A Model, Schema, and Interface for Metadata File Systems

Stijn Dekeyser **Richard Watson** **Lasse Motrøen**

University of Southern Queensland, Australia
{dekeyser,rwatson}@usq.edu.au, lassemot@yahoo.com

Abstract

Modern computer systems are based on the traditional hierarchical file system model, but typically contain large numbers of files with complex interrelationships. This traditional model is not capable of meeting the needs of current computer system users, who need to be able to store and retrieve files based on flexible criteria. A metadata file system can associate an extensive and rich set of data with a file, thus enabling more effective file organisation and retrieval than traditional file systems.

In this paper we review a wide range of existing proposals to add metadata to files and make that metadata available for searching. We then propose a hierarchy of definitions for metadata file systems based on the reviewed prototypes. We introduce a data model for a database-oriented pure MDFSs complete with operations and semantics. The model supports user-initiated instance and schema updates and file searches based on structured queries. We also explore the design space of a set of user interface operations intended to implement the pure model and facilitate the capturing of rich metadata. We argue that without such a simple method for users to create rich metadata, progress in this field will remain limited.

Keywords: Operating systems, Advanced applications of databases, Metadata.

1 Introduction

Traditional file systems store simple file metadata; a predefined set of data, mostly maintained by the operating system, is held in directories and file control blocks (e.g. inodes). Apart from assigning file names, users can effectively specify metadata by creating a directory hierarchy. The file path may encode some metadata. For instance the path `courses/csc2404/07/s2/ass1/1234/sync.c` assigns the following attributes to the file `sync.c`: *course=csc2404*, *year=2007*, *semester=2*, *studentId=1234*, *assignmentNum=1*, *filename=sync*, *filetype=Csource*. The ability to search based on attributes is limited as these attributes are stored hierarchically, and accessed via a path specification. It is a simple matter to build a search query that specifies all attributes in a file's path; this will yield all files in a directory. In our example, it is easy to locate all student assignment submissions for a particular offer of a course. However, a query that seeks to find all submissions for a particular student in a given semester is not supported.

To further explore these problems, consider the following common scenario. Bill has a multitude of music and image files on his personal computer and wants to organise his collection such that he can find and relate files easily. Using a traditional folder approach leads to various problems. The multimedia files can be placed in folders named according to several properties such as genre, year, band name, and location of photo. As discussed above, using hierarchical folders means that Bill loses the ability to search for files from different perspectives. He could populate the folders with soft links (or shortcuts) to the actual music files, but this would create an unacceptable burden of managing such links. Bill has installed third-party applications such as *Google Picasa* (for his image files) and *RealPlayer* (for his music files). These applications manage the organisation of files into groups based on the value of a property like "genre", which addresses the shortcoming of the folder approach. However it offers no solution if Bill wishes to link an image file to to a music file or if he wants to add his own metadata fields to a file.

This scenario demonstrates that organising multimedia using a traditional hierarchical file system, even when enhanced with specific applications, often proves to be impractical. The problem is not limited to multimedia as every type of file can have a large collection of metadata associated to it which can be used to organise the file space.

Problem Statement Simply stated, the first problem we address is that users must be able to manage files such that they can be located effectively at some future time. We need to be able to search for a file using multiple pathways (or search criteria). For example, we may use keywords that have been automatically extracted from the file, or attribute values (assigned by system or user), or links to related files, to seek the target file. A design for a metadata file system must include both the metadata storage model and appropriate user interfaces to allow a user to easily locate a file based on its metadata.

Critically, the second problem that we address is that a successful metadata file system must feature a user interface that allows users to easily assign meaningful and rich metadata. Requiring the user to create every piece of metadata through keyboard entry will almost certainly impede the adoption of such potentially revolutionary systems.

Existing Work Recently the advent of social networking websites that let users share images (e.g. *Flickr*) and video (e.g. *YouTube*) has demonstrated novel ways of organising multimedia. Such applications use the simple concept of *tags* to let users assign

metadata to their files, and allow others to search for files easily. On the users' own computers, more advanced applications such as Picasa and *Google Desktop* offer automated collection of metadata and use localised databases to store metadata and use it in search. Solutions proposed by researchers in the past decade took a more comprehensive approach by extending file systems with metadata functionality. On the commercial side, Microsoft is attempting[1] to implement a metadata file system called *WinFS*. We review these efforts in Section 2.

Contribution It is clear that various approaches to create, manage, and use metadata for files are being considered and developed, and that there is no single solution currently available that has wide adoption or satisfactorily solves all issues. In this paper we review a wide range of existing proposals to add metadata to files and make that metadata available for searching. We then propose a taxonomy for metadata file systems based on the reviewed prototypes. We introduce a data model for a database-oriented pure MDFS complete with operations and semantics. We explore a number of interesting and non-trivial issues that must be solved before a full-scale pure MDFS can be implemented. We also discuss two prototype implementations of our model and outline user interface interactions to capture rich metadata.

As evidenced by the fact that a major software company has not been able to deliver one after many years of work, it is clear that creating a truly useful and powerful MDFS is a daunting task. The problems are likely not only technical, but also of a more human nature (complexity for users, compatibility issues for businesses, etc). We therefore present our work as a modest step and as a basis for future extensions.

Note that the work presented in this paper is, within the context of computer science, of a highly multidisciplinary nature, drawing on results from multimedia systems, databases, programming languages, file systems, and human-computer interfaces.

2 Review of Existing Proposals and Systems

In this section we present existing systems and research proposals that attempt to overcome some of the shortcomings that are present in traditional hierarchical file systems. They include file systems specifically designed to make use of metadata and applications that make use of existing file systems to organise files. Due to space restrictions we refer the reader to [15] for a more detailed review of these and other systems (e.g. *Nebula* [3], *Windows Media Player*, etc).

Google Desktop Google Desktop provides a set of features that allows users to search for content on their computers, based on file name and, for some file types, content as well. When Google Desktop is installed on a system, it automatically indexes files on the computer. The index of words extracted from file names and, where possible, file content are stored in a local database. When new files are added to the system, or files are modified, the index is updated. Google Desktop relies on keyword search rather than structured queries. A search will retrieve a list of documents which are to some extent relevant to the keywords entered by the user. Hence, both the user interface and the kind of results are similar to those in the Google web search engine. A significant limitation is that users are not able to modify any metadata associated with a file. This can only be done by altering the file itself which will result in re-indexation. *Windows Desktop Search* is a similar system, based on the research prototype *Stuff I've Seen* (SIS) [7].

MIT Semantic File System The MIT Semantic File System [11] is one of the first file systems to address the shortcomings of traditional tree structured file systems. The main aim of the MIT Semantic File System (SFS) is to allow users to access files based on file content, as well as accessing files by name.

