Regular Paper

A Unified Framework for Evaluating Data-Dependent Access Control Systems

Bat-Odon Purevjii,† Masayoshi Aritsugi,†
Yoshinari Kanamori† and Cherri M. Pancake††

We present a flexible framework for evaluating data-dependent access control systems. Based on logical formalism, the framework is general enough to simulate all existing systems. In this paper, we analyze and compare currently available access control systems and demonstrate how they can be simultaneously extended and simplified using our framework. A series of examples and a cross-comparative analysis clearly demonstrate the advantages of our framework over previous methods.

1. Introduction

Fine-grained access control has become a crucial issue for some applications of computing and information systems. Medical information systems and e-commerce enterprises are just a few of such applications. A large amount of data in those information systems are stored and manipulated in underlying database management systems. Since data in relational database is accessed through SQL queries and is manipulated by set oriented relational operations, deploying fine-grained access control in database systems is more challenging than systems based on operating systems, flat files, or HTML and XML documents. In particular, enforcing and administering data-dependent access control in the level of a single cell or row in database systems is a non-trivial problem. There is no direct way to assign and manage authorization privileges on a single cell or row of relational tables. Some data-dependent access control systems have been proposed since the origin of the early database systems [10),15),25)]. Recent works improved them and had been implemented within the database systems [18),22)].

Enforcing security policies for fine-grained data-dependent access control directly by application codes is usual in database systems. There is no straightforward way to analyze such application oriented policies. Policy specification, analysis, and maintenance are failure-prone in this approach, since fine-grained authorization objects are specified and manipulated separately from the actual authorization rules. That separation makes the process of analyzing and maintaining policies even more difficult. Because of its complexity, database security administration is often ignored at fine granularity, with security administered mostly at the level of entire tables (not cells or columns [23)]). There is a strong need for a unified framework that is capable of simulating, analyzing, and simplifying data-dependent access control in databases. We believe that formal, yet simple, policy specification languages can provide a practical foundation for achieving this goal.

Various kinds of logic-based access control models and languages have been proposed in the literature [1)]. Their advantages include clean foundations, flexibility, expressiveness, and declarative nature [2),5),24)]. However, since the control models are not targeted to database systems [23)] there is a significant gap between how policy can be specified and the underlying mechanisms that are actually available to enforce policy specifications. To derive a formal framework requires exploration and cross-comparison that can reveal how the systems differ in terms of the granularity of authorization objects, features of policy languages, the relative complexity of policy administration and analysis, and so on.

In this paper, we present a logical framework that is capable of simulating and evaluating existing data-dependent access control systems. The framework is based on a simple logical language. Having a formal description language for data-dependent access control offers a number of advantages. It becomes possible to compare the features of distinct systems and to identify which are most promising in terms of

† Gunma University
†† Oregon State University

practical importance.

The framework allows us to specify and manipulate fine-grained authorization objects and the corresponding authorization rules in a unified way. Consequently, it offers a way to evaluate policy analysis and update complexity in the systems, while remaining independent of them. We used this platform to investigate access control systems that are proposed by research papers [10),15),22)], major database textbooks [8),21)] and vendor documentation [16),18)]. Based on that investigation, we report differences and capabilities of the systems.

The remainder of the paper is organized as follows. Section 2 describes related work. In Section 3, we introduce our access control framework with its authorization description language. We simulate and evaluate existing access control systems in Section 4. In section 5, we report their access control capabilities and their complexities of policy analysis and update compared to our framework. Our conclusions and directions for future work are described in Section 6.

2. Related Work

Fundamental investigations on data-dependent access control for relational databases were done in System R as a view-based mechanism [10)] and in INGRES as a query-modification mechanism [25)]. The former approach is standardized in SQL and is widely employed in many commercial products available on the market today. The latter has successfully been deployed in Oracle under the name of Virtual Private Database (VPD)[16),18)], and recent work [22)] has rectified some drawbacks of the original VPD query-modification approach. None of the systems supports a formal framework that is capable of specifying, analyzing and updating access control policies in a unified way. We partially reported the complexities of policy administration in the existing systems in a previous paper [20)]. (Comprehensive lists of materials on database security were reported in Refs. 6), 8)).

Among the formal approaches, Refs. 4), 11) offer the most flexible frameworks and are therefore the most directly comparable with our work. Reference 11) is a powerful formal language supporting multiple kinds of high-level security policies for centralized data systems. Although it offers richer kinds of policies than our framework, it cannot be used to define fine-grained authorization objects as ours can; the language simply has no appropriate constructs. Similarly, although Ref. 4) introduced a mediator-based enforcement mechanism, its data granularity is defined at the table level only.

The authors of works Refs. 12), 13) employed Prolog to define security policies. They focused on extending an industry standard architecture, however, and did not include policy analysis and update.

