EDMS

Engineering Data Management System

Architecture and Concepts

Hannu Peltonen

Helsinki University of Technology

Department of Computer Science

Laboratory of Information Processing Science

Abstract

The engineering data management system (EDMS) manages CAD drawings and other engineering documents, 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 relational database and 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 written by the users in a high-level language.

Acknowledgements

The system was developed as a joint project with KONE Elevators. The project was partly funded by the Finnish Technology Development Centre (TEKES). The team at Helsinki University of Technology was headed by professor Reijo Sulonen and has included Kari Alho, Tomi Männistö, Vesa Kautto, Tian Jie, Mojgan Otterström and the author of this report. At KONE Elevators the project was started and guided by Asko Martio and Matti Eskola.

Contents

1 Introduction 1

2 System Architecture 2

2.1 Server and Clients 2

2.2 Server Protocol 3

2.3 Client Types and Access Rights 3

2.4 Data Transfer 4

2.5 Other Components 4

3 Tool Integration 5

4 Object Kinds 6

4.1 Documents 7

4.2 Subdocuments 7

4.3 Document versions 8

4.4 Subdocument Versions 8

4.5 Representations 9

4.6 Example of Document Version Use 10

4.7 User Objects 11

4.8 Projects 11

5 Object Types 12

6 Attributes 13

6.1 Attribute Classes 13

6.2 Attribute Names 14

6.3 Attribute Titles 14

6.4 Attribute Descriptions 14

6.5 Attribute Types 15

6.6 Null Values 17

6.7 Display Columns 17

6.8 Display Precision 17

6.9 Attribute Definition Changes 17

6.10 Creation and Deletion of Attributes 17

6.11 Default Objects 18

7 Type Parameters 19

8 Document Management 20

8.1 Main Concepts 20

8.2 Document Creation 22

8.3 Subdocument Creation 22

8.4 Document Version Creation 23

8.5 Document Version Locking 24

8.6 Subdocument Version Creation 25

8.7 Shared Subdocuments 25
8.8 Representation Creation 26
8.9 Representation Copy-Out 27
8.10 Representation Copy-In 27
8.11 Secondary Representations 28
8.12 Representation Checkpoints 29
8.13 Representation Discard 29
8.14 Read-Only Copies 29
8.15 Subdocument Dropping and Adding 30
8.16 Document Version Unlocking 31
8.17 Object Deletion 31

9 States 33

10 Authorization Program 35
10.1 Authorization Program 35
10.2 Sample Authorization Program 35
10.3 Limitations of the Authorization Mechanism 37
10.4 Control of Reading Rights 38

11 Commit Program 39
11.1 Transactions 39
11.2 Commit Program 39

12 References 42

Appendix A : System Attributes 43

Appendix B : Name Constraints 48

Appendix C : Document Name Format 50

Index 52

1 Introduction

The Engineering Data Management System (*EDMS*) stores technical documents and their versions, controls concurrent access to the data, and provides a general authorization mechanism for specifying the rules according to which documents can be modified.

Section 2 describes the main components of EDMS. EDMS has a client–server architecture and it is built in a Unix environment on top of a relational database system. The latter choice was dictated by practical considerations, although it means that some useful concepts, such as general inheritance, have not been included in the system.

A document can store any data, such as a drawing, a text document, or the source code of a computer program. Documents are manipulated with *design tools*. For example, drawings and text documents are manipulated with CAD programs and word processors, respectively. The database is independent of the tools, although some co-operation between EDMS and the tools is desirable, as will be seen in section 3.

In addition to documents and document versions, EDMS defines a number of other concepts for document management. These concepts are introduced in sections 4 and 5.

All objects in EDMS, such as documents and their versions, have attributes for storing various data about the objects. The available attribute types and other properties of the attributes are described in section 6. Section 7 describes a mechanism for storing data which, unlike attributes, are not associated with any individual object.

Section 8 describes the details of document modification, locking, versioning, etc.

Section 9 introduces *state graphs* as a means for tracking the design process. A state graph defines a set of states and the possible transitions between states. A number of different state graphs can be defined, and each database object, such as a document version, can be associated with a suitable graph.

The conditions under which the data can be modified are described by means of a *authorization program*. The program is written by a system administrator in a language which includes operations for examining attribute values and control structures, such as conditional statements and iteration statements. An overview of the authorization mechanism is given in section 10.

Users can also specify that some attributes should be updated automatically as a result of some operations in the database. Section 11 shows how this is accomplished by means of a *commit program*.

2 System Architecture

2.1 Server and Clients

Figure 1 shows the main components of EDMS. All data, including the actual document contents, are stored in a commercial *relational database*. The representation of EDMS data in the relational database in described in the *EDMS Administrator's Guide* [2].

The database is managed by the EMDS *server* program, which runs continuously and receives *service requests* from EDMS *client* programs. Clients include the EDMS *user interface* program [1] and the *administrator's tool*. A design tool can also become a client with the help of the *EDMS C-library*. This will be discussed in section 3.

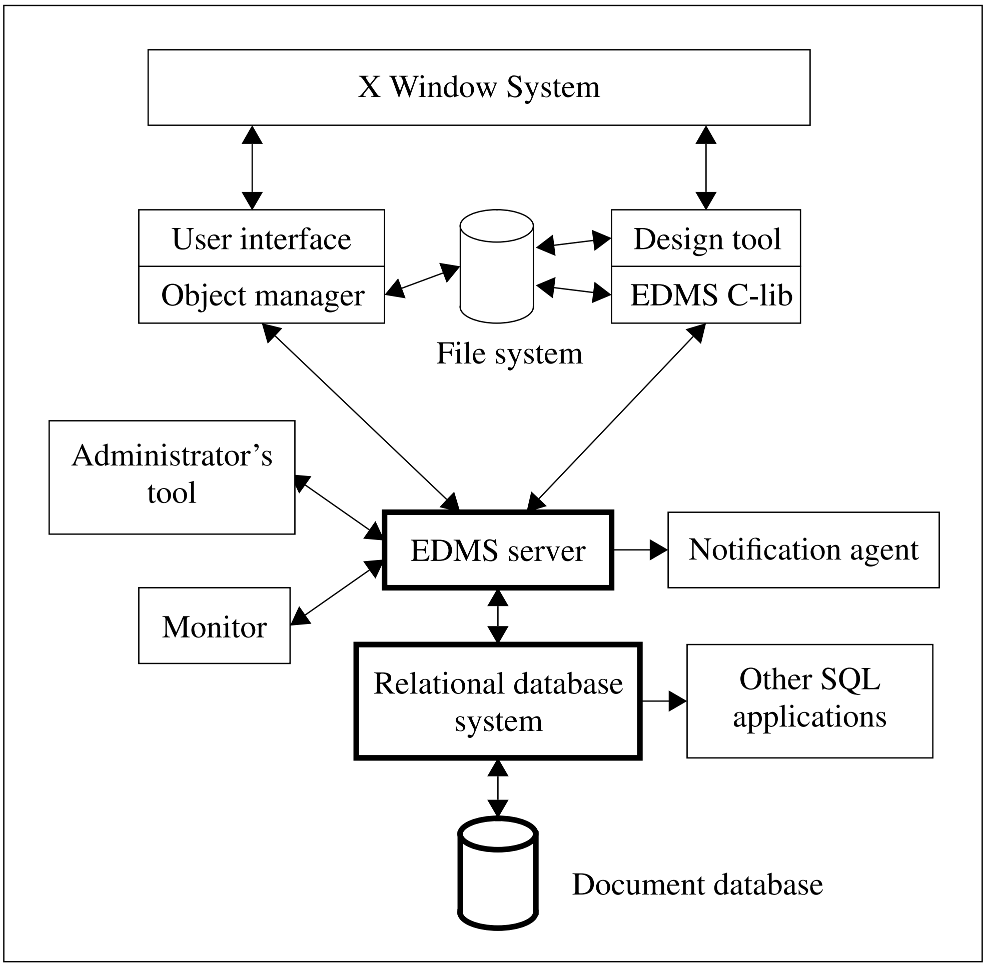

Figure 1: System architecture

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.

Both the server and the clients run on Unix machines which are typically located within a local area network. The programs communicate using TCP/IP sockets. Normally users run the user interface program on their own workstations. However, since the user interface is built with the X Window System, a user can have a non-Unix machine, for example, a PC, which only runs an X server while the interface program is executing on a Unix machine.

2.2 Server Protocol

The requests that the clients can send to the server are defined in the *server protocol* [2]. The protocol only refers to EDMS concepts introduced in this document and says nothing about the relational tables where the server actually stores the data.

The server executes each request as a separate transaction. If a request that modifies the database cannot be carried out completely, the database is not modified at all.

In a future version of the system it may be possible to combine a sequence of operations into a single request, so that either all or none of the operations are executed. Nevertheless, even then one cannot make an atomic transaction out of arbitrary requests which are sent separately by a client, and between which the server may receive requests from other clients.

The protocol includes requests for both "data manipulation" operations, such as changing the state of an object, and "data definition" operations, such as creating a new state graph. The latter requests are sent with a separate client program, the *administrator's tool* [2].

The communications protocol between a client and the server is synchronous. The client sends a request and waits for the server send a reply. However, a client can also ask the server to send asynchronous *notifications* about changes in the database. This is necessary if clients store some data in their own data structures for efficiency reasons. For example, suppose a client stores a list of the versions of a document in its data structures. If another client creates a new version of the document, the first client must be informed about this so that it can add the new version to its own data structures.

2.3 Client Types and Access Rights

The server does not distinguish between different kinds of clients. All clients use the same protocol and can send any requests. However, a client can send a character string as its *client type* to the server. The server stores the types of all clients connected to it, and a client can read this information with a request. The administrator's tool, for example, can use client types to check that only one user at a time uses the program.

As will be explained in section 4.7, the server itself provides only a simple mechanism for controlling who can use EDMS at all. Section 10 describes the more powerful authorization mechanism, which determines whether a particular user is allowed to perform a particular operation.

2.4 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 3.

Document contents are copied from the server to a client with a *copy-out* request. The request can include the name of a file where the server writes the document contents. Obviously this presumes that the server is able to write a file which can then be read by the client, for example using NFS (Network File System). Alternatively, the client can open a separate connection to the server for data transfer. In this case the server and the client need not be able to access common files.

When document contents are copied from a user's file to the database, the present EDMS user interface allows the user to say that the data should be compressed with the standard Unix `compress` program. The user interface automatically decompresses the contents when the document is copied from the database back to a file.

Section 2.1 mentioned the possibility of using EDMS from a PC that runs an X server. Note that in this case the document contents are copied from the server to the Unix machine that runs the user interface program, not to the PC. The design tools on the PC must be able to read the file from the Unix machine.