MIT SFS is designed to be integrated into a tree structured file system and it does so through the concept of *virtual directories*. Each virtual directory is interpreted as a query and contains symbolic links to the actual files stored in the underlying file system.

In order for SFS to provide file access based on file content (to make use of virtual directories as queries) the content of a file needs to be extracted. SFS does this by associating each file type with a *transducer* program that will extract the relevant metadata from files in the system. Each file type will have a specific transducer, and each transducer will be specifically designed to extract desired attributes and values from a file type. For example, a transducer for an email file may extract attributes "To", "From" and "Subject". MIT SFS comes with a set of default transducers that can handle the most common file types, but users are also able to implement their own transducers. A transducer table is used to determine which transducer to use for a certain file type.

Gifford et al. [11] outline some of the shortcomings of MIT SFS. The first point mentioned is that of the query language that each virtual directory can be associated with. MIT SFS offers only a basic query language that prohibits users from using boolean operators (such as 'OR', 'AND', etc.) to specify their queries. Users are also unable to assign metadata to files manually. It is also recognised by [11] that a more expressive data model should be utilised, instead of relying on simple attribute-value pairs.

To some extent, the open-source project MoveMetaFS [16] is similar to MIT SFS. The MMFS allows users to associate a set of tags to a file. The tags can be queried in a simple manner. The project mainly focusses on user interface operations to tag files, using a similar, but more limited, technique as we coined in [6] and describe in more depth in Section 7.

Haystack The Haystack project [12] is mainly based on the argument that developers cannot predict the ways a user wants to utilise information. All users have different needs and preferences when it comes to accessing information. Users should be able to specify relationships between different information objects, how these relationships should be presented, and how information should be gathered. Haystack currently accomplishes the required flexibility by storing all data using RDF [13]. Metadata for information objects in Haystack is initially automatically captured when a file is added to the system. It does so by generating RDF data using an *extractor* similar to MIT's transducer. The user interface for Haystack also allows users to easily modify metadata associated with files.

Aside from significant performance issues, an important shortcoming of the Haystack system is the absence of an API that other applications may use. Users have to interact with Haystack using its own user interface.

WinFS WinFS, like Haystack, attempts to offer a file system that allows information objects to be dy-

[1] Both the history and future of WinFS is relatively opaque. Contrary to earlier plans, it has not been shipped with Microsoft's Windows Vista operating system.

namically related to other information objects. Objects in WinFS can range from files of various types to persons, meetings, locations, etc., and they are all treated as information objects. The WinFS data model offers a rich set of operations that lets applications create, modify and query information objects, and is implemented on a relational database back-end.

WinFS allows applications to modify metadata stored in WinFS along with the schema for each information object, but offers only limited functionality to end-users. We argue that this unnecessarily curtails the usefulness of the system.

A less comprehensive open source project similar to WinFS is GNOME Storage which also uses a relational database backend. However, Storage focuses more on end-user keyword search rather than applications formulating structured queries. For more information on Storage and other systems we again refer the reader to [15].

Graffiti Graffiti [14] is a distributed organisation layer that augments an existing file system to add user-defined metadata and provide sharing of metadata across users and hosts. Graffiti supports the association of a simple text string tag with either a file or a pair of files (a named link). Apart from a linking capability, this differs from the common tagging systems (e.g. Flickr) in that it is generic rather than application-specific. Command line and graphical interfaces are provided.

While its metadata structure is unsophisticated, Graffiti addresses the problem of sharing files and metadata with some success. File checksums are used to ensure that a file and its metadata can be synchronised across multiple platforms. See section 5.2 for more discussion.

Linking File System The Linking File System [1] (LiFS) is a prototype implemented on top of a Linux filesystem. It augments a traditional file system with user specified attributes on files, and links between pairs of files. Links also have an associated set of attributes. The attributes are key/value pairs. LiFS implements the concept of a file trigger, which is an executable file attribute, encoded as a pattern/action pair. When a file operation occurs that matches the pattern, the associated action is executed. This generic mechanism is similar to the MIT SFS's transducer concept and could be used to implement the automatic collection of metadata (see Section 4.3).

The LiFS approach addresses the requirements of storing arbitrary user metadata. However, the absence of a metadata schema is an obstacle to the creation of advanced user interfaces, and implementation of powerful search queries. LiFS does not accommodate either specialisation of metadata for related file types through inheritance, or the ability to create non-file objects that could be linked to files. While the LiFS model is expressible using a database style model like WinFS, or that described in this paper, the reverse is not true.

3 A Taxonomy of Metadata File Systems

The full review of existing proposal and systems [15] brings to light a number of features which can be used to classify the systems into categories, and assist in constructing a taxonomy for a variety of metadata file systems and applications.

Data Model The data model for the reviewed systems ranges from attribute-value pairs over a relational model to RDF graphs.

Metadata–Filesystem integration Some applications maintain metadata in special purpose databases, and offer no filesystem functionality (e.g. for launching files). More advanced systems integrate the storage of metadata within the file system itself, and also offer rich operations on files.

Metadata Capture and Modification Most systems implement the concept of the MIT SFS transducer to capture metadata automatically. Few systems allow users to modify the metadata manually. Hence, rich metadata that is impractical to capture automatically (e.g. appearance of persons in images) is often neglected (some applications will allow users to add information in a predefined "comments" field).

Metadata Schema Modification very few systems allow users (or even applications) to modify the schema of the metadata store. Hence it is often impossible to create new types, new attributes, or new relationships between types.

Dynamic Views Only a few systems support the concept of dynamic views of objects defined by metadata properties. In addition, the expressive power of the view definition language is very limited.

We now propose a hierarchical classification of metadata file systems and applications.

Definition 1 (Metadata Enabled Application). *A Metadata enabled application is a stand-alone software package that runs on top of a host file system and has the following properties:*

1. *Manages its own database of metadata for files of a limited number of types.*
2. *Has a user interface that allows files to be organised based on the metadata, and allows users to search for files using keywords or simple attribute-value comparison.*

Such applications typically lack the ability to relate files of different types and to modify the schema of the metadata store. Examples of such tools include Google Picasa, Windows Media Player, Graffiti, and even MIT SFS.

Definition 2 (Rich Metadata Applications). *A rich metadata application supports the features of a metadata enabled application and also runs on top of an existing file system. It has the following additional features:*

1. *Allows end-users full power to manage metadata previously captured automatically, and allows users to relate files of different types.*
2. *Allows the schema for the metadata store to be modified.*

Such applications typically lack an API that other applications can use, and are not well integrated with the host filesystem. Examples of rich metadata applications include Nebula and Haystack.

