Self-maintained Folder Hierarchies as Document Repositories

Johann Eder, Alexander Krumpholz

Department of Informatics Systems
University of Klagenfurt, Austria
{eder, krumpholz}@isys.uni-klu.ac.at

Alexandros Biliris, Euthimios Panagos*

AT&T Labs Research
Florham Park, NJ, USA
{biliris, thimios}@research.att.com

Abstract

This paper presents a novel approach for the management of large collections of electronic documents. The major technical contribution of our approach is the seamless integration of features found in systems such as file systems, document repositories, directory services, classification hierarchies, and email folders into a single folder-based system. The user may specify a rules for characterizing interesting or relevant document classes. Based on these rules which are checked for consistency, folder are not only populated with documents, but folders themselves are created and deleted based on the occurrence of values in documents.

1 Introduction

The proliferation of the Internet has created strong interest in the management of large collections of electronic documents. Today, the number of documents that are created, stored, and managed (in file systems, directories, databases, messaging systems, etc.) is growing at an exponential rate. Although existing document filing and retrieval systems have their relative merits, they are either very rigid or they do not provide adequate management capabilities. Therefore, the need for powerful and easy to use document filing and retrieval systems is imperative.

File systems are very easy to use and allow users to select where and how to store documents. However, it is difficult to maintain a proper structure and, thus, manual operations are required to a great extent. Directory services, on the other hand, offer automatic document management and support attribute-based document retrieval. However, they do not allow a user to organize documents based on their semantics instead of their explicit attributes. Classification hierarchies place documents in predefined document hierarchies automatically. Typically, the structure of these hierarchies cannot be extended easily, and they often prohibit manual placement of documents. In addition, their strict subset property along the folders in a hierarchy can be cumbersome for shared documents.

In this paper, we introduce a system that provides easy access to large collections of documents. Our approach combines database concepts for storing documents with the well-known user interface of hierarchical file systems. Document hierarchies can be managed by either creating and deleting folders explicitly or by specifying rules that reflect the user's preferred view on the top of a collection of documents. When a document is inserted at some folder in a hierarchy, rules may force it to be automatically propagated to sub-folders. In addition, folders may be created automatically when document insertion triggers appropriate rules. The main contributions of our work are the following.

- Seamless integration of folder hierarchy features found in file systems, directory services and classification hierarchies;
- Implicit and explicit placement of documents into folders;
- Automatic creation and deletion of folders according to actual values of document attributes;
- Implicit and explicit deletion of folders and documents combining deletion and garbage collection semantics for persistence management.

As proof of concept, we implemented a prototype using the expert system shell CLIPS, showing that our approach is indeed helpful for organizing documents and that the specifications of the system are correct. However, space does not allow to present all features and the full formal specification of this system here. Rather, we will introduce the main concepts and contributions of this work in a rather informal way. A detailed description, together with a formal specification of all operations can be found in [5].

*current affiliation: Voicemate Inc., New York, email: thimios@voicemate.com

document d1	document d2
id='1732'	id='46232'
type='mail'	type='mail'
from='alex'	from='michi'
to='herb'	to={'alex', 'hans'}
subject='PantaRhei'	subject='Lunch'

Table 1: Example set of documents *D*

2 Documents, and Folders

Document classification based on hierarchically organized collections is important for two reasons. Firstly, it provides an easy and intuitive way to classify documents into collections and sub-collections. Secondly, it facilitates easy navigations through document collections, especially when the criteria used to classify the documents in the first place are not readily available.

In the rest of this section, we present the definitions we use in subsequent sections, explain the document sets that can be associated with a folder, enumerate the properties attached to each folder and how they are used for document propagation and, finally, state the invariants that hold for each folder.

2.1 Documents

Documents are structured or semi-structured data. Each document is represented as a collection of attribute-value pairs. Each attribute may contain several values, and each value can be structured, i.e. a list of values or even additional attribute-value pairs. Examples of such documents include XML files, electronic mails, entries in directories (such as LDAP), workflow forms, and bibliographic references.

Let *d* be a document. The set of all attributes that are defined for *d* is denoted *attributes(d)*. For an attribute with name *a*, *values(d, a)* returns the set of values of this attribute in document *d*, if *a* is in *attributes(d)*, and the empty set otherwise.