With the emergence of XML technologies, several recent efforts [3),7),19)] have proposed fine-grained access control for XML-based formats, in particular Ref. 14). The authors classified such mechanisms as built-in, view-based, pre-processing and post-processing approaches. (A short introduction of these approaches will be given in Section 4.) A similar classification with slight changes will be employed in our paper, but we subsume "pre-processing" within our query-modification category and we do not consider post-processing. Investigation into policy semantics in several promising XML access control models has been carried out in Ref. 9). The authors claim that a simple and unambiguous language is needed to state the declarative semantics of any XML access control policy. Similarly, our declarative language supports a simple and unified way to define policies in RDBMSs.

3. Formal Access Control Framework

3.1 Preliminaries

Datalog [26)] is a version of Prolog and is employed in databases as a declarative language. Tables are expressed in Datalog by *predicate* symbols. A fixed number of arguments are assigned to each predicate and the arguments correspond to attributes of tables. The predicate followed by its arguments is called an *atom*. An atom consisting of the predicate p with arguments $A_1, A_2, \ldots, A_n$ takes the form $p(A_1, A_2, \ldots, A_n)$. We can assume a predicate as the name of a function that returns a boolean value true or false. If T is a table that has n attributes in some fixed order, then we may use t as the name of a predicate corresponding to this table. The predicate t is true for its arguments if those arguments form a tuple of the corresponding table. That means a tuple in a table corresponds to a $fact$ in a predicate. An argument in Datalog can either be a variable or a constant. In an extended form of Datalog,

given in the area between page 736 and page 753 of the second volume of Ref. 26), function symbols can be used as arguments and an argument takes the form $f(B_1, B_2, \ldots, B_n)$ and, principally, it is as expressive as Prolog. Prolog can be used to declare facts about objects, which are denoted by arguments, and their relationships, which are denoted by predicates, to define *rules* about objects and their relationships, and to ask *questions* about objects and their relationships. Since $B_1, B_2, \ldots, B_n$ are also arguments, they can either be a variable, constant or a function as well. Predicates, functions and constants begin by lowercase letters and variables begin by uppercase letters. We will follow this convention only when defining logical rules throughout the paper; the capitalization is not applied for any other definitions and examples. Numbers are used as constants. The atoms can also be constructed with the arithmetic comparison predicates, $>$, $\leq$, and so on; these predicates are called as *built-in* predicates. Atoms with built-in predicates are written in the usual infix notation, e.g., $X < Y$ instead of $< (X, Y)$. The arithmetic comparisons are equal to the names of relations that contain all the true pairs. Logical statements, called *rules*, are written in the form $q \leftarrow p_1 \ \& \ p_2 \ldots \& \ p_n.$, where q is the *head* and $p_1 \ \& \ p_2 \ldots \& \ p_n$ is the *body* of the rule. As we see a body consists of one or more atoms, called *subgoals*, and the atoms are connected by (logical) AND. The symbol $\leftarrow$ denotes *"if"* condition. We translate the head q is true if all subgoals $p_1 \ \& \ p_2 \ldots \& \ p_n$ in the body are true. An alternative translation is "to solve q solve $p_1 \ \& \ p_2 \ldots \& \ p_n$". Let us assume two tables EMPLOYEES(Name, Dept) and DEPARTMENTS(Dept, Manager) to demonstrate expressive and computational power of Prolog. The logical rule $manages(Name, Manager) \leftarrow employees(Name, Dept)$ & $departments(Dept, Manager)$. defines a new knowledge or table MANAGES on the two other tables. The *manages* predicate returns true if *employees* predicate and *departments* predicate return true, that is, there exists a tuple $(bob, john)$ in MANAGES table if there exist a tuple $(bob, finance)$ in EMPLOYEES table and a tuple $(finance, john)$ in DEPARTMENTS table. In other words, we compute the body of the rule to compute the head of the rule. We can ask a question "? - $manages(bob, Manager)$." from a Prolog interpreter to find who manages Bob. The Prolog interpreter returns names of all managers who manage Bob's department.

3.2 Basic Components

As basic building blocks of our access control framework we assume a set of database users, a set of fine-grained objects derived from database tables, and a set of SQL data privilege modes, or just privileges, *select*, *insert*, *delete*, and *update*. In this framework *authorization rules* describe who can execute which privilege on what objects. The fine-grained authorization objects are implicitly defined within the authorization rules. We give the formalism of the mentioned components in the next subsection. In general, an access control policy is mapped or transformed into a series of several authorization rules; in the simplest case, we may assume an authorization rule as an implementation of a security policy. We consider several domains $d_1, d_2, \ldots, d_n$ in this framework. A non-exhaustive list of the domain set is $d = \{$*integer, real, char, string, boolean, date, time, time-stamp, user-defined type*$\}$. We define here a special domain set sd as $sd = \{$*char, string, user-defined type*$\}$ and the user-defined type here denotes only character and string based types.

3.3 Authorization Description Language (ADL)

We now give the definition of our language and, based on that, the definition of authorization rules. ADL consists of the following constructs:

- A set of constant symbols $a, b, c, d, e, \ldots,$.
- A set of variable symbols $X, Y, Z, V, U, \ldots,$.
- A set of predicate symbols $p, q, r, \ldots,$.

