An Engineering Document Management System[1]

Hannu Peltonen, Tomi Männistö, Kari Alho, Reijo Sulonen
Helsinki University of Technology
Laboratory of Information Processing Science
Otakaari 1, 02150 Espoo, Finland
E-mail: hpe@cs.hut.fi

Abstract

The implementation of an engineering data management system (EDMS) is described. The system models CAD drawings and other engineering documents as objects, which can be composed of subdocuments and have multiple versions and representations.

Documents contain user-defined attributes and the actual data for separate CAD tools. The documents are accessed with a server program, which stores document contents and attributes in a repository built on top of a relational database. The server accepts service requests from various client programs, including a graphical user interface.

Document approval and release procedures can be described by means of user-defined state graphs. Access rights are controlled in a flexible manner with an authorization program. Attributes can be treated as active data by means of procedures which enforce dependencies between attribute values.

1 Introduction

CAD programs make it easy to create and modify drawings and other technical documents. This may lead to uncontrolled proliferation of new documents and their versions. Well-established engineering practices, such as change check and approval procedures, are often ignored when they are not supported by equally powerful computer tools as document editing.

Engineering data management becomes increasingly important for CAD users. The loss of control over engineering documents is further aggravated when documents are stored on disks of individual workstations. It may be difficult to locate a particular document, let alone to know its current status (e.g., draft, approved, etc.).

This paper describes an engineering document management system (EDMS) which we have developed together with a large Finnish elevator manufacturer. EDMS has been

1. Presented at the ASME Winter Annual Meeting, New Orleans, Louisiana, November 28–December 3, 1993 (paper 93-WA/EDA-1)

built for production use at this company, but we believe that the system is flexible enough to be adapted to other similar organizations.

Documents and various attributes are stored in a centralized database. Documents can be searched with various criteria. In addition to keeping track of current work, this helps users to find old documents that could be reused with modifications in new designs.

EDMS mainly manages individual documents and their versions. However, we have started a new project which develops a database for product structures, components, configurations and other product related data. This new system will be linked with EDMS as each component in the product database can be associated with a set of documents in EDMS.

The rest of this paper is organized as follows. Section 2 briefly explains the background of the EDMS project. Section 3 describes the client–server architecture and the system components. The important question of integrating various CAD tools with the document database is discussed in Section 4. Section 5 describes the structure that EDMS imposes on documents. EDMS stores various data about documents as attributes treated in Section 6. Section 7 describes state graphs which can be used to track the design process. Section 8 introduces authorization procedures as a powerful mechanism for access control. Finally, Sections 11 and 12 summarize our plans for future development and provide conclusions.

2 Background

KONE Elevators is the third largest elevator manufacturer in the world with factories in several countries. Most of the drawings and other technical documents are produced on Unix workstations. Final approved drawings are microfilmed and stored in archives. Various attributes of each microfilmed drawing are stored in a database on a mainframe computer.

The goal of the EDMS project was to develop a system which stores the actual document contents in addition to attribute data. Moreover, the system should not only store the final documents but also documents which are still under preparation in various stages of the document approval cycle. The system should also be independent of the design tools with which the documents are prepared.

KONE Elevators investigated commercially available document management systems, but did not find any system that was satisfactory for their requirements. Our research group had previous experience of building databases for engineering data and a joint project with KONE Elevators was started.

3 System Architecture

3.1 Server and Clients

Figure 1 shows the main components of EDMS. All data, i.e., the actual document contents and various attributes, are stored in a commercial relational database. Document contents are stored in the database as *binary large objects* (BLOBs).

The database is managed by the *server* program, which waits continuously for *service requests* from *client* programs. Clients include the *user interface* program [2] and the *administrator's tool* [10]. A design tool can also become a client with the help of the *EDMS C-library.*