The server starts a separate copying process for each client. The process reads document contents from the database and writes them to a file or to a separate communications channel, which is opened between the server and the client. This allows the server to act on requests from other clients while a document is being copied for one client. The copying process is described in more detail in the *Administrator's Guide* [2].

Documents are copied from a client to the server in a similar way. The subprocess which copies documents between the server and a client also executes object queries.

2.5 Other Components

If necessary, other SQL applications (in addition to the EDMS server) can read the relational database directly. These applications cannot, however, update the database because the server does not expect other programs to modify the data it has read from the database to its own data structures. One example is a document delivery system which reads the database directly and determines which new document versions should be delivered to interested users.

The server also accepts simple commands from *monitor* programs. There are commands for printing the names and machines of the clients connected to the server, for printing all requests that the server receives from the clients (mainly for debugging), etc.

3 Tool Integration

To modify a document, a user

(1) copies the document from the database to an ordinary file with the EDMS user interface,

(2) manipulates the file with a design tool, and

(3) copies the modified file back to the database with the user interface.

Steps (1) and (2) can be combined 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.

Some tools allow users to add new push-buttons and menu items which execute user-defined commands. By making these commands send suitable requests to the server, some EDMS operations can be executed directly from the tool. Of course, the tool must also be able to pass parameters, such as the name of the active document, to the user-defined command.

Any EDMS operation can be added to a tool if the tool is modified to send appropriate requests to the server. 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.

The modified tool can either send requests directly to the server, or it can call the functions of the EDMS C-library, which was shown linked to the tool in figure 1 on page 2. The library is written in the "C" language and it includes functions for basic EDMS operations, such as *create document* and *change attribute values*. The functions send requests to the server and read the replies, thus making it unnecessary for the tool developer to know the server protocol.

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.

It is also useful if the EMDS can move data from the database to a document. For example, when the server marks a document as approved, it should also automatically insert the text "Approved by …" in the document.

4 Object Kinds

Data is stored in the database as *objects*. EDMS defines the following *object kinds*:

- document
- subdocument
- document version
- subdocument version
- representation
- user
- project

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.

All objects in the database belong to some object kind and, within the kind, to some *object type*. For example, an object of the kind *document* can be of the type *drawing*. Object types are defined by the users[1] and will be treated in section 5.

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.

1. As was mentioned in section 2.3, the server does not recognize different user categories. When this document says that users, for example, define object types, it means that the server protocol includes request for defining new object types. In practice "ordinary users" do not send these requests. The distinction between "ordinary users" and users who are allowed to define object types with the administrator's tool can be made in the authorization program.

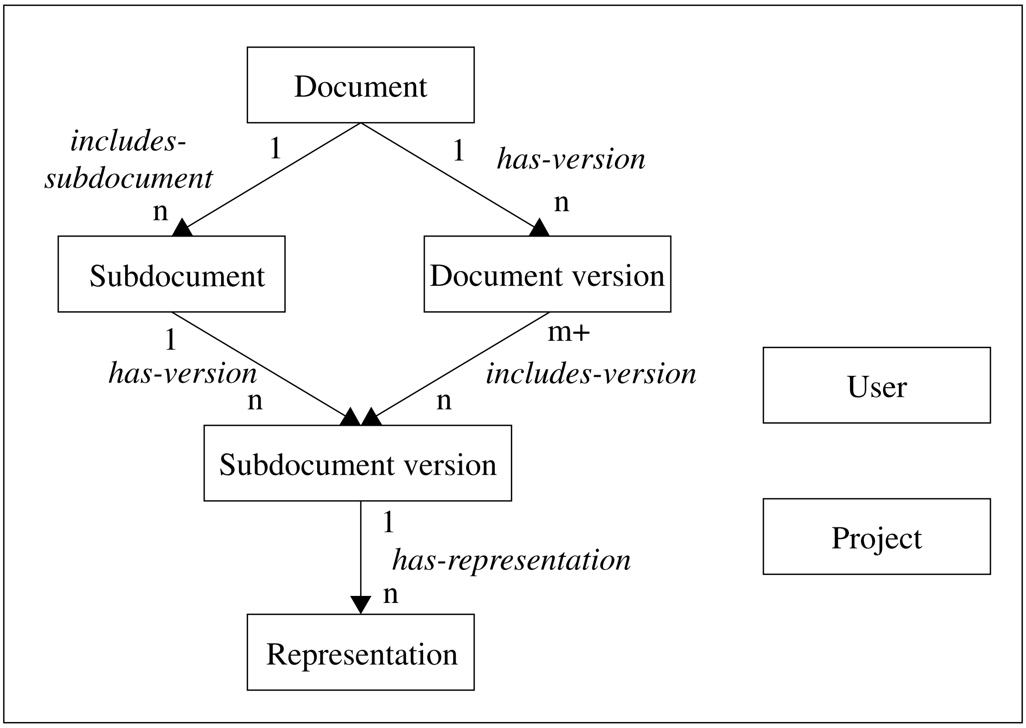

Figure 2: Object kinds

4.1 Documents

Documents represent any data for design tools: drawings, manuals, bitmap images, etc. However, document objects are not manipulated directly with the tools. A document includes a number of subdocuments, each subdocument has a number of subdocument versions, and each subdocument version has a number of representations. Only the last object in this hierarchy, the representation, stores the actual data for a tool, for instance a drawing for a CAD program.

Documents are identified with unique *document names*. The exact rules for the allowed names of documents, object types and other EDMS entities are given in appendix B.

4.2 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.

Note that a document always includes subdocuments. A document which need not be divided into several subdocuments consists of a single subdocument. To be precise, a newly created document does not contain any subdocuments, but at least one subdocument must be created before any data can be stored in the document.

Subdocuments can have arbitrary names as long as all subdocuments of a single document have different names.

4.3 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. Figure 3 shows the versions of a single document. Version A has two children: B and BB, and version B in turn is the parent of C.

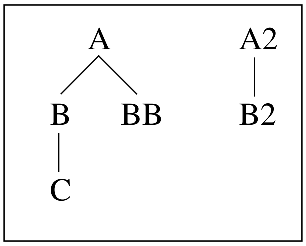

Figure 3: Document version trees

When a child version is created, its contents are copied from 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 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.

4.4 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.

Section 4.3 explained that the original contents of a document version are copied from its parent version. Actually, the system only makes the child document version include the same subdocument versions as the parent document version. No subdocument version are created or copied when a document version is created.[1] 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. For example, suppose document D includes three subdocuments: S1, S2 and S3. Figure 4 shows how document version A includes version A of each subdocument. Document version B has then been created as a child of document version A. Originally document version B has included the same subdocument versions as the parent version A. Later, subdocuments S1 and S3 have been modified in document version B creating subdocument versions B of those subdocuments. Document version B thus includes version B of subdocument S1, version A of subdocument S2, and version B of subdocument S3. Details of this mechanism are given in section 8.

document versions	subdocument versions		
	S1	S2	S3
A	A	A	A
B	B	A	B

Figure 4: Document version compositions

The phrase *subdocument X of version Y of document Z* must be read as *the subdocument version of subdocument X of document Z which is included in document version Y of document Z.* For example, subdocument S2 of document version B in figure 4 refers to version A of subdocument S2.

4.5 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. This was already explained in section 3, although the objects to be copied from and to the database were then identified less precisely as “documents”.

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

1. However, subdocument versions are copied when a document version is created with a source document version (section 8.4).

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. Secondary representations are discussed in more detail in section 8.11.

Figure 5 illustrates the hierarchical structure of a document. Document "100123" has subdocuments"S1", "S2" and "S3". Subdocument "S2" has versions "A", "B" and "C". Version "B" has a primary representation and two secondary representations. Note that the figure does not show document versions, which in a sense exist in another dimension. Version "B" of subdocument "S2" could, for instance, be included in version "C" of the document.

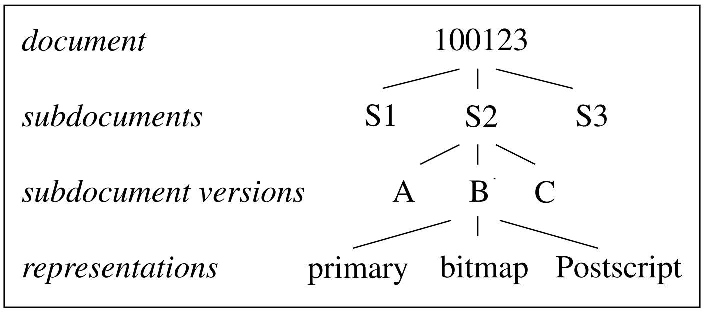

Figure 5: Representations

4.6 Example of Document Version Use

An example of the possible use of document versions is given here. In this example document versions are called *issues* and they are named as "A", "B", …, "Z". The "original" document is not considered an issue and it does not have any issue name. In EDMS, however, the original document is the first version of the document and it has the name "–".

If a document has versions in different languages, the versions are stored as separate document versions of the same document. Version name will then be of the form "*language.issue*", for example, "fi.B" or "en.C".

Sometimes minor changes have to be made to a document although one does not want to create a fully new version. For example, correcting a misspelling in version B should not produce version C. However, if version B has already been approved (see document version states in section 9), it cannot be modified at all. The version name therefore also includes an *edition number*, which starts at 1 and is incremented whenever a new version with the same issue letter is created.

The full document version name is thus "*language.issue.edition*", for instance, "fi.B.3". Figure 6 shows the versions of a typical document.

Note that the parts of a version name have no meaning to the system. From the system's point of view, a document simply has a number of versions, each with a unique name, and the only relationship between versions is the relationship between a version and its children. The system does not know that versions "en.A.1" and "fi.A.2" are more

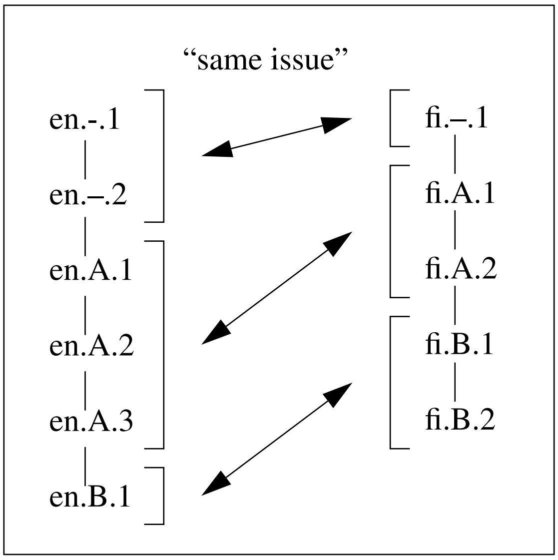

Figure 6: Example of version names

closely related than, for example, "en.A.1" and "fi.B.1". If necessary, one must define document version attributes for storing this information.

