Abstract

Luo Gao. A Toolkit for Automated Fine-Grained Access Control Policy Enforcement in Oracle 9i. (Under the direction of Dr. Ting Yu)

Database access control is indispensable to information system security. As enterprises expand their services to the Internet, it has been widely recognized that traditional relation-level or database-level access control is no longer adequate to handle increasingly complex access control requirements in modern information systems. Instead, fine-grained access control (i.e., row-level access control) is much desired. Though several commercial database management systems support fine-grained access control, it requires security policies to be hard-coded into applications by programmers, which is a very error-prone process. It is very difficult for policy makers to verify whether an application's security requirements are correctly enforced by hard-coded policies. If they fail to detect security flaws in policy implementation, the whole information system may be at grave risk.

To help effectively verify and analyze the enforcement of fine-grained access control, in this thesis we present the design and implementation of a policy management toolkit, access control enforcement toolkit (ACET), which is able to automatically translate formal access control policies to the enforcement program of database fine-grained access control. We discuss the desirable properties of formal policy languages when used to specify database fine-grained access control. We present an automated policy translation algorithm that effectively identifies access control components in formal policies and maps them into basic database access control elements. Our initial evaluation shows that the automatically generated policy enforcement program yields

comparable performance to that developed by programmers. Thus, the toolkit enables policy makers to focus more on fine-grained security policy specification, without worrying the correct and efficient enforcement of database security policies.

A Toolkit for Automated Fine-Grained Access Control Policy Enforcement in Oracle 9i

by

Luo Gao

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH

AUGUST 2004

APPROVED BY

Dr. Annie I. Antón

Dr. Jaewoo Kang

Dr. Ting Yu, Chair of Advisory Committee

To my parents Jialing Shi, ShiZhong Gao and my brother Bo.

Biography

Luo Gao was born in Kunming, Yunnan Province, People's Republic of China. He graduated in 2002 from North Carolina State University with a Bachelor of Science in Computer Science.

Acknowledgments

I would like to take this opportunity to express my sincere appreciation to Dr. Yu, my advisor, for his guidance and constant support. His careful and critical comments significantly improve the content and presentations of this thesis.

My appreciation goes to my other committee members as well, Dr. Antón and Dr. Kang, for their valuable comments, suggestions, and encouragements. I also appreciate C. Powers and Y. Watanabe from IBM who kindly shared their work with me.

I would like to sincerely thank Keith Irwin and Stephen Reece for proof reading my thesis. Thanks are also due to my colleagues and friends at Cyber Defense Lab: Qing Zhang, Kun Sun, and Dingbang Xu

Finally, I would like to thank my parents, brother and my girlfriend Yunhua for their love and support. I am indebted to them and no words can express my appreciation.

Table of Contents

List of Tables

List of Figures

1 Introduction

Databases are widely used to manage and archive large amounts of business information. Proper and effective access control of databases is crucial to enterprise information system security. Unlike system security or network security, which addresses the problem of preventing and detecting attacks from outsiders, the goal of access control is to identify and grant proper privileges to legitimate users.

As the Internet continues to grow, many enterprises offer their services to public via the Internet, Web-service, etc, which makes database access control increasingly challenging. While greatly improving the efficiency, flexibility and availability of enterprises' services, Web-based applications are significantly more complex than traditional information systems. Millions of users may access database services and other resources at the same time, and they may come from different security domains, e.g., part suppliers, partners, customers, etc. As a result, enterprises' access control policies become more and more complex.

It has been well recognized that traditional database-level or table-level access control is not adequate for Web-based applications' security requirements. For example, in a health care system, patients' records are often stored in one table, which may include a patient's identifier, name, date of birth, symptoms, etc. Typically, a user is only allowed to access its own record. Similarly, a doctor should only be able to access his patients' records, but not that of the patients of other doctors. Table-level access control either allows a user to access the whole table (i.e., each record in the table), or have no access to

the table at all. Clearly, such a coarse-grained access cannot express the increasingly complex access control requirements.

To address the above problem, fine-grained access control (FGAC), also known as row level access control, has been proposed [RMS04]. As the name suggests, the basic access control elements in FGAC are the tuples of a table instead of the table itself. FGAC allows a user to access a certain portion of a table. Therefore, it is natural to support flexible access control policies such as those described above.

Several commercial database systems already provide support for FGAC. Representative systems include Oracle's Virtue Private Database (VPD) [Kyt] and DB2's low-level access control [Bir00]. Such features are widely used in Web-based applications. On the other hand, existing FGAC mechanisms require access control policies to be hard-coded into a database by programmers, which is a very error-prone process. To realize the access control requirements of an application, it not only depends on the policy maker to correctly specify access control policies, but also depends on how well programmers understand those policies. If logic errors are introduced in the enforcement code, due to either a programmer' misunderstanding of the policy or his/her negligence, the security of the whole system may be at grave risk. Thus, it is very important to verify that access control policies are correctly enforced by an application. However, since the policy enforcement program is written in general programming languages and is embedded in applications, such verification is very hard.

Taking Oracle 9i as an example, an access control enforcement program may be similar to the following:

```
create function my_security_function( p_schema in varchar2,
                                      p_object in varchar2 ) return varchar2
as
begin
  if (sys_context(“userenv”, “role”) = 'MGR' ) then
        return 'MGR = sys_context(“userenv”, “session_user”)
                OR
                EMP = sys_context(“userenv”, “session_user”)’;
  elsif(sys_context(“userenv”, “role”) = ‘EMP’) then
        return ‘EMP =sys_context(“userenv”, “session_user”)’;
   elseif(sys_context(“userenv”, “role”) = ‘CEO’) then
        return  ‘1=1’
 else return ‘1=0’;
  end if;
end;
```

The above code states that if a login user is an employee, the user can read his/her own record. If the user is a manager, then the user can access the records of all the employees who work under him/her plus his/her own record. If the user is a CEO, he can view everything within the table.

If there are thousands of such lines of codes, one can imagine how hard it will be to verify that they have correctly enforced access control policies.

In this project, we address this challenge by developing an access control enforcement toolkit (ACET) that can simplify the creation of access control enforcement program and make access control policy analysis easier than analyzing policies written in database programming language. The essential idea is that, since it is difficult to analyze and verify access control enforcement code, it is desirable to have policies specified in a high-level policy language, which can be formally analyzed. Then the toolkit automatically generates access control enforcement program based on high-level access

control policies, which will eliminate potential logic errors introduced by programmers. The following figure (Figure 1.1) further illustrates the idea of this toolkit.

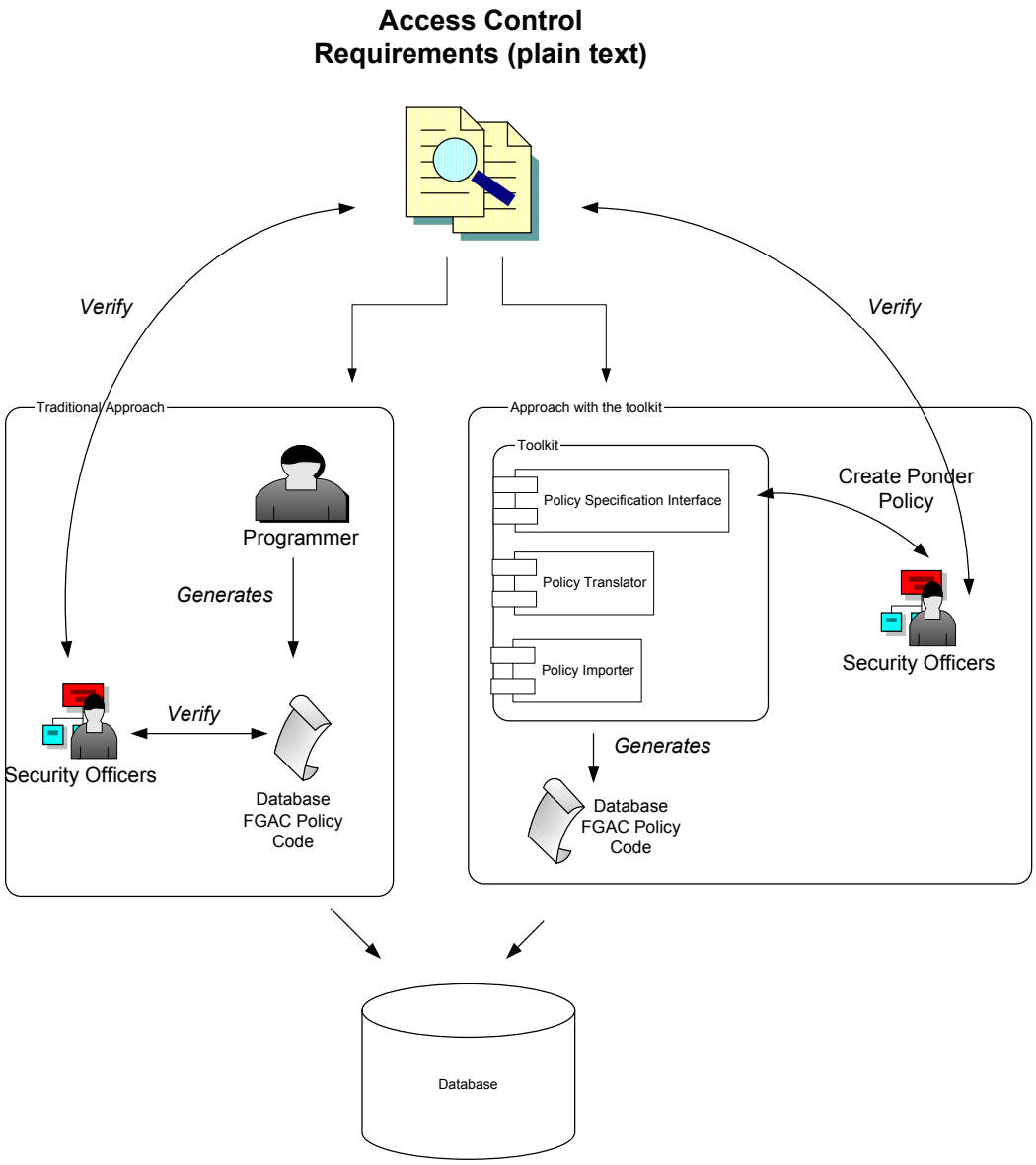


Figure 1.1 General architecture of the ACET

Instead of letting a programmer create policy enforcement program directly from access control requirements, this toolkit allows a policy maker to formally define a policy by using a high-level policy language Ponder [DDL01]. After the formal policy has been

verified and analyzed, they will be automatically translated into the enforcement programand imported into a database server. The key to this toolkit is to develop a policy specification model that is abstract enough so that it can be expressed by formal policy languages. Meanwhile, the policy model also needs to be specific enough so that it can be easily mapped into database management systems where data are stored.

Specifically, this toolkit has three major modules: a policy specification interface; a policy translator that translates formal policies into enforcement code; and an importer that imports generated enforcement program into a database system:

- *Policy specification interface*: This module allows users to define an access control policy by using a high-level policy language such as Enterprise Privacy Authorization Language (EPAL) [AHC03], Rei[KFJ03], or Ponder[DDL01]. This interface will help a policy maker specify formal access control policies.
- *Policy Translator*: This module translates a formal access control into the database enforcement code. In order to do so, the translator will analyze the policy first and identify all the necessary access control elements, which need to be mapped to corresponding database principals, objects and operations.
- Policy *Importer*: This module is responsible for generating auxiliary functions, which are necessary for the automatically generated enforcement program to take effect in database access control.

The developed toolkit offers the following benefits:

- This toolkit provides a user interface to allow a policy maker to create an abstract data model to represent access control policies. We hope it can a policy maker to check whether a policy has been properly specified.