For a value *v*, *denote(v)* is a function returning a unique string representation of *v*. For an attribute *a*, *denote(a)* returns the string representation of the attribute name. For a set of documents *D*, *attributes(D)* returns the attributes appearing in any document of *D*, while *values(D, a)* returns the set of all values of attribute *a* has in any document of *D*.

For the example set of documents listed in Table 1, the following information can be extracted.

```
attributes(d1)={id, type, from, to,
subject}
values(d2, to)={'alex', 'hans'}
attributes(D)={id, type, from, to,
subject}
values(D, to)={'herb', 'alex', 'hans'}
```

A *constraint* is a predicate or Boolean function over the values of documents attributes. We require the following properties of predicates. If *p* is a predicate and *d* is a document, then *eval(p, d)* evaluates to *true* or *false*. If *p* and *q* are predicates, so are the conjunction and disjunction of them, i.e. *p and q* and *p or q*.

Documents can be preprocessed to compute additional attributes using expressions, queries or arbitrary function calls. For this paper we assume that all relevant attributes are already available.

2.2 Folders and Folder Hierarchies

A folder is a container of documents and other folders, referred to as sub-folders. Typically, there are many ways to organize documents into folder hierarchies (as an example, consider the organizational hierarchy of a corporation: business units, centers, departments, and years followed by various document types). In the remainder of the paper, we assume that the folders in some hierarchy form a tree, where each folder (with the exception of the root folder) has exactly one parent folder.

Each folder has a relative name, *FRelName*, which uniquely identifies the folder within all sub-folders of its parent folder. The absolute name of a folder, *FAbsName*, is constructed by concatenating the relative folder names on the path from the root folder to the folder in question, and represented using '/' as the separator for the naming components. The absolute name of a folder uniquely identifies this folder within the system.

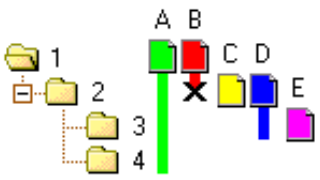
Each folder *F* may be associated with a constraint, called relative filter, *FRelFilter*). The absolute filter of a folder *F*, *FAbsFilter*, is the conjunction of *F*'s relative filter with the absolute filter of *F*'s parent folder. The absolute and relative filter of the root folder is simply *true*.

2.3 Explicit Document Insertion

Documents enter the system when they are inserted into a folders. The attribute *DocumentTimestamp* keeps the insertion time. The following command inserts the document *d3* into the system and assigns it to */myfolder*.

insert document *d3* **into** */myfolder*

Inheritance and rules can place the inserted document in other folders, too as we will explain below. In contrast to classification hierarchies, documents can be inserted at any point in the folder hierarchy, they can be assigned to several



FExpX(2)={C,D}
FRemX(2)={B}
FInhX(2)={A,B}
FFullX(2)={A,C,D}
FCoreX(2)={C}
FCoverX(2)={A,C,D,E}

Figure 1: Types of extents in our system

folders simultaneously, and, finally, they can be removed from a specific folder. Insertions and removals affect the propagation of documents in the hierarchy, as we will see below.

2.4 Folder Extents

A document belongs to a folder *F* either because it is explicitly inserted into *F* or because it is propagated to *F* from its parent folder. The following different notions of extents can be defined for a folder *F* (for examples see Figure 1, where the documents A and B are inserted into folder 1, C and D into folder 2 and E into folder 3. A is inherited to all subfolders, D to folder 3, B is explicitly removed from folder 2).

1. The explicit extent, *FExpX*, of *F* is the set of documents explicitly inserted into *F*;
2. The removed extent, *FRemX*, of *F* is the set of documents explicitly removed from *F*;
3. The inherited extent, *FInhX*, of *F* is the set of documents inherited from *F*'s parent;
4. The full extent, *FFullX*, of *F* is the union of the explicit and inherited extents, excluding the removed extent;
5. The core extent, *FCoreX*, of *F* contains all documents of the full extent that are not inherited by any subfolder of *F*;
6. The cover extent, *FCoverX*, of a folder *F* is the union of *F*'s full extent and the cover extents of all of *F*'s sub-folders. The cover extent of a leaf folder is the same as the full extent of the folder. The cover extent of the root folder contains all documents in the hierarchy.

2.5 Propagation to Subfolders

Insertion and inheritance of documents are controlled by filters and the inheritance options.