In addition, *tab* is a predicate symbol. The terms "predicate symbol" and "predicate" are used interchangeably throughout the paper. The symbols express tables and appear in the body of logical rules. Each predicate symbol has a fixed *arity*, that is, the number of arguments, and the arities correspond to the number of attributes of tables. In addition, $permitted, time, oType, oExclude$ and *constraint* are predicate symbols used within logical rules as well. The definition of these predicate symbols is given separately.

- A set of built-in predicates.
- A set of function symbols $f, g, \ldots, .$

The terms "function symbol" and "function" are used interchangeably throughout the paper. The number of arity of a function symbol is fixed as well. The function symbols help us to

define fine-grained authorization objects within the *permitted* predicate given in the next definition. Additionally, $current, type_k, semantic_k$ and $user$ are function symbols that can be used within the predicates $time, oType, oExclude$ and $constraint$, respectively. The definition of the mentioned predicates is given separately. Moreover, $ftab$ is always a function symbol.

- A predicate *permitted*.

It is a 3-ary predicate that takes the form $permitted(s, o, m)$, where s, o, m are expressions. s and m can either be a constant symbol or a variable symbol and o can be a function symbol. The domain of s is a set of ID's, and the domain of m is data privileges of SQL. The latter domain has fixed number of members, namely, *select, insert, delete and update*. We assume a fixed database schema that has k number of tables. The function o has k number of arguments and each argument is a function in turn. It takes the form $o(g_1(X_1,X_2,\ldots,X_j),\ldots,g_k(Y_1,Y_2,\ldots,Y_l))$, where the function symbols $g_1,\ldots,g_k$ correspond to tables of the database and the variable symbols $X_1, X_2, \ldots, X_j$ and $Y_1, Y_2, \ldots, Y_l$ correspond to attributes of the tables. The arities of $g_1, \ldots, g_k$ are equal to the numbers of attributes of the corresponding tables. The semantics of function o might be defined as follows. It takes k domain elements, functions in turn, and returns an "object" of type "record of k functions". Each function takes the corresponding number of domain elements and returns an "object" of type "record of i elements", where i varies on each function. Then, we can assume o as a function that takes h domain elements and return "object" of type "record of h elements", where h is equal to the number of all attributes of a given database. When specifying logical rules in our framework we omit unnecessary function symbols and arguments from them and use view names as function symbols for space utilization. However, we still assume all function symbols and arguments are in place in each logical rule.

Now we give definitions of other predicates. We call them additional predicates to refer later.

- A predicate *time*.

It is formed as $time(current(X))$. The time instances can be defined by this predicate.

- A predicate *oType*.

It is formed as $oType(type_k(e), d_i)$ where $type_k(e)$ denotes a function that return the type of the k-th attribute of table e and d_i denotes a particular type or domain. This predicate returns true if the type of the attribute is same as d_i.

- A predicate *oExclude*.

It is formed as $oExclude(semantic_k(e), b)$, where $semantic_k(e)$ denotes a function that returns semantics of k-th attribute of table e and b is a value in sd that denotes an arbitrary text. The predicate returns true when the attribute does not contain the given text.

- A predicate *constraint*.

It is formed as $constraint(user(Y))$. IDs of users can be defined by this predicate.

The constant, variable, predicate and function symbols all can be subscripted depending on usage.

Authorization rule is defined in this subsection and examples on it are given in the next subsection.

Definition (Authorization rule). An authorization rule is a logical rule that takes the form:

$permitted(s, o, m) \leftarrow N_1, N_2, \ldots, N_n.$

where s, o and m are defined previously and $N_1, N_2, \ldots, N_n$ are subgoals. A subgoal N_i can either be a predicate, that corresponds to a table, a built-in predicate or an additional predicate.

Before providing some examples of authorization rules, we discuss completeness of the proposed language. In order to verify the complete correctness of the language we need to employ some higher-order logics to specify its constructs. It will be hard to implement such higher-order logic languages in real systems. However, we believe, all necessary functions we need are possible to implement if we make some restrictions in such higher-order languages. This will be included in our future work.

3.4 Examples of ADL Specifications

In this subsection we give some examples of authorization rules defined in ADL.

Example 1. Consider the following authorization rule assuming that there is a table with schema TAB1(X_1,X_2,X_3,X_4). Let us assume $s1$ denotes a user ID. In addition, let us assume tab_1 and $ftab_1$ denotes corresponding predicate symbol and function symbol for TAB1 table. As mentioned before the capitalization convention is not strictly followed in the definition of the table schema.

$permitted(s1, ftab_1(X_2, X_3), select) \leftarrow$

$tab_1(X_1, X_2, X_3, X_4)$ &
$X_3 > 20$.