- Since policy enforcement program is automatically generated and imported into databases, human errors are reduced.
- Instead of worrying about correct implementation of access control policies, policy makers are able to focus more efforts on policy specification.
- We discuss in detail how to optimize the performance of machine-generated code. Our preliminary experiment results show that automatically-generated enforcement program yields comparable performance to that written by programmers. Therefore, by using the toolkit, we hope policy makers can enjoy ease of policy management as well as efficient access control

The rest of the thesis is structured as follows. Chapter 2 provides background information about Oracle's fine-grained access control mechanism and some current existing policy languages. Chapter 2 also provides rationale for choosing policy language. In chapter 3, we discuss how formal policy languages can be used to express fine-grained access control policies for databases. We also describe the algorithm to translate formal policies into Oracle policy enforcement code. Chapter 4 evaluates the performance of the enforcement program generated by the toolkit, and discusses a variety of optimization techniques. We briefly describe the toolkit's user-interface in chapter 5. We conclude this thesis in chapter 6 and discuss possible directions for future work.

2 Background Information

In this section, we will compare the Oracle's FGAC mechanism with DB2's FGAC mechanism and explain why we use Oracle instead of DB2. We will then describe Oracle's fine-grained access control mechanism and analyze its advantages and disadvantages. This will help us identify the desirable properties that a formal policy language should have in order to support database FGAC. We then examine several policy languages in the literature, including Rei[KEF03], EPAL[AHC03] and Ponder[DDL01] and analyze their suitability for specifying database fine-grained access control policies.

2.1 DB2's FGAC mechanism

Several commercial database systems provide mechanisms to support fine-grained access control. Examples include Oracle's virtual private database (VPD) [Kyt] and DB2's low level's access control [Bir00].

DB2 uses views as the primary instrument to implement FGAC. For a list of policies, DB2 creates a view for each policy. In each view, it defines the policy constraints. In order to properly define who can access a view, DB2 binds an authorization ID and a view together to form a *package*. The common representation of authorization ID is a role name. In this package, it defines that the view can be accessed only if a user has the authorization ID that is associated with the view. For example, in a health system, there is a patient role and a doctor role. The policies are defined as the following: a patient can view his own record and a doctor can view his patients' records. To implement these polices in DB2, we need create a view for patients (patient_view)

and a view for doctors (doctor_view). The next step is to associate patient_view with the patient role and doctor_view with the doctor role. These two views are granted to public. Before a user can issue a query on patient_view, the DB2 has to verify that he assumes a patient role first.

Instead of using views for each policy, Oracle uses query-rewriting to enforce policies. When a user issues a query, based on his privileges, the Oracle database server will attach a predicate to the query. This predicate reflects what the access privileges the user has. The details of FGAC implementation in Oracle are described in the next section. Compare with Oracle's approach, DB2's approach has the following disadvantages: first of all, views are not always practical when we need a lot of them to enforce security policy [KD02]. For example, if have want to use views to limit customers' access and there are 100,000 customers, it is not practical to cerate 100,000 views. Second, views may complicate administration of security policies [KD02]. It is hard for a policy maker to tell the difference between a view definition based on database relationship from that for security purpose. Based on above analysis, we decide to use Oracle for this toolkit.

2.2 Access Control Enforcement

In this section, we describe the establishment of the VPD in Oracle.

2.2.1 Define and use of VPD

Traditional database access control is enforced by creating views for individual users based on their privileges. Although it provides a secure environment for a database, such an approach is very inefficient and costly with a large number of users, which is

typical in today's Web-based applications. For example, suppose there are a million patients in a health-care information system. Assume that a user can only access his/her own records. Then a million different views need to be defined. Even if those views do not need to be materialized, creating and managing such a large number of views will be very expensive. In Oracle 8i, a new access control mechanism, called Virtue Private Database (VPD), was introduced. Instead of creating views for each individual, VPD restricts users' access to selected rows of tables through query rewriting. When a user issues a query, based on his/her privilege, a predicate will be generated at run-time and be attached to the query [Kyt]. Access control is enforced when the rewritten query is executed by the database engine, since the attached predicate limits what a user can access.

Figure 2.1 shows an example. The policy is that a user only can see his/her own record unless the user is a 'DBA'. When a common user Alice logs into a database, the policy function will generate a predicate for Alice. Since Alice is not a DBA, the generated predicate will enforce the policy that Alice can only see her own record. When Alice issues a SELECT query against the table, this predicate will be attached to the query. As a result, Alice's access is restricted according to the access control policy.

Role is an important concept in VPD. Since it is infeasible to grant privileges to each individual user, VPD often assigns privileges based on the roles. For each role, VPD defines explicit privileges for it. A user can assume more than one role in VPD. When a user logs into a database, based on the role he/she assumed, he/she will have different privileges. The advantage of using roles in VPD is to simplify access control policy specification.

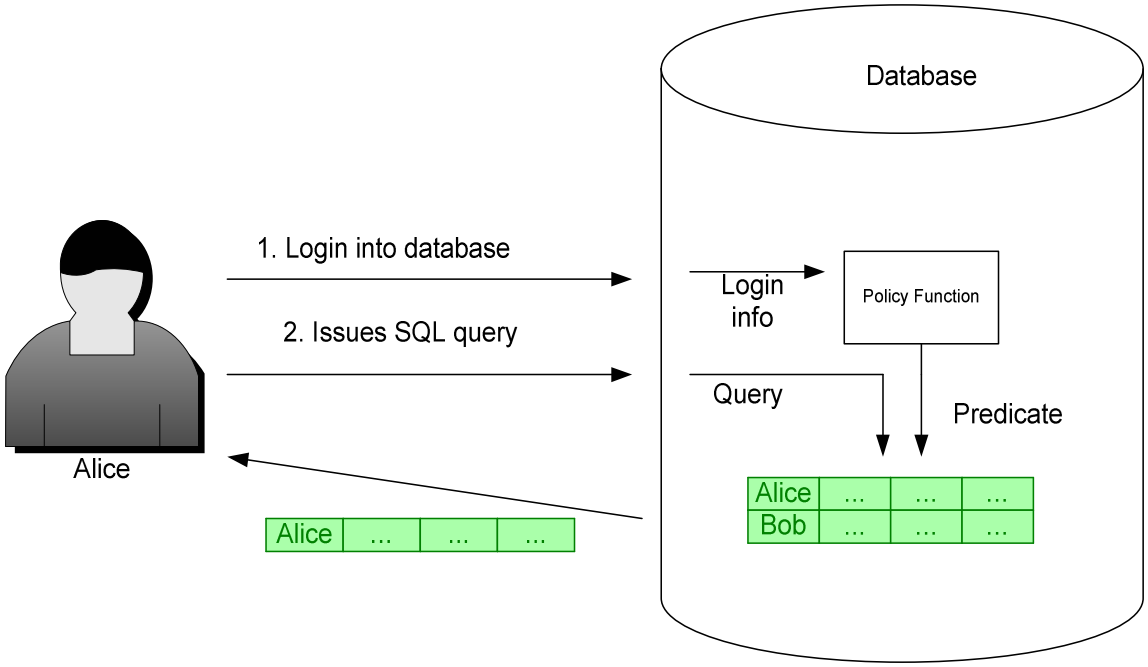


Figure 2.1 Access control enforcement in VPD

Since its introduction, VPD is widely used in Web-based applications, due to its following advantages:

- Multiple security: By using VPD, we can enforce more than one policy to a table at the same time without using views. It avoids using different views to enforce different policies. Thus, it is easy to enforce database security by using VPD.
- Suitable for single user based applications: Single user based applications, such as Web applications, allow a single user to connect to a database. It requires that each individual user can see different results. By using VPD, row level security can easily identify different users and retrieve information for them.
- No Back door: Since each policy is associated with a table, not an application, no users can bypass those policies. By letting the policy directly associate with the

table, regardless what applications the user is using, the database server will always check the user's privileges, before the user's query is executed.

2.2.2 Setting up FGAC

There are five steps to setup fine-grained access control by using VPD.

1. *Policy specification*. In this step, the policy maker needs to state that for each role, what privileges it has. For example, the policy maker may state that an employee can issue SELECT statements on his own records. Managers can issue SELECT statements on the records of employees who work under him as well as his own record. The manager may also have the rights to update any employee's record, but not his own. For anyone else, he cannot perform any actions on that table.
2. *Context creation*. A context value is a text information that can be retrieved by other Oracle functions and PL/SQL queries. It is often used to store some information about the current login user. The second step is to create a context space, which is used to store context values. Each policy can only have one context. This context will be used to store some user information such as login user information and some other values that are defined by a policy maker. Those context values can be used by other Oracle functions. Creating context is usually associate an Oracle procedure. This Oracle procedure is the only way to decide what information will be stored and how to store/set up these context attributes. For example, we may define a context by "*CREATE OR REPLACE CONTEXT example_context USING example_procedure*". *Example_context* is the name of context we create. *Example_procedure* is an Oracle procedure that sets up context values. In other words, in order to set context variables in *example_context*, we

have to use the *example_procedure* to define those variables. By using only one procedure to store context values, it can protect data consistency. For example, if another procedure tries to set the context value and the database server finds out this procedure is not *example_procedure*, the database server will rejects it. .

3. *Create the procedure*. After we defined the procedure name that is used to set the context values, we are now going to show how to create such procedure. The context values are defined by calling DBMS_SESSION.SET_CONTEXT within the procedure. This statement is used to define and populate a context's attributes. It includes three parameters:
    a. Namespace: the name of context that is used by an application.
    b. Attributes: name of the attribute to be set in a context.
    c. Value: the value of a context attributes. Those values can be retrieved by calling function *SYS_CONTEXT*.

    Let us look at the "*example_context"* example again. In this context, we create an attribute in this context named "Role". If we are going to assign a value to the context attribute, we can use the statement:

```
Dbms_session.set_context ('example_context', 'Role', 'Employee')
```

In this example, the namespace is 'example_context', attribute is 'Role', and value is 'Employee'. In other words, this statement defines that in the context 'example_context' the context 'Role' has value 'Employee'(or Role = Employee').

4. *Create a security policy procedure*. This is the central step for establishing FGAC, because the dynamic predicate will be generated by this security policy procedure, which will be called by the database engine at run time. It restricts what a user can see.
5. *Associate the policy with a table***.** The last step is to associate the predicates with each of the DML operations and the table itself. In order to define a policy, we need to define the following attributes:
    a. Object_schema: It defines the owner who created the table. If it is NULL, the database will assume the current login user is the owner of the table.
    b. Object_name: The table name where the policy will take effect.
    c. Policy_name: The name of the policy to be added. The name is defined by a security officer. It must be unique, since there may be more than one policy applied to a table.
    d. Policy_function: It refers to the name of security policy procedure that generates the predicate for users.
    e. Statement_types: Define the SQL statements that will be monitored by this policy, such as SELECT, UPDATE, or DELETE. For each DML operations, it has to have a different predicate function.

As mentioned before, a key step in enforcing access control polices is to create security policy procedures by using Oracle procedure language (PL). Oracle PL is a Turing-Complete language. Thus, it is very powerful and can be used to express very sophisticated policies. On the other hand, access control policies are manually implemented by programmers. Security flaws may be introduced, due to either a

programmer' misunderstanding of a policy specification or his/her negligence. Since the Oracle PL is a Turing-Complete language, it is very hard for security officers to verify implemented policies and detect potential vulnerabilities.

The above observation is the major motivation for this project. If policy specification can be automatically implemented and enforced in the database, policy verification and security flaw detections only need to be performed in policy specification phase, which is much more manageable than checking policies written in Oracle PL. The key question is to choose the appropriate formal policy language to specify fine-grained access control in Oracle 9i. We have identified the following desirable properties for policy languages. These properties are based on the characteristics of VPD. At the same time, this policy language should also be easy to understand for a security officer.

- A policy language should have an easy-to–understand and concise syntax. Basic access control elements, such as subjects, objects, access rights, etc., should be clearly identified by using policy key words.
- The semantics of a policy language should support the close model, i.e., a subject does not have access to a certain object unless it is explicitly allowed by access control policies. This is desirable because VPD assumes a closed model.
- Negative policies should take precedence over positive policies. In other words, if a subject is allowed by one policy to access an object, but is denied by another policy, then the subject cannot access that object. This is also due to the semantics of VPD.
- The access control policy language needs to support role-based access control. In Oracle VPD, privileges are often granted based on users' roles. And there is often

more than one role defined in a policy. The policy language also needs to support role hierarchies. Otherwise, the relationship between roles cannot be properly identified in a formal policy specification.