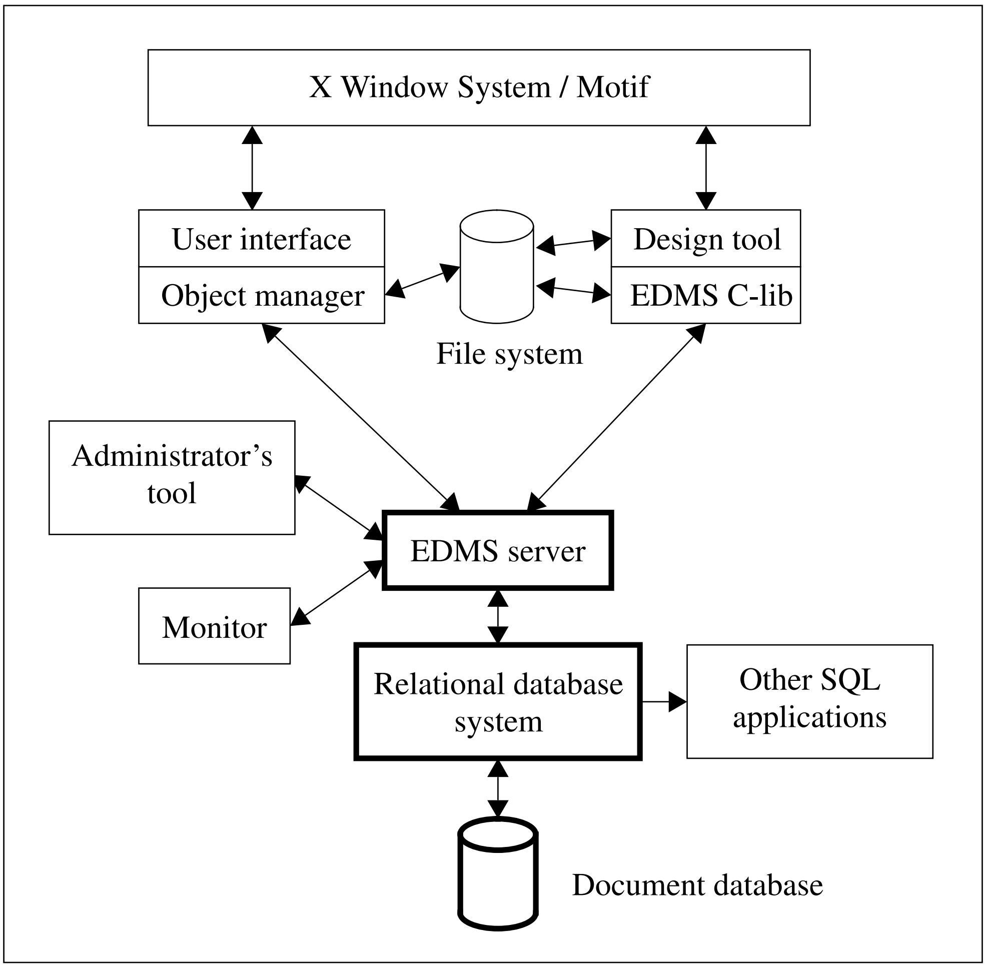

Figure 1: System architecture

To access the database, a user starts the user interface or some other client program on a workstation. The program connects to the server and sends it service requests according to user actions. Several clients can be connected to the same server at the same time.

The server and the clients run on Unix workstations which are typically located within a local area network. The programs communicate using TCP/IP sockets.

3.2 Server Protocol

The requests that the clients can send to the server are defined in the *server protocol* [10]. The protocol only refers to EDMS concepts introduced in this document and is independent of the relational tables where the server actually stores the data. The server could therefore be implemented on top of another database system without changing the server protocol.

Each request corresponds to a single logical operation, such as "create document version", which typically involves updating many tables in the database. Since the server executes each request as a separate transaction, the database is not modified at all if a request cannot be carried out completely.

A client can send a number of requests as a *compound request*, which is executed as a single transaction by the server. For example, the user interface provides operation "create a new document and its first subdocument". This effect is achieved with two *create* requests which are sent as a single compound request to the server.

The protocol also provides a uniform interface to various attributes. For example, a document can contain an attribute which stores a single character string and another attribute which has a set of strings as its value. Although the server stores the attributes in different ways, they are treated in much the same way in the protocol.

3.3 Data Transfer

Unless a design tool communicates directly with the server, a user who wants to read or modify a document must first copy the contents of the document from the server to an ordinary file with the EDMS user interface. The file can then be read and written by the tool. Finally, the user copies the modified file back to the server with the EDMS user interface. Some possibilities for simplifying this procedure will be presented in Section 4.

The server starts a separate copying process for each client. The process reads document contents from the database and sends them to the client over a separate communications channel. This allows the server to act on requests from other clients while a document is being copied for one client.

Documents are copied from a client to the server in a similar way. The user interface compresses the data before sending it to the server. Compressed data is automatically restored by the user interface when the data is copied back to the client.

3.4 User Interface and Object Manager

The user interface program is divided into two parts. The actual user interface part communicates with the user using X Window System and Motif toolkit, and accesses the database by means of the *Object Manager*, which is a class library written in C++ language. The actual user interface thus sees a high-level interface and does not know anything about the server protocol.

To reduce data transfer between the user interface and the server, the object manager includes a *cache*, which stores data that user interface has read from the server. A cache creates a consistency problem. For example, suppose the cache of a client contains a list of the versions of a particular document. If another client creates a new version of the same document, the first client must be informed about this so that it can add the new version to its own cache.

To solve this problem, a client can ask the server to send asynchronous *notifications* about changes in the database [1, 6]. In the above example, the object manager of the first client *registers* the document in the server. Whenever the document is modified, for example by creating a new version of it, the server sends a notification to all clients that have registered the document. When the object manager receives the notification, it updates the cache to reflect the current state of the database. The object manager also sends a further notification to the actual user interface so that it can update the windows.