Meaning of the rule: Assume that the variables in the rule ranging over all possible values. If there is a fact (a_1, a_2, a_3, a_4) that makes tab_1 true and $a_3 > 20$ is true then there is a fact that says "$s1$ is permitted to "*select*" a_2 and a_3". In other words, if there is a tuple (a_1, a_2, a_3, a_4) in table TAB1 and $a_3 > 20$ then user $s1$ is permitted to select the subset (a_2, a_3) of tuple (a_1, a_2, a_3, a_4).

Example 2. Consider the following authorization rule assuming that there is a table with schema TAB2(Y_1,Y_2,Y_3). In addition, let us assume tab_2 and $ftab_2$ denotes corresponding predicate symbol and function symbol for TAB2 table.

$permitted(s2, ftab_2(Y_3), select) \leftarrow$
$tab_2(Y_1, Y_2, Y_3)$ &
$oExclude(semantic_3(tab_2), c)$.

This rule states that user $s2$ is permitted to *select* the value of attribute Y_3 if it does not include c. c can be a text like cancer and so on.

4. Investigation of Access Control Systems

In this section we analyze the access control systems given in Refs. 8), 10), 15), 16), 18), 21), 22). The approaches they use can be classified generally as built-in [22], view-based [8),10)] or query-modification [15),18)]. In the first, authorization evaluation is done during query processing, while in the other two it is performed ahead of time. (The Non-Truman model [22] is the only system that actually performs authorization evaluation during query processing; evaluation is done by the query optimizer.) In the following subsections, we analyze the granularities they support, the languages used for access control, and the complexity of policy administration, analysis and update.

The authors of the system [22] considered a very specific kind of authorization or inference information that cannot be expressed within view definitions. The authors claim that the inference information depends on some database states and users may be aware of these states. Because the inference information depends on users' knowledge, that is not completely predictable, it is hard to express it in logical forms. The issue needs more investigation and we do not consider it here. The authors' examples on views, without the inference information, can similarly be expressed with view-based systems.

Database schema:
EMPLOYEE(Name, Salary, Manager, Department)
DEPT(Department, Floor)

View definitions:
a. CREATE VIEW VEMP AS
SELECT Name, Salary
FROM EMPLOYEE
WHERE Department = 'toy'

b. CREATE VIEW LOCEMP AS
SELECT Name, Floor
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.Department = DEPT.Department

Authorization rules in SYSAUTH table:

No	Userid	Tname	Type	Privilege
1	user1	EMPLOYEE	base table	select
2	user1	DEPT	base table	select
3	user2	VEMP	view	select
4	user3	LOCEMP	view	select

Fig. 1 Database schema, view definitions and authorization rules.

Thus, we do not provide concrete examples; however, we introduce some useful concepts and unique features from the system at the end of this section.

4.1 View-based Systems

Griffiths et al.'s system [10)] is typical of the view-based approach. Fine-grained authorization objects are defined in terms of views, with authorization rules defined on the views and are stored in a separate system table. Examples of database schema, authorization rules and view definitions are given in **Fig. 1**.

Granularity and language for access control: The smallest authorization object is a view on a table. A view can be defined on a row of a table as well as an individual cell; however, users must write individual definitions for each view in order to implement row-level protection. This is not practical for systems with large number of users and objects. The system does not support a formal language for specifying and analyzing access control policies. The author of a recent view-based system [8)] described a hypothetical language to specify authorization rules, but the language has no facilities to analyze and maintain authorization rules. One interesting idea the author mentioned is to assign a name to each authorization rule to possibly help in checking policy conflicts in database systems. He also noted the possibility of integrating context-based access control into view-based systems. As an example, the author provided the following view definition.

Example 3a. A view definition based on context.

```
CREATE VIEW S_NINE_TO_FIVE AS
```

SELECT *
FROM EMPLOYEE
WHERE Current_time ≥ Time '09:00:00'
AND Current_time ≤ Time '17:00:00';

We assume N, S, M, D, and F correspond to Name, Salary, Manager, Department, and Floor, respectively.

Example 3b. The corresponding authorization rule on the previous view can be specified by ADL as follows:

$permitted(user1, employee(N, S, M, D), select) \leftarrow$
$employee(N, S, M, D)$ &
$time(current(X))$ &
$X \geq 9$ &
$X \leq 17$.

(Note that for readability and space utilization we shorten the names of arguments in logical specifications; this convention is followed throughout the paper.)

Authorization evaluation: The authorization evaluation is very simple in this system. Authorization objects and the corresponding authorization rules are defined in advance. Generally, queries are written in terms of views and the authorization subsystem evaluates each query by checking user privileges on the requested views. A query is accepted if the user holds all corresponding privileges on the views (and is rejected otherwise).

The following example shows how ADL handles authorization rules defined on the views and base relations in Fig. 1. It demonstrates that authorization objects and rules are specified in a unified way in our framework.

Example 4.1. The corresponding specifications in ADL framework are:

(1) $permitted(user1, employee(N, S, M, D), select) \leftarrow$.
(2) It can be specified as 1.
(3) $permitted(user2, vemp(N, S), select) \leftarrow employee(N, S, M, toy)$.
(4) $permitted(user3, locemp(N, F), select) \leftarrow employee(N, S, M, D)$ & $dept(D, F)$.