Keeping in mind the above properties, we analyze three representative policy languages: Rei [KFJ03], EPAL[AHC03], and Ponder[DDL01], from which we select the most suitable one for specifying fine-grained access control in databases. We also considered other policy languages such as Web Services Policy Language (WSPL) [AA04] developed by SUN Microsystems and REWERSE (for REasoning on the WEb with Rules and SEmantics) [REW04], but both of them are still under development.

2.3 Formal Policy Languages

This subsection provides an overview of the three most relevant policy specification languages: Rei [KFJ03], EPAL[AHC03], and Ponder[DDL01].

2.3.1 Rei

Rei, a Japanese word that means “universal”, is a policy language developed by Kagal, Finin, and Joshi [KFJ03, Kag02, KaFJ03]. It is an action based policy language. It includes constructs for rights, prohibitions, obligations, and dispensations [KFJ03]. Since Rei is not designed for any specific applications, it permits domain specific information to be added without modification of the language itself. Rei is implemented in Prolog. The creators of Rei believe that a policy could be expressed as what an object can/cannot and should/should not do in terms of actions, services etc. Rei includes two parts: domain independent ontologies and domain dependent ontologies. The former includes concepts such as permissions, obligations, actions, and operators etc. The latter

is a set of ontologies, shared by the entities in a specific system, which defines domain classes and it's properties such as file directories and file names [KFJ03]. Rei includes three types of constructs for domain independent ontologies: policy objects, meta policies, and speech acts. Speech acts includes: request, cancellation, delegation and revocation. They are used for decentralized control. For example, a user may have the rights to send a *request* (request for an action) to other user, but he may not have the rights to cancel the request he has sent.

The constructs are the core of the policy language. It describes the concepts of rights, prohibitions, obligations, and dispensations. In this part, we are only to describe rights and probations, because are not essential to authorization.

- Rights define the permission that a subject has. It allows a subject to perform one or more actions. Rei defines rights as follows:
    - has (*subject*, right (*actionname, conditions*)). This defines that if the *subject* wants to perform the action, it has to satisfy *conditions*.

      Actionname is defined separately which contains the object of the action.

      Example:

      has (employee, right(print, rank =3))

      It defines that if an employee's rank is 3, then he/she can perform print action.
- Prohibitions define negative authorizations, meaning that a subject cannot perform certain actions.
    - has (*subject*, prohibition (*actionname, conditions*)). The *subject* is prohibited from performing *actionname* if *subject* satisfies *conditions*.

In Rei, actions can be represented as a tuple with parameters, as shown in the following format:

action (*actionname*, *targetobject*, pre-*conditions*, *effects*)

In this tuple, actionname defines the name of action. It is used in the rights and prohibitions policies to define the action that a user can perform. Targetobject is a list of objects on which the action is performed. Pre-conditions are the conditions that need to be true before the action can be performed and Effects are the results of the action. The pre-conditions are defined only for the action not for any subjects.

Rei proposes two ways to resolve conflicts in policies. The first is to set priorities, by using statement overrides (A, B), meaning policy A has priority over B. The second way is to use precedence relations. The policy maker may decide certain precedences for a set of actions, e.g., negative policies are stronger than positive ones.

The advantage of Rei is that it provides a variety of action primitives for access control specification. Access control policies can be defined as what actions a user can take; and many perimeters can be associated with this action. For example, when we define what action a user can perform, we can also define operators for the action, such as: repetition (allows the user to repeat the action) and once (defines that a user can only perform this action once). Although Rei is relatively simple syntax, it does have some disadvantages and therefore is not quite suitable for specifying access control policy for databases. First of all, subjects are treated as un-interpreted symbols. No role hierarchy can be defined in Rei. As a result, it will be difficult to define role-based fine-grained access control policies.

2.3.2 EPAL

Enterprise Privacy Authorization Language (EPAL) is developed by IBM [AHC03, ASP02, KSW02]. EPAL is a "formal language to specify fine- grained enterprise privacy policies. It concentrates on the core privacy authorization while abstracting from all deployment detail such as data model or user-authentication"[AHC03]. It is an XML-based policy language and allows developers to enforce privacy policies directly into enterprise applications.

An EPAL policy is a set of privacy rules ordered with descending precedence. In EPAL, rules are used to determine if the request is granted or not. If a rule applies, subsequent rules are ignored. A rule may contain conditions and obligations. There are four elements in a rule: a *user category, an action*, a *data category,* and *a purpose*.

- *A user category* defines the subject of a rule, such as an employee, a manager.
- *Data category*: the data category provides a high level classification of data, such as employee information, medical record, etc. By classifying data into different category, based on the privacy requirements, data can be treated differently. EPAL itself does not define any actual data. Instead, it uses data category to categorize data.
- *Purpose* is an important part in EPAL, because information should only be disclosed for particular purposes. For each rule in EPAL, it has to state the purpose for the use of certain information**.** Similarly, each information access request also needs to specify the purpose for the access. Otherwise, it will be rejected automatically.
- *Action* defines a privacy relevant action that can be referenced in rule definitions.

Typically, privacy authorization rules also require context conditions. Each container defines a data structure that contains context data that can be evaluated by conditions associated with the context data. The container defines a list of attributes that can be evaluated by conditions. Such attributes may include e g., one's name, employee number, and department. And based on instances of the attributes, the conditions will be evaluated to be 'true' or 'false'. Each *condition* statement represents one condition. If there is more than one condition, all the conditions have to be true before the rule can be applied. Otherwise the rule will be ignored.

Since EPAL does not define any specific data types, it is necessary to define a vocabulary set defined as "*infoType*" , which contains all the vocabularies that will be referenced in rules . There are three attributes in the definition of a vocabularies set. The "id" attributes defines the name of the vocabulary. The "issuer" defines who issues these vocabularies. And the "version-info" defines the version of this rule and other management information such as date.

In order to establish an EPAL policy, a policy maker has to create a set of user categories *U*, a set of data categories *T*, a set of purposes *P*, and a set of actions *A*. All of them have to be defined in vocabulary and will be referenced by other parts of the policy. A request to the system is in the form "Is the given user-category allowed to perform the given action on the given data category for the given purpose?" The system determines the ruling by processing each rule with descending precedence. By analyzing the tuple (*U*, *T*, *P*, *A*), the system's output will be either "allow", "deny" or "not-applicable".

If an access control policy is “An employee can only see his own record and a manager can view the records of all the employees who work in his department. In EPAL, the policy will be defined as shown in Figure 2.2.

```
<rule id =”Oracle_policy” ruling =”allow”>
<user-category refid = “employee table”/>
<data-category refind = “employee_record_table”/>
<prupose refind = “view table” />
<operation refined =”SELECT” />
<condition refid =  “condition1”/>
<condition refid = “condition2”/>
</rule>

<rule id =”Oracle_policy_manager” ruling =”allow”>
<user-category refid = “employee table”/>
<data-category refind = “employee_record_table”/>
<prupose refind = “view table” />
<operation refined =”SELECT” />
<condition refid =  “condition1”/>
<condition refid = “condition3”/>
</rule>

<container
id= “employeeTAB”>
<attribute
id =” employee _table.userID”
simpleType = http://www.w3.org/2001/XMLSchema#string>
</attribute>
<attribute
id = “employee _table.name”
simpleType = http://www.w3.org/2001/XMLSchema#string>
</attribute>
</container>

<container
id= “employeeREC”>
<attribute
id =” employee_record_table.userID”
simpleType = http://www.w3.org/2001/XMLSchema#string>
</attribute>
<attribute
id = “employee_record_table.name”
simpleType = http://www.w3.org/2001/XMLSchema#string>
</attribute>
<attribute
id = “employee_record_table.manager”
simpleType = http://www.w3.org/2001/XMLSchema#string>
</attribute>
</container>
```

```
<condition id = “condition1”>
<predicate refid =http://www.research.ibm.com/privacy/epal#string-equal”>
<function
refind =”http://www.research.ibm.com/privacy/epal#string-bag-to-value”>
attributes-reference
container-refid = “employeeTAB”
attribute-refid = ““employee _table.name”/>
</function>
<attribute-value simType =http://www.w3.org/2001/XMLSchema#string>
<value> context.name</value>
</attribute-bag>
</predicae>
</condition>

<condition id = “condition2”>
<predicate refid =http://www.research.ibm.com/privacy/epal#string-equal”>
<function
refind =”http://www.research.ibm.com/privacy/epal#string-bag-to-value”>
<attributes-reference
container-refid = “employeeTAB”
attribute-refid = ““employee _table.ID”/>
<attributes-reference
container-refid = “employeeREC”
attribute-refid = ““employee_record_table.userID”/>
</attribute-bag>
</predicae>
</condition>

<condition id = “condition3”>
<predicate refid =http://www.research.ibm.com/privacy/epal#string-equal”>
<function
refind =”http://www.research.ibm.com/privacy/epal#string-bag-to-value”>
<attributes-reference
container-refid = “employeeTAB”
attribute-refid = ““employee _table.name”/>
<attributes-reference
container-refid = “employeeREC”
attribute-refid = ““employee_record_table.manager”/>
</attribute-bag>
</predicae>
</condition>
```

Figure 2.2 EPAL Policy Example

As we can see, EPAL requires a formal definition for each attribute and condition before they can be applied into policies. This requirement does offer the advantage of keeping data references consistent, because each attribute has to be formally defined in the vocabulary set. The drawback of this language is that it does not fully support role hierarchy. EPAL only allows each role to have a single parent. As a result, we can not use EPAL to express a relationship that a node has more than one parent. For example, suppose a manager is also considered as an employee and a team leader. Then, it is difficult to define the parent node for the manager in EPAL, since the manager role is an extension of both the employee role and the team leader role.

2.3.3 Ponder

Ponder is a policy language developed by researchers at Imperial College[Dam02, DDL01, DSL01]. It is a declarative, object-oriented language for specifying security and management policies for distributed object systems[DDL 01]. Ponder is designed for non-discretionary access control, where administrators have the authority to specify security policies that are enforced by the access control system. Ponder supports authorization, delegation, information filtering, refrain policies, and obligations.

In Ponder, the term *subject* refers to users. The term *target* refers to objects (resources). The term *action* defines what action/actions can be performed on the target and the term *when* states the constraints/conditions where a policy can be applied..

An authorization policy defines what actions a subject could perform against a set of targets. Ponder allows two kinds of authorization policies. A positive authorization police defines the actions that subjects are permitted to perform on target objects.

A negative authorization policy defines what actions that subjects are not allowed to perform on target objects. The syntax of an authorization policy is shown in Figure 2.3.

```
Inst (auth+ | auth-) policyname{
Subject         expression;
Target          expression;
Action          expression;
When            constraints
}
```

Figure 2.3 Authorization Policy Syntax

Example 1. Positive and negative authorization policies

Inst auth+ employee_view
{
subject manager
target employee_record_table
action select
}
It defines that a manager can issue select statement on employee record table.

Inst auth- employee_view
{
subject manager
target employee_record_table
action delete, update
}
It defines a manager is forbidden to issue delete and update statement on the employee record table.

Ponder explicitly supports the definition of roles and role hierarchies. Policies can be grouped together based on roles to reflect the privileges of a group of users instead of individuals. The syntax of roles is showed in Figure 2.4. For example, a manager will always have the same privileges regardless who is assigned to this role.

```
Type  role roleName
{
{basic-policy-definition}
{group-definition}
{meta-policy-definition}
}[@ subject-domain]
```

Figure 2.4 Role Policy Syntax

Example 3. Role Policy

```
type role employee
{
}
 inst auth+ emp_select
 {
 target  /IBM/RECORD_TABLE/record
 action SELECT
 when subject.name=/IBM/REC_TB/record.name
 }
```

The above policy specifies that an employee only can view his own record in the record_table. A role may include more than one basic policy, group or meta-policy. A group definition groups related policies together for the purpose of policy organization. Meta policies define policies about the policies within a composite policy and are used to define application specific constraints. For example, the meta policy could be used to define that the same person cannot submit and approve a budget. Subject domain defines the set of subjects. The subject domain is specified following the @ sign. If a subject domain is undefined, then a subject domain will be created with the same name as role.