3.5 Access Interfaces

To summarize, the document database can be accessed through various interfaces. The lowest interface is provided by SQL. To use this interface, the user or application programmer must know in detail how EDMS stores the data in the tables of the relational database. For example, we have built a document delivery system which reads the database directly and determines which new document versions should be delivered to interested users.

The second interface is the server protocol. It is safe and easy to use because the server maintains database consistency and the protocol hides implementation details. One of the clients that use the server protocol is a "batch" loader program, which reads a formatted text file and sends requests to create documents in the database.

The third interface is the Object Manager. It is a convenient class library for C++ programmers. The Object Manager reduces communication between the server and the client by means of the cache and server notifications. Design tools can also use a C-library to send requests to the server.

Finally, ordinary users access the database with the graphical X/Motif interface.

4 Tool Integration

Section 3.3 explained how a document must be copied from the database to a file before it can be accessed with a design tool. This can be simplified to some extent by storing the name of the design tool as an attribute of the document. After retrieving a document from the database to a file, the user can ask the EDMS user interface to start the tool with the file name as a command line argument. However, the user must still use the EDMS user interface to copy the document back to the database, and to retrieve new documents after the tool has been started.

Any EDMS operation can be added to a tool by modifying the tool to send appropriate requests to the server with the EDMS C-library. The library is written in the "C" language and includes functions for basic EDMS operations, such as *create document* and *change attribute values*. The functions send requests and read the replies, thus making it unnecessary for the tool developer to know the server protocol.

Although this solution allows smooth integration between the tool and EDMS, it is often infeasible in practice because it requires close co-operation with the developers of the tool. Fortunately, we are able to test this concept because most of the drawings at KONE Elevators are made with a CAD program developed by a company which takes part in our project. We have supplied the EDMS C-library to this company and they are building a direct link between the CAD program and EDMS.

5 Object Kinds

EDMS defines *object kinds*, such as *document* and *document version*. The object kinds, their relationships, and the operations that can be applied to objects of each kind are fixed in EDMS. Users cannot define new object kinds or new operations for the existing kinds. The suitability of EDMS for a particular organization depends very much on how well the engineering concepts and procedures in the organization match, or can be made to match, the EDMS object kinds. The data model may seem rather complicated, but it is based on the requirements of a large industrial company.

Note that the term *object kind* refers to the fixed classification of objects in EDMS. Within each object kind, however, the EDMS administrator can define various *object types* as will be explained in Section 6. For example, object kind *document* can have object types *drawing* and *manual*.

Figure 2 shows the object kinds and their relationships. The cardinality of a relationship is marked as “x:y” where “1” stands for “exactly one”, “m” or “n” stands for “zero or more”, and “m+” or “n+” stands for “one or more”. The names of the relationships are written in italics and should be read in the direction of the arrows. For example, the arrow from a document version to a subdocument version with cardinality “m+:n” and the name *includes-version* means that a document version can include many subdocument versions, and one subdocument version can be included in many document versions. A document version need not include any subdocument version, but a subdocument version is always included in at least one document version.

Note that there is no established terminology in this field; the EDMS terms, such as *document* and *version*, may mean something quite different in other similar systems. For example, what we call *versions* are sometimes known as *issues* or *revisions*.

More details of the EDMS objects and their behavior can be found in EDMS Architecture and Concepts [9].

5.1 Documents

Documents represent any data for design tools: drawings, manuals, bitmap images, etc. Originally we only used the concepts of a document and document version. However, our industrial partner turned out to require a considerably richer document structure. A document therefore includes a number of subdocuments, each subdocument has a number of subdocument versions, and each subdocument version has a number of representations.

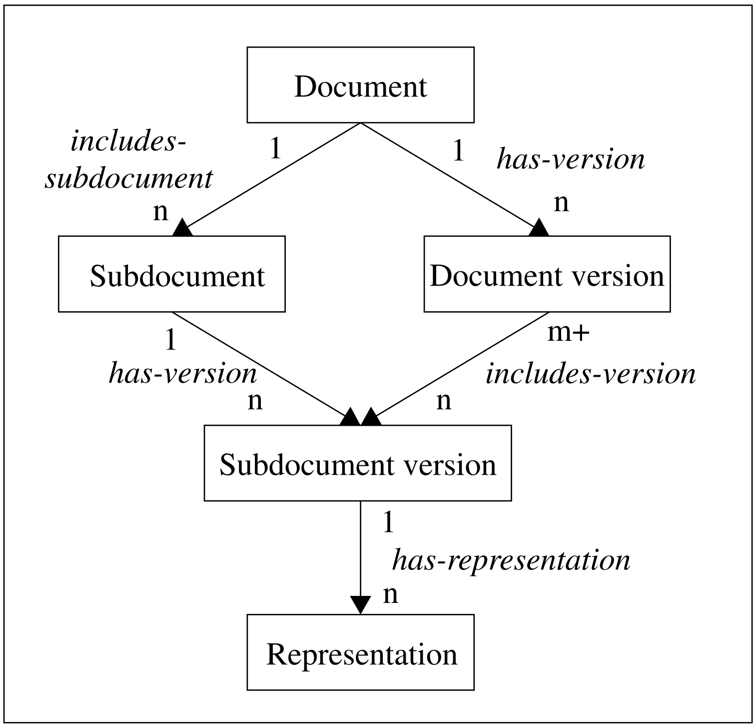

Figure 2: Object kinds

Only the last object in this hierarchy, the representation, stores the actual data for a tool, for instance a drawing for a CAD program.