The option *FCheckInsert* controls the explicit insertion of documents into a folder, specifying which filters are checked at insertion time. If it is *false*, no documents can be inserted explicitly into this folder. If it is *relative* or *absolute*, the explicitly inserted documents have to satisfy the relative or absolute filter of the folder, respectively. If the *FCheckInsert* option is *true*, then there are no restrictions on the insertion.

property name	property type
FRelName	relative folder name
FAbsName	absolute folder name
FRelFilter	constraint
FAbsFilter	constraint
FAbsParent	absolute folder name
FSubfolders	set of rel. folder names
FInhX	set of documents
FExpX	set of documents
FRemX	set of documents
FFullX	set of documents
FCoreX	set of documents
FCoverX	set of documents
FRuleName	identifier
FInherits	[all|new|none]
FCheckInherit	[absolute|relative|true|false]
FCheckInsert	[absolute|relative|true|false]
FPermanent	[true|false]
FTimestamp	timestamp

Table 2: Properties of folders

The inheritance of documents from a parent folder to a sub-folder is controlled by the filters, the *FInherits* option and the *FCheckInherit* option. If the *FInherits* option of a folder is *all*, then all documents of the parent folder are considered for inheritance. If it is *new*, then only those documents of the parent folder that have been inserted after the creation of the folder are considered. If the inherits option is *none*, then the folder does not inherit any document from it's parent.

The *FCheckInherit* option specifies which filter is used to determine the inherited documents, and it is defined according to *FCheckInsert*. In particular, if *FCheckInsert* is *absolute* or *relative*, then the documents have to satisfy the absolute or relative filter, respectively. If it is *true*, every document can be inherited. If the option is set to *false*, there cannot be a match and inheritance is disabled.

The *FPermanent* option is used to define when a folder is deleted. If it is *true*, then the folder exists until it is explicitly deleted. If it is *false*, the folder only exists, if it contains sub-folders or documents. This implies that the folder will be deleted when its last document is removed. A complete list of the properties is given in Table 2.

The above folder options lead to powerful combinations of explicit and implicit placement of documents. For example, if the *FInherits* option for all folders is set to *none*,

and the *FCheckInsert* option is set to *all*, then the folder hierarchy has file-system semantics: all documents in a folder have to be explicitly inserted and no document is propagated to sub-folders. On the other extreme, when the *FCheckInsert* option is set to *none* for all folders except the root and the *FCheckInherit* option is set to *absolute*, the typical classification hierarchy semantics are achieved, where all documents are contained in the explicit extent of the root (the document base) and all other folders contain inherited documents only.

Arbitrary combinations of these concepts are feasible, e.g. to have a hierarchy of insertion points according to the organizational structure and classification hierarchies below these insertion points. All combinations of property settings have a precisely defined semantics through the invariants described below.

2.6 Folder Invariants

The following invariants hold for each folder *f*.

1. $FAbsName(f) = FAbsName(FAbsParent(f))/FRelName(f)$
   The absolute name of a folder is the absolute name of its parent folder, followed by a '/' and the folder's relative name.

2. $FAbsFilter(f) = FRelFilter(f) \wedge FAbsFilter(FAbsParent(f))$
   The absolute filter of a folder is the conjunction of the absolute filter of its parent and the folder's relative filter.

3. $f \in FSubfolders(FAbsParent(f))$
   A folder is an element of the sub-folders of its parent folder.

4. $FFullX(f) = (FExpX(f) \cup FInhX(f)) \backslash FRemX(f)$
   The full extent of a folder is the union of its explicit extent with its inherited extent minus the removed extent.

5. $FCoreX(f) = \{d \in FFullX(f) | \neg\exists g \in FSubfolders(f) : d \in FInhX(g)\}$
   The core extent of a folder includes all documents of its full extent, that are not inherited to one of its sub-folders.

6. $FCoverX(f) = FFullX(f) \cup FCoverX(c_1) \cup \ldots \cup FCoverX(c_n), c_i \in FSubfolders(f)$
   The cover extent of a folder is the union of its full extent with the cover extent of all sub-folders of the folder.

