, 1964.

5. A. Celentano, M. Fugini, and S. Pozzi, "Knowledge-Based Document Retrieval in Office Environments: The Kabiria System," *ACM Transactions on Office Information Systems*, vol. 13, no. 3, pp. 237–268, July 1995.

6. S. Chen, *Document Preprocessing and Fuzzy Unsupervised Character Classification*, Ph.D. Dissertation, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, May 1995.

7. S. Chen, F. Shih, and P. Ng, "A Fuzzy Model for Unsupervised Character Classification," *Information Science, An International Journal*, vol. 2, no. 2, 1994.

8. S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria, "Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and System," *ACM Transactions on Office Information Systems*, vol. 4, no. 4, pp. 345–383, October 1986.

9. H. Clifton, H. Garcia-Molina, and R. Hagmann, "The Design of a Document Database," in *Proceedings of the ACM Conference on Document Processing Systems*, pp. 125–134, December 1988.

10. W. Clocksin and C. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1981.

11. W. Croft and D. Stemple, "Supporting Office Document Architectures with Constrained Types," in *Proc. of ACM SIGMOD International Conf. on Management of Data*, pp. 504–509, 1987.

12. J. December, *Presenting Java: An Introduction to Java and Hotjava*, Sams.Net, Indianapolis, IN, 1995.

13. U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Communication of the ACM*, vol. 39, no. 11, 1996.

14. P. Fischer and S. Thomas, "Operators for Non-First Normal Form Relations," in *Proc. of the 7th International Computer Software Applications Conf.*, pp. 464–475, 1983.

15. J. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, Publishers, Inc., New York, NY, 1986.

16. S. Gibbs and D. Tsichritzis, "A Data Modeling Approach for Office Information Systems," *ACM Transactions on Office Information Systems*, vol. 1, no. 4, pp. 299–319, October 1983.

17. U. Gupta, *Validating and Verifying Knowledge-Based Systems*, IEEE Computer Society, Los Alamitos, CA, 1991.

18. R. Guting, R. Zicari, and D. Choy, "An Algebra for Structured Office Documents," *ACM Transactions on Information Systems*, vol. 7, no. 4, pp. 123–157, April 1989.

19. X. Hao, *Automatic Office Document Classification and Information Extraction*, Ph.D. Dissertation, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, August 1995.

20. X. Hao, J. Wang, M. Bieber, and P. Ng, "Heuristic Classification of Office Documents," *International Journal of Artificial Intelligence Tools*, vol. 3, no. 2, pp. 233–265, 1994.

21. X. Hao, J. Wang, and P. Ng, "Nested Segmentation: An Approach for Layout Analysis in Document Classification," in *Proc. of the Second International Conference on Document Analysis and Recognition*, Tsukuba Science City, Japan, pp. 319–322, October 1993.

22. C. Hewitt, "Office are Open Systems," *ACM Trans. on Office Information Systems*, vol. 4, no. 3, pp. 271–287, July 1986.

23. P. Hoepner, "Synchronizing the Presentation of Multimedia Objects – ODA Extensions," *ACM SIGOIS Bulletin*, vol. 12, no. 1, pp. 19–32, July 1991.

24. W. Horak, "Office Document Architecture and Office Document Interchange Formats – A Current Status of International Standardization," *IEEE Computer*, vol. 18, no. 10, pp. 50–60, October 1985.

25. J. Hughes, *Object Oriented Database*, Prentice Hall, New York, NY, 1990.

26. R. Hunter, P. Kaijser, and F. Nielsen, "ODA: A Document Architecture for Open Systems," *Computer Communication*, vol. 12, no. 2, pp. 139–151, 1989.

27. T. Imielinski and H. Mannila, "A Database Perspective on Knowledge Discovery," *Communication of the ACM*, vol. 39, no. 11, 1996.

28. G. Jaeschke and H. Schek, "Remarks on the Algebra of Non-First Normal Form Relations," in *Proc. of the ACM SIGACT-SIGMOD Sysmposium on PODS*, pp. 124–138, 1982.

29. W. Kim and F. Lochousky, *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, New York, NY, 1989.

30. Q. Liu and P. Ng, "A Browser of Supporting Vague Query Processing in an Office Document System," *Journal of Systems Integration*, vol. 5, no. 1, pp. 61–82, 1995.

31. Q. Liu and P. Ng, "A Query Generalizer for Providing Cooperative Responses in an Office Document System (revised version)," Submitted to Data and Knowledge Engineering Journal, October 1995.

32. Q. Liu and P. Ng, *Document Processing and Retrieval: Text Processing*, Kluwer Academic Publishers, Norwell, MS, 1996.

33. Q. Liu, J. Wang, and P. Ng, "An Office Document Retrieval System with the Capability of Processing Incomplete and Vague Queries," in *Proc. of the Fifth Intl. Conf. on Software Engineering and Knowledge*, San Francisco, CA, pp. 11–17, June 1993.