Role hierarchy can be defined through role extension. When a role extends from another role (base role), it inherits all the privileges from the base role. New policies can

also be added to the extended role. If two policies have the same name, then the new one will overwrite the old one. The keyword "*extends*" is used when a role extends another. Formal parameters define the parameters for the newly created role and actual parameters define the parameters that may have included in the base role. The inheritance syntax as shown in Figure 2.5.

```
Type Role roleTypeNmae { formal Parameters }
Extends parentRoleType { atucalparameters}
{
role body
}
```

Figure 2.5 Role Extension Syntax

Example 4. Role inheritance

```
type role manager extends employee
{
        inst auth+ mgr_select
        {
target  /IBM/REC_TB/record
action SELECT
when subject.name=/IBM/REC_TB/record.mgr
}
}
```

The above policy specifies that a manager role is extended from an employee role. It not only inherits all the privileges of an employee role, but also extends the privileges by allowing a manager to view the records of all the employees who work under him.

After analyzing the above three policy languages, we believe that Ponder meets the requirements for specifying fine-grained access control policies for databases. Ponder can be easily used to support role-based access control, which is the crucial for Oracle VPD. While Rei and EPAL have their own advantages, they fall short in defining role

hierarchies. Further, Ponder has a clear and concise syntax. Thus, a Ponder policy can be mapped to Oracle policy enforcement program in a relatively straightforward manner.

Since Ponder is originally designed for distributed network service management, it has features that are not completely suitable for database access control. For example, the Ponder role policy requires the subject-domain to be formally defined, but FGAC does not have such definition. Based on the above analysis, in this project we adopt Ponder as a preliminary high-level policy language database fine-grained access control policy specification. Although Ponder supports four types of policies: authorization, obligation, delegation, and refrain policy. In our toolkit, as explained in this section, we are only use authorization policy to express database access control policies. For other type of the policies, they will not be used to define the access control policies. A further study is needed to decide whether these type of polices can be properly translated into database access control policies.

3 Translating Access Control Policies to Oracle Enforcement Script

In this section, we first discuss how Ponder is used to specify database fine-grained access control policies. Then, we introduce the new algorithm that translates these Ponder policies into Oracle FGAC enforcement code.

3.1 Creating Access Control Policy in Ponder

This subsection provides an overview of how we use the Ponder policy languages to create FGAC policies.

3.1.1 Choose the Policy Type

As mentioned in previous chapter, Ponder allows various policy types. Clearly, authorization policies are the most relevant for access control policy specification. Since roles are important component of FGAC, the Ponder role hierarchy should be used to create access control policy specifications.

To illustrate how to create FGAC specifications, we assume we are going to enforce a policy on the following table (HR table):

Name	Manager	Rank
…	…	…

Table 3.1 HR Table

Table 3.1 has three attributes: *name* stores an employee's name, *manager* stores the employee's manager name, and *rank* stores the employee's rank. This table will be also used in the following examples, unless otherwise specified. There are three roles in the policy: employee, manager, and CEO. The employee can only view his own record. A

manager can view an employee's record whose manager is him. A CEO can view the whole table.

By setting the employee role as the base role for manager role, when a user is trying to perform the manager role, he not only can have the manager privileges but also can have employee privileges. A Ponder policy and role hierarchy could be expressed as the following figure (Figure 3.1). It defines that role manager and role CEO are extended from the employee role. Each of them has individual privileges in addition to employee privileges. By using Ponder role definition and role hierarchy, we believe that an access control relationship can be properly identified.

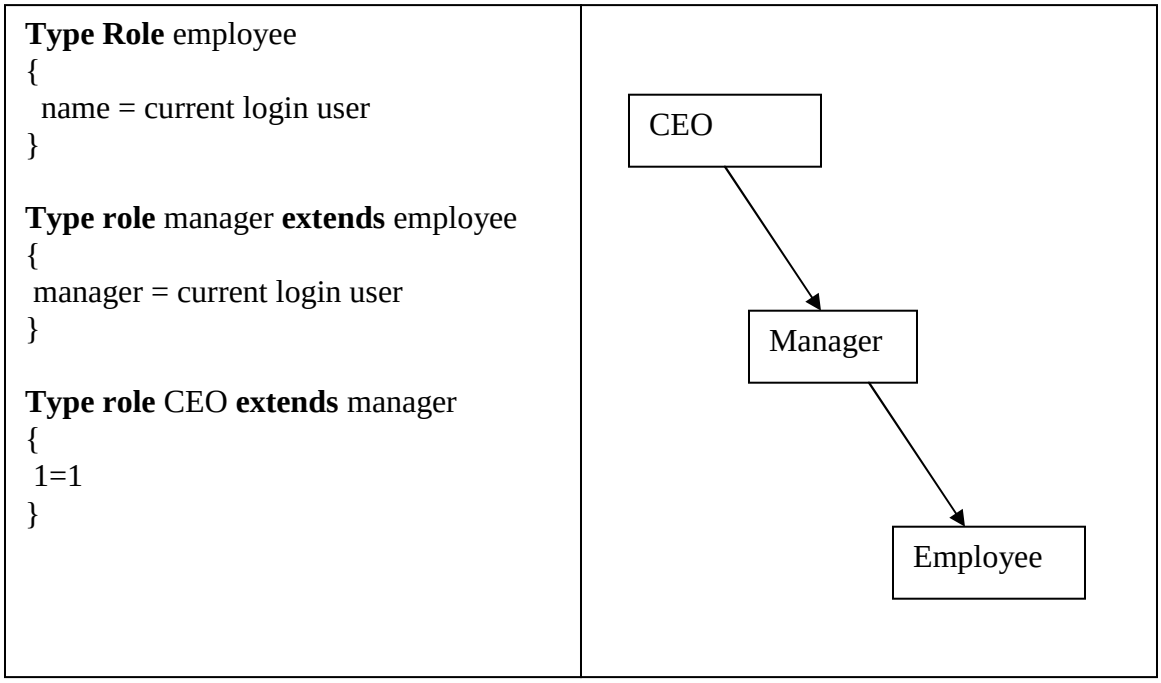


Figure 3.1 Role Hierarchies

For each access control policy, its subject, target, action and constraints need to be defined properly. In our case, since we are trying to define the privileges for different roles, the subject of an access control policy will be roles. In general, a target could refer to a table, or the whole database. Since FGAC only supports a single tale, the target in

this toolkit will also refer to a single table. For access control action keywords, there are only a few action commands in Oracle database that a user can issue against a table, such as “SELECT”, “INSERT”, “UPDATE” and “DELETE”. All these commands will act as the action key words in Ponder. By using default Oracle action keywords as policy action keyword, the translation between the Oracle and policy action key words can be simplified, therefore it reduces the complexity of policy specification and increase the performance of the toolkit.

3.1.2 Access Control Policy Interpretation

Given a set of authorization policies, it is possible that several policies concern about a role’s privileges on the same object under different conditions. When enforcing those policies in a database, we need to consider the overall effects of the set of authorization policies.

Formally, positive authorization policies define a set of positive authorization tuples ($s, o, +a, c$), where s is a role, o is a database object, a is an action, and c is a predicate that specifies the constraints when s can take action a on o. Similarly, negative authorization policies define a set of negative authorization tuples ($s, o, -a, c$), which means that s cannot take action a on o if c is true. Let s be the current role of a user. Given two authorization tuples ($s_1, o_1, *a_1, c_1$) and ($s_2, o_2, *a_1$ and c_2), where * can be either + or –, if s1 and s2 are either the same as s or are extended from s, o1=o2, a1=a2, then we say the two tuples are *relevant* to s. Otherwise, they are *irrelevant* to s.

In order to determine a user’s privileges, we need consider the combined effect of authorization tuples. Suppose a user’s current role is s. Let T= {($s_1, o, +a, c_1$), …, ($s_n, o, +a, c_n$)} be a set of positive authorization tuples that are relevant to s. Then the user is

allowed to take action *a* on object *o* as long as c_1 OR … OR c_n is true. Intuitively, since the authorization tuples are positive, as long as one of the constraints is satisfied, the user obtains the corresponding privilege. Similarly, let T= {(s_1, *o*, -*a*, c_1), …, (s_n, *o*, -*a*, c_n)} be a set of negative authorization tuples that are relevant to *s*. Then the user is not allowed to take action *a* on object *o* as long as c_1 OR … OR c_n is true.

When there are both positive and negative authorization tuples, we take the closed authorization model, i.e., negative authorization overrides positive authorization. Formally, given two relevant authorization tuples (s_1, *o*, +*a*, c_1) and (s_2, *o*, -*a*, c_2), the user is allowed to take action *a* on object *o* only if *c1* AND ¬ *c2* is true.

In general, let T={(s_1, *o*, **a*, c_1), …, (s_n, *o*, **a*, c_n)} be a set of authorization tuples relevant to s. The combined authorization constraints for s can be determined as follows. Let T^+ and T^- be the sets of all the positive and negative authorization tuples in T respectively. We can get the combined constraints C^+ and C^- of T^+ and T^- for *s* respectively, as described above. Then the user is allowed to take action a on object o only if C^+ AND ¬ C^- is satisfied.

3.2 The Architecture of the Toolkit

This toolkit has three components: Ponder policy specification interface, policy translator, and policy importer. A policy maker can use the Ponder policy specification interface to create Ponder authorization policies. The created Ponder policies are the input for the policy translator, which translates Ponder policies into Oracle policy enforcement program. The translation includes three steps: identifying role hierarchy, access control elements translation and set context values. The policy importer is

responsible for importing the created Oracle policy enforcement program into an Oracle database by using JDBC. Figure 3.3 shows the architecture of the toolkit. In the following, we describe the Policy translator and the importer in detail.

