FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# A Review of Technologies supporting Dynamic Fine-Grained Data Access Control in Relational Databases

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**Gerhard Schraml, BSc.**
Registration Number 00728067

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Robert Sablatnig

Vienna, 20th November, 2018

Gerhard Schraml    Robert Sablatnig

# Erklärung zur Verfassung der Arbeit

Gerhard Schraml, BSc.
1100 Laxenburger Straße 29/8

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. November 2018

Gerhard Schraml

# Kurzfassung

Die Menge an sensiblen Daten, die in Enterprise-Applikationen verarbeitet werden, wächst kontinuierlich. Gleichzeitig kann ebenso beobachtet werden, dass die Anzahl der Anwender dieser Applikationen steigt. Aus diesem Grund ist es notwendig, Mechanismen zu entwickeln, die die effiziente Verwaltung von Zugriffskontrolle auf sensible Daten ermöglicht. Feingranulare Datenzugriffskontrolle und Attributbasierte Zugriffskontrolle sind junge Konzepte die die Umsetzung dieser Anforderung unterstützen. Noch komplexer wird es, wenn sowohl die Struktur der Datenbank als auch die Zugriffsregeln zur Entwicklungszeit einer Applikation noch nicht bekannt sind. Es wird ein Prototyp eines Systems vorgestellt, welcher Dynamische Feingranulare Datenzugriffskontrolle im Forschungsfeld des Sustainable Non-Bureaucratic Government ermöglicht. Zu diesem Zweck wird Virtual Private Database eine Funktionalität der Oracle-Datenbank, herangezogen, um erfolgreich einen Prototyp einer Applikation mit dynamischen Zugriffsregeln zu entwerfen. Die Applikation unterstützt ihre eigene Modifizierung zur Laufzeit durch demokratische, gemeinschaftliche Entscheidungsfindung ihrer Anwender. Der replizierende Vergleich mit einem bestehenden Prototyp zeigt, dass mit Hilfe von Virtual Private Database eine Implementierung eines Systems mit Dynamischer Feingranularer Datenzugrifsskontrolle möglich ist. Im Gegensatz dazu wird gezeigt, dass es nicht möglich ist, standardisierte Definition von Zugriffsregeln mittels der eXtensible Access Control Markup Language im selben Prototyp zu erreichen.

# Abstract

The amount of sensitive data processed by enterprise applications is constantly increasing. Simultaneously, a growing number of users of these applications can be observed. Thus it is necessary to provide means for efficiently managing access control to sensitive data. Fine-Grained Data Access Control and Attribute Based Access Control are new concepts assisting with fulfilling this requirement. The complexity of access control reaches an even higher level, if database structure and authorization rules are not known at application design time. A proposed prototype system in the field of Sustainable Non-Bureaucratic Government aids as a decision criterion for which technologies can be used to implement Dynamic Fine-Grained Data Access Control. Virtual Private Database is a feature provided by Oracle Database that is successfully utilized in the prototype system to build a dynamic access control aware application. The application supports its own alteration during runtime by means of democratic collaborative decision making of the users. In a comparison with an existing prototype system it is shown that it is possible to replicate the feature set of Dynmic Fine-Grained Data Access Control using Virtual Private Database. In contrast, it is found that it is not possible to integrate standardized access policy definition with the prototype using the eXtensible Access Control Markup Language.

# Contents

CHAPTER 1

# Introduction

Evolving database technology has enabled applications to store and process practically uncountable amounts of person-related data [5]. The implementation of access control mechanisms aims at protecting sensitive data from being accessed by unauthorized users or user groups. *Relational Database Management Systems* (RDBMS) provide means for storing all the application data in tabular form. Built-in access control in common RDBMS (such as Oracle Database [28]) is traditionally implemented only at relation level, that is, access can only be granted for tables or views as a whole [22]. This level of access control is referred to as *coarse-grained*[23].

In contrast, *Fine-Grained Data Access Control* (FGDAC) allows for managing authorizations each on row, column and cell level of tabular data [41]. As a short example, consider a database table containing all the employees records of a company including names, salaries and department affiliations. With a FGDAC implementation deployed, access constraints as following could be enforced: "employees may only view their own salary", "employees may only see the names of employees belonging to the same department".