Definition 3 (Metadata File Systems). *A Metadata file system (*MDFS*) supports the features of a rich metadata application but is tightly integrated with the traditional features of a filesystem. In addition it uses an expressive data model (i.e. relational, object-relational, object-oriented, or semi-structured), and has a comprehensive API to be used by third-party applications.* Examples of such systems include Microsoft's WinFS and GNOME's Storage.

Finally we present the definition used in the remainder of this paper.

Definition 4 (Pure Metadata File Systems). *A Pure metadata file system is an* MDFS *built on an object-oriented data model and features a powerful generic graphical user interface allowing end-users to fully manage metadata and schema modification.*

WinFS is not a pure MDFS because it is aimed towards software developers rather than end-users. While Microsoft's policy has the advantage of simplifying implementation and has the potential of making the introduction of WinFS on the desktop more palatable, we take the view that it is unnecessarily restrictive and misses the opportunity to present end-users with a potentially revolutionary new approach to file management. Indeed, users of WinFS will need to rely on applications to capture and use metadata and especially on their provisions to associate files of various types. Hence, users will not have access to a generic file browser for this functionality.

4 A Model for a Pure MDFS

In this section we present the formal definition of a data model for a metadata file system. What form should such a model take? A model must support the association of named attributes to files, and also have the ability to record relationships between files. A metadata file system must also support non-file entities (e.g. Persons) in order to be able to store complex metadata. A key feature of a pure metadata file system is that users can extend the metadata structure, typically by specialising an existing entity. Specialisation can be implemented using inheritance. The features described so far correspond closely with the entity-relationship data model. However, we will also need to provide special behaviour to entities (see Section 4.3).

In essence then, the data model we use at the lowest level is almost a subset of the ODMG Object Model [17, 5]. A significant departure from the ODMG model is that the deletion of a class from the schema has novel and unusual semantics. As class deletion implies object deletion, removing a class in the usual manner could also remove files. We define a more sophisticated delete operation (section 4.2.1) that does not result in file deletion.

We support the functions of a metadata file system by defining a series of schemas over the data model. Such schemas naturally form a hierarchy, with the base level ignoring concrete issues such as built-in classes and relationships, an fundamental file attributes. Schemas at higher levels are defined through extending the base level with new classes and attributes, mostly through inheritance. The next level in the hierarchy shown in Figure 1, the minimal schema, provides generic classes, file attributes and relationships. It is the minimum system that could be employed by users of an operating system which includes an MDFS file system.

In many cases this vanilla file system will be extended by operating system vendors and distributors to meet their requirements, for example shipping standard multimedia and word processing document classes and built-in relationships between email and person classes. Organisations could further extend the metadata schema to meet corporate and project specific needs. Finally, individual users can add to the hierarchy of classes if needed.

The rich hierarchy of built-in classes and relationships in the schema that end users will obtain with their OS, means that a possible proliferation of mutually incompatible schemas is somewhat mitigated. In addition, any two MDFS volumes will share at least part of the class hierarchy, making data and schema integration issues less of a problem. We discuss this further in Section 5.

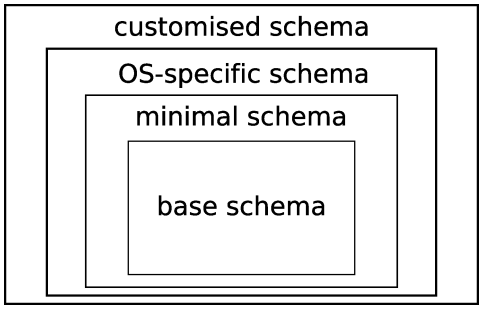

Figure 1: Filesystem schema levels

We will now define models for a schema and instance of a schema. The operations defined over the schema and instance are the application programming interface (API) to the metadata file system. Like their counterparts in a traditional file system, these would be implemented as system calls and execute in kernel space.

4.1 Data Model

The data model is class based, with relationships and simple inheritance. A schema defines classes and relationships between classes. Classes are also elements of an inheritance tree that is part of a schema. In the minimal schema we distinguish between classes that are associated with files (fileable) and those that are not (abstract). An instance over a schema defines instances of classes (objects) and relationship instances.

4.1.1 Names, identifiers, and values

The sets of possible class, relationship, attribute, and type names are respectively: *class*, *rel*, *att* and *type*. Objects and files are identified by members of the sets *oid* and *fid*, whereas *value* is the set of values. If $type = \{t_1 \dots t_n\}$ and $\mathrm{dom}\, t$ is the set of values associated with type t, then $value = \mathrm{dom}(t_1) \cup \mathrm{dom}(t_2) \cup \dots \mathrm{dom}(t_n)$. Each type contains a NULL value.

4.1.2 Schema definitions

A schema defines classes, organised into a specialisation hierarchy, and relationships between classes. We define the schema as the triple (class function, hierarchy relation, relationship relation). We define three supplementary functions on classes P (parent), S (superclass), and A (attribute), that are used in defining the schema operation semantics, and the instance semantics.

Each class introduces new attributes (class function C below), and instances of a class (objects) will contain the union of attributes defined by all ancestor classes. A class also inherits relationships from its ancestors.

$\mathcal{S}$ The *schema* $\mathcal{S} = (C, R, H)$.

C The *class* function $C : class \nrightarrow (att \nrightarrow type)$.

H The class *hierarchy* $H \subseteq class \times class$ such that H is the set of branches between classes, forming a single, rooted tree. Hence $(c_1, c_2) \in H$ means

that c_2 inherits directly from c_1, or c_1 is the parent class of c_2 (see P below).
$\forall(c_1, c_2) \in H \bullet \mathrm{dom}(C(c_2)) \cap \mathrm{dom}(A(c_1) = \emptyset$.
That is, subclasses can only extend a superclass definition.

A few classes and a small hierarchy is predefined in the minimal schema (the base schema is empty). There is a single root class (T_C), which has two subclasses (F_C and A_C). All user-created classes in the schema inherit from one of these two subclasses.

T_C The root of H. $T_C \in \mathit{class}$.
$C(T_C) = \{\mathit{creationTime} \mapsto \mathit{time}\}$

F_C The 'fileable' superclass $F_C \in \mathit{class}$.
$(T_C, F_C) \in H$.
$C(F_C) = \{\mathit{fileId} \mapsto \mathit{fid}\}$

A_C The 'abstract' superclass $A_C \in \mathit{class}$.
$(T_C, A_C) \in H \wedge T_C \neq F_C$

R The set of *relationships* $R \subseteq \mathit{rel} \times \mathit{class} \times \mathit{class}$.
A class inherits its ancestors' relationships, so the same relationship cannot be defined for the descendant classes.
$\forall(r, c_1, c_2) \in R \bullet \neg\exists(r_1, c_3, c_4) \in R \bullet r = r_1 \wedge c_3 \in S(c_1) \wedge c_4 \in S(c_2)$.