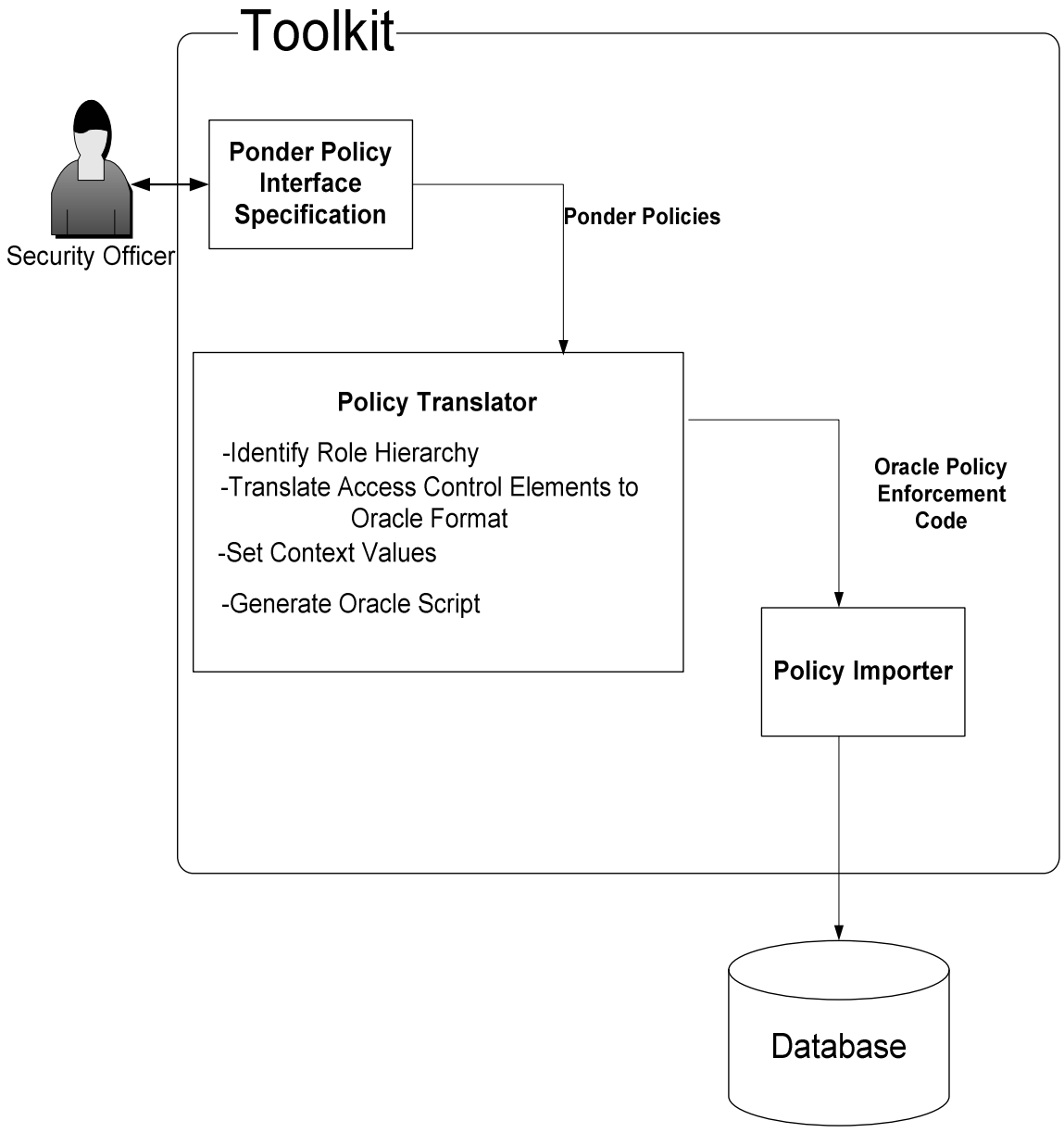


Figure 3.2 Toolkit Architecture

3.3 Policy Translator and Policy Importer

Policy translator is the core of this toolkit. It is responsible for translating Ponder policies into Oracle policy enforcement program. This program is written in Java and contains three functions: identifying role hierarchy, access control elements translation and set context values. The following subsections discuss how the policy translator translates Ponder policies into Oracle scripts.

3.3.1 Role Hierarchy Identification Function

The purpose of role hierarchy identification function is to identify the role relationship in a policy. This function analyzes the Ponder policy and records all the roles existing in the policy. A text file will be generated and contain all the roles pairs generated by the translator If a role is an extended from another one, it will be recorded as (senior role, junior role) in the text file, where the senior role is the extended role and the junior role is the base role. If a role is not extended from any other roles, this role will be saved as (Senior Role, NULL) in the text file. This text file will be imported into an Oracle server and a new table will be generated by the translator based on this text file. In this table, it has two attributes: senior role and junior role. All the role pairs are mapped into this table. The reason to create such a table is to allow the Oracle function to perform a role hierarchy search among roles. The reason to create a text file for the policy roles is to allow a policy maker to check whether the policy role relationship has been correctly built. By doing so, a policy maker can check the role hierarchy first before a FGAC being fully implemented.

The created role table is used by a role comparison function defined by this toolkit The role comparison function takes two roles r1 and r2 as parameters and returns true if

r1=r2 or r1 is senior to r2. This comparison function is based on the role table created by the translator. When a role comparison function is called, the function will search the role table by using SQL "*start with…connect by…*" query. An example of the syntax is shown in Figure 3.5.

```
select  count(*)
from role table
where junior_role in (select junior_role from role table
                      start with senior role = current user role
                      connect by prior junior role = senior role)
and junior role = Give role name
```

Figure 3.3 Role Hierarchy Search Query

If a user's current role is senior to a given role, it means that the user can have the privileges of the given role. For example, if a role table contains two pair of roles: (Manager, Employee) and (CEO, Manager), and a user sets his role to CEO. The role comparison function compares role CEO with role Employee and role manager. It returns true in both cases. As the result, the CEO will have the privileges of both manager and employee.

3.3.2 Mapping of Access Control Elements

The task of the second function is to scan through Ponder policies and identify all the access control elements, including subject, target, action, and constraints. The translator identifies each one of them and stored them in a Java array, which is used to later create Oracle policy enforcement program.

In this project, we assume the roles defined in Ponder policies are from the same ontology as those in a database. Further, we assume actions are the same as data manipulation operations. Therefore, for subjects and actions, the translation is

straightforward. If roles and actions are not defined using the same ontology, then necessary mapping is needed. Additionally if a Ponder action is “retrieve”, then it needs to be mapped to SELECTION operation to a database. The translation of target and constraints is more challenging. In Ponder, objects are often organized into hierarchies. Thus, a target not only includes the table name but also includes the directory path. For example, in a Ponder policy, the target statement may be expressed as:

Target: /IBM/HR/record

It refers to the table *HR* under *IBM* directory. The keyword *record* defines the particular tuple/tuples that satisfied the constraints of the policy. On the other hand, in Oracle, a table is simply referred by a unique name. We cannot directly map the target name written in Ponder to a database object, because the Ponder table name is not uniquely identified. In Ponder, two different tables can have the same name, as far as they have different directory paths. In Oracle, a table name must be unique.

To solve this problem, we assumed there is a *mapping file* which explicitly maps Ponder policy targets to database objects. A mapping file is a text file is created by a policy maker and stores target names that are written in both Ponder format and Oracle format. When the translator maps a target element, it looks up the mapping file and finds the corresponding Oracle database objects. If no mapping exits in the file for a given target element, an error message is returned.

For example, assume the mapping file has the following name pair:

(/IBM/HR/record, HR)

Once the translator encounters this pair, it knows "/IBM/HR/record" in a Ponder policy refers to table HR in an Oracle database. Thus, the HR table will be used to generate Oracle policy enforcement program.

The mapping file contains not only the mapping information for tables, but also

```
Subject.name = /IBM/HR/record.name
```

that of policy constraints. For example, if we have the following constraints:

This constraint states that the *name* attribute of a tuple in the HR table must match the user's name. When the translator translates this constraint, it breaks the constraint into two parts; *subject.name* and */IBM/HR/record.name*. For the */IBM/HR/record.name* part, as described above, the translator looks up the mapping file and maps it to database HR table.

Thus, the constraint is translated as:

```
Subject.name = HR.name
```

For the *subject.name* part, the translator needs to retrieve it from the subjects' (i.e., the current user) context value.

3.3.3 Setting context values

A context stores information about the current user connected to an Oracle database. Oracle 9i provides functions to retrieve a user's properties from its current context In a Ponder policy, part of constraint is usually defined as "subject. X", where 'X' represents an attribute of a user. For example, subject.name refers to the login user's

Subject.name = HR.name

name and subject.rank refers to the rank of the user. Using the example from above again, the constraint

refers to that the current login database user can only view his own information. When

```
sys_context('userenv', 'session_user')= HR.name
```

the translator translates "*subject.name*", it creates a context value to store the current login user's name. In Oracle, a context value is already created for a login user's name, which can be retrieved by calling "*sys_context('userenv', 'session_user')*". Thus, the constraint is translated into Oracle format as

It is necessary to set a context value for each individual "*subject.X*" elements to avoid unnecessary nested query. For example, if we set current login user's name as the only context value. For the following constraint:

```
Subject.rank < /IBM/HR/record.rank
```

It states that the current login user can only view employee record whose rank is lower than him. By using the name as the only context value, the above constraint has to be translated as:

```
Rank < ANY (select rank from HR
        where name= sys_context('userenv',
        'session_user') )
```

In above query, if we use name as the only context value, we need a nested query in order to fully express the constraint. However, this nested query can be avoided. Instead of using name as the only context value, we can create another context value for “rank”. When the translator reads “subject.rank”, it will create a new context value for ‘rank’. As the result, the constraint can be written as:

```
Rank <  sys_context('db_context', 'rank')
```

in above query, ‘*db_context’* is the name we defined for the context, and ‘*rank’* is a context value. By storing “*subject.X*” as a context value and eliminating unnecessary nested queries, the efficiency of query can be improved.

3.3.4 Oracle Script Generation

Once three functions have been executed, the translator is ready to generate Oracle script. There are three Oracle functions need to be generated including: role setting function, predicate setting function, and policy generating function. In the following graphs, we are going to give a brief description about how the scripts are generated.

The first Oracle function that needs to be generated for VPD is the setting role function. The main goal for this procedure is to set context values for the user. Since each user may have more than one role in a database, when a user logs in a database; a role has to be set for him before he can issue any SQL query. In this procedure, it allows a user to set his role and this role will be stored as an Oracle context value. After a user’s role has been set, the role name will be available to other Oracle functions. Only after the role has been set, the database server can attach the corresponding predicate to any user issued

query. For example, if a user requests to assume the employee role, the role setting procedure will set the requested role for him. The assumption is that the user has passed the role checking mechanism. This mechanism is to ensure a user will only assume roles assigned to him. Since the role-checking mechanism is not the goal of this toolkit, the toolkit will assume that role checking is done by other part of the system. Other context values also need to be set in this function, such as name, rank etc.

For each database application, a new context should be created and the role setting function should bind to it. This function is the only way to set a context value. By doing so, it ensures data integrity. Once a context value has been set, we know this value has been validated and properly assigned to a user.

The second Oracle function is for setting predicates. A predicate is dynamically attached to user's query during run time. This function defines the predicate based on a user's privileges. In this toolkit, instead of returning a different predicate for each different role, only one predicate is returned for all the roles. In this single predicate, it contains all the role constraints. This predicate is generated by retrieving all the role constraints that already translated into Oracle format by the translator. The generated predicate has the following format:

```
Predicate =
Role1 constraints and
role comparison function (current user role, role1)

OR

Role2 constraints and
Role comparison function (current user role, role2)
.
.
.
RoleN constraints and
Role comparison function (current user role, roleN)
```

Each role constraint is associated with the role comparison function generated by the translator. The role comparison function compares the current user role with a given role. Only if the role comparison result returns true, the associated role constraint will be executed.

The reason, to return one predicate contains all the constraints instead of returning a different predicate for each role, is the relationship between each role is disjunction. A user could be an employee. He also could be an employee and manager at the same time. An OR relationship ensures the current login user be able to view all the information allowed by his privileges. For example, if the predicate is setting like following:

```
Predicate =
Employee constraints and
role comparison function (current user role, employee)

OR

Manager constraints and
Role comparison function (current user role, manager)
```

Assume the user's role is employee. When this predicate is attached to the query the user issue, the database server checks the Boolean variable returned by role comparison functions. Since the result for role comparison function (employee, manager) returns false, as the result, the predicate is equal to

```
Predicate = Employee constraints
```

This is because manager constraints are not executed due to the employee role is junior to the manager role.

The last Oracle function is to associate predicates with each of the DML operations (SELECT, UPDATE, DELETE and UPDATE) and the targeting table itself. In this function, it requires to define the policy name, function name(the function generates the predicate), table name, and which DML statement it associates with. All these information can be retrieved from policy model and previous Oracle functions. This Oracle adding policy function will ensure for every each DML operation there is a predicate setting function associate with it.

After all the necessary Oracle scripts have been generated, they will be imported into Oracle database sever by using the script importer provided by the toolkit.

3.3.5 Oracle Script Importer

The importer connects the toolkit and an Oracle database by using JDBC. The function of importer is to import all the generated scripts into an Oracle database sever.

3.4 A Simple Scenario

The example of employee, manager and CEO is used at here again. The policy is defined in section 3.1. When the translator reads the Ponder policy, it creates the role hierarchy file first. In this case, the role hierarchy is shown as the example in Table 3.1.

Senior Role	**Junior Role**
Employee	NULL
Manager	Employee
CEO	Manager

Table 3.2 Role Hierarchy Table

The translator imports this role hierarchy relationship into Oracle and inserted into role table.