4.7 User Objects

Information about people can be stored in the database as *user objects*. In addition to a unique *user name*, each user object also has an optional *login name*. Two user objects cannot contain the same login name.

When a client connects to the server, it sends the user's login name to the server. The client is allowed to use the server only if the database contains a user object with the given login name.[1] If a user object is found, the name of that object is established as the user's identity. For example, each object has an attribute which tells who has created the object. This attribute contains a user name; the login name is never stored in the database. It is also possible to define new attributes which always have a user name as their value (section 6.5).

4.8 Projects

The database also stores *project* objects. Projects have attributes and can only be manipulated with standard operations, such as *create project* and *change attribute values*.

1. An empty database contains no user objects. It seems impossible for any client to add the first user because the client's login name cannot be found in the database. To solve this problem, a client with the special login name `edms` is always allowed to use the server.

5 Object Types

Each object in the database belongs to one of the object kinds presented in the previous section, and is of a particular *object type*. Each document, for example, is of some document type.

An object type has a *name* and a *title*. Object type name is a unique name for the object type, and its format is determined by the relational database system.[1] Ordinary users, however, refer to an object type with its *title*, which is an arbitrary string.

The type of a document determines the type of the its versions, subdocuments and subdocument versions. For example, all versions of a subdocument of a document of the document type T are of the subdocument version type T. The type of a subdocument version, however, does not determine the type of its representations. The type of each representation is stored separately.

There are four independent sets of type names: document type names, representation type names, user type names, and project type names. For each document type, there is also a document version type, a subdocument type, and a subdocument version type with the same name. Two object types of different object kinds can have the same name. There can, for example, be a document type and a representation type with the same name.

There are no relationships between types. In particular, the system does not support type inheritance; one cannot create an object type by adding new properties (e.g., attributes) to an existing type. This probably means that the number of different types in a single database will be fairly small and fixed (perhaps about ten object types). At the moment the only information associated with an object type is a list of attributes for objects of the type (section 6).

1. See appendix B for details.

6 Attributes

All objects have *attributes*, which store various data about the objects.

6.1 Attribute Classes

Each object has three sets of attributes:

- system attributes,
- common attributes, and
- type-specific attributes.

All objects of the same object kind have the same *system attributes*. These attributes are defined by EDMS, and their values can only be modified by the system.

Document name is a typical system attribute. All documents must have this attribute for identifying the document. A user cannot directly modify a document name because the database contains many references to the document. The system attributes for each object kind are listed in appendix A.

All objects of the same kind in the same database have the same *common attributes*. These attributes are defined by the users and have no special meaning to the system. Objects of the same kind in two different databases always have the same system attributes, but they can have quite different common attributes.

Finally, all objects of the same type have the same *type-specific attributes*, which are also defined by the users.

Common attributes were said to have no meaning to the system. To be more precise, the word *system* refers here to the server. Typically the user interface defines attributes for its own use. For example, as was mentioned in section 3, each representation could have an attribute that stores the name of the tool with which the representation is created. When a representation is copied from the database to an ordinary file, the user interface can automatically start the tool with the name of the file as an argument.

This attribute is a common attribute since it is not defined or understood by the server. However, an ordinary user probably considers the “system” to comprise both the server and the user interface. In this sense the name of the tool is a system attribute since all representations have it and it has a special meaning to the system (i.e., to the user interface component of the system). Although the server allows the attribute to be deleted, this must not be done because the user interface expects representations to have this attribute.

6.2 Attribute Names

The system identifies attributes with *attribute names*. The format of an attribute name is determined by the relational database system.

All attributes of an object must have unique names. No common attribute can thus have the same name as a system attribute or another common attribute of the same object kind. A type-specific attribute must not have the same name as a system attribute or a common attribute of the same object kind or another type-specific attribute of the same object type. Different types of the same object kind, however, can have type-specific attributes with the same name.

Attributes of different object kinds are independent of each other. For example, a document and a document version can have an attribute with the same name.

6.3 Attribute Titles

Ordinary users need not be aware of attribute names because the user interface only shows *attribute titles*. The title can contain any characters, including spaces.

The user interface can show messages, menu items and other text in various languages. In order to also show attribute titles in the same language as the rest of the user interface, an attribute definition can give the title in different languages. An English title must always be given and it is used if there is no title in the language selected for the user interface.

Multilingual attribute titles have been defined to make the user interface consistent in each language. If the user interface displays menu items in Finnish instead of English, it seems that attribute titles should also be displayed in Finnish. Note, however, that the user interface does not become fully Finnish unless the system is modified so that many other texts can be given in Finnish. These texts include object type titles (section 5) and the names of states (section 9). It may not be necessary, or even desirable, to allow the users to specify all these texts in different languages.

6.4 Attribute Descriptions

Attribute definition also includes an *attribute description*. The description is arbitrary text, which is shown by the user interface when the user wants information about the attribute. Descriptions can be given in different languages in the same way as attribute titles. The description text should not contain information which is already available in the attribute definition. For example, the type of the attribute should not be repeated in the description.

6.5 Attribute Types

The possible values that can be given to an attribute are specified by the *type* of the attribute. The following attribute types are available:

- Integer
- Integer range
- Floating point
- Floating point range
- Character string
- Date
- Time
- List item
- User name
- Project name
- Set of any of the above types
- Sequence of any of the above types
- Text
- Data
- Multilingual string

The value of an *integer* attribute is stored in 32 bits. *Integer range* as an attribute type is identical to an integer, but the type includes a minimum and maximum value between which all attribute values must lie (inclusive range).

The value of a *floating point* attribute is stored in 64 bits, i.e., the type corresponds to the type “double” in the “C” language (in typical workstation implementations). *Floating point range* is analogous to integer range.

The definition of a *string* attribute includes a maximum length for the attribute value.

The value of a *date* attribute comprises year, month, and day. A *time* attribute also stores hours, minutes and seconds.

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*. The definition of a *list item* attribute only includes the name of such value list. The attribute will actually be a string attribute with a maximum length equal to the maximum length of the choices in the value list, which is specified when the value list is created. 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 *user name* attribute must the name of some user object in the database (section 4.7). This attribute is thus similar to a list item attribute with a value list that contains the names of all user objects. Attribute type *project name* is similar to user name except that the value must be the name of a project instead of a user.

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 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 *set of T* or *sequence of T* where *T* is any of the above types except *text* or *data*[1]. For example, the type of an attribute can be

- set of strings of at most 20 characters,
- set of floating point numbers,
- sequence of integers in the range 0–10,
- sequence of list item values.

Both set-valued and sequence-valued attributes have a number of separate items as their value. The items in a set are always returned from the database in a sorted order regardless of the order of the items when the attribute was given a value. Duplicate items are ignored in a set. A sequence, however, maintains the original order of the items and allows duplicate items. There is no limit on the number items in a set or sequence.

For example, if an attribute of type *set of integers* is given value “(3 1 2 1)”, the value is stored in the database as “(1 2 3)”. If the attribute type were *sequence of integers*, the value would be stored as “(3 1 2 1)”.

Finally, an attribute of the type *multilingual string* stores a string value in different languages. One can, for example, use a multilingual string as a document title that must be stored in English and Finnish.

A multilingual string is actually a set of strings with the additional constraint that each item in its value must be of the form *language*:*text* where *language* is a string found in the value list with the fixed name `languages`. A value list with this name must be created before multilingual attributes can be defined. The definition of a multilingual string specifies the maximum length for the text that follows the language name and the colon.

1. This limitation comes from the database system used to implement EDMS.

For example, if the value list `languages` contains values `en` and `fi`, a multilingual string could be given value

```
("en:Lift door" "fi:Hissin ovi").
```

6.6 Null Values

The actual value of an attribute can be null as well as any of the values specified by the attribute type. For example, the value of an integer attribute is either null or an integer. A null value as the value of a string attribute is distinct from an empty string.

Null values can be used to represent undefined or unknown values. When an attribute with a null value is used in a search condition, for example “weight < 1000”, the condition is always false.

A set or a sequence cannot have a null value as distinct from an empty set or sequence.

6.7 Display Columns

An attribute definition can include an optional integer, which specifies the default width for the attribute value in the user interface. For example, a string attribute with a maximum length of 100 characters can have 20 as the default display width.

6.8 Display Precision

The definition of a floating point attribute includes *display precision*, which tells how many digits the user interface should display after the decimal point.

6.9 Attribute Definition Changes

The type of an existing user-defined attribute, i.e. a common or specific attribute, can be changed 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.

Attribute title, description, display columns and display precision can be changed freely. Titles and descriptions can also be changed by giving them in new languages. These properties can also be changed for system attributes. The name of an attribute, however, cannot be changed.

6.10 Creation and Deletion of Attributes

A new common attribute can be added to an object kind, and a new type-specific attribute can be added to an object type. If objects of the kind or type already exist, the initial value of the new attribute in those objects will be null. The initial value of a new set or sequence attribute will be an empty set or sequence in all existing objects.

Common and type-specific attributes can be deleted freely. However, note what was said at the end of section 6.1 about common attributes which may actually be "system" attributes for the user interface.

System attributes cannot be created or deleted.

6.11 Default Objects

A server request which creates a new object, for instance a new document, can contain initial values for the common and type-specific attributes of the new object. The request need not, however, specify a value for all attributes. Values for the unspecified common and type-specific attributes are copied from the *default object* of the object type.

When the server creates a new object type, it also creates the default object with the same attributes as the other objects of the type. Initially all attributes of the default object have a null value. After the default object has been created, its attributes can be read and modified in the same way as the attributes of ordinary objects.

7 Type Parameters

For each object type, the database can store a number of *type parameters*. Each parameter has a name and a value, which is an arbitrary string. As an example of the use of type parameters, suppose that the user interface should check that the name of a document has a particular format, e.g. "XXX-999" where the X's and 9's stand for letters and digits. If the allowed format depends on the type of the document, the format to be used for a particular document type could be stored as a parameter of the type.

Type parameters can also be associated with an object kind as a whole instead of a specific object type. In the previous example, suppose there is a default document name format and only some document types have a specific format. The default format could then be stored as a parameter of the object kind *document*.

Note the difference between attributes and type parameters. Both can be defined for a specific object type or for an object kind as a whole. The definition of an attribute only tells that objects of the given object type or kind have the attribute; each individual object has a different value of the attribute. The value of a type parameter, however, is associated directly with the object type or kind. A type parameter can be given a value even if there are no objects of the object type or kind.

