Published in the proceedings of
The 1994 ASME Database Symposium
September 11–14, 1994
Minneapolis, Minnesota

# AN AUTHORIZATION MECHANISM FOR A DOCUMENT DATABASE

**Hannu Peltonen, Kari Alho, Tomi Männistö and Reijo Sulonen**
Laboratory of Information Processing Science
Helsinki University of Technology
Espoo, Finland

## ABSTRACT

A programmable authorization mechanism for an engineering document database is described. The database stores documents and their versions in a commercial relational database. Documents and other objects contain user-defined attributes. Release and approval procedures of the documents are described by means of state graphs that specify the possible states of a document and the transitions between them. The system consists of centralized server and client programs, which communicate with the server using a specific protocol.

The database administrator must be able to formulate rules that tell who is allowed to do particular database operations; rules that tell which operations are allowed in particular document states; and rules that describe consistency invariants between attributes.

These rules are expressed by means of authorization procedures written in a special language. Whenever the server receives an operation request, it executes the corresponding authorization procedure. The system can thus be easily adapted to an organization without actually modifying the client or server programs.

## 1 INTRODUCTION

An engineering document database must include a flexible mechanism for controlling access privileges, which determine whether a particular user is authorized to perform a particular operation (e.g., *modify*) on a particular object (e.g., a document version).

The authorization usually depends on the identity of the user. Typically access privileges are not granted directly to individual users. Instead, users are divided into groups, which are authorized for different operations. Alternatively, one can assign users to different *roles*, and grant privileges to roles.

Often documents, document versions and other objects are associated with *states* that correspond to the document approval and release procedures. The set of permissible operations then depends on the current state of the object to be manipulated. For example, a document version may be in the state *approved*, which means that the version cannot be modified.

The authorization rules vary considerably between organizations. In addition to the current state, the authorization may depend on other properties of the object. It is, for example, possible that documents are assigned to departments, and the manager of each department has special privileges on documents assigned to his or her department.

Authorization may further depend on objects that are related to the object being manipulated. For example, if a document is composed of subdocuments, there may be a rule that prevents the document from being approved unless all its subdocuments have already been approved.

This paper describes the authorization mechanism of an engineering document management system (EDMS) that we have developed in co-operation with a large industrial company (Peltonen 1993, Peltonen et al. 1993). The earliest versions of EDMS controlled operations only by means of object states. Each possible state of an object, e.g., of a document version, was associated with a set of allowed operations. This arrangement can be expressed as a table shown in Figure 1, where the columns correspond to states and operations applicable to document versions.

| state | document version operations | | | |
|---|---|---|---|---|
| | read | modify | create new version | … |
| draft | yes | yes | no | … |
| … | … | … | … | … |
| approved | yes | no | yes | … |

FIGURE 1: DOCUMENT VERSION OPERATION TABLE

One possibility for including user information in this table would be to associate each user with one or more *roles* and replace each "yes/no" cell in Figure 1 with a set of roles in which the operation is allowed.

This approach, however, seems too rigid. Although the *contents* of the table can be modified easily, one cannot change the *rule* that the authorization is always determined with this table. As an example, consider the assignment of documents to departments as mentioned earlier in this section. The department is represented as an attribute, which is defined by the EDMS administrator using the dynamic schema definition facilities of EDMS and does not appear anywhere in the program code. Consequently, it should also be possible to define an authorization rule that refers to this attribute without modifying the program.

Although the document database is built on top of a commercial relational database system, we cannot simply use its authorization mechanism. The operations defined for the document database typically manipulate several tables and cannot be controlled by granting read and modify privileges on individual database table columns. Moreover, the relational database only allows the administrator to grant privileges to specific users.

Our solution to this problem is a separate *authorization program*. The program contains an authorization procedure for each possible database update operation (e.g., creation of a new document version). Before the database is updated, the corresponding procedure is executed to determine whether the operation can be executed. The procedures are written in a special language, which includes constructs found in most procedural programming languages (e.g., assignments, loops and procedures), as well as some useful extensions, such as SQL queries. This approach should provide the required flexibility more easily than authorization schemes based on more fixed concepts and rules (Rabitti 1991).