34. Q. Liu, J. Wang, and P. Ng, "On Research Issues Regarding Uncertain Query Processing in An Office Document Retrieval System," *Journal of Systems Integration*, vol. 3, no. 2, pp. 163–194, 1993.

35. E. Lutz, H. Kleist-Retzow, and K. Hoernig, *Multi-User Interface and Applications*, ch. MAFIA - an Active Mail-Filter-Agent for An Intelligent Document Processing Support, North Holland: Elsevier Science Publishers, Amsterdam, S. Gibbs and A.A. Verrijn-Stuart ed., 1990.

36. D. Maier, *The Theory of Relational Database*, Computer Science Press, Potomac, MD, 1983.

37. J. McHugh, *Algorithmic Graph Theory*, Prentice Hall, NJ, 1990.

38. F. Mhlanga, *D_Model and D_Algebra: A Data Model and Algebra for Office Documents*, Ph.D. Dissertation, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, May 1993.

39. F. Mhlanga, J. Wang, T. Shiau, and P. Ng, "A Query Algebra for Office Documents," in *Proc. of the 2nd Intl. Conf. on Systems Integration*, Morristown, NJ, pp. 458–467, June 1992.

40. F. Mhlanga, Z. Zhu, J. Wang, and P. Ng, "A New Approach to Modeling Personal Office Documents," *Data and Knowledge Engineering*, vol. 17, no. 2, pp. 127–158, November 1995.

41. N. Naffah, *Integrated Office Systems*, North-Holland, Amsterdam, 1980.

42. J. Peckham and F. Maryanski, "Semantic Data Modles," *ACM Computing Survey*, vol. 20, no. 3, pp. 100–120, 1988.

43. B. Pernici and A. Verrjin-Stuait, *Office Information Systems: The Design Process*, North-Holland, Amsterdam, 1989.

44. S. Pozzi and A. Celentano, "Knowledge-Based Document Filing," *IEEE Expert*, pp. 34–45, October 1993.

45. D. Shasha and J. Wang, "Optimizing Equijoin Queries in Distributed Databases Where Relations Are Hash Partitioned," *ACM Transactions on Database Systems*, vol. 16, no. 2, pp. 279–308, June 1991.

46. G. Shaw and S. Zdonik, "A Query Algebra for Object-Oriented Databases," in *Proceedings of the Sixth International Conf. on Data Engineering*, Los Angles, CA, pp. 154–162, February 1990.

47. F. Shih, S. Chen, D. Hung, and P. Ng, "A Document Segmentation, Classification and Recognition System," in *Proceedings of 2nd International Conference on Systems Integration*, Morristown, NJ, pp. 258–267, June 1992.

48. S. Su, M. Gou, and H. Lam, "Association Algebra: A Mathematical Foundation for Object-Oriented Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 775–798, October 1993.

49. C. Thanos, *Multimedia Office Filing: The MULTOS Approach*, North-Holland, Amsterdam, 1990.

50. J. Wang, F. Mhlanga, Q. Liu, W. Shang, and P. Ng, "An Intelligent Documentation Support Environment," in *Proc. of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, pp. 429–436, June 1993.

51. J. Wang, F. Mhlanga, and P. Ng, "A New Approach to Modeling Office Documents," *ACM SIGOIS Bulletin*, vol. 14, no. 2, pp. 46–55, December 1993.

52. J. Wang and P. Ng, "TEXPROS: An Intelligent Document Processing System," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, pp. 171–196, April 1992.

53. C. Wei, *Knowledge Discovering for Document Classification Using Tree Matching in TEXPROS*, Ph.D. Dissertation, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, May 1996.

54. C. Wei, Q. Liu, J. Wang, and P. Ng, "Knowledge Discovering for Document Classification Using Tree Matching in TEXPROS," Submitted to Information Sciences, An International Journal, March 1996.

55. C. Wei, J. Wang, X. Hao, and P. Ng, "In Deductive Learning and Knowledge Representation for Document Classification: The TEXPROS Approach," in *Proceedings of 3rd International Conference on Systems Integration*, Sao Paulo, SP, Brazil, pp. 1166–1175, August 1994.

56. D. Woelk, W. Kim, and W. Luther, "An Object-Oriented Approach to Multimedia Databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington D.C., pp. 311–325, May 1986.

57. Z. Zhu, "Document Filing Based upon Predicates," Ph.D. Dissertation Proposal, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, October 1994.

58. Z. Zhu, Q. Liu, J. McHugh, and P. Ng, "A Predicate Driven Document Filing System," *Journal of Systems Integration*, vol. 6, no. 3, pp. 373–403, September 1996.

59. Z. Zhu, J. McHugh, J. Wang, and P. Ng, "A Formal Approach to Modeling Office Information Systems," *Journal of Systems Integration*, vol. 4, no. 4, pp. 373–403, December 1994.

60. J. Zobel, J. A. Thom, and R. Sacks-Davis, "Efficiency of Nested Relational Document Database Systems," in *Proc. of the 17th International Conf. on Very Large Databases*, Barcelona, Spain, pp. 91–102, September 1991.