7. $FInhX(f) = \{d \in FFullX(parent(f)) |$
   $(FInherits(f) = all \vee (FInherits(f) = new$
   $\wedge DocumentTimestamp(d) \geq FTimestamp(f)))$
   $\wedge((FCheckInherit(f) = absolute$
   $\wedge eval(FAbsFilter(f), d) = true)$
   $\vee(FCheckInherit(f) = relative$
   $\wedge eval(FRelFilter(f), d) = true))\}$
   The inherited extent of a folder contains all documents of the full extent of its parent folder according to the *FInherits* option: (*all*): all documents are inherited, (*new*): only documents newer then the folder are inherited. Additionally the documents must match the filter defined by the *FCheckInherit* option to be inherited.

8. $FCheckInsert(f) = absolute \Rightarrow \forall d \in FExpX(f) : eval(FAbsFilter(f), d) = true$
   If the *FCheckInsert* variable is set to *absolute*, all documents of the explicit extent must match the folder's absolute filter.

9. $FCheckInsert(f) = relative \Rightarrow \forall d \in FExpX(f) : eval(FRelFilter(f), d) = true$
   If the *FCheckInsert* variable is set to *relative*, all documents of the explicit extent must match the folder's relative filter.

10. $FCheckInsert(f) = false \Rightarrow FExpX(f) = \emptyset$
    If the *FCheckInsert* variable is set to *false*, the explicit extent should not contain any document.

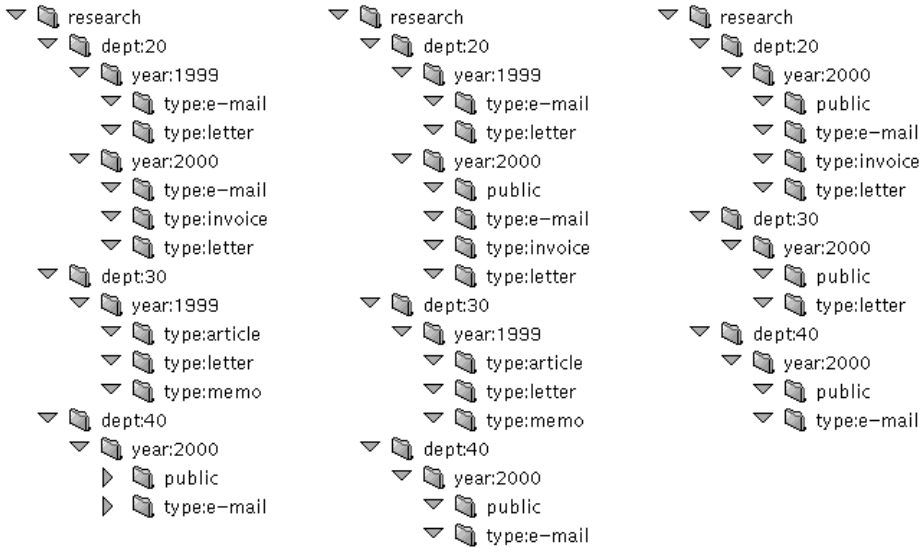
11. $FPermanent(f) = false \Rightarrow FFullX(f) \neq \emptyset \vee FSubfolders(f) \neq \emptyset$
    If a folder's permanent option is set to *false*, the folder must contain a document or a sub-folder.

3 Folder Operations

3.1 Creating folders

A folder name is represented by a string and a folder path is a sequence of folder names separated by the slash character. A generic path (*gpath* for short) is a sequence of names where some of the names or parts of the names have been replaced with the wild-card characters * or +.

A path matches a generic path, if each * in the generic path can be substituted by a string and each + by a path such that this substituted generic path equals the path. For example, */research/dept:30/year:1998* is a path, while */research/*/year:1998* is a generic path that matches the first when the wild-card * is replaced with *dept:30*. The same path would also match the generic path */+/year:1998*, where + substitutes *research/dept:30*.



Tree 1 Tree 2 Tree 3

Figure 2: Example trees

3.2 Explicit Folder Creation

The simplest way of creating a folder is by an explicit creation command. For example, the following statement will create a folder named **research** under the folder **/**, the top level folder, and the (relative) filter of the new folder will be **org=lab**.

create folder research under / with org=lab

If the name of the parent folder includes wild-cards (generic paths), a folder is created under all folders that match the generic path. For example, the following command will create a folder with the (relative) name **2000** under all folders matching **/*/new**.

create folder 2000
under /*/new with year=2000

3.3 Template-based Folder Creation

Folders are frequently created according to attribute values appearing in documents. Therefore, we provide operations to create a folder for each value of some distinguished attribute given in the **by** clause.

create folder under /research
by dept named D-$val

This statement will create a folder under **/research** for each distinct value of the attribute **dept** of any document in the full extent of the folder **/research**. The name of such a folder will be **D-$val**, where **$val** is a placeholder for the actual value of the attribute **dept**, and the relative filter of the folder will be **dept=val**(e.g., the name of the folder having **dept=20** becomes **D-20**).