The rest of the paper is organized as follows. Section 2 briefly presents the main components of the document management system in which the authorization mechanism is incorporated. Section 3 illustrates the actual authorization mechanism with examples. Section 4 deals with another similar mechanism, the *commit program*, by which attributes can be updated automatically as a result of database operations. Section 5 shows how the authorization and commit programs can send electronic mail. Finally, Section 6 provides conclusions and discusses issues left open by our current system.

## 2 SYSTEM COMPONENTS

### System Architecture

Figure 2 shows the main components of EDMS. All data, i.e., the actual document contents and various attributes, are stored in a commercial relational database. Document contents are stored in the database as *binary large objects* (BLOBs).

The database is managed by the *server* program, which waits continuously for *service requests* from *client* programs. Clients include the *user interface* program (Alho 1993) and design tools that use the *EDMS API (Application Programming Interface) library*, which is a set of functions written in the "C" language.

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.

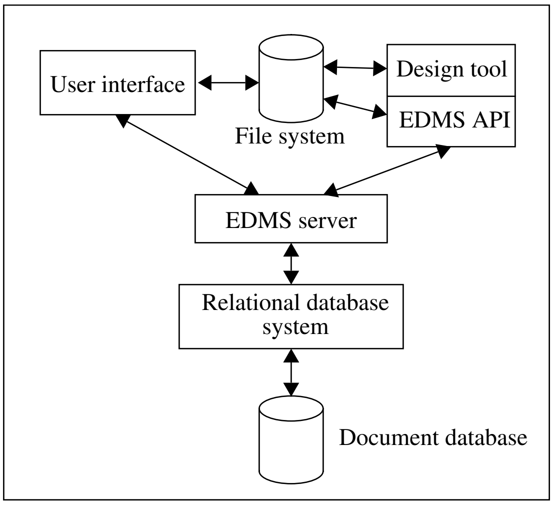


FIGURE 2: SYSTEM ARCHITECTURE

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.

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

The requests that the clients can send to the server are defined in the *server protocol* (Peltonen and Alho 1993). The protocol only refers to EDMS concepts introduced in this document and is independent of the relational tables where the server actually stores the data. The server could therefore be implemented on top of another database system without changing the server protocol.

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

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

### Document Model

EDMS document model is described here very briefly. For more details, see the references (Peltonen 1993, Peltonen et al. 1993).

*Documents* represent any data for design tools: drawings, manuals, bitmap images, etc. The structure of a single sample document is shown in Figure 3. This document is assumed to be a drawing which comprises two sheets, stored as *subdocuments* Sheet1 and Sheet2. Each *document version* contains a particular version of each subdocument of the document. Document versions are shown as ovals on the left-hand side. Document version A contains *subdocument version* A of both subdocuments. Someone has created document version B by modifying subdocument Sheet2. As a result, a new version B of this subdocument is created and included in document version B. Since the first subdocument has not been modified, document version B still contains version A of this subdocument.

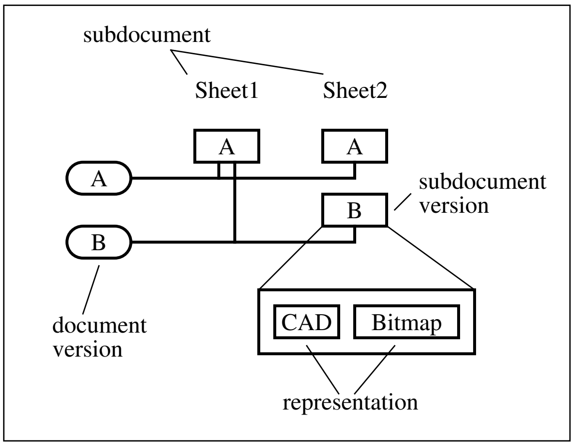


FIGURE 3: DOCUMENT STRUCTURE

The version names (A, B, etc.) in this example are arbitrary. The server allows any version names as long as all versions of a single document have different names, although usually versions are named with consecutive numbers or letters.

The actual data for the design tools are stored in the *representations* of the subdocument versions. A subdocument version has a single *primary representation* (e.g., a file edited with a CAD tool) and optionally a number of *secondary representations* (e.g., a Postscript file), which are generated from the primary representation.

Various data about the documents, document versions and other objects are stored as *attributes*. The EDMS administrator can define a number of document types, such as *drawing* and *manual*, and attributes for each type. The objects also have certain system attributes. Each document, for example, has system attributes for document name and creation time.