5.2 Subdocuments

Documents are composed of *subdocuments*. For instance, suppose a drawing comprises several sheets, each of which is manipulated as a separate file by the drawing tool. The sheets of the drawing as a whole form a single document, and each sheet is a subdocument. Subdocuments are also needed for a text document which includes figures made with a separate drawing program. The text file and the graphics files are stored in the database as separate subdocuments.

5.3 Document versions

A document has a number of *document versions*. Each version represents the document as it has existed at some point during its history through successive modifications. Document versions are generated explicitly by the users; the modification of document does not automatically create a new document version.

Each version can have another version of the same document as the *parent version*. All versions that have a certain version as the parent are *child versions*, or simply *children*, of the parent version.

A newly created document version has the same contents as the parent version. The contents of the child version can then be modified until the child is used as the parent of another version.

Document versions can have arbitrary names, which do not have to reflect the parent–child relationships between versions. Typically the versions are named with consecutive numbers or letters. This is implemented in the user interface program, which suggests a suitable name for a new version according to the parent version name and the adopted version naming policy.

5.4 Subdocument Versions

On the one hand a document is composed of subdocuments, on the other hand it has a number of document versions. Each subdocument therefore has *subdocument versions*, and each document version may include one version of each subdocument of the document.

When a new document version is created, the version is made to include the same subdocument versions as the parent document version. Only when a user modifies a particular subdocument for the first time in the child document version, the system creates a new subdocument version, and includes it in the child document version.

As a result, many versions of a document may include the same subdocument version. This is illustrated in Figure 3, which shows the document version window of the user interface. The upper part of the window displays the two versions, “fi.-.1” and “fi.A.1”, of document “0000123”.

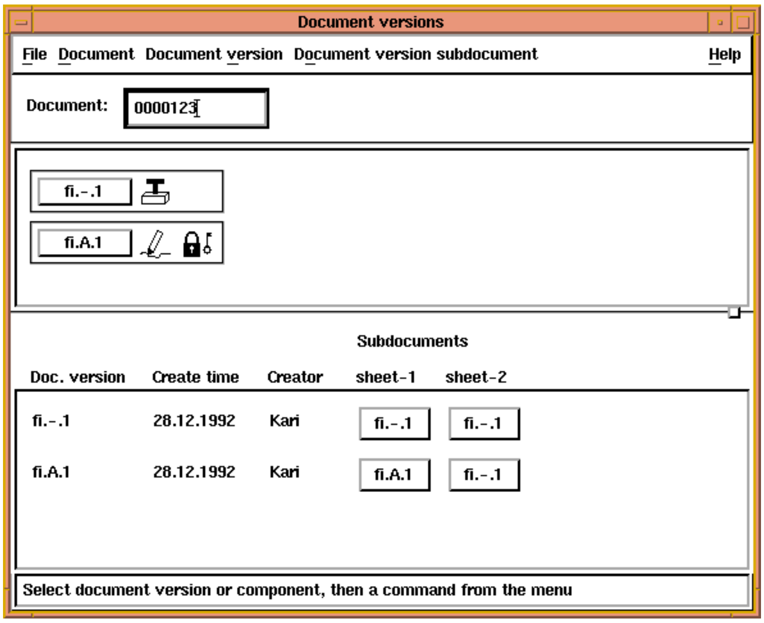

Figure 3: Document version window

For certain operations on document versions, the user must first *lock* the version. The lock and key next to version “fi.A.1” tell that this version is locked by the current user. (A lock without the key would mean that the version is locked by some other user.) The “rubber stamp” and “pen” indicate the current state of the document (Section 7).

The lower part of the window tells that the document has two subdocuments: “sheet-1” and “sheet-2”. Document version “fi.-1.” includes version “fi.-.1” of both subdocuments while document version “fi.A.1” includes version “fi.A.1” of the first subdocument and version “fi.-.1” of the second subdocument.

5.5 Representations

The actual data for the design tools are stored in the *representations* of the subdocument versions. A subdocument version has a single *primary representation* and optionally a number of *secondary representations*.

Only the primary representation is edited directly by the users; the secondary representations are generated from the primary one. In order to modify a primary representation, a user copies its *contents* from the database to an ordinary file, which can then be edited with a design tool.