Compared to access control systems with static database structure and policies, the problem under discussion gets more complex, if both database structure and access control policies to apply are not known in advance, i.e. at application design time. This is due to the fact that it is impossible for application designers and developers to anticipate all possible future changes to the system.

Such is the case in the emerging research field on *Sustainable Non-Bureaucratic Government* (SNBG), which aims at creating a system of structured, machine-readable law. The following statements refer to the work of Paulin (2015) [35]. SNBG involves democratic creation, enactment and modification of machine-readable access control policies to system-relevant data during runtime, as well as democratic creation and modification of the underlying database structure - likewise, during runtime. Accordingly, database access policies in SNBG can neither be defined nor abstracted (e.g. in form of templates) at design-time. In SNBG, the absence of administrative super users, who could corrupt the democratically created policies when implementing them in the system, is of crucial importance. Furthermore, authorization decisions can no longer solely depend on the identity of a requestor. This is due to the usage of attributes describing user, resource and the environment of an authorization request. These attributes allow for a less maintenance-intensive and more general definition of access control policies, compared to systems utilizing Identity Based Access Control [23]. This relatively new approach is referred to as *Attribute Based Access Control* (ABAC) [23].

In order to provide a name for the functional requirements defined above the term *Dynamic Fine-Grained Data Access Control* (dFGDAC) is introduced by Paulin [35]. The main purpose of this thesis is expected to aid as a decision criterion on which technologies can be utilized to set up a dFGDAC system. The technologies under comparison are *Oracle Virtual Private Database* (VPD) [39] and *Secure SQL Server* (SecSQL) [35], a prototype system specifically designed to meet the criteria as required by SNBG. In this work a new prototype is proposed trying to replicate the dFGDAC features of the existing SecSQL solution by utilizing the fine-grained query rewriting features of VPD.

To establish comparability, performance indicators are identified within this work by considering both functional and non-functional features necessary for implementing a dFGDAC system. Examples for such indicators are provided access control granularity, ABAC support and the ability to implement a dynamic system that does not require the knowledge of database structure an access constraints in advance. Subsequently, a detailed hands-on investigation using the candidate technologies focusing on these performance indicators is executed.

In addition, the possibility of adding aspects of standardization to a dFGDAC system is evaluated. It can be conceived that dFGDAC is applicable to several similar problem domains in terms of public, collaborative usage of sensitive data [35]. Especially in the area of *Open Data* (control of access to governmental data) using standards enables systems being compatible and interoperable. To this end, *eXtensible Access Control Markup Language* [15] (XACML), an XML based language standardizing the way ABAC policies are defined, is evaluated in terms of possible integration with a dFGDAC system.

As a result of the basic definitions and concepts introduced above, it is possible to formulate the following main research question, this work aims to answer:

> "Is it possible to build a dFGDAC system in the application scenario of SNBG utilizing Oracle VPD?"

In addition, another research question is stated as follows:

> "Is it possible to add aspects of standardization by integrating XACML as language for definition of fine-grained access constraints?"

The main contribution of this work is to aid as a decision criterion on which technology is best-suited for use as an enabler of SNBG regarding dynamic fine-grained data access control. In addition, evaluating the integration of standardization approaches provides insight into possible couplings with other similar access-constrained applications. Finally, the thesis presents an important contribution to fostering technological development of applications covering similar domains in terms of sensitivity and collaborative usage of data, such as *Smart City* (governance of access to public resources), *Open Data* (control of access to governmental data) or *e-Health* (governance of access to Hippocratic data).