At the moment the server does not define any type parameters. Although the document name formats mentioned in section 8.2 are implemented with type parameters as outlined above, the implementation is completely in the user interface; the server only stores type parameters without understanding their contents.

8 Document Management

This section describes in detail how documents are created and modified with EDMS. The relevant concepts were already introduced in sections 4.1 to 4.5 (see figure 2 on page 7). Section 8.1 gives an overview of the operations on documents and document versions. Details of each operation are described in later sections.

8.1 Main Concepts

After a document is created, it has no versions. A document can exist without any versions because the document is an object with its own attributes.

As was explained in section 4.3, each document version has either no parent version or exactly one parent version, which is another version of the same document. The initial contents of a document version can also be copied from a version of another document. There are thus three ways to create a new document version:

(1) *New version without a parent version or a source version.* The new document version is created "from scratch" with empty contents. A document can have several document versions without a parent, i.e., there can be several independent development lines within a document. The creation of a document version without a parent is illustrated in figure 7. The document versions are named in the same way as in section 4.6.

(2) *New version with a source version.* The new document version does not have a parent version. However, its initial contents are copied from a version of another document. This document and the copied version are called the *source document* and the *source version* of the new document version, respectively. The source version is not affected by the creation of the new document version. Neither does the possible modification of the source version affect any version that has used it as a source version. The creation of a document version with a source version is illustrated in figure 8.

(3) *New version with a parent version.* The new document version has another version of the same document as a parent version. Initially the new version, i.e., the child version, shares its contents with the parent version. The parent version can no longer be modified after it has at least one child. A single version can have many children, i.e., many versions can have the same parent version. The creation of a document version with a parent version is illustrated in figure 9.

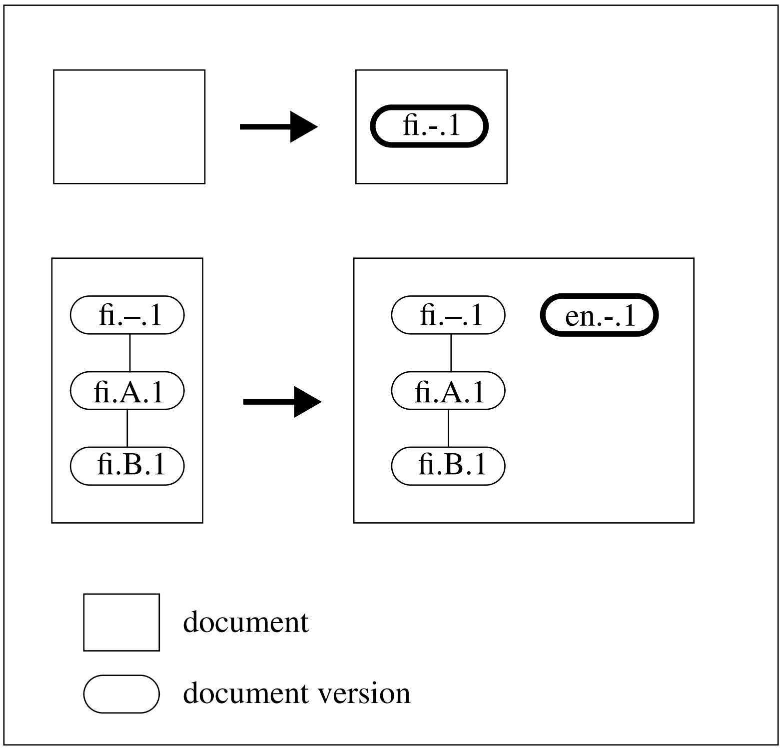

Figure 7: New document version without a parent

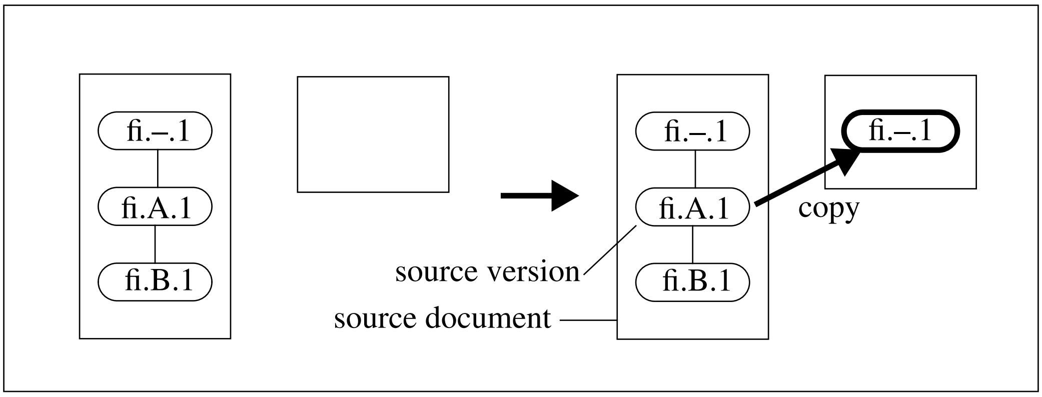

Figure 8: New document version with a source version

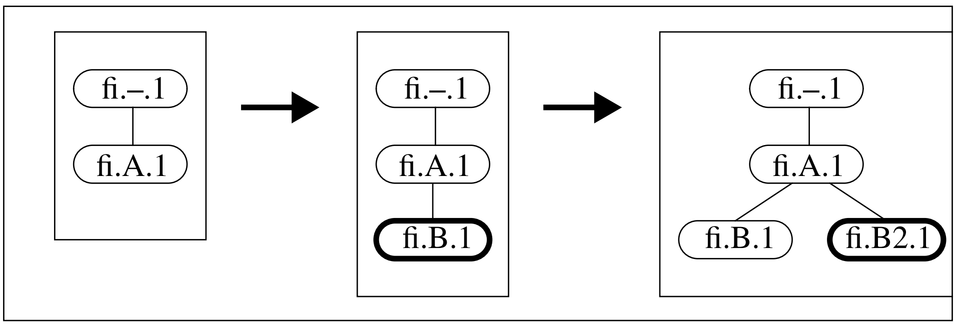

Figure 9: New document version with a parent

In order to modify a document version, a user copies one or more of its subdocuments from the database to ordinary files, for instance in his or her home directory, modifies the files with design tools, and copies the files back to the database. (The previous sentence was somewhat imprecise since one actually copies representations of subdocument versions. The details will be discussed later.)

The operation of copying data from the database to ordinary files for modification is called *copy-out*. The reverse operation of copying the modified files back to the database is called *copy-in*.

Before a user can copy out subdocuments of a document version for modification, he or she must *lock* the version. Other users cannot lock the same version until the first user has *unlocked* it. A document version cannot be unlocked if any of its subdocuments is still copied out. Although users copy individual subdocuments of document version, the whole document version is always locked by a particular user.

Note that modification of a document version does not automatically create a new document version. After a document version has been created, it has the same contents as its parent version or source version. A user can then lock the version and modify its subdocuments with the copy-out/copy-in mechanism. After the first user has unlocked the version, other users can lock and modify it.

A document version is thus added to the database when its construction begins, not when it becomes ready. Whether a particular version is under construction or ready is recorded as the *state* of the version. (As will be seen in section 9, there are actually more than two possibilities for the state of a document version.)

8.2 Document Creation

When creating a document, a user must supply the name and type of the document. Two documents cannot have the same name even if the documents are of different document types.

Document names are arbitrary strings. For more precise control, one can define the allowed format for the names of documents of a particular document type. It is for example possible to define that the name must be of the form "xxx–99–999" where "xxx" is taken from a set of possible codes and each "9" is a digit. The details are given in appendix C.

The new document includes no subdocuments and has no document versions.

8.3 Subdocument Creation

A new subdocument can be added to a document provided that the document does not already have a subdocument with the same name. The name of the subdocument is an arbitrary string. All subdocuments of a document of document type D are of the subdocument type D.

8.4 Document Version Creation

A new version can be created in the three different ways described in section 8.1. All versions of a document of document type D are of document version type D.

The new version can be given any name as long as all versions of the document have different names. Typically the user interface generates an appropriate default name for the new document version.

No parent

Each document version is composed of zero or more subdocument versions. A new document version with no parent or source version, i.e. case (1) in section 8.1, includes no subdocument versions. New subdocument versions must be created with a separate operation (section 8.6).

Source version

Figure 10 illustrates the creation of a document version with a source version from another document, i.e. case (2) in section 8.1. Document versions are named here simply as “A”, “B”, … instead of the more complicated names used in section 4.6. Document X on the left includes two subdocuments, S1 and S2, and has three versions: A, B and C. The small rectangles represent subdocument versions and the ovals represent document versions that include a subdocument version for each subdocument of the document. For example, document version B includes version B of the first subdocument and version A of the second subdocument. The heavy vertical arrows between subdocument versions indicate successive versions of the versions. For example, version C of subdocument S2 has been created by modifying version A of that subdocument. The modification of subdocument versions will be described in sections 8.9 and 8.10.

Suppose that a new document version D of document Y is created with version B of document X as the source version. The subdocument versions included in version B of document X are copied to document Y and included in its version D. Each copied subdocument version gets the same name as the new document version. Version B of the first subdocument of document X is thus copied to document Y where it becomes version D of the first subdocument of document Y. Similarly, version A of the second subdocument of X is copied to Y where it becomes version D of the second subdocument of X. The copied subdocument versions will then be included in the new document version D.

Subdocument versions which have been dropped from the source document version (section 8.15) are not copied.

A version of document Y cannot include a version of subdocument S if document Y does not include subdocument S. Therefore, if the source version includes a version of a subdocument which does not exist in the document in which the new document version is created, the subdocument from the source document is also copied. For example, if Y does not have a subdocument with the name S2, subdocument S2 of X is copied to Y. Only the subdocument object with its attributes is copied; the versions of the subdocument are not copied (except the subdocument version included in the source document version).