As an example, suppose a subdocument version contains a drawing made with a CAD program. The file that can be edited with the CAD program would be stored as the contents of the primary representation of the subdocument version. Suppose further that the CAD program also generates a Postscript file and a bitmap file of the drawing. These files could then be stored as secondary representations for tool-independent document viewing and distribution. The secondary representations are never edited directly; they can only be re-generated after the primary representation has been modified.

6 Object Types and Attributes

Each object (document, document version, representation, etc.) in the database is of a particular *object type*. The EDMS administrator can define several object types, such as *drawing* and *manual*, with different properties.

Various data about objects are stored as *attributes*. All objects of the same object kind (e.g., all document versions) have the same *system attributes*. These attributes are defined by EDMS, and their values can only be modified by the system.

The EDMS administrator can define both *common* and *type-specific* attributes. A common attribute is defined for an object kind; for example, all documents can have a particular attribute. A type-specific attribute is defined for an object type; for example, all versions of documents of the type *drawing* can have a particular attribute.

The possible values that can be given to an attribute are specified by the *type* of the attribute. In addition to the familiar integers, character strings, etc., EDMS includes some less common types described below.

Sometimes the value of an attribute must be chosen from a set of character strings. The allowed choices can be stored in the database as a separate *value list* (e.g., the names of the departments of a company). The definition of a *list item* attribute (e.g., the *department* attribute of a document) only includes the name of such value list. Several attributes can use the same value list. When a list item attribute is given a new value, the system checks that the value exists in the corresponding value list. Values can be added to the value list when new choices for the corresponding list item attributes become available.

The value of a *text* attribute can be any printable ASCII text. Attribute definition does not specify any maximum length for the value. A *data* attribute is similar but can contain any data. The actual documents, for example, are stored as an attribute of this type. The main limitation of text and data attributes is that they cannot be used in database queries. It is impossible to search for objects where the value of a text attribute contains a particular substring.

The type of an attribute can also be a *set* of values. For example, the type of an attribute can be "set of integers in the range 0–10" or "set of values from the value list called *departments*".

The EDMS administrator can change the type of an existing common or type-specific attribute in a "sensible" way. For example, the maximum length of a string attribute can be freely increased, but before decreasing the length, the system checks that the current value of the attribute would not have to be truncated in any object. The administrator can also add a new common attribute to an object kind or a new type-specific attribute to an existing object type.

Figure 4 shows the window displayed by the user interface for creating a new document. The list at the top of the window shows the available document types, from which the user has selected the type *drawing*. The user should then enter values for the common and type-specific attributes of this document type, which are shown in the lower part of the window. The zoom buttons open new windows for entering attribute values which are not written directly to the input field. For example, a zoom button for an attribute with the type "item from the value list called *drawing_types*" could open the window shown in Figure 5.

When started, the user interface reads the object types and their attributes from the server. After the EDMS administrator has added a new attribute to the database, the attribute automatically becomes visible the next time the user interface is started.

7 States

An object can be in different *states* during its lifetime. The possible states and the allowed transitions between states are specified by with a *state graph*. Figure 6 shows a possible state graph for a document version.

After a version has been created, it is in the state *draft*. The version is moved by its designer to the state *ready* when he or she considers it ready to be checked. If the person responsible for checking the version accepts it, he or she moves it to the state *checked*; otherwise he or she returns it back to the state *draft*. A checked version can be approved, moving it to the state *approved*, from which it never moves to another state.

The EDMS administrator can define several state graphs and associate them with different object types. Whenever a new object is created, it is moved to the first state of the graph associated with the object type.