The possible values that can be given to an attribute are specified by the *type* of the attribute. Besides the familiar integers, character strings, etc., EDMS allows some less common types, such as list items (the allowed choices for the attribute value are stored in named list in the database) and set-valued attributes (e.g., a set of strings).

### Users

In addition to the various objects related to documents, the database stores *user objects*. One can define several user types, and for each type, a set of attributes.

Each user object contains a login name. A user with a particular Unix login name is allowed to connect to the server only if the database contains a user object with the same login name.

One of the available attribute types is *user name*. The value of an attribute of this type is always the name of some user in the database.

### States

An object, such as a document or document version, can be in different *states* during its lifetime. The possible states and the allowed transitions between states are specified by with a *state graph*. Figure 4 shows a possible state graph for a document version.

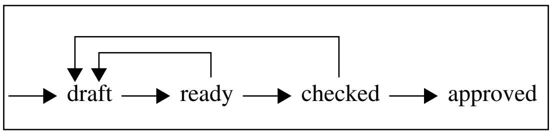


FIGURE 4: SAMPLE STATE GRAPH

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 person responsible for checking the version accepts it, he or she moves it to the state *checked*; otherwise he or she returns it back to the state *draft*. A checked version can be approved, moving it to the state *approved*, from which it never moves to another state.

The EDMS administrator can create many different state graphs. If the administrator defines a state graph as the default graph for an object type, new objects of the type are automatically associated with that graph. An individual object can later be associated with another graph if necessary.

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

At the moment the database only stores the current state of each object and the name of the user who has moved the object to its current state. The system could easily be modified to record more comprehensive state histories. Each object would then be associated with an audit trail that tells all state transitions and other important operations that have been applied to the object during its life time.

## 3 AUTHORIZATION PROCEDURES

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

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

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

The server reads the authorization program when the server is started. To modify the authorization rules, the server must be restarted with a new authorization program. It might be useful to change the system so that the server could be instructed to read a new authorization program without restarting the server.

If the authorization program does not contain a procedure for a particular operation, the operation is automatically accepted. Alternatively, the server could check that the authorization program contains a procedure for every possible operation. (A simple procedure consists of a single *accept* statement.) This would prevent accidental omission of authorization checking.

## 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. In this example we further assume that the names of department managers are stored in the table `dept_managers` with columns `dept` and `manager`. We want to express the rule that a document version can only be approved, i.e., moved to the state "approved", by the manager of the department to which the document belongs.

When the server receives a request for changing the state of any object, it calls the following authorization procedure written by the EDMS administrator. (The keyword `event` marks a procedure which is executed when the server receives the corresponding request from a client. The keyword `procedure` starts an auxiliary routine which can be called from other procedures. A routine started with the keyword `function` is similar except that it returns a value to the calling procedure.)

```
(1)  event changeState (user, object, newState)
(2)    test kind(object)
(3)    when "doc-version":
(4)      newDocVersionState
(5)           (user, object, newState);
(6)    when …
(7)    end;
(8)  end;

(9)  procedure newDocVersionState
(10)      (user, version, newState)
(11)   test newState
(12)   when "approved":
(13)     mgr := managerOf(version);
(14)     if user = mgr then
(15)       accept;
(16)     else
(17)       reject "must be approved by ",mgr;
(18)     end;
(19)   when …
(20)   end; -- test newState
(21) end; -- newDocVersionState

(22) function managerOf (docOrVersion)
(23)   if kind(docOrVersion) = "document" then
(24)     doc := docOrVersion;
(25)   else
(26)     doc := document(docOrVersion.doc);
(27)   end;
(28)   dept := doc.dept;
(29)   mgr := [manager from dept_managers
(30)           where dept = :dept];
(31)   return mgr;
(32) end;
```

The authorization procedure has arguments for the name of the user attempting to change the state (i.e., the name of a user object in the database), the identifier of the object, and the state to which it should be moved.

The authorization procedure first determines what kind of object is being moved to a new state (line 2). If the object is a document version, the procedure on line 9 is called. The function on lines 22–32 takes a document or document version as argument and returns the name of the manager of the department to which the document has been assigned. On line 26, the argument is known to be a document version. The name of the corresponding document is retrieved from the attribute `doc`, and a reference to a document with that name is created with the built-in function `document`.

On line 29, 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 14, the name of the user who tries to change the object state is compared to the name of the department manager. If the names are equal, the request is accepted. Otherwise, the request is rejected by means of a `reject` statement on line 17 and the error message in the statement is displayed by the user interface.