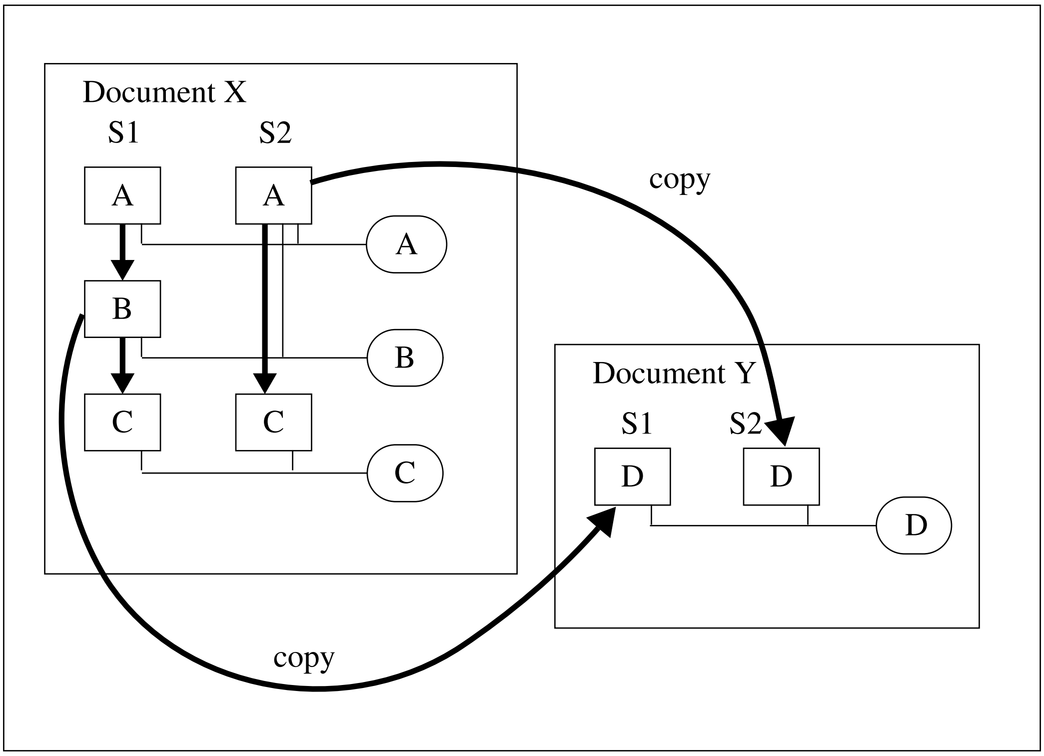

Figure 10: New document version with a source version

The copying of a subdocument version means that the attributes and all representations of the subdocument version are copied. The attributes of a subdocument version are determined by its type, and the type of the subdocument version in turn is determined by the type of the document to which the subdocument version belongs. This means that when a document version is created from a source version of another document, both documents must be of the same document type; otherwise the attributes could not be copied.

Parent version

The third possibility is to create a document with a parent version from the same document, i.e. case (3) in section 8.1. The new document version includes the same subdocument versions as the parent. No subdocument versions will thus be copied until the new version is modified. An example is shown in figure 11. When version D is created with version C as a parent, version D includes the same subdocument versions as document version C, i.e., version C of subdocument S1, version C of S2, and version A of S3. A locked document version (see the next subsection) cannot be used as a parent for a new document version.

8.5 Document Version Locking

A document version can be locked by a user if it is not already locked by someone else. The database also records the time when the document version was locked. A document version with any child versions cannot be locked. In other words, one cannot modify a

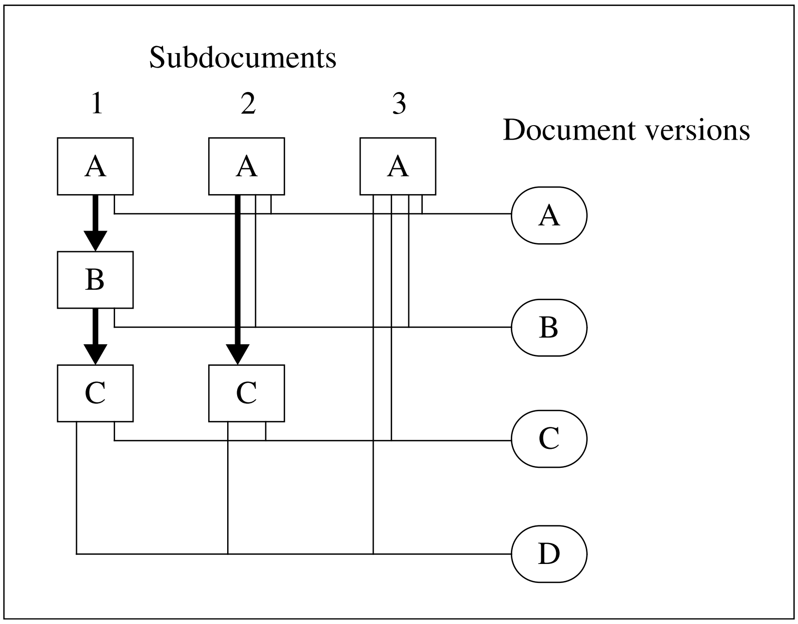

Figure 11: New document version with a parent

document version after it has been used a basis for a new version of the same document because the modification of the parent version would affect its children. Note that a document version *can* be modified if it has only been copied as a source version for the creation of version of another document.

The locking of document versions can be further controlled with the authorization mechanism (section 10).

8.6 Subdocument Version Creation

A user who has locked a document version can create a new subdocument version and include it in the document version. The document must already include that subdocument. The new subdocument version will have the same name as the document version in which it is included. The new subdocument version does not have any representations.

Figure 12 shows an example. At first the document includes subdocuments S1 and S2, and has two versions, A and B. After subdocument S3 has been created, one can create a version of it and include it in the document version B. A subdocument version is always created to a specific document version.

Note that a document version need not include a version of each subdocument of the document. In the above example, document version A does not include any version of subdocument S3 because the subdocument was created after the document version.

8.7 Shared Subdocuments

At this point it is necessary to define the term *shared* subdocument. A subdocument of a document version is shared if an earlier document version includes the same subdocument

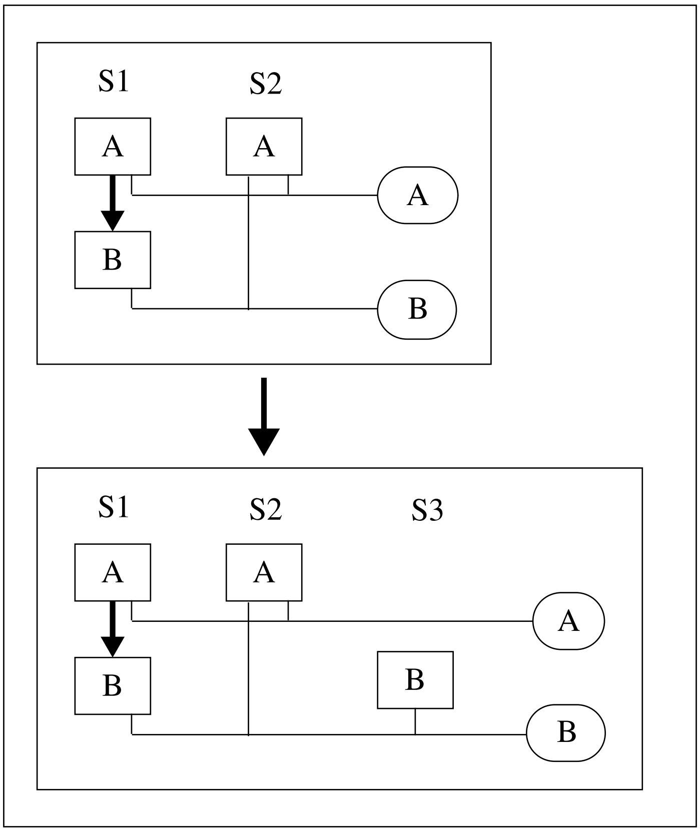

Figure 12: Creation of a new subdocument version

version. For example, consider again figure 12. Subdocument S2 of document version B is a shared one because the same subdocument version (version A of subdocument S2) is also included in document version A. Subdocument S1 of document version B is not shared because the subdocument version (version B of subdocument S1) is not included in any earlier version of the document. Subdocument S2 of version A is not considered shared either because the same subdocument version is not included in any *earlier* document version.

A subdocument version included as a non-shared subdocument in a document version always has the same version name as the document version. For instance, in the above example subdocument S1 of document version B is not shared, and it means that the subdocument version is version B of subdocument S1.

8.8 Representation Creation

A new representation with empty contents can be created and added to a subdocument of a document version. The document version must have been locked by the user who creates the new representation. The representation can have an arbitrary name as long as the subdocument version does not already a representation with the same name. The primary representation must be created before any secondary representations.

The new representation is marked as being copied out (section 8.9). The original contents of the representation are stored in the database with a copy-in operation (section 8.10).

The creation of a new representation of a subdocument version is considered to modify the subdocument version. Therefore, a new subdocument version is created when a new representation is created in a shared subdocument version.

8.9 Representation Copy-Out

The primary representation of a subdocument of a document version can be *copied out* by the user who has locked the document version. The contents of the representation are copied from the database to an ordinary file, for example in the user's home directory.

8.10 Representation Copy-In

After a user has modified the file to which the primary representation was copied out, he or she can copy in the representation. Copy-in is the reverse of copy-out: the new contents of the primary representation are copied from an ordinary file back to the database. A new representation must also be copied in after it has been created.

The representation is always a representation of a particular subdocument of a particular document version. If the subdocument is not shared, the data from the file simply replace the old contents of the representation.

If the subdocument is shared, a new subdocument version is created as illustrated in figure 13. In the upper part of the figure, subdocument S2 of document version C is shared because it is the same subdocument version as subdocument S2 of document versions A and B. The subdocument version, i.e., version A of subdocument S2 has a primary representation R_P and a secondary representation R_S. Now, representation R_P of subdocument S2 of document version C is copied out to a user's file, the file is modified with a design tool, and the representation is copied back in.

The new representation cannot replace the contents of representation R_P of version A of subdocument S2 because that would also affect document versions A and B which include that subdocument version. A new subdocument version C of subdocument S2 is therefore created and included in the document version C. Attributes of subdocument version A are copied to the new subdocument version (except, of course, system attributes like version name and creation time). The modified representation R_P is copied to the new subdocument version C. The secondary representations of version A of subdocument S2 are also copied to the new subdocument version. The copied secondary representations are marked as invalid in the new subdocument version (section 8.11).

Subdocument S2 of document version C is no longer shared. When one of its representations is next time copied out and back in, a new subdocument version need not be created and the new contents of the representation replace the old contents of the same subdocument version.