P The *parent* function on classes $P : \mathit{class} \nrightarrow \mathit{class}$.
$P = H^{-1}$.

S The *superclass* function $S : \mathit{class} \rightarrow \mathbb{P}\,\mathit{class}$.
$S = P^{+}$.

A The *attribute* function on classes.
$A : \mathit{class} \nrightarrow (\mathit{att} \nrightarrow \mathit{type})$.
$A(c) = \{(a, t) \mid c' \in S(c) \wedge (c'', f) \in C \wedge c' = c'' \wedge (a, t) \in f\}$. $A(c)$ defines the attributes, some inherited from superclasses, of an object instance of c.

4.1.3 Instance

The instance of a schema is modelled by a set of functions that encode the state of objects (O), and provide a means of identifying objects (I_C), and relationship instances (I_R). We also present definitions for two functions on object identifiers that are used to reveal the type (T) of the associated class, and its attributes (A_O). These are used in defining the semantics of the instance operations.

$\mathcal{I}$ The instance of schema $\mathcal{S}$. $\mathcal{I} = (O, I_C, I_R)$.

O The object function $O : \mathit{oid} \nrightarrow (\mathit{att} \nrightarrow \mathit{value})$.
The familiar *object.attribute* field access notation can be used as a shorthand:
$\forall\, o : \mathit{oid}, a : \mathit{att} \bullet o.a = O(o)(a)$

I_C The *instance* function identifies objects that have been created as an instance of a class.
$I_C : \mathit{class} \nrightarrow \mathbb{P}\,\mathit{oid}$.
$\forall\, c_1, c_2 \in \mathit{class} \bullet c_1 \neq c_2 \Rightarrow I_C(c_1) \cap I_C(c_2) = \emptyset$.
(Object is instance of just one class.)
$\forall(c, s) \in I_C \bullet \forall\, o \in s \bullet \mathrm{dom}(A(c)) = \mathrm{dom}(O(o))$
$\forall(c, s) \in I_C \bullet \forall\, o \in s \bullet \forall\, a \in \mathrm{dom}(O(o)) \bullet O(o)(a) : A(c)(a)$
(A class instance contains precisely the attributes of its instantiating class.)

I_R The *relationship instance* function
$I_R : (\mathit{rel} \times \mathit{class} \times \mathit{class}) \nrightarrow \mathbb{P}(\mathit{oid} \times \mathit{oid})$.
$\forall(r, c_1, c_2) \in R \bullet \forall(o_1, o_2) \in I_R(r, c_1, c_2) \bullet c_1 \in T(o_1) \wedge c_2 \in T(o_2)$.

T The *type* function on object identifiers
$T : \mathit{oid} \nrightarrow \mathbb{P}\, C$.
$T = S \circ I_C^{-1}$
$T(o)$ is a set that includes the class c that was used to instantiate o as well as all the superclasses of c.

A_O The attribute function on object identifiers.
$A_O : \mathit{oid} \nrightarrow (\mathit{att} \nrightarrow \mathit{type})$.
$A_O = A \circ I_C^{-1}$.

4.2 Operations

The following operations are sufficient to maintain the schema and an instance of a metadata store. Their semantics are defined with respect to the data model.

For each operation we show state transformations, the return value and, if the operation is partial, the exception condition. A transformation of a relation X is typically described as $X' = f(X)$, where X is the state of X before the operation and X' the post-operation state.

4.2.1 Schema Operations

The following operations populate a schema. Note that the schema "delete" operations also affect the instance. The delete class operation is very powerful and has interesting semantics. It removes the nominated class and all subclasses, and any relationships that relate those classes. Class deletion removes instances of deleted relationships, but *objects are not deleted*. Objects are instead recast as instances of the parent of the class being deleted. This will result in deletion of some attribute values.
The initial state of the minimal schema is:

$$\mathcal{S} = \{\{\ T_C \mapsto \{\mathit{creationTime} \mapsto \mathit{time}\},\ F_C \mapsto \{\mathit{fileId} \mapsto \mathit{fid}\},\ A_C \mapsto \emptyset\ \},\ \emptyset,\ \{\ (T_C, F_C), (T_C, A_C)\}\}$$

createClass(c, p, m): $\mathit{class} \times \mathit{class} \times \mathbb{P}(\mathit{att} \times \mathit{type}) \rightarrow \mathit{bool}$
$C' = C \cup \{c \mapsto m\}$
$H' = H \cup \{(p, c)\}$
Returns: $c \notin \mathrm{dom}\, C \wedge p \in \mathrm{dom}\, C$

deleteClass(c): $\mathit{class} \rightarrow \mathit{bool}$
$C' = C \lhd D$
$H' = \{(p, c) \mid (p, c) \in H \wedge p \notin D \wedge c \notin d\}$
$R' = \{(r, c_1, c_2) \mid (r, c_1, c_2) \in R \wedge c_1 \notin D \wedge c_2 \notin D\}$
$O' = O \oplus \{(o, O(o) \lhd \mathrm{dom}\, A(P(c))) \mid c \in T(o)\}$
$I'_C = I_C \lhd D$
$I'_R = I_R \lhd \{(r, c_1, c_2) \mid (r, c_1, c_2) \in R \wedge (c_1 \in D \vee c_2 \in D)\}$
Returns: $c \in \mathrm{dom}\, C$
where $D = \{c\} \cup \{x \mid c \in S(x)\}$

createRelation(r, c_1, c_2): $\mathit{rel} \times \mathit{class} \times \mathit{class} \rightarrow \mathit{bool}$
$R' = R \cup \{(r, c_1, c_2)\}$
Returns: $(r, c_1, c_2) \notin R$

deleteRelation(r, c_1, c_2): $\mathit{rel} \times \mathit{class} \times \mathit{class} \rightarrow \mathit{bool}$
$R' = R \setminus \{(r, c_1, c_2)\}$
$I'_R = I_R \lhd \{(r, c_1, c_2)\}$
Returns: $(r, c_1, c_2) \in R$