## Rule Types

Data which is expected to change should not be coded directly in the procedures. Instead, the variable data should be stored in the database tables, which are read by the procedures. For example, suppose only certain users are allowed to create new documents. It is possible to list these users directly in the procedure as follows:

```
event createDocument (user, …)
  if user in { "Bill", "Mary", … } then
    accept;
  else
    reject;
  end;
end;
```

It is, however, much better to create a table that stores the names of users who are allowed to create documents. The procedure will then only express the rule "there is a list of users allowed to create documents" without specifying the actual users, who are retrieved with an SQL query (the keyword `all` in the SQL expression tells that the query returns a list of values):

```
event createDocument (user, …)
  if user in [all user from
              users_to_create_docs] then
    accept;
  else
    reject;
  end;
end;
```

Alternatively, user objects could contain an attribute which stores a list of operations that the user is allowed to perform:

```
event createDocument (user, …)
  if "create doc" in
        user(user).permissions then
    accept;
  else
    reject;
  end;
end;
```

The type of the `permissions` attribute would be a set of values from a list that contains all possible operations.

The authorization program defines for each operation the conditions under which the operation is allowed. Typically the conditions refer to the current state of the object. The following procedure, for example, says that a document version can be locked for modification only in the *draft* state (the current state of an object is available as a system attribute):

```
event lockDocVersion (user, version)
  if version.current_state != "draft"
  then
    reject "can only lock versions ",
           "in 'draft' state";
  end;
  …
end;
```

Sometimes it would be more natural to tell for each state which operations are allowed in that state. This information can be coded in a database table `allowed_ops` with two columns for the states and the allowed operations. For example, the fact that a document version in the draft state can be locked would be represented with a row with values "draft" and "lock" in columns `state` and `op`, respectively. The authorization program would look as follows:

```
event lockDocVersion (user, version)

  checkState("lock", version);
end;

procedure checkState (operation, object)
  state := object.current_state;
  if not (operation in
         [all op from allowed_ops
          where state = :state]) then
    reject "operation `", operation,
           "' not allowed in state `",
           state, "'";
  end;
end;
```

## Built-in Rules and Authorization Procedures

Some database consistency rules are dictated by our data model and are therefore coded directly in the server program. For example, a document version can never be locked for modification if a new version has already been derived from it.

Otherwise, we try to replace server program code with authorization procedures. For example, earlier versions of the server enforced the rule that a document version could only be unlocked by the user who had originally locked it. Although in general this rule makes sense, it may be useful to allow an EDMS administrator to unlock a document version if the original user has for example forgotten to do it before leaving for a vacation.

The server now allows any user to unlock any document version; the operation must be restricted with an authorization procedure. If the administrator user has the name "edms_admin", one can for example write the following procedure (the system attribute `locked_by` of a document version tells who has locked the version):

```
event unlockDocVersion (user, version)
  if user = "edms_admin" or
     user = version.locked_by then
    accept;
  else
    reject "must be unlocked by ",
           "'edms_admin' or `",
           version.locked_by, "'";
  end;
end;
```

Originally the goal of authorization procedures was to provide extra control beyond the rules of the basic system. The system would work in a sensible way even without any authorization program. However, it is now virtually mandatory to have some authorization procedure for unlocking. One possibility is to change the server to provide default rules. For instance, if no authorization procedure is given, the server uses the original rule that a version can only be unlocked by the user who has locked it.

## Attribute Dependencies

The authorization procedures can also check attribute values. For example, suppose documents have attribute `language`, which tells the language in which the document is written, and attribute `size`, which tells the paper size. Now we want to enforce the following rule: language and size must be specified and if the language is “Finnish”, the size must be “A4” or “A5” (the European “letter size” or half of it). This corresponds to the following authorization procedure. (Argument `attrNames` contains the names of the attributes to which the user has given initial values and argument `attrValues` contains the corresponding values.)

```
event createDocument
        (user,
         docName, docType,
         attrNames, attrValues)
  lang := valueOf("language",
                  attrNames, attrValues);
  if lang = NULL then
    reject "language must be specified";
  end;
  size := valueOf("size",
                  attrNames, attrValues);
  if size = NULL then
    reject "size must be specified;
  end;
  if lang = "Finnish" and
     (size != "A4" and size != "A5") then
    reject "size must be A4 or A5";
  end;
  …
end;

function valueOf (key, names, values)
  foreach name,val in names,values do
    if name = key then
     return val;
    end;
  end;
  return NULL;
end;
```

The function `valueOf` finds the value of a given attribute from two lists of attribute names and values. The `foreach` statement scans the lists `names` and `values` in parallel and assigns each pair of values from the two lists to variables `name` and `val`.