The remainder of this work is organized as follows. In Chapter 2, theory and concepts of access control are discussed in detail. It includes an overview on the evolution of access control and extensive introductions to FGDAC and dFGDAC. In addition, the XACML standard is introduced and described. Chapter 3 presents a new prototype dFGDAC system making use of Oracle VPD. Documentation of the prototype includes data models, application-specific business logic as well as a brief overview of the provided web-based user interface. In Chapter 4, efforts to add standardization aspects to the system using the XACML language for access constraint definition are documented. It contains descriptions of utilizing high-level XACML implementations, direct integration of XACML policies with relational databases as well as a discussion of both approaches. In Chapter 5, both prototype dFGDAC solutions are discussed. The chapter comprises the definition of the performance indicators, description of the considered technologies in terms of these indicators and a tabular, easily readable by-feature comparison. Finally, results are concluded in Chapter 6. The chapter presents both findings of the current work as well as a brief outlook on future work.

CHAPTER 2

# Concepts of access control

This chapter outlines the most important facts and basic concepts of access control in order to provide a solid foundation for further research presented in the following chapters. Section 2.1 draws a line from the first efforts to latest development of access control. In Section 2.2, the foundations of Fine-Grained Data Access Control are presented in detail. Section 2.3 introduces the term *Dynamic Fine-Grained Data Access Control* and provides an overview of an existing prototype implementation. Finally, Section 2.4 outlines the fundamentals of the *eXtensible Access Control Markup Language*.

## 2.1 Evolution of access control

Beginning from the 1970s, several access control models were presented by researchers and system engineers [27]. They all have in common to provide means for controlling the way, access to digital resources is granted [27]. Starting off with basic access control models, growing data volume as well as rapidly increasing user numbers lead to the need for sophisticated mechanisms [20]. The steadily evolving models may be grouped in low-level models, consisting of *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC), followed by first approaches of standardized access control mechanisms, namely *Role-Based Access Control* (RBAC). DAC utilizes lists of allowed actions per user, attached to the object under protection [46]. In contrast, MAC defines an ordered labelling system describing the sensitivity and trustworthiness of each objects and users, respectively [9, 47]. RBAC introduces the concept of roles to act as an intermediary between users and objects [3, 45]. Lately, developments tend towards *attribute-based access control*, trying to make decisions based on attributes describing the involved participants rather than utilizing the identity of a requestor [23, 8, 48, 49].

### 2.1.1 Conceptual distinction: authentication vs. authorization

Speaking simplified, access control is based upon two steps, namely *authentication* and *authorization* [17]. Authentication is defined as verifying the identity claimed by the subject requesting access to a certain resource [23]. This verification can in general be achieved using one or more of the following authentication factors:

- *knowledge factor* - something the subject knows, e.g. a password
- *possession factor* - something the subject possesses, e.g. an electronic key card
- *inherence factor* - something the subject is, e.g. physical characteristics such as finger prints

Requiring two or more of these factors at the same authentication request is referred to as *multi-factor authentication* and obviously leads to a stronger form of protection than relying on just one single authentication factor.

In contrast, authorization describes the process of determining, whether a subject is allowed to gain access to a requested resource (e.g. is Bob allowed to view health records?) [23]. Hence, authorization depends on proper authentication, as granting access to a resource at least requires to be sure about the identity of the subject asking for access.

The following sections focus on describing concrete models of the authorization step, leaving the concept of authentication out of consideration. Thus, the terms *access control* and *authorization* are used synonymously within the scope of this work.

### 2.1.2 Discretionary Access Control

Basic elements of DAC are *subjects*, *objects* and *actions* [20]. Subjects describe active elements of a system, demanding request on certain resources. These resources are referred to as objects, whereas actions describe the kind of access, e.g. read or write. The most-known implementation of DAC is via *Access Control Lists* (ACL), which basically attach a list of allowed actions to concrete objects. Early versions of Unix systems already included that kind of access control (user/group/world, read/write/execute) [17]. Another common example is the privilege system implemented in relational databases (e.g. Oracle Database), including granting and revoking privileges (SELECT, INSERT, etc.) on certain database objects, such as tables or views.

Basically, subjects and objects are recognized by their identity [46]. Presence of an entry in the ACL of the object to be protected, containing the requestor (subject) and the desired type of access (action) results in a positive access decision. The model is called *discretionary*, as the owner of an object is in charge of managing who may or may not gain access to it. Furthermore, owners may also delegate (*grant*) other users the privilege of managing access to a given resource. Such privileges may also be *revoked*. These actions are regulated by a general, administrative access control policy.