4.2.2 Instance Operations

Operations that access the files, but do not update metadata, are not presented here. A small set of operations (*open*, *close*, *read*, *write*, and *position*) would be required; only *write* is likely to affect the contents

of the metadata store; we discuss this is section 4.3. Note also that the *createFile* and *deleteFile* operations defined below reflect only the metadata store semantics. *createFile* would create a zero length file in the file store, and *deleteFile* would remove a file from the file store. Initially, $\mathcal{I} = \{\emptyset, \emptyset, \emptyset\}$

createObject(c): $class \nrightarrow oid$
$O' = O \cup \{(o, \{(a, NULL) \mid a \in \mathrm{dom}(A(c))\})\}$ where $o \in oid \wedge o \notin \mathrm{dom}\, O$
Exception: $c \notin \mathrm{dom}\, C$
Returns: o
deleteObject(o): $oid \rightarrow bool$
$O' = O \lhd \{o\}$
$I'_R = \{(r, x_1, x_2) \mid (r, x_1, x_2) \in I_R \wedge o \neq x_1 \wedge o \neq x_2\}$
Returns: $o \in \mathrm{dom}\, O$
createFile(o): $oid \rightarrow bool$
Returns:
$F_C \in T(o) \wedge o.fileId = NULL \wedge setAtt(o, fileId, f)$
where $f \in fid \wedge \neg \exists\, o \bullet o.fileId = f$
deleteFile(o): $oid \rightarrow bool$
Returns: $F_C \in T(o) \wedge setAttr(o, fileId, NULL)$
setAtt(o, a, v): $oid \times att \times value \rightarrow bool$
$O' = O \oplus \{(o, O(o) \oplus \{(a, v)\})\}$
Returns: $o \in \mathrm{dom}\, O \wedge a \in \mathrm{dom}\, A_O(o) \wedge v : A_O(o)(a)$
getAtt(o, a): $oid \times att \nrightarrow value$
Exception: $o \notin \mathrm{dom}\, O \vee a \notin \mathrm{dom}\, O(o)$
Returns: $o.a$
relate(r, o_1, o_2): $rel \times oid \times oid \rightarrow bool$
$I'_R = I_R \cup \{((c_1, c_2, r), (o_1, o_2)) \mid (r_1, c_1, c_2) \in R \wedge r = r_1 \wedge c_1 : T(o_1) \wedge c_2 : T(o_2)\}$
Returns:
$\exists (r_1, c_1, c_2) \in R \bullet r = r_1 \wedge c_1 : T(o_1) \wedge c_2 : T(o_2)$
unrelate(r, o_1, o_2): $r \times oid \times oid \rightarrow bool$
$I'_R = I_R \setminus \{(r, o_1, o_2)\}$
Returns: $(r, o_1, o_2) \in I_R$
isClass(c): $class \rightarrow bool$
Returns: $c \in \mathrm{dom}\, C$
isAtt(c, a): $class \times att \rightarrow bool$
Returns: $a \in \mathrm{dom}\, A(c)$
isInstance(o, c): $oid \times class \rightarrow bool$
Returns: $c \in T(o)$
isRelated(r, o_1, o_2): $rel \times oid \times oid \rightarrow bool$
Returns:
$\exists (r_1, c_1, c_2) \in R \bullet c_1 \in T(o_1) \wedge c_2 \in T(o_2) \wedge r1 = r$
search(c, f): $class \times F \rightarrow \mathbb{P}\, oid$
Returns: $\{o \mid o \in isInstance(o, c) \wedge F\}$

F is a formula with the following syntax

$$\begin{array}{rcl} F & ::= & atom \mid F \wedge F \mid F \vee F \mid \neg F \\ & & \mid \exists\, o \bullet F \mid \forall\, o \bullet F \\ atom & ::= & isInstance(o, c) \mid isRelated(r, o, o) \\ & & \mid o.a\, \Theta\, o.a \mid o.a\, \Theta\, k \end{array}$$

where $c : class, a : att, k : value$ are constants, $\Theta : value \times value \rightarrow bool$ is a comparison operator and $o : oid$ is a object variable. All variables except o in a query expression must be bound, and expressions must be *safe*, meaning that they should not return infinite sets.

Example Suppose classes `Audio` and `Wedding` exist, as does a relationship `MusicUsedInWedding`. The following query returns all music by the band Abba listened to while attending a wedding in Oslo:

$search($`Audio`, o`.artist = 'Abba'`
$\wedge\, \exists\, w(isInstance(w,$ `Wedding`$)$
$\wedge\, w$`.location='Oslo'`
$\wedge\, isRelated($`MusicUsedInWedding`$, o, w)))$.

4.3 Extending the Model with MDFS Transducers

In [15] we detail the extension of our model with an active component; due to space constraints we give an informal description.

Firstly, the model's class attributes are extended with *system* flags. The flag implies access restrictions that work in both the instance and the schema of the model. On the instance side, values of system attributes can be read but not modified by applications and end-users. Modification can only be done by the MDFS system as described further in this section. On the schema side, system attributes cannot be removed from classes, nor can they be modified.

Note that this is not a form of security (see Section 5), as access is not determined on the basis of a user's ID. Instead, this flag only denotes that some attributes are owned by the system and cannot be modified through the interface.

Secondly, classes are extended with behaviour in the form of MDFS *Transducers*. These are functions[2] that modify the values of an object's system attributes. The transducers are called each time an object's associated file stream is modified (hence also when the file is first created); to this end, the semantics of the file *write* operation mentioned in section 4.2.2 is modified.

Transducers automatically assign metadata that can (and indeed, should) be captured without user interaction. For example, keywords for a text document, the `from` field of email messages, or the dimensions of an image. A transducer recursively calls the transducer of its parent class, up to the top-most class T_C. This ensures that system attributes such as file size and access time are updated automatically, in this case by the transducers of F_C and T_C respectively.

Note that relationships between objects are not set by MDFS Transducers. Indeed, these constitute metadata that requires user interaction. Also note that if an application or end-user creates a new class that contains system attributes, a transducer for the class should also be supplied, otherwise the values of these attributes will always be set to NULL. Finally, transducers may (but are not required to) also set values of non-system attributes if they are currently NULL, but these may be overwritten by end-users.

5 Multiple Users, Multiple Volumes

The model defined in Section 4 is intended for a single user, single volume pure metadata file system. As Figure 2 shows, this basic (or personal) environment is but one of four possible combinations when the twin axes of number of users and number of volumes are considered. The ultimate goal of our research project is to define a metadata file system that allows multiple users and communication with other volumes, whether they be MDFS or other systems on the same computer or accessible over a network. In this section we briefly outline some of the issues involved.

5.1 Security Aware MDFS

When several users have access to the information stored in a single MDFS volume, the system will need to provide security. Traditional file systems employ a role-based security method [8] to control read and write access of files' contents. They also impose a very limited degree of security on metadata, in that they can make directories unreadable for some users, thereby hiding file names, sizes, etc. However, the main focus is on securing access to the file's bitstream.

[2] An MDFS transducer is a reserved class method similar to a *Constructor* or *Destructor* in Object-Oriented programming languages.