Example 4.2a. Policy analysis in a view-based system. Let us assume that the security administrator needs to know which users are SELECTing values of *Name* and the *Salary* attribute of employees who work in the toy department. The goal can be achieved in view-based system, but it can be quite complex way. The process goes as follows:

(1) Search the system catalog for all view definitions that include *Name* and *Salary* attribute in the SELECT clause and "Department = 'toy' " predicate in the WHERE clause.
(2) If there exists such a view definition make a query on the system authorization table to retrieve user IDs who have the select privilege on this view.
(3) Make a query on VEMP to retreive the names and salaries of the employees.

Example 4.2b. Same policy analysis using the ADL framework. This goal can be achieved in our framework in two steps as follows:

(1) Search for an authorization rule that includes *Name* and *Salary* arguments in the second argument of the head and the constant "toy" in the fourth argument of predicate "employee" (in the body).
(2) Ask the following question to retrieve the values.
?- $permitted(X, vemp(N, S), select)$.

Example 4.3a. Policy update in a view-based system. Let us assume that an new individual ($user4$) is hired for the toy department. Consequently the administrator needs to update the definition of VEMP as follows:

CREATE VIEW VEMP AS
SELECT Name, Salary
FROM EMPLOYEE
WHERE Department = "toy" and Salary ≤ 2000;

and assign the select privilege to the new user. The goal can be achieved in view-based systems as follows:

(1) Define the new view definition.
(2) Define the new authorization rule on the view.

Example 4.3b. Same policy update using the ADL framework. This goal can be achieved by defining a new rule in one step as follows:

$permitted(user4, vemp(N, S), select) \leftarrow$
$employee(N, S, M, toy)$ &
$S \leq 2000$.

4.2 Query-modification Systems

In this section we consider Oracle VPD [16),18)] as an examplar of query-modification systems. With this approach, authorization objects are not defined in terms of views; instead, fine-grained authorization objects are generated dynamically.

Granularity and language for access control: Row-level access control is enforced

in VPD with the help of a proprietary procedural programming language, PL/SQL. To enforce access control at the cell-level, this system must be combined with a view-based mechanism. Authorization policies and corresponding policy functions are created in PL/SQL. The system does not support a formal language for specifying and analyzing access control policies. However, there are some facilities for getting information on defined policies and the corresponding policy functions.

Domain-relational calculus is used to define authorization rules in an alternative query-modification system [15]. In this case, however, the language has no facilities to analyze or maintain authorization rules. The author of the system noted an interesting access control case, where a policy was defined for instances from the same table, which had the schema EMPLOYEE (Name, Title, Salary) and the following view definition.

Example 5a. A view definition based on recursion.

CREATE VIEW EST AS
SELECT Name1, Title, Name2
FROM EMPLOYEE
WHERE EMPLOYEE:1.Title =
EMPLOYEE:2.Title;

In our next example, we assume N1, N2, T, S1 and S2 correspond to Name1, Name2, Title, Salary1 and Salary2, respectively.

Example 5b. The corresponding authorization rule on the previous view can be specified in ADL as follows:

$permitted(user5, est(N1, T, N2), select) \leftarrow$
$employee(N1, T, S1)$ &
$employee(N2, T, S2)$.

Authorization evaluation: Authorization policies are defined separately for each table. Taking the user query, the system reads corresponding policies and calls policy functions. The functions return WHERE clause predicates to be assigned to the original query. The modified query is then executed. For instance, when the doctor whose ID is N makes a query on the PATIENTS table as "SELECT * FROM PATIENTS" the original query will be modified to "SELECT * FROM PATIENTS WHERE doctor_id = N". This database schema and the policy and policy function are portrayed in **Fig. 2**. The values of doctor_id in GET_DOCTOR_ID policy function change dynamically depending on the value of doctor_name, a value assigned through the function call USER (which returns the login ID of the current user).

```
Database schema:
DOCTORS(Doctor_id, Doctor_name, Group_id)
PATIENTS(Patient_id, Doctor_id, Patient_name, Disease)
TREATMENTS(Patient_id, Treatment_dt, Treatment)

A policy on PATIENT table:
begin
  dbms_rls.add_policy
  (
    object_schema     => 'HOSPITAL',
    object_name       => 'PATIENTS',
    policy_name       => 'PATIENT_VIEW_POLICY',
    policy_function   => 'GET_DOCTOR_ID',
    function_schema   => 'HOSPITAL',
    statement_types   => 'SELECT, INSERT, UPDATE, DELETE',
    update_check      => true,
    enable            => true
  );
end;

The corresponding policy function:
create replace function get_doctor_id
(
  p_schema_name     in      varchar2,
  p_table_name      in      varchar2
)
return varchar2
is
  l_doctor_id   number;
begin
  select doctor_id
  into l_doctor_id
  from doctors
  where doctor_name = USER;
  return 'doctor_id = ' || l_doctor_id;
end;
```

Fig. 2 Database schema, a policy on PATIENTS table and the corresponding policy function.