The majority of DAC implementations uses *closed policies*, that is, every permitted access of any subject to any object has to be explicitly defined [46]. Requests, for which no entry in the ACL can be derived, result in a negative access decision. In contrast, *open policies* grant every requested access, unless it is explicitly defined to be forbidden [46, 9]. In general, security-critical systems should rather be protected using a closed policy access control model ensuring better protection [46], whereas open policy models are able to reduce administration effort for systems with small need for protection. It is also possible to use a mixture of both approaches. However, this may result in conflicting access requests, for which both a positive and a negative decision might be derived. To this end, *combining algorithms* are necessary to come to an unambiguous decision.

Defining access rights at the granularity of users and resources enables detailed control over what a user is allowed to access as well as who is allowed to access a specific resource. In return, that leads to higher administration effort, compared to generic approaches of access control definition, and poor scalability in case of growing user numbers. One approach to get rid of that problem is the introduction of *user groups*. Users belonging to a specific user group inherit all the access rights granted to the group. Another negative aspect of the DAC model is the propagation of the identity of a user to a process executed by her. That means, every program executed by a user gains the privileges of that user. Thus, an attacker might make use of that privileges by providing a piece of potentially malicious code and forcing the victim to execute it (*Trojan Horse*) [46].

### 2.1.3 Mandatory Access Control

A key element of Mandatory Access Control (MAC) is the so-called *security level* [20, 9]. Both objects (resources) and subjects (requestors) within a system are classified using a security level, describing the need for protection and the permission to access a certain class of protected resources, respectively. Security levels may be defined using a partially ordered set, that is, a ranking scale in the following form: $SecLevel1 < SecLevel2$, which reads as follows: “SecLevel1 is lower than SecLevel2”. An example for a security level definition is:

$$unclassified < confidential < secret < top\ secret$$

MAC using above definition is implemented as follows: subjects given a certain security level are not allowed to gain access to objects classified as higher security level. Consequently, they are allowed to gain access to all objects classified as lower or equal security level. This rule is referred to as *no read-up* [9]. In return, the *no write-down* rule says that no subject is allowed to write to objects at a lower security level, in order to prevent leakage of highly sensitive information. Similarly to DAC, Trojan Horse attacks would be possible otherwise by exposing information of high confidentiality to objects that may be read by subjects of low security level.

One way of implementing MAC is via *Multilevel Security Policies* (MLS) [9]. Security levels assigned to objects are called *classifications*, whereas security levels assigned to subjects are referred to as *clearances* [9]. The classification of a resource defines the sensitivity within an organization, while the clearance of a user gives evidence concerning the trustworthiness of the user with respect to his ability on keeping discretion of the protected information. MLS were already used for military and governmental purposes before first efforts were undertaken at the beginning of the 1970s to implement the model in electronic information systems [9]. MLS policies can be divided into two types:

- *confidentiality policies*
- *integrity policies*

Security levels in confidentiality policies are called *confidentiality levels* and are defined as a tuple comprising a *sensitivity level* (secret, top secret, etc.) and one or more *compartments* the level applies to [9]. Compartments can be seen as some kind of application-specific domain, e.g. health care, transport, etc. A positive authorization decision may only be derived if both of the following conditions are met: (1) the sensitivity level of the requested resource is lower or equal to the requestors sensitivity level; (2) the set of compartments the requestor is allowed to gain access to contains all elements of the set of compartments attached to the requested resource, in other words: the requestor has to be "cleared" for all domains the resource is involved in. MLS additionally contains the discretionary approach of DAC, that is, the owner of a resource must additionally grant a user of sufficient clearance the access to contained information.

Security levels in integrity policies are called *integrity levels*. Classifications of resources therefore refer to the level of confidence that may be put in the information contained. Clearance of users is defined by the trustworthiness that may be placed in information originated from this user.