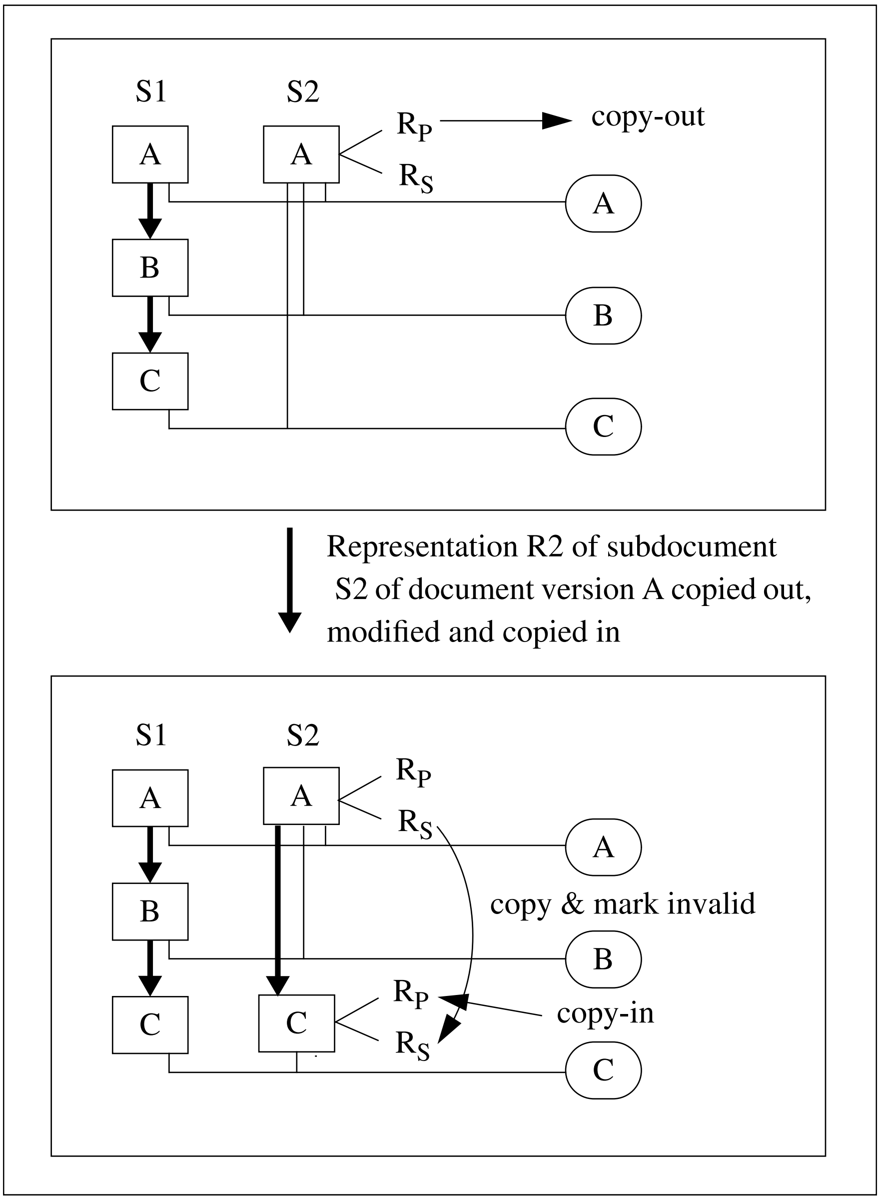

Figure 13: Representation copy-in

To summarize: When a new document version is created, it includes the same subdocument versions as the parent document version. When a representation of a particular subdocument of the new document version is copied out and back in for the first time, a new subdocument version is created and included in the document version. Subsequent modifications of representations of that subdocument version overwrite old data because the subdocument is no longer shared with earlier document versions.

8.11 Secondary Representations

The copy-out/copy-in mechanism described above is only used for modifying the primary representation of a subdocument version. Secondary representations are not copied out because they, unlike primary representations, are not modified by editing an older version of the representation. Instead, the contents of a secondary representation are always created directly from the corresponding primary representation.

The contents of a secondary representation can therefore be *replaced* with data from an ordinary file. This is similar to a copy-in operation of a primary representation, but the replacement operation is not preceded by any corresponding copy-out operation.

Each secondary representation has a system attribute which tells whether its contents are valid with respect to the primary representation. Whenever the primary representation of a subdocument version is copied in, this attribute is set to *false* in all secondary representations of the subdocument. When the contents of a secondary representation are replaced, the attribute is set to *true*.

Like copy-in, the replace operation also creates a new subdocument version if the subdocument version was shared.

8.12 Representation Checkpoints

A *checkpoint* operation can be applied to a primary representation which has been copied out. Like the copy-in operation, *checkpoint* copies data from a user's file to the database as the new contents of the representation. Nevertheless, the representation remains copied out. Checkpointing can be used, for example, to have the file backed up with the database or to allow other users to read the unfinished file from the database.

Checkpoint of a representation of a shared subdocument version creates a new subdocument version in the way that copy-in does.

8.13 Representation Discard

A primary representation which has been copied out can also be *discarded*. This means that the representation is marked as no longer being copied out, although no new contents are copied to the database. The contents of the representation remain in the database as they were when the representation was copied out (or as they were after the last checkpoint).

8.14 Read-Only Copies

The contents of any representation can be copied from the database to a user's file for reading. This is similar to *copy-out*, but the representation cannot be copied in later.

The operation retrieves the current contents of the representation from the database. In case of a primary representation which has been copied out, the *copy-for-read* operation returns the contents as they were when the representation was copied out, or the contents that have been copied to the database with the latest checkpoint operation.

The *copy-for-read* operation does not update the database.

8.15 Subdocument Dropping and Adding

If a document version includes a version of a subdocument, that subdocument can be *dropped* from the document version. The document version still includes the subdocument version, which is only marked as dropped. Representations of a dropped subdocument version cannot be copied out.

As an example, suppose a text document includes the text and a number of pictures as separate subdocuments. If a picture is removed from a version of the document, the corresponding subdocument can be dropped from the document version.

When a child version is created from a document version with dropped subdocuments, the new document version also includes the dropped subdocument versions (marked as dropped, of course).

A dropped subdocument can be *added* back to the document version. This operation only changes the marking of the included subdocument version from *dropped* to *normal.* No new subdocument version is created.

Subdocuments can be dropped and added only by the user who has locked the document version.

Figure 14 shows the versions of a document at six different times. The document has three subdocuments with names "1", "2" and "3". Each row of the tables corresponds to a document version. For example, the first table tells that the document has version A, which includes version A of each subdocument. The tables show the successive compositions of the document when new document versions are created, and a subdocument is dropped and later added back to a document version. The names of dropped subdocument versions are written in parentheses.

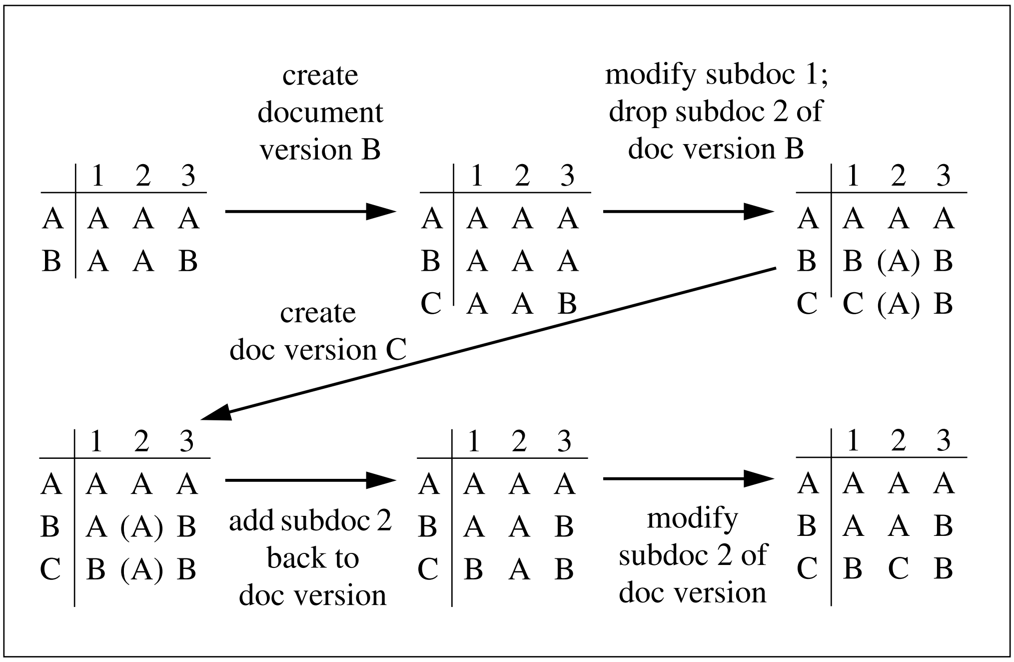

Figure 14: Dropped subdocuments

8.16 Document Version Unlocking

A document version can be unlocked by the user who has locked it. The document version cannot be unlocked if any of its subdocuments has a representation copied out.

8.17 Object Deletion

Documents or other objects are usually never deleted from the database because the database is supposed to be permanent archive. Nevertheless, there are exceptions. For example, it must be possible to delete a document which has been accidentally created with a wrong name.

Since object deletion is an uncommon operation, the system makes effort to prevent a user from accidentally deleting important data. For example, a document cannot be deleted until all its versions have been deleted separately.

The rules for object deletion are as follows.

Document

A document cannot be deleted if it has any versions or subdocuments. This means that no other objects are ever deleted with a document.

Subdocument

A subdocument cannot be deleted if it has any versions.

Document version

A document version cannot be deleted if it has any child versions or it is locked. When a document version is deleted, all non-shared subdocument versions that are included in it are also deleted. For example, consider the "after" situation in figure 12 on page 26. When document version B is deleted, version B of subdocument S1 and version B of subdocument S3 are also deleted since they are non-shared subdocuments versions in the document version.

The previous paragraph explained which subdocument versions are deleted with a document version. If any of these subdocument versions has representations, the subdocument version cannot be deleted, and consequently the document version cannot be deleted either.

Note that subdocument versions are only deleted with document versions.

Representation

A representation of a subdocument version of a particular document version can be deleted. If the subdocument version is non-shared, the representation is simply deleted.

If the subdocument version is shared, the deletion of a representation is again regarded as a modification of the subdocument version. The system therefore creates a new subdocument version, which is identical to the original version except that it does not have the deleted representation.

The primary representation of a subdocument version cannot be deleted if the subdocument version has any secondary representations.

Example

Figure 15 shows a document with subdocuments S1 and S2, and versions A and B. There are three subdocument versions represented as rectangles: S1/A, S1/B and S2/A. The primary representations of the subdocument versions are marked as R_P, and the secondary ones as R_S.

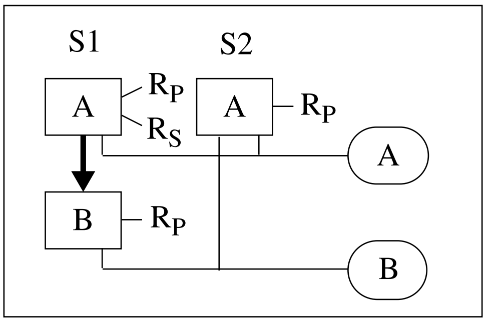

Figure 15: Document deletion

The whole document can be deleted in following steps:

(1) The primary representation of subdocument S1 in document version B is deleted.