Example 6. The corresponding ADL specifications for the policy and policy function appear below. We assume Did, Dname, G, Pid, P, and D correspond to Doctor_id, Doctor_name, Group_id, Patient_id, Patient_name, and Disease, respectively.

$permitted(Did, patients(Pid, Did, P, D),$
$\quad select) \leftarrow$
$doctors(Did, Dname, G)$ &
$patients(Pid, Did, P, D)$ &
$constraint(user(Dname))$.

Policies in VPD can also be dependent on other policies. For instance, the policy for the TREATMENTS table can depend on the policy for the PATIENTS table. The corresponding policy and its policy function appear in **Fig. 3**.

When a doctor (doctor_id of 2) makes a query against the TREATMENTS table as "SELECT * FROM TREATMENTS" the original query will be modified to "SELECT * FROM TREATMENTS WHERE patient_id IN (SELECT patient_id FROM PATIENTS WHERE doctor_id = 2)".

Example 7.1. The corresponding ADL specifications of the policy and policy function are

```
A policy on TREATMENTS table:
begin
  dbms_rls.add_policy
  (
    object_schema     => 'HOSPITAL',
    object_name       => 'TREATMENTS',
    policy_name       => 'TREAT_VIEW_POLICY',
    policy_function   => 'GET_PATIENT_ID',
    function_schema   => 'HOSPITAL',
    statement_types   => 'SELECT, INSERT, UPDATE, DELETE',
    update_check      => true,
    enable            => true
  );
end;

The corresponding policy function:
create replace function get_patient_id
(
  p_schema_name     in      varchar2,
  p_table_name      in      varchar2
)
return varchar2
is
  l_patient_id  number;
begin
  return 'patient_id_in (select patient_id from patients)' ;
end;
```

Fig. 3 A policy on TREATMENTS table and the corresponding policy function.

shown below. We assume Td and T correspond to Treatment_dt and Treatment respectively:

$permitted(Dname, treatments(Pid, Td, T),$
$\quad select) \leftarrow$
$\quad doctors(Did, Dname, G)\ \&$
$\quad patients(Pid, Dname, P, D)\ \&$
$\quad treatments(Pid, Td, T)\ \&$
$\quad constraint(user(Dname)).$

Example 7.2a. Policy analysis in a query-modification system. Let us assume that the administrator needs to know who is updating *Treatment_dts* and *Treatments* of patients who are under the treatment *refresh*. We have observed that no authorization rules are stored physically. This makespolicy analysis very complex in VPD. The goal could be achieved by the following steps:

(1) Check the policy and policy function corresponding to the TREATMENTS table to know who has update privilege on the particular columns. This cannot be accomplished directly, however, since the policy function itself depends on the policy for the PATIENTS table.
(2) Check the corresponding policy and policy function of the PATIENTS table. After doing this, we know how users are assigned to the privilege, but more effort is required in order to derive the doctor_ids.
(3) Make a query against the TREATMENTS table to retrieve *patient_ids*, *treatment_dts* and *treatments* that correspond to rows having a refresh value in their treatment column.
(4) Make a query on PATIENTS table to retrieve the corresponding *doctor_ids* of the retrieved *patient_ids*.

Example 7.2b. Same policy analysis, using the ADL framework. This is achieved easily with a single step, as follows:

?- $permitted(Dr, treatments(Pid, Td,$
$\quad refresh), update).$

Policy update. To update a policy in VPD it is neceseeary to delete the policy and its policy functions and re-create them. As we showed in the previous subsection, such a policy update is straightforward using our framework.

4.3 Built-in Systems

In this subsection we consider the Non-Truman model [22] as an examplar of built-in systems. This is an improvement of view-based and query-modification systems. Fine-grained authorization objects are defined in terms of (authorization) views in this system. The authors introduced the concept authorization view. Such views can be parameterized, and therefore, there is no need to define a view for each individual user. This helps to reduce policy administration complexity in this system.

Granularity and language for access control: A row-level access control can be enforced in this system. The system does not support a formal language for specifying and analyzing access control policies.

Authorization evaluation: The authorization evaluation of this system is completely different from the previous systems. Queries can be written in terms of views or base relations. The authors proposed a set of inference rules to check query validity. If the query is valid, the system returns the answer by processing the original query. Obviously, the query is rejected when it is not valid. As mentioned in the beginning of this section, the evaluation process is done during query processing in query optimizer.

4.4 Discussion on Updating Costs for Authorization Rules

There is a tradeoff between the costs associated with the unified and separated approaches to updating authorization rules. When simultaneous updates are required to two or more components of an authorization rule, the unified approach is less expensive; however, the second approach is better when components of an authorization rule are updated individually.

Table 1 Comparison on access control systems.