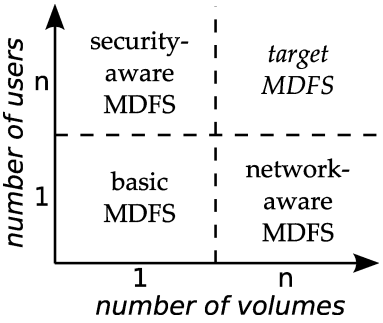

Figure 2: Extending the basic MDFS model to support multiple users and multiple volumes.

Interestingly, database systems also employ role-based security, but almost exclusively on the schema level. A database's INFORMATION_SCHEMA meta schema contains a table that stores users' privileges on such objects as tables and attributes. Databases do not offer security on the instance-level, where for example a user could be prohibited to read or write a tuple if the value of one of its attributes is a specific string.

For metadata file systems, we need role-based security mechanisms that work on the file contents level, the metadata level, and the schema level. It should also allow the use of instance-dependent access rules. For example, we may want to allow user A to write the content of file f_1 if the metadata field 'owner' is set to A's id, and the size of the file is less than 100 KB. A second user B may be allowed to read metadata associated with a file, but not to update it. Finally a user C may be allowed to create a new class by inheriting from an existing class in the MDFS schema, while other users are not.

We have studied instance-dependent access rules in the context of another research project [4] and aim to investigate their use in the context of metadata file systems.

5.2 Network Aware MDFS

There are several scenarios for the use of more than one file system volume. Consider that user A wants to copy a file from a remote network volume into his own MDFS volume. The remote volume may or may not be a MDFS system on its own, but will have some metadata associated to the file. In this scenario, where both the file content and metadata is copied to the local MDFS volume, the system, probably assisted to some degree by the user, needs to decide on a suitable mapping of metadata from the remote volume to the local volume.

Consider a second scenario, where two MDFS volumes exist and communicate via a network. Objects and files exist in one location, but both systems may use each other's objects and files. Clearly this scenario must solve the inherent schema integration problem [2].

In a third scenario user B may want to store all objects and files locally, but the metadata for an object is stored in a remote database. To make the example more concrete, suppose that a university stores data on its students in a central database, and individual lecturers have objects in their MDFSs for each of the students they teach. They can then create relationships between word processing documents and students in their local system. However, the student's metadata is accessed from the central database.

We note that Graffiti[14] is able to solve at least the problems of the first two scenarios, but only because of the simple and uniform tag metadata structure. The Graffiti server is able to synchronise files and metadata on two or more hosts because it does not have to deal with the issue of integrating file systems with different metadata structures.

These scenarios outline different solutions to the problem of letting a single MDFS volume communicate with other data systems. Ideally, a complete MDFS system will support each of the scenarios. We are currently investigating a range of solutions in this context.

6 Implementation Issues

6.1 Implementation Choices

Aside from storing files' binary content, a Metadata File System needs to store metadata and make it accessible through a query mechanism. The obvious target for implementing an MDFS is by using a database management system for the metadata, combined with a traditional file system to store file content. However, there are several possible avenues for implementing the model described in this paper. We briefly list some of them.

Employing Databases Since at the core our MDFS model is a subset of the ODMG Object Model, the preferred implementation platform is an ODMG-compliant object-oriented or object-relational database. Such a database is ideally equipped to handle class hierarchies and relationships. The actual binary content of individual files need not be stored in the database, however. It is sufficient to store a pointer to a file's inode in the database, and let the underlying file system handle subsequent file access. The database can natively handle queries, and also determine whether views (or *Virtual Folders*, as described in Section 7) are updatable. However, significant care must be taken in optimising the database and integrating it with the underlying file system.

Using WinFS Microsoft's WinFS was a data storage and management system based on a relational database. It is therefore similar to the previous option. We have reviewed the system in Section 2. The main advantage of using WinFS as an implementation base for a pure MDFS is that it natively supports many of the features that we require. However, the main disadvantage, apart from the incompleteness of the project, is that the metadata schema is not updatable at run time. This means that new classes must be compiled and made available as Dynamic-link Libraries. We argue that end users must be able to modify the schema (in particular the ability to create relationships) in order to achieve the full potential of metadata file systems.

Extending File System architecture It is possible to modify the way in which current file systems store metadata. For example, the POSIX inode structure could be extended to include a set of attribute-name/value pairs, and a set of pointers to other files. However, the notion of class with its specific attributes and its inheritance hierarchy would still be lacking, reducing users' power to model the metadata schema. On the other hand, query efficiency would be much less of an issue.

Use of links Soft links (or shortcuts) could be used to virtually place one file in various directories, the name of which represents a property of the file. As we

mentioned in the Introduction, managing the proliferation of links and directories would be a significant burden. In addition, relationships between files are difficult to represent in this manner.

Tagging File and directory names can be used to associate files with a set of tags. As mentioned in the Introduction, tagging is currently enjoying significant popularity in social web applications. Very likely this is due to the inherent simplicity of the method. Some of the metadata modelling power that we propose in this paper can be simulated by tagging, but concepts such as relationships pose a problem. In addition, querying and keeping tags consistent becomes rather convoluted.

6.2 Efficiency

This paper does not discuss efficiency issues as it concentrates on a model without bias towards any of the possible implementation platforms discussed in the previous section. However, it is clear that efficiency will be one of the deciding factors in the success or lack thereof of metadata file systems. We argue that current database technology is sufficiently far evolved to support the real-time data access needs required for this application. In addition, the LiFS [1] approach of storing metadata in new types of non-volatile main memory is a promising avenue of research.

6.3 Prototypes

Lasse At present we have finished work on a first tentative prototype of our model. The *Lasse* prototype was developed on top of a technology preview of Microsoft's WinFS. The main deliverable of *Lasse* was an MDFS File Browser application which allowed (1) the listing of objects in the MDFS file store, (2) a simplified mechanism to capture rich metadata (see Section 7), and (3) the creation of Virtual Folders (view definitions). The MDFS File Browser underwent a usability analysis by a number of staff in our department, which provided our project with crucial feedback to continue work on the model. The prototype also revealed the limitations of using WinFS as an implementation platform, mainly in the difficulty of letting users change the MDFS schema at run time. Screenshots of *Lasse* are included in Figures 3 and 4.

Sam We are currently working on a second proof-of-concept prototype, dubbed *Sam*. It is developed in Linux using the FUSE project which allows us to work in user-space and reuse common file browser components. We are using PostgreSQL as the database backend to store metadata, while file content itself will be stored in the default Linux file system. The goal of *Sam* is to further explore user interface issues; at first through a command-line interface, and later through a graphical layer.