The next step is to create a function that sets roles in the application context. The context will contain role name for current login user. The function allows users to set the role to be “employee”, “manager”, or “CEO”. Assume that the current login user sets to his role to “employee”. The role name “employee” is stored in application context variable ‘rolename’ and if the application context name is “context_name’, the role-name ”employee” can be retrieved by calling following statement:

```
Sys_context (context_name, ‘rolename’)
```

Once a role has been set, the predicate function generates the predicate for the employee role. Since we include all three roles constraints in single predicate and the role name can be retrieved by calling “Sys_context(context_name, ‘rolname’)” statement
The predicate is generated like the following:

```
Predicate :=

Name = sys_context('userenv', 'session_user')
And role_comparsion function (current user RoleName, ‘Employee’)

OR

Manager = name= sys_context('userenv', 'session_user')
And role_comparsion function (current user RoleName, ‘Manager’)

OR

1=1
role_comparsion function (current user RoleName,  ‘CEO’)
```

Those condition expressions are generated by using the mapping file, which stores policy constraints in Ponder format and Oracle format in pair. Assume such mapping file

already created. The translator extracts policy constraints from policy model and replaces with the corresponding Oracle format one by scanning through the mapping file. After these constrains have been translated, they are put into Oracle predicate setting function. By using these condition expressions, the Oracle predicate setting function generates the predicate for all the roles.

After the user assumes an employee role, the comparison function compares three pairs of roles. For (employee, employee) pair, the comparison function returns true. But for (employee, manager) and (employee, CEO), the role comparison function returns false due to employee role is not senior to either manager or CEO role. As the result, the predicate becomes

```
Predicate :=

Name = sys_context('userenv', 'session_user')
And true

OR

Manager = name= sys_context('userenv', 'session_user')
And false

OR

false
```

This predicate ensures a user that is assuming employee role only have the employee privileges, but not manager and CEO privileges.

The last step is to add a policy. This is be achieved by calling Oracle add_policy function. The parameters of the function include: table name, predicate function name, and action type. Once this policy is added, the Oracle FGAC is fully implemented.

When a user issues a query, assume the table name is “table_name” and the query is following:

```
Select * from table_name;
```

The predicate generated by predicate setting function will be attached to this query. As the result after the predicate is attached, the query will look like:

```
Select * from table_name;
Where  name = sys_context('userenv', 'session_user')
And TURE
OR
Manager = name= sys_context('userenv', 'session_user')
And FALSE
OR
FALSE
```

Which is equal to

```
Select * from table_name;
Where name = sys_context('userenv', 'session_user')
```

As the result, an employee can only view his own record.

In this section, we have discussed how to define a VPD policy by using Ponder. We also discussed the necessary steps for a translator to translate a Ponder policy into Oracle scripts. In next section, we are going to analyze the performance of this toolkit.

4 The Toolkit User Interface

The interface of the toolkit is designed to help policy makers easily translate formal access control policies to policy enforcement programs. It allows a policy maker to specify access control policies, create corresponding enforcement programs and import the program into Oracle database. In this section, we give a brief description of the interface.

After successfully logging in, the policy maker is prompted with the major working interface, as shown in Figure 4.1. The policy maker can create new access control policies or edit previously saved policies on the left side text window. After he is satisfied with the contents of the policy, he can click the *translate* button, which invokes the policy transaction function of the toolkit. The resulting Oracle policy enforcement program will be displayed in the text pad located at right side of the window. The generated program has three parts: role setting function, predicate setting function, and policy generating function. The policy maker can review any of them by clicking *view* and choosing different functions. By displaying the access control policy and the corresponding enforcement program side by side, the toolkit gives the policy maker a visual view on how the policy is interpreted by Oracle.. Once the policy maker is satisfied with the enforcement program, he/she can click the *import* button, and the enforcement program will be imported into an Oracle database.

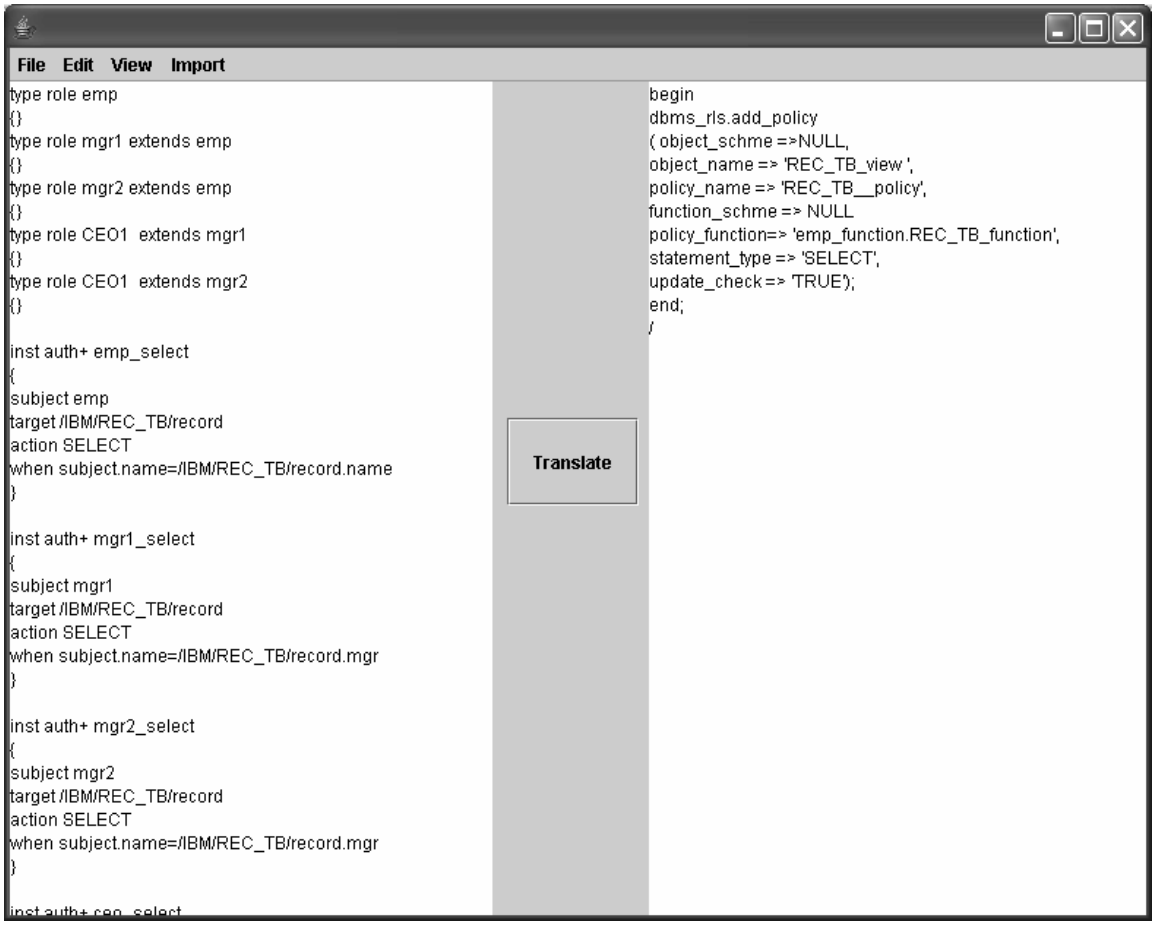


Figure 4.1 Generating Policies

This toolkit connects to the Oracle database by using JDBC with the same username/password that the policy maker uses to login into the toolkit. After the enforcement program has been successfully imported, the policy maker can test its effectiveness. The testing window is shown in Figure 4.2. During the testing, different user names with different roles can be used to issue queries. The testing window serves as a simple front end to the database management system. It submits testing questions to the database engine, and retrieve and display query results.

Figure 4.2 Testing Window

5 Performance Analysis and Optimization

In this chapter, we are going to examine the correctness of the automatically generated policy enforcement program. It is important that this toolkit can translate various policies correctly. If this toolkit cannot perform such duty, then there is a little value for this toolkit. It is also very important to consider its quality, i.e., whether access control policies can be enforced *efficiently* by using the toolkit. If after adopting the machine-generated policy enforcement program, the performance of a database management system deteriorates severely, then the toolkit is of little value. In the rest of the thesis, we refer to the policy enforcement program written by a programmer *the human-generated program*, and that generated by the toolkit *the machine-generated program.* It is reasonable to assume that human-generated program is efficient, since a programmer can carefully analyze a policy first and find the optimal way to create the policy enforcement program.

5.1 Experiment Setup

In this experiment, we use a P4 2.8 GHz computer with 512 Mb RAM. The database management system is Oracle 9i version 9.2.0.1.0.

5.2 Policy Translation

In this subsection, we are going to analyze several policies and demonstrate these policies can be correctly translated by the toolkit.

5.2.1 Hospital Example

This example is for a hospital system and we only consider the information retrieval process. The schema of the table and its access control policies are defined as the following:

- The patient record table (patient_table) contains following attributes: “Doctor_ID” defines which doctor this patient belongs to; “patient_ID “stores the id number for the patient; “patient_name” stores the name of the patient; “disease” stores the disease name of the disease.
- There are two roles in this example: doctors and patients.
- Each user in the system has to assume at least one of the above two roles before she can access the database. We assume that login authentication and role authentication are handled by other part of the system.
- The policy is like the following: a patient can only view his own information. A doctor can view all his patients’ records.
- The Ponder policy will be generated like the following:

```
Type role patient {}
Type role doctor{}

inst auth+ patient_select
{
subject patient
target patient_table/record
action SELECT
when subject.name = patient_table/record.name
}

inst auth+ doctor_select
{
subject doctor
target patient_table/record
action SELECT
when subject.ID = patient_table/record.Doctor_ID
```

In the above policy, we only defined two roles and there is no role hierarchy relationship between them. The reason is patients may have different doctors. If we let the doctor role extend from the patient role, it means a doctor can view every patient's record even if that patient does not belong to him.

5.2.2 School Example

This example is for a school system. It demonstrates how the role hierarchy relationship can be expressed. The schema of the table and its access control policies are defined as the following:

- The student record table (student_table) contains following attributes: "name" stores the name of a student; "teacher_name" stores the name of the teacher; "class_name" stores the class the student is currently taken; "TA" stores the teaching assistant's name for that class; and "grade" stores the grade for the student.

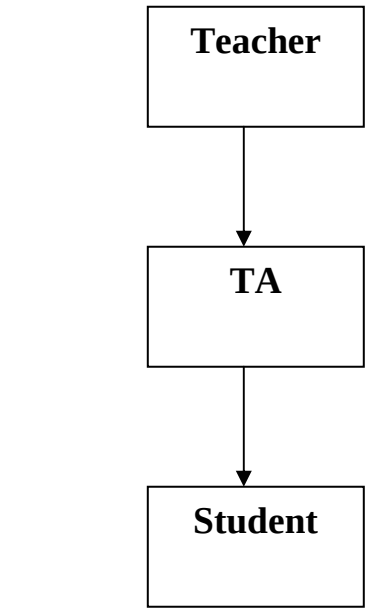
- There are three roles can issue select statement to this table: teacher, TA, and student.
- Each user in the system has to assume at least one of the above three roles before she can access the database. We assume that login authentication and role authentication are handled by other part of the system.
- The policy is like the following: a student can only view his own record. A TA can view everybody's record who he is TAing for. A teaching assistant is also a student. A teacher can view everybody's record.
- The ponder policy will be generated like the following:

```
Type role student {}
Type role TA extends employee{}
Type role teacher extends TA{}

inst auth+ student_select
{
subject student
target student_table/ record
action SELECT
when
subject.name= student_table/ record.name
}

inst auth+ TA_select
{
subject TA
target student_table/ record
action SELECT
when subject.name= student_table/ record.TA
}

inst auth+ Teacher_select
{
subject teacher
target student_table/ record
action SELECT}
```



In above example, we demonstrate that the role hierarchy relationship can be properly expressed. In this example, a teacher can have all the privileges that a student and a TA can have.