Capabilities / Systems	Context-based access control	Dynamic generation of authorization objects	Recursive access control	Policy administration at different element levels	Policy analysis	Policy update
View-based	+	-	-	Moderate to Complex	Moderate to Complex	Moderate to Complex
Query-modification	-*	+	+	Easy to moderate	Moderate to Complex	Moderate to Complex
Built-in	-*	+	-	Easy to moderate	Moderate to Complex	Moderate to Complex
ADL-based	+	+	+	Easy	Easy to Moderate	Easy to Moderate

This issue will be examined in our future work.

5. Comparison on Access Control Systems

Based on our investigation, we report the different features of access control systems in **Table 1**. Asterisks indicate systems that have no concrete examples of the corresponding features.

Context-based access control can be enforced by defining views in view-based systems. A view definition that depends on context is demonstrated in example 3a in Subsection 4.1. The corresponding specification in ADL is given in example 3b in the same subsection. In addition, our type-based and semantic-based access control scenario, presented in example 2, can be assumed as context-based access control as well. The dynamic generation of authorization objects depending on user IDs can be observed in examples 6 and 7.1 in Subsection 4.2. The first component of the predicate *permitted* in these specifications, denoted by variable symbols, can appear in (relation) predicates as well. This means that the corresponding object and subject components will be changed simultaneously, depending on the attribute values found in some other table. An scenario of recursive access control - access control defined on multiple authorization objects from a single table - was given in example 5a, for query-modification systems, and 5b, for the ADL framework, in Subsection 4.2. All these specifications can be written in a straightforward way in our ADL framework.

The complexity of administrating policies with different granularity levels in the systems was reported in our previous work [20]. The complexities of analyzing and updating policies in the systems are compared in examples 4.2a, 4.2b, 4.3.a, 4.3b, 7.2a and 7.2b.

6. Conclusion

We proposed a rule-based fine-grained access control framework based on a logical language. The language formally specifies policies in terms of the underlying data elements of relational databases. The framework allowed us to simulate the existing systems and to evaluate the complexities of policy administration, analysis and update within the systems. We found that existing systems have no flexible facilities for policy analysis and updating.

Our framework provides a simple way to specify, analyze and update policies. The only things we could not simulate were the inference rules given in Ref. 22). We will explore this issue in the future. We intend to implement a prototype system to prove the feasibility of our framework. The comparisons of policy analysis and update given in subsections 4.1 and 4.2 and the update costs of authorization rules will be examined by conducting experiments on the prototype system as well. Comparing our framework to an industrial XML standard, i.e., the XACML [17] is another interesting avenue we intend to explore.

Acknowledgments This research was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 15300029, and Special Project for Earthquake Disaster Mitigation in Urban Areas (MEXT).

References

1) ACM SIGSAC: ACM Symposium on Access Control Models and Technologies (SACMAT). http://www.sacmat.org/.
2) Adabi, M.: Logic in Access Control, *Proc. 18th IEEE Symp. on Logic in Computer Science* (*LICS 2003*), Ottawa, Canada, pp.228–233, IEEE Computer Society (2003).
3) Bertino, E. and Ferrari, E.: Secure and selective dissemination of XML documents, *ACM Trans. Inf. Syst. Secur.*, Vol.5, No.3, pp.290–331 (2002).
4) Bertino, E., Jajodia, S. and Samarati, P.: Supporting Multiple Access Control Policies in Database Systems, *Proc. 1996 IEEE Symposium on Security and Privacy*, pp.94–108, IEEE Computer Society (1996).
5) Bonatti, P.A. and Samarati, P.: Logics for Authorization and Security, *Logics for Emerging Applications of Databases [outcome of a Dagstuhl seminar]*, Chomicki, J., van der Meyden, R. and Saake, G.(Eds.), Springer, pp.277–