Without the **named** option in the template statement we use the default name pattern **att:$val**, where **att** is the name of the distinguished attribute.

Template-based folder creation can specify a sequence of attribute names to be used for creating several levels in a folder hierarchy. For example,
create folder under /research by dept/year
is a shorthand for the following statements:
create folder under /research by dept
create folder under /research/dept:* by year

Tree 1 in Figure 2 shows an instance of a folder hierarchy created by the above template.

3.4 Rule-based Folder Creation

For maintaining folder hierarchies we may define rules that create folders when the folder hierarchy changes or documents with new attribute values are inserted into a hierarchy. We can think of these rules as folder creation operations that are executed whenever the folder hierarchy or the underlying document base changes.

rule r1: create folder old
under /* with year $<$ 1998

The example above shows a rule with the rulename **r1** will create a folder named **old** under each folder with an absolute name matching **/*** (i.e. all folders in the second layer of the folder hierarchy). When a new folder (e.g. **lab**) is created afterwards, a folder with name **old** will be automat-

ically created under this folder (**/lab/old**) in our example).

Rules can also create folders based on templates. Tree 2 of Figure 2 shows the folder hierarchy created by the following operations and rules.

create folder research under /
with org=research
rule r1:
create folders under /research
by dept/year/type
rule r2:
create permanent folder public
under /research/*/year:*
with clearance $<$ secret and year $>$ 1999

When a document with the attributes (dept=30, clearance=unclassified, year=2000, type=letter) is inserted into the hierarchy shown in Tree 2 of Figure 2, a new folder is created under **dept:30**. When a folder that has been created by a rule becomes empty, the folder is automatically removed from the hierarchy. Tree 3 of Figure 2 shows the hierarchy after the removal of all document with **year** $<$ **2000**.

Rule-based folder creation allows the classification of documents into various folders at the same time. Each rule is inserted into the "rule base" of the system and creates a generic invariant that is enforced whenever the folder hierarchy or the document base changes. The invariant for the maintenance of folders with template rules is presented below. RuleType indicates that it is a template rule, $\doteq$ is the matching relation between generic paths, RuleAttribute contains the distinguished attribute of the template, RuleRelfilter the relative filter given in the rule definition, and RuleNamePattern represents the string given in the named option.

Invariant:
$\forall rn \in R : r(rn).RuleType =' template' \wedge$
$\forall afn \in F_{exi} : f(afn).FAbsName \doteq$
$r(rn).RuleGenericParent \wedge$
$\forall d \in f(afn).fullX : r(rn).RuleAttribute \in$
$attributes(d) \wedge$
$\forall v \in values(d, r(rn).RuleAttribute)$
$\Rightarrow \exists f : f.FAbsName \in F_{exi} \wedge$
$f.FAbsParent = afn \wedge$
$f.FRelFilter = (r(rn).RuleRelFilter \wedge$
$(r(rn).RuleAttribute = denote(v))) \wedge$
$f.FRelName =$
$stringrepl(r(rn).RuleNamePattern,' \$val', denote(v)) \wedge$
$f.FRuleName = r(rn).RuleName$

The above invariant states that there has to be a subfolder with appropriate name and filter for all folders matching the parent folder given in the rule for all values of the distinguished attribute of documents in this (parent) folder.

3.5 Avoiding Conflicts

A conflict arises when a folder with the same name is defined twice. While this is easy to check for explicit folder creation operations, it has also to be checked for template-based folder creation and rule-based folder creation. In particular, in the case of rules, we need to check for potential conflicts at rule creation time, although the conflict might arise some time in the future, when a new document is inserted or a new folder is created.

We do not execute an operation when it (potentially) creates a folder that already exists or a folder that could be created by an existing rule. In order to determine whether a folder name may be created by a rule, we use the notion of *patterns*. In particular, the following patterns are being used.

- Absolute names of existing folders are patterns;
- A path-rule with (generic) parent *p* and folder name *f* has the pattern *p/f*;
- For a template-rule with the (generic) parent *p* and the name-pattern *f* we define the pattern *p/g* by substituting $val in *f* with *.