We must also write procedure `changeAttributeValues`, which is executed when a user tries to update attributes of an existing document.

## 4 COMMIT PROCEDURES

EDMS requests can be made to update user-defined attributes automatically by means of a *commit program*. A commit program is similar to an authorization program. However, whereas an authorization program allows an administrator to describe conditions under which a server request is accepted or rejected, a commit program allows the administrator to associate additional attribute modifications with a server request.

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.

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, create new objects or update attribute values.)

(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.

As a variation of the language and size assignments in the previous section, we allow the user to leave the size empty, in which case the following commit procedure sets the size to “A4” if the language is “Finnish”.

```
event createDocument (user, docName, …)
  doc := document(docName);
  if doc.language = "Finnish" and
     doc.size = NULL then
    update doc.size := "A4";
  end;
  …
end;
```

We must of course also modify the authorization procedure to accept an unspecified size.

At the moment commit procedures can update the database only by changing values of attributes defined by the administrator. Commit procedures could easily be extended to include operations for changing object states. For example, suppose state graphs are used for both document versions and individual subdocuments. One could then have a commit procedure that moves the document version to the *draft* state whenever any subdocument is moved to the *draft* state.

## 5 ELECTRONIC MAIL

An authorization procedure or a commit procedure can contain a `mail` statement, which sends electronic mail to a given user. For example, the following authorization procedure sends mail to the department manager when a user tries to make an unauthorized state transition.

```
event changeState (user, obj, newState)
  ...
  mgr := managerOf(obj);
  if user != mgr then
    mail mgr : -- recipient
         "EDMS violation", -- subject
         user, " tried to move ",
         obj, " to state ", newState;
    reject "you cannot do that";
  end;
  ...
end;
```

More typically, though, mail is sent from commit procedures. The following commit procedure sends mail to a manager when a document version in his or her department is moved to the state "ready" (indicating that the version should be checked for approval).

```
event changeState(user, obj, newState)
  if kind(obj) = "doc-version" and
     newState = "ready" then
    mail managerOf(obj) :
         "EDMS check",
         user, " marked ", obj,
         " ready for checking";
     -- should be 'deferred_mail'
     -- (see the text below)
  end;
end;
```

Remember, however, that the commit procedure is executed immediately before the transaction is committed. In theory, it is possible that the commit procedure is executed and then the actual commit operation fails so that the state transition is not recorded in the database after all. Therefore, the mail statement in the above commit procedure should be replaced with a `deferred_mail` statement. Instead of sending mail immediately, this statement saves the message and sends it only after the transaction has been committed successfully.

## 6 CONCLUSIONS

A programmable authorization mechanism provides the necessary flexibility for implementing many different kinds of authorization rules. In contrast to the authorization mechanism in many other data management systems, the administrator not only modifies the data in some pre-defined authorization tables but also writes the rules that tell what kind of tables are used and how their contents are interpreted. Since the rules can access attribute values, one can also write authorization rules that depend on the properties of the objects being manipulated.

The authorization procedures described in this paper are only called when a user tries to update the database. However, it is often also necessary to control read operations. A user can, for example, ask for all documents with certain attribute values, and for each found document, 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.

In principle, 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 needed. This is still an open question.

## ACKNOWLEDGEMENTS

The system is developed as a joint project with KONE Elevators. The project is partly funded by the Finnish Technology Development Centre (TEKES) as part of the SIMSON program of the Federation of the Finnish Metal, Engineering and Electrotechnical Industry. We gratefully acknowledge the guidance and support from Asko Martio and Matti Eskola at KONE Elevators.

## REFERENCES

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

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

Peltonen, H. and Alho, K., 1993, "EDMS Administrator's Guide", Technical Report TKO-C60, Laboratory of Information Processing Science, Helsinki University of Technology.

Peltonen, H., Männistö, T., Alho, K., and Sulonen, R., 1993, "An Engineering Document Management System", *ASME Winter Annual Meeting*, paper 93-WA/EDA-1.

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