323 (2003).
6) Castano, S., Fugini, M.G., Martella, G. and Samarati, P.: *Database Security*, Addison-Wesley & ACM Press (1995).
7) Damiani, E., di Vimercati, S.D.C., Paraboschi, S. and Samarati, P.: A fine-grained access control system for XML documents, *ACM Trans. Inf. Syst. Secur.*, Vol.5, No.2, pp.169–202 (2002).
8) Date, C.J.: *An Introduction to Database Systems*, 8th edition, Addison-Wesley (2003).
9) Fundulaki, I. and Marx, M.: Specifying access control policies for XML documents with XPath, *Proc. 9th ACM Symposium on Access Control Models and Technologies* (*SACMAT 2004*), Yorktown Heights, New York, Jaeger, T. and Ferrari, E.(Eds.), pp.61–69, ACM (2004).
10) Griffiths, P.P. and Wade, B.W.: An Authorization Mechanism for a Relational Database System, *ACM Trans. Database Syst.*, Vol.1, No.3, pp.242–255 (1976).
11) Jajodia, S., Samarati, P., Subrahmanian, V.S. and Bertino, E.: A Unified Framework for Enforcing Multiple Access Control Policies, *Proc. ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, Peckham, J.(Ed.), pp.474–485, ACM Press (1997).
12) Lin, A.: Integrating Policy-Driven Role Based Access Control with the Common Data Security Architecture, Technical Report HPL–1999–59, HP Labs (1999).
13) Lin, A. and Brown, R.: The Application of Security Policy to Role-Based Access Control and the Common Data Security Architecture, *Computer Communications*, Vol.23, No.17, pp.1584–1593 (2000).
14) Luo, B., Lee, D., Lee, W.-C. and Liu, P.: A Flexible Framework for Architecting XML Access Control Enforcement Mechanisms, *Proc. International Workshop on Secure Data Management in a Connected World* (*SDM'04*) Toronto, Canada, Jonker, W. and Petkovic, M.(Eds.), pp.141–155, Springer (2004).
15) Motro, A.: An Access Authorization Model for Relational Databases Based on Algebraic Manipulation of View Definitions, *Proc. Fifth International Conference on Data Engineering*, Los Angeles, California, pp.339–347, IEEE Computer Society (1989).
16) Nanda, A.: Fine Grained Access Control (2003).
http://www.proligence.com/nyoug_fgac.pdf.
17) OASIS Technical Committees: OASIS eXtensible Access Control Markup Language (XACML).
http://www.oasis-open.org/committees/xacml/.
18) Oracle Corporation: The Virtual Private Database in Oracle9ir2: An Oracle Technical White Paper (2002).
http://www.oracle.com/technology/deploy/security/oracle9ir2/pdf/VPD9ir2twp.pdf.
19) Purevjii, B., Amagasa, T., Imai, S. and Kanamori, Y.: An Access Control Model for Geographic Data in an XML-based Framework, *Proc. 2nd International Workshop on Security In Information Systems* (*WOSIS 2004*), Porto, Portugal, Fernández-Medina, E., Castro, J. C.H. and García-Villalba, L.J.(Eds.), pp.251–260, INSTICC Press (2004).
20) Purevjii, B., Aritsugi, M., Imai, S., Kanamori, Y. and Pancake, C.M.: Protecting Personal Data with Various Granularities: A Logic-Based Access Control Approach, *To be published in the Proc. International Conference on Computational Intelligence and Security* (*CIS 2005*), Xi'an, China, Hao, Y., et al.(Ed.), Springer (2005).
21) Ramakrishnan, R.: *Database Management Systems*, WCB/McGraw-Hill (1998).
22) Rizvi, S., Mendelzon, A.O., Sudarshan, S. and Roy, P.: Extending Query Rewriting Techniques for Fine-Grained Access Control, *Proc. ACM SIGMOD International Conference on Management of Data*, Paris, France, Weikum, G., König, A.C. and Deßloch, S.(Eds.), pp.551–562, ACM (2004).
23) Rosenthal, A. and Winslett, M.: Security of Shared Data in Large Systems: State of the Art and Research Directions, *Proc. ACM SIGMOD International Conference on Management of Data*, Paris, France, Weikum, G., König, A.C. and Deßloch, S.(Eds.), pp.962–964, ACM (2004).
24) Samarati, P. and di Vimercati, S.D.C.: Access Control: Policies, Models, and Mechanisms, *Foundations of Security Analysis and Design* (*FOSAD*) Focardi, R. and Gorrieri, R.(Eds.), pp.137–196, Springer (2000).
25) Stonebraker, M. and Wong, E.: Access control in a relational data base management system by query modification, *Proc. 1974 ACM/CSC-ER Annual Conference*, pp.180–186, ACM Press (1974).
26) Ullman, J.D.: *Principles of database and knowledge-base systems*, Vol.I and II, Computer Science Press (1988).

(Received June 21, 2005)
(Accepted October 11, 2005)

(Editor in Charge: *Kaname Harumoto*)

Bat-Odon Purevjii received his B.E. and M.E. degrees from Mongolian University of Science and Technology, Mongolia, in 1997 and 1999, respectively. He is currently a Ph.D. student at the Department of Computer Science, Gunma University, Japan. His research interests include database and XML security.

Masayoshi Aritsugi received his B.E. and D.E. degrees in computer science and communication engineering from Kyushu University, Japan, in 1991 and 1996, respectively. Since 1996, he has been working at the Faculty of Engineering, Gunma University, Japan, where he is now an Associate Professor. His research interests include database systems and parallel and distributed data processing. He is a member of IPSJ, IEICE, IEEE-CS, ACM, and DBSJ.

Yoshinari Kanamori received his D.E. degree from Tohoku University in 1975. Since 1991, he has been a Professor at the Department of Computer Science, Gunma University. His research interests include database systems and image processing. He is a member of IPSJ, IEICE, ACM, and IEEE-CS.

Cherri M. Pancake received the Ph.D. from Auburn University in 1986. She is Professor and Intel Faculty Fellow in the School of Electrical Engineering and Computer Science at Oregon State University, where she also serves as Director of the Northwest Alliance for Computational Science & Engineering (NACSE). Her research focus is usability engineering, specifically applying user-centered design to improve software and data systems used by practicing scientists and engineers. She is a Fellow of both the ACM and the IEEE.