For an explicit folder creation operation, which would create a folder with the absolute name *n*, the operation is only executed when *n* does not match any existing pattern. When a rule is inserted, the pattern of the rule is checked against all existing patterns. If there is a match, then the rule is rejected. It is easy to see that using these precautions we can guarantee that no rule will generate a folder which already exists, or that no two rules would generate the same folder.

Since we introduced the wildcard + to represents a path of any depth, we must avoid rules which trigger themselves in a loop, directly, or indirectly via several rules. We avoid such scenarios by maintaining a graph showing (potential) triggering relationships between rules. Rules inducing a cycle in this graph will be rejected.

3.6 Execution Order

The invariants defined in section 2.5 and in section 3.4 represent the semantics of valid instances of a folder hierarchy and control the creation of folders and the placement of documents in folders. Whenever the document base or the folder hierarchy changes, these invariants are interpreted as rules establishing a consistent state as a consequence of the change.

We distinguish *producing invariants* that cause creation of additional folders or insertions like inheritance or rules and *reducing invariants* that delete folders or documents (garbage collection).

We define the priority of rules maintaining the invariants to guarantee deterministic behaviour of our system in the following sequence: first the operation is performed, then the producing invariants are triggered and finally the reducing invariants are executed.

Since producing invariants may only depend on the existence of facts and never on their absence, this method will ensure that the process will always terminate and the results of applying reducing invariants will never will invalidate producing invariants.

3.7 Deleting Folders

For maintaining the folder hierarchy, it is important to be able to delete folders and rules. When a folder does not have the *permanent* option, it is deleted automatically once it becomes empty, i.e. when it contains no documents and has no subfolders. Delete operations, of course, have to provide the combined garbage collection and deletion semantics for the definition of persistence as it is laid down in the folder and rule invariants. For an example:

delete folder /research

This operation will delete the folder **/research** when it has no subfolders. When a folder is generated by a rule, it is not deleted because the rule will create it again instantaneously. The operation

delete folder /research recursively

will delete the folder **/research** and all its subfolders even when they have been created by a rule, since the parent folder is also deleted and the rule will not recreate these folders.

We also provide operations to delete a rule. Here we have to specify how the folders generated by this rule have to be treated,: they may be deleted as well, their type may be changed to created folders.

4 Related Work

4.1 File Systems

In modern operating systems [10], like UNIX [11], files can be placed in folders created manually or via scripts, and folder hierarchies are created to organize files. User can structure their home directories (i.e. their part of the global file system hierarchy) in a way that represents their sense for organization. Problems arise when parts of the file system have to be accessed by different users, with different preferences. The structure of the hierarchy has to be well defined to find files that have been put there by a different user. In our approach, multiple organization hierarchies can be defined to access the same document. For example, the 'Call for papers' for the VLDB'2000 could be found in folders like the following:

```
.../Conferences/2000/VLDB/CFP
.../re-
search/Conferences/VLDB/2000/CFP
.../actualCFPs/VLDB
```

Files need not be put in those directories by hand - they are created automatically when a new CFP enters the system, if rules are used to specify the hierarchy. Every user will find the files by navigating through the system in his most intuitively way. Finally, our approach provides all the functionality and, additionally, allows inheritance according to filters and automatic maintenance of the folder structure.

4.2 Classification hierarchies

Classifications hierarchies are used to categorize documents according to their content. They have been used in the mid-90s for special purpose documents in the medical field, later they have been used in several search engines like Yahoo or Infoseek to categorize the content of the World Wide Web. Because of the huge amount of documents, automatically classifications was being developed using Machine Learning algorithms [4].

Our system provides the additional feature to add documents wherever the user wants them to be (associative placement) and let the user adjust the inheritance by setting control options.

4.3 LDAP Directory Services

The Lightweight Directory Access Protocol (LDAP) is a vendor-neutral standard to access common directory information [3]. It provides an extendable architecture for storage and management of information that needs to be available for today's distributed systems and services.

Similar to a filesystem or a classification hierarchy, all information in LDAP is organized in a tree, where each node represents an attribute/value pair [2]. The name of the node is always `Attribute=Value` and cannot be chosen freely. An object inside LDAP is always characterized by the path to the node where it is located, i.e. all documents in a node must match the conjunction of all expressions defined by the nodenames of its path. In this facets LDAP is closer to a classification hierarchy than to a filesystem.