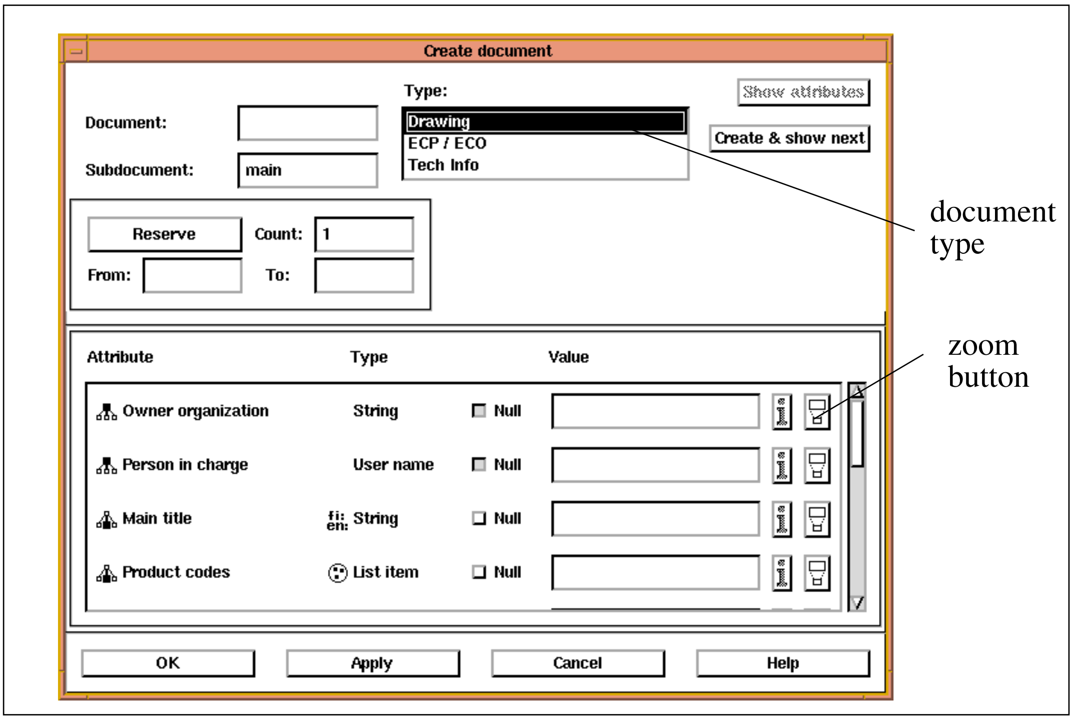

Figure 4: Document creation window

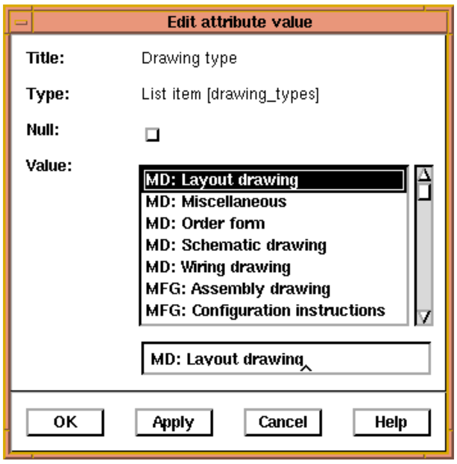

Figure 5: Attribute value entry window

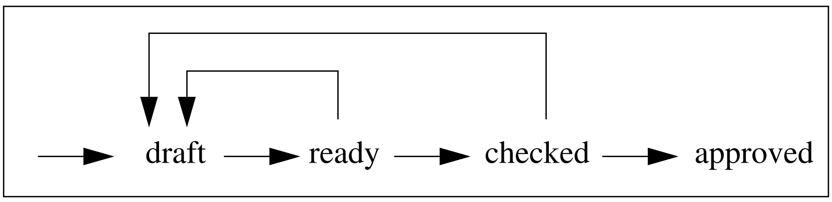

Figure 6: Sample state graph

Often the allowed operations on an object depend on its state. For example, one may say that a document issue cannot be locked unless it is in the state *draft*, or that the parent issue of a new document issue must be in the state *approved*. This kind of rules can be defined with the authorization program described in the next section. The states and rules can, for example, be used to implement procedures for the ISO 9000 standard.

8 Authorization and Commit Programs

This section describes how EDMS determines whether a particular user is allowed to apply a particular operation to a particular object. For example, one must be able to specify who is allowed to approve a document version, i.e., to move it to the state *approved* in the sample state graph of Figure 6.

Since the authorization rules vary considerably between organizations, we did not want to define a fixed format for them. For example, it would be too rigid to define the rules as a table with entries of the form "user X is allowed to perform operation Y on an object of the type Z which is in the state W".

8.1 Authorization Program

Access rights are controlled by means of an *authorization program*. The program consists of *authorization procedures* written by the database administrator in a powerful procedural language which includes operations for reading attributes and other properties of objects. An example of an authorization program is shown later.

When a client attempts to perform an operation, the server executes an authorization procedure to determine whether the operation should be allowed. The procedure terminates when it executes either a *reject* or an *accept* statement. A *reject* statement can include a text string which tells more specifically why the operation was rejected.

In addition to attributes of EDMS objects, the procedures can read additional authorization data from separate relational tables, which are stored in the same database with the documents.

8.2 Sample Authorization Program

As an example, suppose that each document belongs to a particular department and has an attribute called `dept`. Each department has a manager. The names of department managers are stored in the table `dept_manager(dept, manager)`. We want to express the rule that a document version can only be approved, i.e., moved to the state *approved*, by the manager of the department to which the document belongs.

When the server receives a request for changing the state of any object, it calls the `changeState` authorization procedure in the authorization program. The procedure has parameters for the name of the user attempting to change the state, the identifier of the object, and the state to which it should be moved.

```
(1)   event changeState (user, object, new)
(2)     test kind(object)
(3)     when "document":
(4)       …
(5)     when "doc-version":
(6)       newDocVersionState(user, object, new);
(7)     when "subdoc":
(8)       …
(9)     end;
(10)  end;
(11)
(12)  procedure newDocVersionState (user, version, new)
(13)    test new
(14)    when "ready":
(15)      …
(16)    when "approved":
(17)      doc_name := version.doc;
(18)      doc := document(doc_name);
(19)      dept := doc.department;
(20)      mgr := [manager from dept_manager
(21)              where dept = :dept];
(22)      if user = mgr then
(23)        accept;
(24)      else
(25)        reject "version must be approved by ", mgr;
(26)      end;
(27)    end; -- test new
(28)  end; -- newDocVersionState
```

On line 20, the manager of the department is found with an SQL *select* statement written in square brackets without the key word `select`. The item `:dept` is replaced with the value of the variable `dept`. The statement must select exactly one value (one column of a single row) from the database.