5.3 Toolkit Performance Evaluation and Analysis

The database of the experiment is for an employee management system. In particular, we consider the access control for the employee record table of the database. For simplicity, we only consider the enforcement of access control policies for information retrieval (i.e., SELECTION statements). The schema of the table and its access control policies are defined as the following:

- The employee record table (emp_table) contains the following attributes: “name”, the name of an employee; “manager”, the name of an employee’s manager; “department_ID”, the department that an employee is in; “rank”, the rank of an employee; “salary”, an employee’s salary; and “ project”, the project that an employee is currently working on. The integrity constraints of the table require that an employee can only belong to one department and work on one project at a time.
- There are seven roles defined in the database: employee (EMP), manager (MGR), human-resource staff (HR), research and development staff (RD), human resource manager (HR_MGR), research and development manager (RD_MGR), and CEO.
- Each user in the system has to assume at least one of the above seven roles before she can access the database. We assume that login authentication and role authentication are handled by other part of the system.

- A user with an employee role is allowed to view her own record. The predicate generated by a programmer is:

  name = sys_context (‘userenv’, ‘session_user’);

- A user with a manager role is allowed to view the records of all the employees that he manages. The predicate generated by a programmer is:

  manager = sys_context (‘userenv’, ‘session_user’)

- A user with a human resource staff role is allowed to view the records of all the employees whose ranks are lower than hers. The predicate generated by a programmer is:

  rank < ANY (select rank from emp_table

  where name= sys_context('userenv', 'session_user'))

- A user with a research and development staff role is allowed to view the records of all the employees who work on the same project as him. The predicate generated by a programmer will be:

  where project in (select project from emp_table

  where name = sys_context (‘userenv’, ‘session_user’)

- A user with a human resource manager role has the privileges of a manger and a human resource staff. No further privileges are given to this role.
- A user with a research and development manager role has the privileges of a manager and a research development staff. No further privileges are given to this role.
- The CEO is allowed to view the entire table.

  The figure below shows the role hierarchy of the experiment.

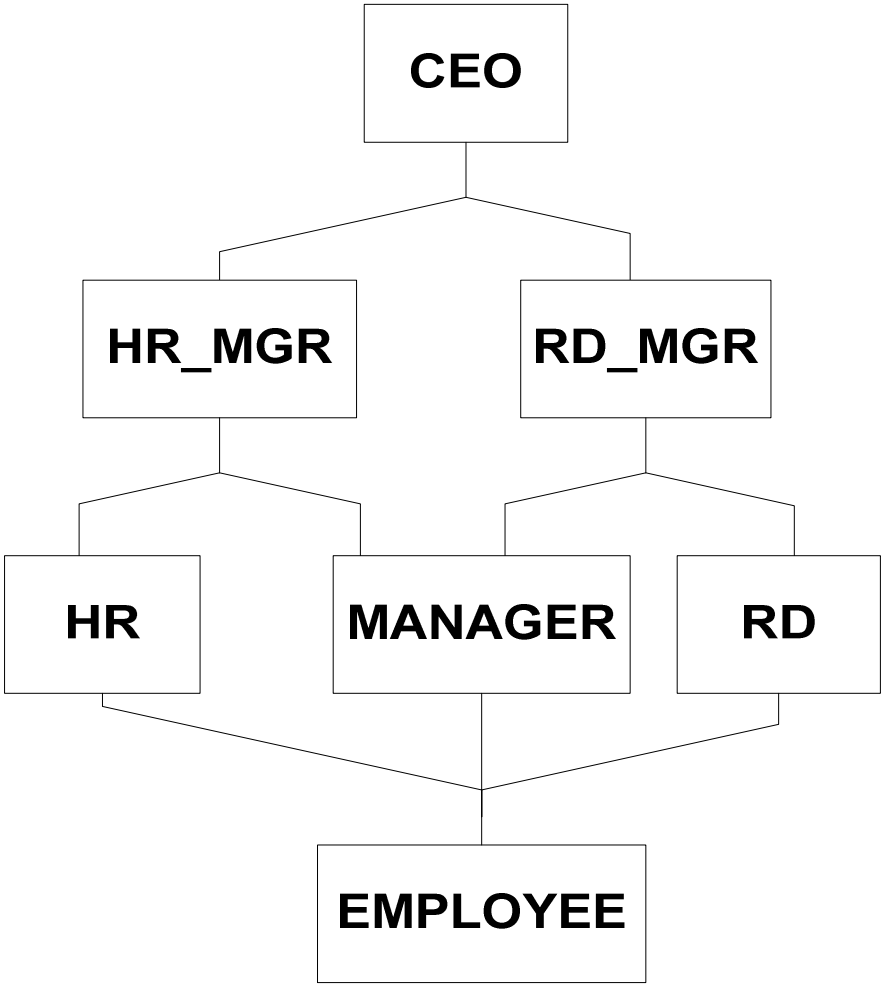


Figure 5.1 Role Hierarchies for the Experiment

5.3.1 A Review of Machine-Generated Policy Enforcement Program

Once a user issues a query, a human-generated program typically attaches a single predicate to the query. The predicate is specific to the user's current role. On the contrary, the machine-generated program described in Chapter 3 takes a simple and holistic approach. No matter what role the user assumes, the program attaches to the query a constant predicate, which encodes the access control constraints for all roles. Remember that the predicate contains invocation of the role comparison function. Therefore, during query execution, when the predicate is evaluated by the database engine, the returned value of role comparison functions will dynamically determines which constraints should take effect, based on the user's current role.

Table 5.1 shows an example to illustrate the difference between the predicates of a human-generated program and that of a machine-generated program. In this example, the user's current role is 'employee' and the issued query is "select * from emp_table".

Manually Generated Code	Machine Generated Code
Select * from table_name Where name = sys_context('userenv', 'session_user');	Select * from table_name Where name = sys_context('userenv', 'session_user') And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “EMP’’) OR Manager= sys_context('userenv', 'session_user') And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “MGR’’) OR Rank = Sys_context (context_name, ‘rank’) And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “HR’’) OR Project = sys_context(contxt_name, ‘project’ And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “RD”) OR Manager= sys_context('userenv', 'session_user') AND Rank = Sys_context (context_name, ‘rank’) And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “HR_MGR’’) OR Manager= sys_context('userenv', 'session_user') AND Project = sys_context(contxt_name, ‘project’ And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “RD_MGR’’) OR And 1= role_comparison_function (‘‘context_name’’, ‘rolename’’, “CEO’’)

Table 5.1 Comparison of Manually Generated Code and Machine Generated Code

5.3.2 Performance Evaluation and Analysis

We compare the performances when adopting human-generated policy programs and machine generated programs. In the experiment, the employee record table has 2000 tuples. We measure the query execution time of users with different roles, when issuing query “Select * from emp_table”. The performance results are shown in Table 5.2.

Table Size /Role	500 (Manually Generated)	(Machine Generated)	1000 (Manually Generated)	(Machine Generated)
EMP	0	44.04	0	92.07
MGR	0.01	43.01	0	83.07
HR	0.01	3.02	0	5.07
RD	0	31.09	0	59.09
HR_MGR	0	16	0	31.07
RD_MGR	0	32.07	0	71.11
CEO	0	15.06	0.01	34.08
	1500 (Manually Generated)	**(Machine Generated)**	**2000 (Manually Generated)**	**(Machine Generated)**
EMP	0	135.07	0	193.03
MGR	0	118	0	174.01
HR	0.01	8.02	0.01	11.06
RD	0	101.07	0	141.02
HR_MGR	0.01	57.04	0.01	73.02
RD_MGR	0	109.06	0.01	149.05
CEO	0.01	56.02	0.01	63.04

Table 5.2 Results of Initial Approach with Table Size 500-2000

The results show that the machine-generated program is far inferior to the human-generated program in terms of efficiency. For example, for a user with an employee role, the performance of the machine-generated program is more than 150 seconds slower than that of the human-generated program.

We need to identify the performance bottleneck and optimize the machine-generated program accordingly. Remember that each clause of the predicate produced by the machine-generated program contains two parts: the constraint for a specific role and

an invocation of the role comparison function. Only when the role comparison function returns true will the corresponding constraint limit a user's access Since the role comparison function is also implemented through a SELECT query, a user's query after rewritten becomes a nested query, which usually yields sub-optimal performance. For example, if a role constraint returns 200 tuples, the role comparison function will be executed 200 times. If a tuple does not satisfy the role –specific constraint, the role-comparison function will be invoked.

Note that a user's role remains the same during a session (i.e., from the user logons into the database until she logs out). Therefore, repeated invocation of the expensive role comparison function is unnecessary during query execution. One way to optimize the machine-generated program is to pre-set the results of the role comparison function as context values. Once a user sets her role, we will compare the user's role with all the defined roles in the system. The results returned by the role comparison function are stored as context variables for the user. For example, for the employee record management system used in the experiment, we define the following context variables for each user: isEmployee, isManager, isHRManager, isRD, isRDManager, isHR and is CEO**.** Intuitively, if the user's current role is the same or senior to a given role, then the corresponding context variable is set to be *true*. We call such context variables *role membership* variables.

To determine whether a constraint for a specific role should take effect, the generated predicate simply needs to retrieve the corresponding role membership variable for the context. Since retrieving a context value is much faster than the invocation of the role comparison function, the query execution time should be significantly reduced.

Based on this observation, we further improve the predicate setting function. Instead of including the context variables as part of the generated predicate, the context variables are used to decide whether the corresponding constraints should be included int the predicate. The major part of the predicate setting function is like the following:

```
Pred = ‘’;
Employee conditions = ‘name = sys_context('userenv', 'session_user')’;
Manager conditions = Manager= sys_context('userenv', 'session_user')
HR conditions = ‘Rank = Sys_context (context_name, ‘rank’)’
CEO conditions = ‘1=1’;


If (1= Sys_context (context_name, ‘compared_value1’) )
Then predicate := CONCAT (pred, ‘employee conditions’);
End if;

If (1= Sys_context (context_name, ‘compared_value2’) )
Then predicate:= CONCAT (pred, ‘manager conditions’);
End if;

If (1= Sys_context (context_name, ‘compared_value3’))
Then predicate: = CONCAT (pred, ‘HR conditions’);
End if;
.
.
.
IF(1= Sys_context (context_name, ‘compared_value7’))
Then predicate:= CONCAT (pred, ‘CEO conditions’);
End if;
```

By using this approach, if a user’s role is employee, the role comparison function sets the role membership variable *isEmployee* to be true, but those for other roles to be false. As the result, only constraint specific for the employee role will be included in the returned predicate. The query after rewritten will be the following, which is very similar to the one returned by the human-generated program:

```
Select * from table_name
Where name = sys_context('userenv', 'session_user');
```

The following table shows the time difference between the *human-generated program* and *machine-generated program*:

Table Size /Role	500 (Manually Generated)	(Machine Generated)	1000 (Manually Generated)	(Machine Generated)
EMP	0	0	0	0
MGR	0.01	0.01	0	0.01
HR	0.01	0.01	0	0.01
RD	0	0.01	0	0.01
HR_MGR	0	0	0	0
RD_MGR	0	0.01	0	0.01
CEO	0	0.01	0.01	0.01
	1500 (Manually Generated)	**(Machine Generated)**	**2000 (Manually Generated)**	**(Machine Generated)**
EMP	0	0	0	0
MGR	0	0.01	0	0.01
HR	0.01	0.01	0.01	0.01
RD	0	0.01	0	0.01
HR_MGR	0.01	0.01	0.01	0.01
RD_MGR	0	0.01	0.01	0.01
CEO	0.01	0.01	0.01	0.01

Table 5.3 Third Approach Results

We see the performance difference between the *human-generated* and *machine-generated program* is almost negligible.

6. Related Work

This work is related to many areas, including: relational database management system (RDBMS), access control, and policy languages. Much work has been done in each of these areas. In this chapter, we describe the work that most heavily influences database access control.

6.1 Access Control Policies

There are two types of access control policies: discretionary access control (DAC) [BJS 95] [TY03] and mandatory access control (MAC) [BJS 95] [TY03].