4.4 Email Systems

Modern email clients let users organize their mails into folder structures. Usually this has to be done manually. Although the user can define filters in most of the client applications, e.g. Netscape Messenger [7], these filters cannot create new folders according to incoming mail. More sophisticated mail filters like procmail [9] are able to call

procedures to take care of new situations, but the rule base needs an experienced user to manage it and reorganization of the folder structure is not trivial.

Several studies have examined how to support the user by automatically filing the mails using Machine Learning [8], but new folders cannot be created here either, and there is also no way to find the same document using different access paths. SaveMe [1] offers an SMTP interface that allows a group of users to send mail to a defined folder which can be mounted by the users via the IMAP ACL extension [6]. SaveMe provides a way to share documents and access them like well known mail folders but the target folder has to be specified when sending the mail to SaveMe.

Our system can display the emails by browsing through folders that are created automatically by rules that are easy to define. New rules can be added to create new folder structures to access the same documents using different ways to go there. Therefore, our folder tree can be seen as nested views assorting messages.

4.5 Feature Matrix

Table 3 shows a comparison of the features the different systems support. A plus sign shows that the feature at the left is well supported by the proposal printed at the top of the column. A minus sign indicates insufficient support for this feature.

	F	C	D	E	O
free names	+	+	-	+	+
explicit placement	+	-	-	+	+
implicit placement	-	+	+	+	+
inheritance	-	+	+	-	+
configuarable inheritance	-	-	-	-	+
automatic maintenance	-	-	-	-	+

Table 3: Comparison of approaches
F: File Systems, C classification hierarchies, D: Directory services, E: email folders, O: our approach

5 Conclusion

In this paper, we presented a system that provides easy access to large collections of documents. Our approach combines database concepts for storing documents with the well-known user interface of hierarchical file systems and the automatic document placement of classification hierarchies. In addition, we provide a mechanism that is used to automatically maintain the structure of the hierarchy.

Rules can easily be specified to define interesting or relevant structures to access large collections of documents. Once such rules are defined, the system can automatically create new folders and delete existing ones when new documents are inserted or deleted, respectively.

The very generic definition of documents in our system makes it possible to support various kinds of different documents, e.g. email, office documents, forms of various kinds, network devices, or worklist entries of workflow management systems. In addition, the fact that we only handle document metadata allows us to store the actual document in any kind of database and avoid duplicates when the same document is inserted into multiple folders.

Finally, we implemented a prototype to prove our concept, to validate our specifications and to gain experience in the use of such a system. The evaluation of the behavior of the system has been quite successful so far. A high performance implementation is subject of ongoing research.

References

[1] Stefan Berchtold, Alexandros Biliris, and Euthimios Panagos. SaveMe: A system for archiving electronic documents using messaging groupware. In Dimitrios Georgakopoulos, Wolfgang Prinz, and Alexander L. Wolf, editors, *Proceedings of the International Joint Conference on Work Activities and Collaboration: WACC '99*, pages 167–176. ACM Press, 1999.

[2] Timothy A. Howes, Mark C. Smith, and Gordon S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, 1999.

[3] Heinz Johner, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, and Johan Westman. *Understanding LDAP*. IBM International Technical Support Organization (IBM Redbooks), 1998.

[4] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In *Proceedings of the 14th International Conference on Machine Learning ICML97*, pages 170–178, 1997.

[5] Alexander Krumpholz. Self-Maintained Document Hierarchies. Master's thesis, Institut für Informatik-Systeme, Universität Klagenfurt, June 2000.

[6] J. Myers. RFC2086: IMAP4 ACL extension. `http://www.imap.org/docs/rfc2086.html` (current 26/04/2000), 1997.

[7] Netscape, Inc. Netscape Messenger. `http://www.netscape.com` (current 26/04/2000), 2000.

[8] Jason Rennie. ifile: An application of machine learning to e-mail filtering, 1998.

[9] Stephen R. van den Berg. Procmail. `http://www.procmail.org` (current 26/04/2000), 1998.

[10] Andrew S. Tanenbaum and Albert S. Woodhull. *Prentice-Hall*. Operating Systems, 1997.

[11] Uresh Vahalia. *Prentice-Hall*. UNIX Internals, 1996.