On line 22, the name of the user who tries to change the object state is compared to the name of the project manager. If the names are equal, the request is accepted. Otherwise, the request is rejected by means of a `reject` statement on line 25 and the error message is displayed by the user interface.

8.3 Commit Program

Sometimes users would like certain EDMS requests to automatically update some attributes. For example, suppose document versions are associated with a state graph that includes states *draft* and *checked* besides various other states. Document versions include attribute *checked_by*, which should be updated automatically as follows:

- When a user moves a document version to the state *checked*, his or her name is stored in *checked_by*.

- When a document version is moved to the state *draft*, attribute *checked_by* is given a null value.

To achieve these kinds of effects, the EDMS administrator can specify a *commit program* for the server. A commit program is similar to an authorization program. However, whereas an authorization program allows users to describe conditions under which a server request is accepted or rejected, a commit program allows the administrator to associate additional attribute modifications with a server request.

The name *commit program* comes from the fact the server executes the commit procedure immediately before committing the transaction. Attributes that are modified by a server request thus have their new values when the corresponding commit procedure is executed. This is in contrast to authorization procedures, in which the attributes still have their original values.

The automatic modification of the *checked_by* attribute is achieved with the following commit procedure:

```
(1)     event changeState (user, obj, newState)
(2)       if kind(obj) = "doc-issue" then
(3)         test newState
(4)         when "draft":
(5)           update obj.checked_by := NULL;
(6)         when "checked":
(7)           update obj.checked_by := user;
(8)         when "approved":
(9)           designer := document(obj.doc).designer;
(10)          mail designer: obj, " was approved by ", user
(11)        end;
(12)      end;
(13)    end;
```

Attribute value is changed with the *update* statements on lines 5 and 7. A commit procedure can also send electronic mail. For example, the above procedure sends mail to the designer when his or her document version is approved.

9 Database Design

This section describes very briefly how EDMS stores its data in the relational database. The mapping between the EDMS data model and the relational tables is hidden inside the server; EDMS clients do not need this information.

Information about object types and attributes (Section 6) is stored in various tables which are created together with an EDMS database.

For each object type (e.g., a document type or a representation type) there is a table that has a row for each object of that type. Upon receiving a request for a new object type, the server creates the corresponding table with columns for all scalar (i.e., non set-valued) attributes of the type.

In addition to the scalar attributes, the table also has a column for an object number, which is an arbitrary integer created by the server. For each set-valued attribute there is a separate table with two columns: object number and a single element of a value of the attribute.

For example, suppose document type *drawing* has attributes *doc*, *organization* (string) and *delivery* (set of strings). The database has the following drawings:

doc	organization	delivery
abc	R&D	manufacturing, sales
xyz	personnel	manufacturing

This data would be stored in the following tables:

types (object types)		
type_no	kind	type_name
1	d	drawing

attrs (attributes)				
attr_no	kind	obj_type	attr_name	…
123	d	SYSTEM	doc	…
124	d	drawing	organization	…
125	d	drawing	delivery	…

d_drawing (documents of the type *drawing*)		
obj_no	a_doc	a_organization
1	abc	R&D
2	xyz	personnel

set_125 (set-valued attribute number 125)	
obj_no	value
1	manufacturing
1	sales
2	manufacturing

The actual document contents are stored in the system attribute *contents* of a representation. This attribute is of the the type *binary large object* (BLOB). If EDMS is ported to a database system without BLOBs, one could store representation contents in ordinary files and the names of these files in the database. BLOBs, however, have simplified the implementation because contents can be treated in many respects in the same way as other attributes. In particular, the contents are updated under the same transaction mechanism as any data in the database.

The server creates indexes for all identification attributes (e.g., system attributes *doc* and *version* of document versions). Other indexes can be created with server requests.

10 Related Work

According to Katz [7], engineering data is characterized by three relationships between data objects. In Katz's paper these data objects mainly correspond to products instead of documents. The general concepts, however, can be applied to documents as well.

Firstly, an object can be a *version* of a generic object. Two versions of the same object can further related by the fact that one version has been *derived* from the other one. Documents and document versions in EDMS obviously correspond closely to generic objects and their versions.

Secondly, an object can be a *component* of another object [8]. EDMS subdocuments can be regarded as components of a document. However, a subdocument cannot be shared as a component in multiple documents, and a subdocument cannot be divided into smaller components.