### 2.1.4 Role-Based Access Control

As information systems became larger in terms of number of users as well as covered functionalities, administration of simple DAC and MAC policies started to get cumbersome [17]. Thus, efforts were undertaken to develop models for being able to cope with upcoming access control requirements. According to [3] and [17], Ferraiolo and Kuhn were the first to introduce a standardized feature set of *Role-Based Access Control* (RBAC) in 1992 [16]. Built upon this work, Sandhu et al. introduced a whole framework of models describing different types of possible RBAC implementations in 1996 [42]. In the following years, standardization attempts were undertaken resulting in the approval of a complete proposal as an official INCITS standard by the U.S. *National Institute of Standards and Technology* (NIST) [17]. It is worth mentioning that RBAC does not aim at completely substituting any DAC or MAC implementations but rather extend basic access control mechanisms by new functionalities.

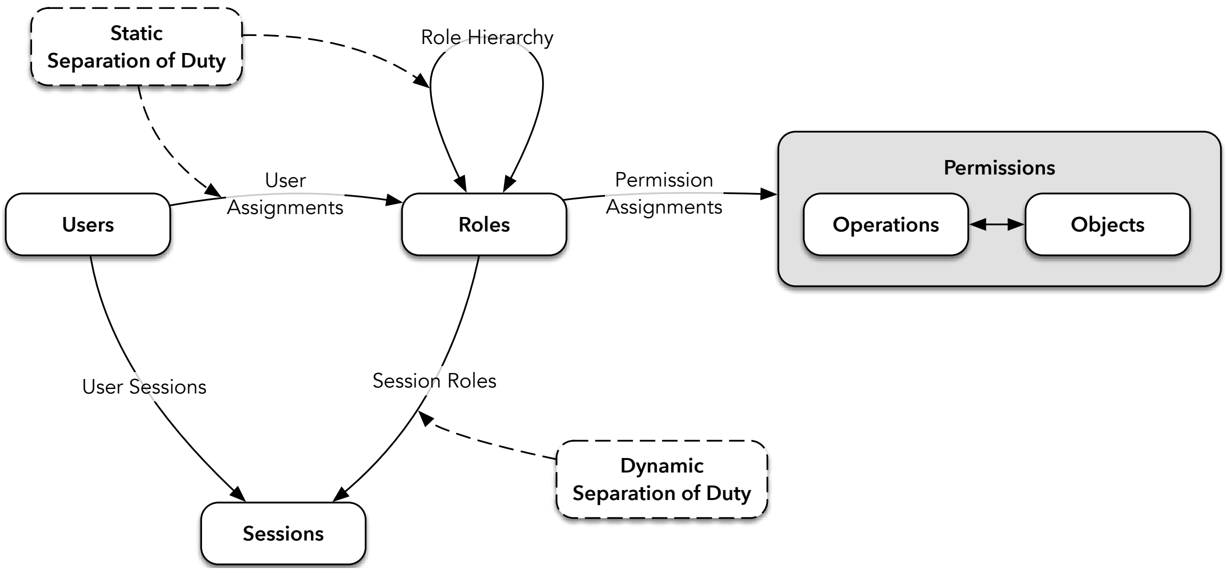


Figure 2.1: Core elements of RBAC, adopted from [3]

### Core elements of RBAC

The RBAC scheme basically consists of the modules *Core RBAC*, *Hierarchical RBAC* as well as *Constrained RBAC*. While Core RBAC comprises the basic mandatory elements of an RBAC implementation, the latter components add optional functionalities. Figure 2.1 depicts the core elements of all three modules and their relations amongst themselves. It contains the elements of Core RBAC (“Users”, “Roles”, “Sessions”, “Operations” and “Objects”), Hierarchical RBAC (“Role Hierarchy”) and Constrained RBAC (“Static Separation of Duty”, “Dynamic Separation of Duty”). The purpose of the individual elements and their relations is described in the following paragraphs.