7 User Interface Design Decisions

In Section 4.3 we introduced MDFS Transducers which, as in many commercial applications, handle the automatic capture of metadata without the need for user interaction. However, we argue that rich metadata such as links between objects cannot be captured automatically and requires user interaction. We also claim that without an efficient and effective generic GUI technique that helps end-users to rapidly capture rich metadata, MDFS technology will not be successful. In future work we will substantiate our claim by performing a usability study on a number of prototypes which we are implementing. In this section we briefly describe several aspects that have guided us in designing GUI operations for capturing metadata as well as using it in search. Again we refer the reader to [15] for more details.

Central in our approach is the concept of a virtual folder, or dynamic *View*. The visual presentation of a View is similar to a file browser (i.e. a table with one row per file and columns for metadata attributes) available in popular platforms, but Views have significantly different semantics. Files (or rather, fileable objects) may appear in more than one View, while being stored only once. A View is defined on the basis of a structured query (using the `search` function described in Section 4) and is refreshed each time an object (or set thereof) is "moved" in or out of the View.

The *move* operation has a new meaning compared to traditional file browsers, and plays a central role in our approach for capturing rich metadata through user interaction. Objects that are dragged into a View will acquire metadata that is needed for the objects to be in the result of the View's definition. If this process is successful the View, when refreshed, will include the new objects. Hence, the View is updatable. There are several cases in which the process can fail; (1) if the objects dragged into the View are of a different type than the View's definition, (2) if the View definition is an unupdatable query, and (3) if read-only attribute values must be changed in order for the objects to appear in the result.

Following [9] we allow more complex queries to be updatable than only those ranging over a single relation (class) and not using aggregation. In particular, the joins that we use in Views contain the equivalent of a `where exists` subquery and relate primary key (object identifiers) only. Consider the query expression given in Section 4.2.2 and the end-user wanting to drag a set of audio files into a View defined by that query. If there is more than one wedding in the system with `location = 'Oslo'`, the GUI will prompt the user for clarification. He may want to link the new audio files to just one such wedding (the GUI will display a list to choose from), or in some cases opt for associating the audio files with all weddings that took place in Oslo. As membership in the relationships is decidable at run time, such Views become updatable as well.

Figure 3 illustrates the use of Virtual Folders as a means to capture metadata through a move operation. The screenshots of the *Lasse* prototype show that initially four Photo objects were selected from the "Photos" Folder and subsequently dragged into the Virtual Folder "*Photos with Comments 'Family Holiday'*". The second screen then shows the content of the latter, and shows that the four objects have obtained the necessary metadata to belong in the Virtual Folder.

Views are persistent and can be organised into an arbitrary hierarchy by end-users but can also be organised automatically. Since all Views are subsets of the object store, a natural hierarchy based on set containment is implicit in the model. However, as View definitions are First-Order Logic expressions, View containment on the basis of their query definitions is undecidable. Hence, an automatic organisation of Folders will need to work on the instance level and reorganise the hierarchy each time the content of a View is changed. This is a potentially expensive operation that end-users should be able to switch off, but can be optimised by pruning parts of the hierarchy that have not changed.

A large number of other non-trivial issues are asso-

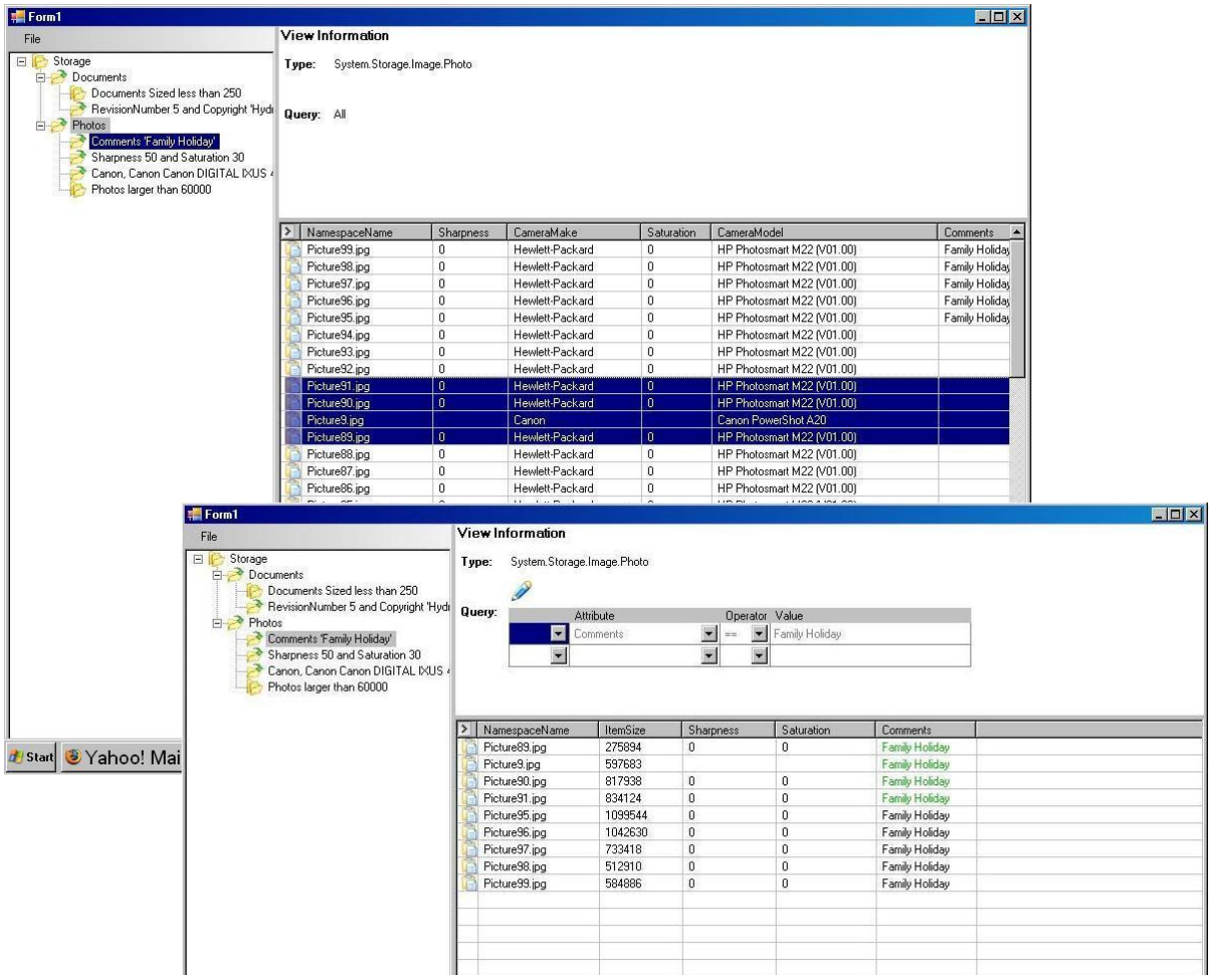

Figure 3: (a) Dragging photos into the Virtual Folder *Photos with comment 'Family Holiday'*, (b) Result after the drag operation, showing that metadata has been updated to make the photos appear in this Virtual Folder.

ciated with the operation of Views as a generic dual mechanism of querying the file store and acquiring new metadata for objects. For example, for a variety of reasons Views should return homogeneous sets of objects (as there always exists a common superclass for objects contained in a view), and it should be straightforward to decide which metadata attributes should be displayed to the user.