(2) Document version B is deleted. This also deletes the non-shared subdocument version S1/B (which now does not have any representations).

(3) The secondary representation of subdocument S1 in document version A is deleted.

(4) The primary representation of subdocument S1 in document version A is deleted.

(5) The primary representation of subdocument S2 in document version A is deleted.

(6) Document version A is deleted. This also deletes the non-shared subdocument versions S1/A and S2/A (which now do not have any representations).

(7) Subdocuments S1 and S2 are deleted. (They have no versions.)

(8) The document itself is deleted. (It has no versions or subdocuments.)

9 States

An object can be in different *states* during its lifetime. The possible states and the allowed transitions between states are specified by associating the object with a *state graph*. Figure 16 shows a possible state graph for a document version.

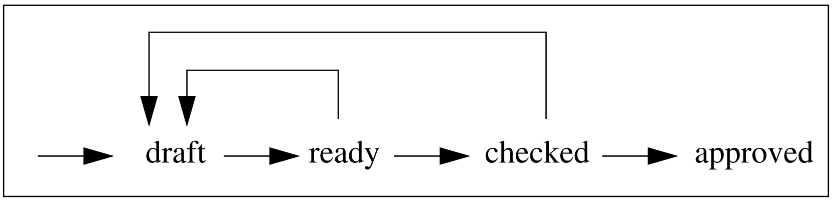

Figure 16: Sample state graph

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 version passes the check, it is moved to the state *checked*; otherwise it is returned 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 object kinds with which states can be used are determined when the database is created. Each such kind has the following system attributes:

- the name of the state graph associated with the object,
- the current state of the object,
- the time when the object was moved to its current state, and
- the name of the user who moved the object to its current state.

States and state graphs are identified with arbitrary names. The specification of a state graph consists of the name of the graph, a list of state names, and a set of state transitions, i.e., a set of pairs with the names of the old and the new state. The first state in the state list is the *initial state* of the graph; the order of the other states is insignificant, although usually the states form a logical sequence.

For each object type, one can define a *default state graph*. When an object of the type is created, it is automatically associated with this state graph. If necessary, the object can later be associated with another graph instead. Whenever an object is associated with a new state graph, the object is moved to the initial state of the graph.

The creation of a new document version is an exception to the above rule. If the new version is created with another version of the same document as a parent, the new version is associated with the same graph as the parent version. If the new version has no parent or its contents are copied from a source version of another document, the new version is associated with the default graph of its document version type. In both cases the new document version is moved to the initial state of the 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* (using the sample state graph in figure 16). This kind of rules can be defined with the authorization program of section 10.

10 Authorization Program

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 16 on page 33.

This section only gives an overview of the authorization mechanism, the details are again found in the *Administrator's Guide* [2].

10.1 Authorization Program

Access rights are controlled by means of an *authorization program*. The program consists of *authorization procedures* written by users (typically by a database administrator) in a simple procedural language which includes operations for reading attributes and other properties of objects. Examples of authorization programs are shown later.

Each authorization procedure corresponds to a service request which updates the database. Whenever the server receives such a request, it calls the corresponding authorization procedure with appropriate arguments before updating the database.

The statements of the authorization procedure determine whether the operation is 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.

The server never performs an operation which is rejected by an authorization procedure. The acceptance of an operation by an authorization procedure, however, does not guarantee that the operation will be performed because a request must also satisfy other conditions.

The authorization procedures may have to refer to entities for which there are no corresponding EDMS objects. This data can be stored in relational tables, which are read from the authorization program with SQL statements. The tables are located in the same database that the server uses. For example, the sample authorization program in the next subsection reads information about departments from such an "extra" table.

10.2 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 department managers are stored in the table `dept_manager(dept, manager)`. Now 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 following authorization procedure in the authorization program:

```
event changeState (user, object, new)
```

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. The authorization procedure is shown below.

```
(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
```

The variables in these statements have two kinds of values: strings and object identifiers. An object identifier identifies an object, such as a document or a document version. Parameters `user` and `new` have a string value, and `object` is an object identifier.

On line 2, the built-in function `kind` takes an object identifier as argument and returns object kind as a string. The `test` statement is similar to the `switch` statement in the "C" language. If the object is a document version, procedure `newDocVersionState` is called on line 6.

On line 17, the value of the attribute `doc` of the document version is assigned to the variable `doc_name`. On line 18, a reference to the document with the name `doc_name` is assigned to the variable `doc`. The function `document` takes a string as argument and returns an object identifier. The name of the department to which the document belongs is retrieved from the attribute `department` of the document on line 19.

Lines (17)–(19) could also be written simply as

```
dept = document(version.doc).department;
```

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 not the same, the request is rejected by means of a `reject` statement, which can include an explanatory string. If the names match, the request is accepted.

As a further example, suppose that a manager can have a number of deputies, who are allowed to approve the same document versions as the manager. The deputies of managers are stored in the table `deputies(manager, deputy)`. Lines 22–26 of the original authorization procedure can now be replaced with the following statements:

```
(1)      if user = mgr then
(2)        accept;
(3)      else
(4)        deps = [all deputy from deputies
(5)                          where manager = :mgr];
(6)        if user in deps then
(7)          accept;
(8)        else
(9)          reject "version must be approved by ", mgr,
(10)                 " or any of ", deps;
(11)       end;
(12)     end;
```

On lines 4 and 5, the SQL expression with the key word `all` assigns a *list* values to the variable `deps`. On line 6 one then tests if the user is member of this list.

10.3 Limitations of the Authorization Mechanism

The actual authorization rules at any given time are determined by the authorization program and the tables that the program accesses, such as the tables `dept_manager` and `deputies` in the previous example. The server only executes statements of the authorization program; it does not "understand" concepts like department or manager in the example.

As a result, the authorization agent is an external "black box" to the server, which cannot modify the authorization program or the tables used by the program. Neither can the server control who is allowed to make this kind of modifications. In particular, the server does not include operations for granting access rights, not to mention operations for granting rights to modify authorization rules.

Modifications of the authorization program and the data tables can be controlled with the operating system and the database system. Nevertheless, it seems that this kind of authorization mechanism is not very effective against users who deliberately want to break the rules. The mechanism is more suitable when it is only necessary to prevent users from accidentally doing something they should not do.

10.4 Control of Reading Rights

The authorization procedures are only called when a user tries to update the database. Read operations may also have to be controlled. For example, a user can ask for all documents with certain attribute values. For each found document, the user wants to see the values of certain attributes. There are different levels of detail to restrict this query. One could specify that the user is only allowed to read certain documents, or only certain attributes. In a more general case one should be able to specify whether the user is allowed to read a particular attribute of a particular document.

One could call an authorization procedure for each individual attribute value that the user wants to read, i.e., once for each (document, attribute) pair. This would, however, be far too inefficient. A less general but more efficient mechanism is clearly needed. This is still an open question.

11 Commit Program

11.1 Transactions

As a background for this section, a brief summary of the transaction concept in database systems is given below.

A single server request often results in several modifications in the database. To maintain the consistency of the database, it is important that either all modifications are carried out or, if something fails during the request, the database is not modified at all.

The server achieves this behaviour by collecting the database modifications of a single request into a *transaction*. When a transaction is *committed*, the relational database system used by the EDMS ensures that all modifications within the transaction are recorded permanently in the database. If the transaction is *aborted*, the modifications have no effect on the database at all.

11.2 Commit Program

Sometimes users would like certain EDMS requests to automatically update some user-defined 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 a user-defined 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, EDMS users can specify a *commit program* for the server. A commit program is similar to an authorization program (section 10). However, whereas an authorization program allows users to describe conditions under which a server request is accepted or rejected, a commit program allows users to associate additional attribute modifications with a server request.

A commit program contains *commit procedures* in the same way as an authorization program contains authorization procedures. The main purpose of a commit procedure is to change attribute values with an *update* statement. Additionally, a commit procedure can abort the whole request with a *reject* statement, although typically invalid requests are already rejected by the corresponding authorization procedure. A commit procedure cannot contain *accept* statements.

A commit procedure can examine attribute values and execute SQL queries like an authorization procedure.

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 server thus executes a request in the following steps:

(1) Check that the request meets the general EDMS conditions for the particular request. (For example, for a *change state* request, check that the object exists, the object is associated with a state graph, and the state graph includes a transition from the current state of the object to the new state given in the request.)

(2) Call the corresponding authorization procedure. If the procedure executes a *reject* statement, send an error reply to the client and stop processing the request. If an *accept* statement is executed, continue. (It is an error if an authorization procedure ends without executing either *reject* or *accept*.)

(3) Begin a transaction.

(4) Update the database according to the request. (For example, a *change state* request updates attributes *current_state*, *transition_time* and *transition_user*.)

(5) Execute the commit procedure if the commit program contains one. If the procedure executes a *reject* statement, abort the transaction and send an error reply to the client.

(6) Commit the transaction.