The third main concept is *equivalence*. Two database objects can be equivalent in the sense that they are representations of the same concrete design artifact. The representations of a subdocument version (Section 5.5) implement a simple form this concept in EDMS. Note that the term *equivalence* can be considered somewhat misleading because it suggests that the relationship between the equivalent representations would be symmetric. Often, however, there is a fixed order in which one representation can be generated from another. For example, a Postscript representation can be generated from a CAD drawing, but not vice versa.

Many versioning systems incorporate some notion of version states. For example, the version model of Chou and Kim [5] defines the states *transient*, *working* and *released* (corresponding roughly to the states *draft*, *ready* and *approved* in the sample EDMS state graph of Figure 6). The version model also defines the rules for state transitions and the operations that can be applied to versions in different states. In contrast to this approach, EDMS provides a general mechanism for defining appropriate state graphs. The authorizing and commit programs allow the EDMS administrator to specify how versions should behave in different states.

Different organizations have very different authorization requirements. We believe that the required flexibility is more easily achieved with the authorization program of EDMS than with authorization schemes based on more fixed concepts and rules [12].

EDMS maintains the consistency of data accessed concurrently by several users with the notification mechanism [1]. This problem has been investigated in a more general setting by Hall [6].

EDMS is implemented with a traditional relational database system. The maturity of the relational technology encourages one to build a richer data model on top of them [3, 11]. On the other hand, object-oriented databases may prove more appropriate for this kind of application in the future [4].

11 Further Work

So far we have assumed that all documents are stored in a single database managed by a single server. Large organizations, however, often want to distribute the data in many separate databases, which may even be located in different countries. This leads to the common problems of distributed data management.

The data model must be developed further. The current version of EDMS may be unnecessarily complex for some organizations. For example, subdocuments or multiple representations may not be needed by all potential users of the system. On the other hand, a new mechanism is clearly needed for shared subdocuments. For example, users may want to include a single graphics file in many documents.

Independence from design tools is one of the main principles of EDMS. Nevertheless, it is useful for the system to know something about the stored documents. For instance, various attributes of a drawing are stored in the database, and some of these attributes are also available in the drawing itself. EDMS should be able to extract this information from the drawing with a separate tool-specific program. Similarly, data should also be moved in the other direction.

12 Conclusions

We have defined a data model for engineering documents. The model has been implemented for production use on top of a relational database system. The system makes it possible to establish and enforce document approval and release procedures. These procedures, however, become useless unless they are integrated in the applications.

The centralized repository helps to solve the problem of locating documents which would otherwise be dispersed in personal workstations. Organizing documents like this gives new possibilities for document retrieval, searching, delivery, etc.

The system has been built to be dynamic so that it could be adapted to other organizations that handle large amounts of engineering documents.

Acknowledgments

The system was developed as a joint project with KONE Elevators. The project was partly funded by the Finnish Technology Development Centre (TEKES). We gratefully acknowledge the guidance and support from Asko Martio and Matti Eskola at KONE Elevators.

References

[1] Alho, K., , H. Peltonen, Männistö, T., and Sulonen, R.. "An Approach for Supporting Inter-Application Consistency". In Proc. of *The 2nd IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 1993.

[2] Alho, K.. *EDMS User's Reference*. Technical Report TKO-C58. Laboratory of Information Processing Science, Helsinki University of Technology. 1993.

[3] Barsalou, T. and Wiederhold, G. "Complex Objects for Relational Databases." *Computer-Aided Design*, Vol. 22, No. 8, October 1990.

[4] Cattell, R.G.G. *Object Data Management*. Addison-Wesley, 1991.

[5] Chou, H.-T. and Kim, W. "A Unified Framework for Version Control in a CAD Environment." In *Proc. of the 12th International Conference on Very Large Databases*. 1986.

[6] Hall, K. *A Framework for Change Management in a Design Database*. PhD thesis, Stanford University, 1991.

[7] Katz, R. "Toward a Unified Framework for Version Modeling in Engineering Databases." *ACM Computing Surveys*, Vol. 22, No. 4, December 1990.

[8] Kim, W., Bertino, E., and Garza, J. "Composite Objects Revisited." In *Proc. of the International Conference on the Management of Data (SIGMOD)*. 1989.

[9] Peltonen, H. *EDMS Architecture and Concepts*. Technical Report TKO-C59. Laboratory of Information Processing Science, Helsinki University of Technology. 1993.

[10] Peltonen, H. *EDMS Administrator's Guide*. Technical Report TKO-C60. Laboratory of Information Processing Science, Helsinki University of Technology. 1993.

[11] Premeriani, W.J., Blaha, M.R., Rumbaugh, J.E., and Varwig, T.A. "An Object-Oriented Relational Database." *Communications of the ACM*, Vol. 33, No. 11, November 1990.

[12] Rabitti, F., Bertino E., Kim, W., and Woelk, D. "A Model of Authorization for Next-Generation Database Systems." *ACM Transactions on Database Systems*, Vol. 16, No. 1, March 1991.