Deletion of Views should not result in the deletion of objects contained in the view, whereas the deletion of an object should also delete all relationship instances in which the object participated.

Creating Views should be possible in the GUI in a effective and simple manner. Users could be given a Wizard to create queries, use a Query-by-Example [19] interface (this is the approach we took in *Lasse*, see Figure 4 for a screenshot), employ a graph-based query language such as PaMaL [10], or orienteer [18] their way through class relationships to construct Views. Regardless of how views are created, users should see immediately whether their view is updatable. This facilitates the creation of a mental model so that users can employ the system more effectively.

Even when a very good structured query-definition interface exists, some users may opt to create a query consisting of key words. This could be either translated to a structured query, or information-retrieval algorithms can be used instead. Virtual Folders with an unstructured query definition can also be saved and placed in a hierarchy, but they are not updatable, and hence cannot be used to capture metadata.

These and other issues are detailed in [15].

8 Conclusion and Further Work

We have proposed a hierarchy of definitions for metadata applications and file systems based on a comprehensive review of existing implementations and research proposals. We then formally defined a model for a pure MDFS including data model, operations, and behaviour. Finally we described design aspects of a generic graphical user interface for capturing and using metadata in a pure MDFS. We are currently implementing a number of GUI prototypes, including those described here, and are planning a usability study with the aim to discover which strategies are most effective and efficient for capturing rich metadata through end-user interaction.

References

[1] Alexander Ames, Nikhil Bobb, Scott Brandt, Adam Hiatt, et al. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST05)*. IEEE, 2005.

[2] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

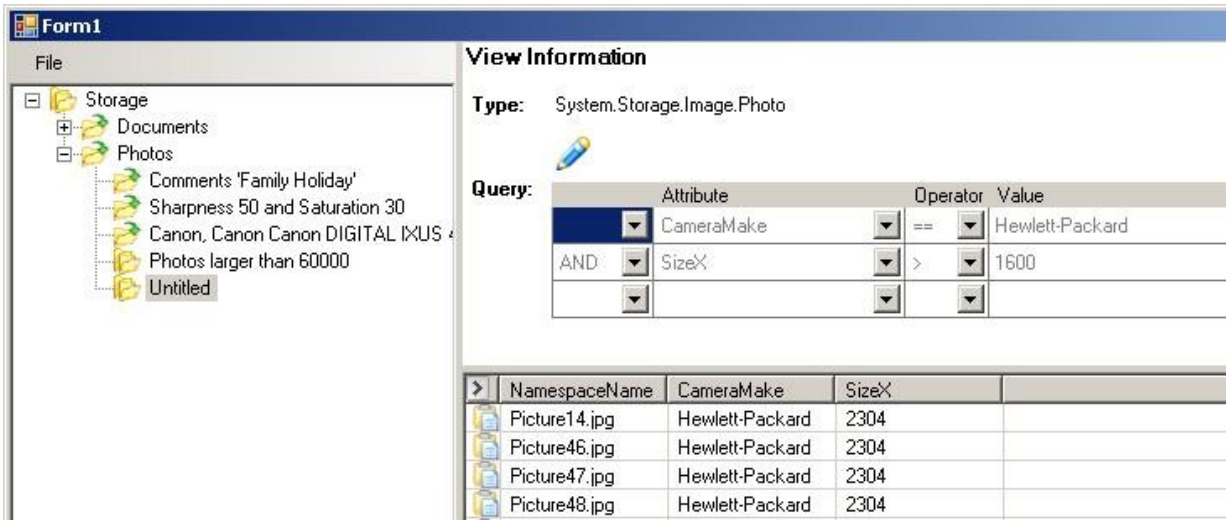

Figure 4: Creating a new Virtual Folder: query-by-example–like view definition interface.

[3] C. Mic Bowman, Chanda Dharap, Mrinal Baruda, Bill Camargo, and Sunil Potti. A file system for information management. In *Proceedings of the International Conference on Intelligent Information Management Systems, Washington D.C., USA, March 1994*, March 1994.

[4] T. Calders, S. Dekeyser, J. Hidders, and J. Paredaens. Analyzing workflows implied by instance-dependent access rules. In *ACM SIGMOD/PODS 2006 Conference*, Chicago, June 2006.

[5] R. Cattell, D. Barry, M. Berler, J. Eastman, et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, January 2000.

[6] S. Dekeyser. A metadata collection technique for documents in WinFS. In *Proceedings of the 10th Australasian Document Computing Symposium*. School of Information Technologies, University of Sydney, 2005.

[7] Susan T. Dumais, Edward Cutrell, Jonathan J. Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. Stuff I've Seen: a system for personal information retrieval and re-use. In *SIGIR*, pages 72–79. ACM, 2003.

[8] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[9] Antonio L. Furtado and Marco A. Casanova. Updating relational views. In *Query Processing in Database Systems*, pages 127–142. Springer, 1985.

[10] Marc Gemis and Jan Paredaens. An object-oriented pattern matching language. In *ISOTAS*, pages 339–355. Lecture Notes in Computer Science, Springer, 1993.

[11] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James O'Toole. Semantic file systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Asilomar Conference Center, Pacific Grove, California, October 13-16, 1991*, pages 16–25. ACM, 1991.

[12] David R. Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A general-purpose information management tool for end users based on semistructured data. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, pages 13–26, 2005.

[13] Graham Klyne. Resource description framework (RDF), February 2004. W3C Recommendation.

[14] Carlos Maltzahn, Nikhil Bobb, Mark W. Storer, Damian Eads, Scott A. Brandt, and Ethan L. Miller. Graffiti: A framework for testing collaborative distributed metadata. In *Proceedings in Informatics*, number 21, pages 97-111, March 2007.

[15] Lasse Motrøen. Metadata file systems and GUI operations. Master's thesis, University of Southern Queensland, Australia, 2007. Draft.

[16] Szabó Péter. MoveMetaFS – a searchable filesystem metadata store for linux. Freshmeat project `http://freshmeat.net/projects/movemetafs`, 2007.

[17] Richard Soley and William Kent. The OMG Object Model. pages 18–41, 1995.

[18] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In Elizabeth Dykstra-Erickson and Manfred Tscheligi, editors, *CHI*, pages 415–422. ACM, 2004.

[19] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.