This subsection started with an example as motivation for the commit program. 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)        update obj.approved_by := NULL;
(7)      when "checked":
(8)        update obj.checked_by := user;
(9)      end;
(10)   end;
(11) end;
```

Attribute value is changed with the *update* statements on lines 5, 6, 8 and 10.

Since representation *copy-in* and *copy-out* operations (including *checkpoint* and *replace*) are executed in a separate subserver process, commit procedures cannot be defined for these operations.

For details of the commit program, see the *Administrator's Guide* [2].

12 References

[1] Alho, K. *EDMS Engineering Data Management System—User's Reference*. Technical Report TKO-C58. Helsinki University of Technology, Laboratory of Information Processing Science. 1993.

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

Appendix A: System Attributes

System attributes were explained in section 6. The system attributes of each object kind are listed in this appendix. The list shows for each system attribute, its name (section 6.2), title (section 6.3), type (section 6.5) and description (section 6.4). Users can change titles and descriptions, and give them in new languages.

The maximum lengths of string attributes are defined as constants MAX_XXX (see appendix B).

Each object kind for which the state mechanism is used (typically at least document versions) also has the following attributes.

Name:	state_graph
Title:	State graph
Type:	string(MAX_STATE_GRAPH_NAME_LEN)
Description:	The name of the state graph with which the object is associated, or NULL if the object is not associated with any state graph.

Name:	current_state
Title:	Current state
Type:	string(MAX_STATE_NAME_LEN)
Description:	The current state of the object, or NULL if the object is not associated with any state graph.

Name:	transition_time
Title:	State transition time
Type:	time
Description:	Time when the object was moved to its current state, or NULL if the object is not associated with any state graph.

Name:	transition_user
Title:	State transition user
Type:	user name
Description:	User who moved the object to its current state, or NULL if the object is not associated with any state graph.

The object kinds have the following system attributes:

Document

Name:	doc
Title:	Document
Type:	string(MAX_DOC_NAME_LEN)
Description:	The unique name of the document.

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the document.

Name: create_user
Title: Creator
Type: user name
Description: User who created the document.

Document Version

Name: doc
Title: Document
Type: string(MAX_DOC_NAME_LEN)
Description: The version is a version of this document.

Name: version
Title: Version
Type: string(MAX_VERSION_NAME_LEN)
Description: Name of the version.

Name: source_doc
Title: Source document
Type: string(MAX_DOC_NAME_LEN)
Description: Name of the source document. NULL if the version has no parent or if the parent is a version of the same document.

Name: parent_version
Title: Parent/source version
Type: string(MAX_VERSION_NAME_LEN)
Description: The name of the parent or source version. NULL if the version has no parent.

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the document version.

Name: create_user
Title: Creator
Type: user name
Description: User who created the document version.

Name: lock_time
Title: Lock time
Type: time
Description: Time when the version was locked. NULL if the version is not locked.

Name: lock_user
Title: Locked by
Type: user name
Description: User who locked the version. NULL if the version is not locked.

Subdocument

Name: doc
Title: Document
Type: string(MAX_DOC_NAME_LEN)
Description: Name of the document.

Name: subdoc
Title: Subdocument
Type: string(MAX_SUBDOC_NAME_LEN)
Description: Name of the file.

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the document file.

Name: create_user
Title: Creator
Type: user name
Description: User who created the document file.

Subdocument Version

Name: doc
Title: Document
Type: string(MAX_DOC_NAME_LEN)
Description: Name of the document.

Name: subdoc
Title: Subdocument
Type: string(MAX_FILE_NAME_LEN)
Description: Name of the file.

Name: version
Title: Subdocument version
Type: string(MAX_VERSION_NAME_LEN)
Description: Name of the subdocument version.

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the subdocument version.

Name: create_user
Title: Creator
Type: user name
Description: User who created the subdocument version.

Representation

Name: doc
Title: Document
Type: string(MAX_DOC_NAME_LEN)
Description: The version is a version of a file of this document.

Name: subdoc
Title: Subdocument
Type: string(MAX_SUBDOC_NAME_LEN)
Description: Name of the subdocument.

Name: version
Title: Subdocument version
Type: string(MAX_VERSION_NAME_LEN)
Description: Name of the subdocument version.

Name: rep
Title: Representation
Type: string(MAX_REP_NAME_LEN)
Description: Name of the representation.

Name: rep_class
Title: Primary / secondary
Type: string(1)
Description: Tells whether the representation is a primary ("p") or secondary one ("s").

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the representation.

Name: create_user
Title: Creator
Type: user name
Description: User who created the representation.

Name: contents
Title: Contents
Type: blob (binary large object, arbitrary data up to 2 Gb)
Description: Contents of the representation.

Name: compression
Title: Compression
Type: string(MAX_COMPRESSION_FORMAT_LEN)
Description: If non-NULL, tells how the contents have been compressed.

Name: valid
Title: Valid
Type: string(1)
Description: In a secondary representation tells whether the contents are valid ("y") or not ("n"). Always NULL in a primary representation.

Name: last_mod_time
Title: Last modification time
Type: time
Description: Time of the last modification of the representation contents.

Name: last_mod_user
Title: Last modified by
Type: user name
Description: User who has last modified the representation contents.

Project

Name: proj
Title: Project
Type: string(MAX_PROJ_NAME_LEN)
Description: The unique name of the project.

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the project.

Name: create_user
Title: Creator
Type: user name
Description: User who created the project.

User

Name: user
Title: User
Type: string(MAX_USER_NAME_LEN)
Description: The unique name of the user object.

Name: login_name
Title: Login name
Type: string(MAX_LOGIN_NAME_LEN)
Description: User's login name (NULL if the user is not allowed to use EDMS)

Name: create_time
Title: Creation time
Type: time
Description: Creation time of the user object.

Name: create_user
Title: Creator
Type: user name
Description: User who created the user object.

Appendix B: Name Constraints

This appendix describes the allowed format of document names, version names, etc. The rules refer to system parameters, such as MAX_VERSION_NAME_LEN. The values of these parameters can be chosen freely within some limits, but after a database has been created with particular values, it may be difficult to change the parameters.

Object names

The name of an object of any kind can contain any printable ASCII characters except the following characters:

(space) `*` `?`

Lower and upper case letters are distinct. The maximum name lengths for the different object kinds are defined as the following system parameters:

Object kind	*Maximum name length*
document	MAX_DOC_NAME_LEN
subdocument	MAX_SUBDOC_NAME_LEN
document version	MAX_VERSION_NAME_LEN
subdocument version	MAX_VERSION_NAME_LEN
representation	MAX_REP_NAME_LEN
user	MAX_USER_NAME_LEN
project	MAX_PROJ_NAME_LEN

Object types

Object type *name* can contain at most 16 lower case letters, digits and underline characters. The first character must be a letter.

Object type *title* can contain at most MAX_TYPE_TITLE_LEN printable characters (spaces are allowed).

Attributes

The rules for attribute *names* are the same as for object type names.

Attribute *title* can contain at most MAX_ATTR_TITLE_LEN printable characters (spaces are allowed).

Attribute *description* can contain at most MAX_ATTR_DESCR_LEN printable characters (spaces are allowed).

States

The names of state graphs and individual states have the same rules as object names except that a state graph name can contain at most MAX_STATE_GRAPH_NAME_LEN characters and the name of an individual state within a graph can contain at most MAX_STATE_NAME_LEN characters.

Value Lists

The rules for value list names are the same as for object type names.

Type Parameters

The the rules for type parameter names are the same as for object type names.

Appendix C: Document Name Format

A *document name format* can be defined for each document type. The format is specified as a list of format strings. When a new document is created, the user interface checks if the name of the new document matches any of the format strings in the list. If name does not match any format, the user is given an warning message and he or she can decide if the document should be created with the invalid name.

The name format string can also specify that the name given by the user should be slightly modified before the new document is created.

In addition to any non-space characters, a name format string can contain following special codes:

`9`	Matches any single digit "0"–"9".
`a`	Matches any single lower case letter "a"–"z"
`A`	Matches any single upper case letter "A"–"Z"
`x`	Matches any single digit or lower case letter "a"–"z"
`X`	Matches any single digit or upper case letter "A"–"Z"
α`(`*n*`)`	Matches exactly n characters matched by α, $n \geq 1$, α is one of "9aAxX"
α`(`*m*`, `*n*`)`	Matches a sequence of at least m and at most n characters matched by α, $0 \leq m < n$
α`(`*m*`, `*n* `L` *x*`)`	Matches a sequence of at least m and at most n characters matched by α, $0 \leq m < n$; moreover, if there are less than n characters, a sufficient number of characters x are added so that the original value is left-justified in a field of n characters.
α`(`*m*`, `*n* `R` *x*`)`	The same as above except that the value is right-justified in the field.
`{`s_1 s_2 ... `}`	Matches any of the given strings, which cannot contain special codes.
`[ s ]`	Specifies an optional part.

If the format string starts with the characters "`U:`", all lower case letters in the document name are translated to upper case before the name is examined.

As an example, suppose the document name must be of the form

xxx–99.99.999[–*yyy*]

where "xxx" is one of the following codes: "AM", "AME", "AR", "AS", "DL1", "DL2" or "KOS"; each "9" is a digit; and the optional "yyy" are from one to three lower case letter. Users must be able to enter any of the three numbers without leading zeroes, and it must be possible to enter the code “xxx” in lower case letters. This format could be specified with the following name format string:

```
U:{AM AME AR AS DL1 DL2 KOS}-
9(1,2 R0).9(1,2 R0).9(1,3 R0)[-a(1,3)]
```

If a user creates a new document and enters document name

```
am-1.2.3
```

the new document will actually be created with the name.

```
AM-01.02.003
```

Index

A

accept 35
adding, see subdocument
administrator 1, 35
administrator's tool 2
attribute 12, 13
  creation 17
  data type 16
  date type 15
  deletion 17, 18
  description 14, 17, 43, 48
  display columns 17
  display precision 17
  floating point type 15
  integer type 15
  list item type 15
  multilingual string type 16
  name 14, 17
  project name type 16
  range 15
  sequence type 16, 18
  set type 16, 18
  string type 15
  text type 16
  time type 15
  title 14, 17, 43, 48
  type 15
  type change 17
  user name type 16
authorization 1, 3, 25, 34, 35

C

checkpoint, see representation
child version 8, 24, 31
C-library 2, 5
client 2, 11
client type 3
commit program 1, 39
common attribute 13, 17
compression 4
copy-in, see representation copy-in
copy-out, see representation copy-out

D

default graph 34
default object 18
default state graph 33
delivery 4
discard, see representation
document 6, 7, 20
  creation 22
  deletion 31
  name 19, 22, 48, 50
  type 12
document issue
  locking 22, 24, 31, 34
  unlocking 22, 31
document version 6, 8, 20
  creation 23, 33
  deletion 31
  name 23
  state 33
  type 12, 23
dropping, see subdocument

E

edition number 10

F

file 4, 5, 7, 9, 13, 22

I

inheritance 12
initial state 33

L

language 10, 14, 16, 17
locking, see document issue
login name 11

M

monitor 4

multilingual string, see attribute

N

network 3
NFS 4
notification 3
null value 17, 18

O

object type 12
 name 12, 48
 title 12, 48

P

parent version 8, 9, 20, 24, 25, 33
PC 3, 4
primary representation 9
project 6, 11

Q

query 4

R

reject 35
representation 6, 9, 25
 checkpoint 29
 contents 9
 copy-in 22, 27
 copy-out 4, 22, 27
 creation 26
 deletetion 31
 discarding 29
 type 12

S

secondary representation 9
server 2
server protocol 3
service request 2
shared subdocument 25
socket 3
source version 20, 23, 25, 34
SQL 4, 35, 37
state 1, 22, 33, 43, 48
subdocument 6, 7
 adding 30
 creation 22
 deletion 31
 dropping 23, 30
 name 22
 type 12, 22
subdocument version 6, 8
 creation 25, 29, 31
 name 25
 type 12
subprocess 4
system attribute 13, 18, 43

T

TCP/IP 3
tool 1, 5, 7, 13
transaction 3, 39
type parameter 19, 49
type-specific attribute 13, 17

U

unlocking, see document issue
user 6, 11
user interface 2, 8, 13, 18, 19, 23

V

value list 15, 16, 49

X

X server 3, 4
X Window System 3