DAC restricts a user's access privileges to an object. Access policies are decided by the owner of the object. Different user may have different access privileges for a same object. Most database systems support DAC [BJS 95]. We can define a database table as an object. The creator of the table will automatically get all privileges on it. The creator can pass different access privileges of this table to other users.

In MAC, access control policies are decided by administrators instead of object owners. In MAC, each object has an access level such as secret, classified, and unclassified, etc. Each user is assigned to have a clearance level. A user can only access those objects that he has clearance. The difference between MAC and DAC is in MAC the privileges are static, not based on content. However, an organization structure cannot be easily interpreted by using classification levels. As the result, role based access control (RBAC) [GB98], [BBU99] is introduced.

In RBAC, access control privileges are associated with roles and users are assigned to roles based on their responsibilities and qualifications. For RBAC, the roles and role hierarchy are based on the structure of an organization. For example, roles in a

school may include teacher, student, teaching assistant, etc. Based on a user's responsibility, he/she may assume more than one role. In the above example, a user can be a student and a teaching assistant at the same time. When assigning roles, the principle of least privilege should be followed. A user should only have the minimum privileges that are enough to perform his duty. With RBAC, a user's roles can be easily changed. Privileges can be granted or revoked from roles as needed. Further, role and role hierarchy can be mapped to the operational activity of an organization.

6.2 Database Access Control

Although most database systems support RBAC, MAC, and DAC, many different strategies have been proposed to provide a more secure environment for database systems. In this section, we examine some of these approaches, including rule based access control [TD97], IBM's sticky policy [AHC03, AA04] and information disclosure management [YW04]

The general idea of rule-based access control [TD97] is as follows. An enterprise's organizational structure is created as a table. For example, if we have a structure like "manager-> employee" where manager is the parent node of employee, the table will be created as followings:

Role Name	Symbol
Manager	A
Employee	A1

Table 6.1 Role Hierarchy Table

The employee role uses the symbol (A1) that is similar to the manager's symbol(A). When a manager issues a query, the SQL keyword 'LIKE' will be used to

identify that employee node is the sub-node of the manager. For each unit in an organizational structure, a set of privileges is defined with the organizational structure table symbol attributes. The database users' name/roles will be mapped to nodes in the organizational structure. When a user issues a query, the database server will check whether the user has clearance to retrieve data by looking up the organizational structure table. The problem for the rule based access control is that if we have a very large organization structure, the symbol we used to represent each node will be getting very long and complicated. Since the SQL 'LIKE' operation is a string comparison function, the performance of this access control mechanism is going to be decline.

Several access control projects have been under going in IBM. One of them is called "sticky policy paradigm"[AHC 03]. In sticky policy, the policy will be enforced on data. "Policy" includes the conditions and requirements of data usages and must be always associated with data instances. Even when the data is transferred from one database to another, the policy is still attached with the data and always true. The only time that policy could be invalidated is when data owner issue a policy invalidation statement. The disadvantage of stick policy is when the policy attached to the data is updated, the user's data is still managed by the old policy, not to the new policy. As the result, it may cause security risk. The first attempt to implement such policy is on Tivoli Privacy Manager, a privacy policy management tool developed by IBM. According to IBM, this is the first enterprise privacy management solution that automates privacy policy enforcement and monitoring.

Watanabe[YW04] from IBM purposed a model for information disclosure management. The model has two parts: a centralized information disclosure decision

center and policy enforcement agents. There are two functions in the model: Access Enforcement Function (AEF) and Access Decision Function (ADF). The AEF always associates with data and ADF is located with the central server. When a user is trying to access a database, AEF will check his login information and send it to ADF. The ADF will check the user's information with the stored policy. If the login information meets the access control requirements, the user can precede and successfully retrieve information from the database. However, since all the decision will be made by the central server, if a central server has a large number of enforcement agents associate it, it may cause a bottleneck issue.

7 Conclusions and Future Work

7.1 Conclusion

In this thesis, we present the design and implementation of the access control enforcement toolkit (ACET) that automatically translates database access control policy specifications to Oracle policy enforcement programs. Instead of letting a programmer create access control code, which is an error prone process and may introduce security risks because the access control code is hard to analyze, this toolkit may help a policy maker create policy enforcement programs more easily. We evaluate the performance of the toolkit and identify a variety of ways to optimize the performance of the automatically generated policy enforcement program.

This toolkit is composed of two parts: a policy specification model and a policy enforcement program translator. After comparing several formal policy specification languages, we choose Ponder as a preliminary language for this project because it has a concise and intuitive syntax for policy specification. It also supports role hierarchy definition and role-based access control in a straightforward manner as previously discussed.

Taking the Ponder policy as the input, the enforcement program translator generates three functions: role setting, predicate setting, and policy generation.

The automatically generated policy enforcement program can correctly enforce database access control policies. At the same time, the preliminary results suggest it may do it efficiently. After applying several optimization techniques, our preliminary experiment suggests that the machine-generated program yields comparable performance to that created by a database programmer.

7.2 Limitation of this Toolkit

Although this toolkit offers basic functionalities to automatically translate abstract database access control policies into Oracle policy enforcement programs, it has several limitations that require further investigation.

First, because the policy language is limited, this toolkit cannot express some complicated queries. For example, suppose a policy states that a manager can view the records of the employees who report directly or indirectly to her. Such a policy can be easily expressed by SQL by using the following statement:

```
SELECT *
FROM table_name
START WITH manager = sys_context
(…)
CONNECT BY PRIOR name = manager
```

However, this query cannot be directly expressed in Ponder, unless a more complex abstract data model is defined to represent the relationship between tables in a database.

Second, the mapping file for translating Ponder policy variables to Oracle database variables has to be predefined. Generating such a mapping file requires comprehensive knowledge of both the formal access control policies and the details of the database implementation. Thus, in some sense, it shifts the burden from programmers to policy makers. Also, since it requires a policy maker to create such file and stores them a safe place, the situation of mishandling such file could happen.

Third, this toolkit assumes that a policy maker has some database privileges, such as granting procedures, adding and dropping policies, which typically are the privileges of database administrators. Although the access control enforcement program is

automatically generated, and it does not require the policy maker to manually create any Oracle functions, the possibility of misusing these privileges does exist.

7.3 Future Work

To make the toolkit completely suitable for practical usage, several issues that need to be further investigated.

First of all, a formal policy model and a policy language for database fine-grained access control are needed. The existing policy languages are not designed specifically for databases. Therefore, they are not expressive enough to fully support database access control policies. For example, though Ponder is the preliminary formal language for the toolkit, certain extension to Ponder is required in order to support some commonly used database access control policies.

Besides row-level access control, many web applications further require cell-level access control, i.e., even a single record may be only partially accessible to a user. For example, though an employee can access some common attributes of other employees in the same group, their salary information should only be accessible to the manager of the group. Cell-level access control can be achieved through the combination of row-level access control and column-based access control. Another approach is to partition a table vertically into several sub tables. After applying row-level access control on each sub table, we may enforce cell-level access control through outer joins between those sub tables. How to translate abstract access control policies to automatically enforce cell-level access control is a challenging problem we would like to investigate in the future.

Reference

[AA04] A. Anderson**.** An Introduction to the Web Services Policy Language (WSPL). *Sun Microsystems Laboratories, Jun 2004*

[AHC03] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, Enterprise Privacy Authorization Language (EPAL 1.1). *IBM Research Report*, October 1, 2003.

[ASP02] P. Ashley, M. Schunter and C. Powers, "From Privacy Promises to Privacy Management - A New Approach for Enforcing Privacy Throughout an Enterprise", Proceedings of the New Security Paradigms Workshop (NSPW), Virginia, 23-26 September 2002.

[BBU99] J. Barkley, C. Beznosov, Uppal, "Supporting Relationships in Access Control using Role Based Access Control" , *Fourth ACM Workshop on Role-Based Access Control* 1999

[BCF03] E. Bertino, B. Catania, E. Ferrari and P. Perlasca. A Logical Framework for Reasoning about Access Control Models. In *ACM Transactions on Information and System Security*, 6(1), pp.71-127, February 2003.

[Bir00] P. Bird. Implementing Low Level Access Control with DB2 UDB. The *IDUG solution journal*, Volume 7 Number 3, Winter 2000

[BJS 95] E.Bertino, S.Jajodia, and P.Samarati. Database Security: Search and Practice. Information Systems. VOL 20 No. 7 pp 537-556

[Dam02] N. Damianou .A Policy Framework for Management of Distributed Systems, University of London, Feb 2002.

[DDL01] N. Damianou, N. Dulay, E. Lupu, M Sloman, The Ponder Specification Language. *Workshop on Policies for Distributed Systems and Networks* (Policy2001), HP Labs Bristol, 29-31 Jan 2001.

[DHH02] G. Della-Libera,P. Hallam-Baker, M. Hondo etc. Web Services Security Policy (WS-SecurityPolicy). http://www.ibm.com/developerworks/library/ws-secpol/index.html Dec 2002

[DLS01] N. Dulay, E. Lupu, M Sloman, N. Damiano. A Policy Deployment Model for the Ponder Language *Proc. IEEE/IFIP International Symposium on Integrated Network Management (IM'2001)*, Seattle, May 2001.

[FGL92] D. Ferraiolo., D. Gilbert, and N. Lynch Assessing Federal and Commercial Information Security Needs. NISTIR 4976. Gaithersburg, MD: National Institute of Standards and Technology. 1992.

[FKC95] D.F. Ferraiolo, J. Cugini, D.R. Kuhn "Role Based Access Control: Features and Motivations" , *Computer Security Applications Conference, 1995*

[GB 98] S. Gavrila, J. Barkley, "Formal Specification for Role Based Access Control User/Role and Role/Role Relationship Management" (1998), *Third ACM Workshop on Role-Based Access Control.*

[GOK02] M. P. Gallaher, A. C. O'Connor, and B.Kropp. The Economic Impact of Role-Based Access Control Research Triangle Park, NC: Research Triangle Park Institute. 2001

[Kag02] Lalana Kagal, "Rei : A Policy Language for the Me-Centric Project, TechReport, HP Labs, September 2002.

[KD02] Kristy Browder and Mary Ann Davidson, The Virtual Private Database in Oracle9*i*R2 Oracle Corporation, Redwood Shores, CA 94065 2002

[KFJ03] L. Kagal, T.Finin, and A.Joshi. A Policy Based Approach to Security for the Semantic Web, InProceedings, *2nd International Semantic Web Conference (ISWC2003),* September 2003.

[KaFJ03] L. Kagal, T. Finin, and A. Joshi. A Policy Language for A Pervasive Computing Environment, InCollection, *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, June 2003.

[KSW02] G. Karjoth, M. Schunter and M. Waidner. Platform for Enterprise Privacy Practices: Privacy-Enabled Management of Customer Data. In *Proceedings of the Second International Workshop on Privacy Enhancing Technologies (PET 2002),* LNCS 2482, pp. 69-84, 2003.

[Kyt] T. Kyte, Fine Grained Access Control and Application Contexts. http://govt.oracle.com/~tkyte/article2

[REW04] REWERSE (Reasoning on the Web with Rules and Semantics) http://rewerse.net/

[RMS04] S.Rizvi, A.Mendelzon, S.Sudarshan,and P.Roy. Extending query rewriting techniques for fine-grained access control, *International Conference on Management of Data Proceedings of the 2004 ACM SIGMOD international conference on Management of data,* Pages: 551 - 562 2004

[Sch01] D. Scherer. Fine Grained Access Control with Oracle 8i's virtual private

database features.
http://www.coreparadigm.com/conferenceDocs/vpd/vpd.ppt.pdf

[TD97] T. Didriksen. Rule based database access control—a practical approach ACM Workshop on Role Based Access Control Proceedings of the second ACM workshop on Role-based access control Fairfax, Virginia, United States Pages: 143 - 151 1997

[TY03] Ting Yu: "Automated Trust Establishment In Open Systems". University of Illinois at Urbana-Champaign, October, 2003.