**Core RBAC** contains the well-known basic elements inherent to all access control models [17]. These are *users*, *objects* and *operations* that users can execute on objects. But in contrast to the models described in the preceding sections, users are not directly granted permissions. Instead, the concept of *roles* is introduced. Users can get assigned multiple roles based on their job, position or responsibilities within a company using RBAC in its information system. Conversely, roles may be assigned to an arbitrary number of users. For example, all employees of the accounting department of a company could be assigned each the “global employee” role as well as the “accountant” role. Consequently, all defined roles are equipped with the permissions (i.e. allowed operations on objects) necessary for a member to successfully carry the role out. Here, too, multiple assignments of a single permission to roles are allowed. Roles serve as an intermediary, concrete user permissions are derived on each authorization request. Continuing the example mentioned earlier, let us assume there are two permissions “read canteen menu” and “transfer salary”. One would assign the “global employee” role only the former permission, whereas the “accountant” role will likely be also granted the latter. Based on the chosen example it is obvious, that RBAC permissions do not necessarily describe only technical access to resources (e.g. read, write, execute resource XY) but rather define higher level use cases coming up within an enterprise-level information system.

Furthermore, the concept of *sessions* is contained in the feature set of Core RBAC [42]. Sessions can be used to reduce the amount of privileges a user needs to carry out a certain task or process of tasks. Therefore the concept of *least privilege* is embedded to RBAC. Least privilege states that a user should be granted as little permissions as possible and as much as necessary to fulfil a task [42]. That is, during a session only a subset of the users roles get activated. A user might simultaneously use multiple sessions with different roles activated within.

**Hierarchical RBAC** refers to the possibility of implementing permission inheritance by basing roles upon each others [17]. As a result, organizational hierarchies within a company and the respective permissions can be mapped to a role hierarchy, where senior roles are assigned more privileges than junior roles. Considering the previously mentioned example, the “accountant” (senior) role is a classic example for the applicability of an inheritance relation to the “global employee” role (junior) [17]. Accountants are not directly granted the “read canteen menu” permission but rather inherit it from the “global employee” role. To continue the example, a new role “head of accounting” might be imaginable, which inherits all permissions from “accountant” and, as it is a transitive relation, also all permissions from “global employee”. Additionally, “head of accounting” could be assigned a new permission “hire employee”, which concretises the higher competences of this role. Back to the formal definition, the set of the assigned permissions per role is a superset of the set of assigned permissions of the role it is derived from.

**Constrained RBAC** introduces the ability to implement simple rules that might restrict the assignment of roles to users. As a consequence, the *Separation of Duty* (SOD) concept is addressed. According to [17], the American National Standards Institute (ANSI) defines the term as follows:

> “Dividing responsibility for sensitive information so that no individual acting alone can compromise the security of the data processing system.”

That means in other words, that no user should be granted a set of permissions simultaneously that would enable her to execute fraudulent actions due to an unnecessary surplus of competency. In constrained RBAC, SOD can be implemented both statically (*Static Separation of Duty - SSD*) and dynamically (*Dynamic Separation of Duty - DSD*) [17]. SSD is defined at design time of the user/role assignments and is valid at any time from its definition. For example, an employee of the accounting department can never assume the roles “release salary transfer” and “transfer salary” at the same time. Considering rather small companies, defining such a constraint statically might be a too strong condition and not practical, as the company could lack on a sufficient amount of personnel. In order to prevent threatening the quick execution of all tasks of the payment process due to possible illness or vacation, DSD introduces the possibility of constraints within a single session. Members of the accounting department then could be assigned both above described roles, but could only activate one of them within a single session, i.e. although a user can generally be assigned multiple roles, only a subset of all assigned roles can be held at a time.

### RBAC administration

The ability to control and maintain access to a whole information system by a central authority was a prime motivation towards the implementation of a new access control model [17]. In addition, straightforward auditing and reporting of currently assigned permissions is one of the major advantages RBAC involves, especially over the discretionary approach of DAC [17].

When RBAC was first introduced it was designed for use in systems of manageable size, e.g. company-wide systems [17]. Role administration was meant to be done by security and system administrators [17]. However, unexpectedly increasing user numbers lead to high administration efforts [17]. On one hand, a single administrator or even teams of administrators have to ensure that new users are assigned their allowed roles as fast as possible in order to prevent slowing down business. On the other hand, growing systems covering more and more different kinds of functionalities require the extension of the existing RBAC configuration [17]. Role engineering of strategic vision is necessary in order to prevent role explosion and chaotic implementations [23].

Initial versions of RBAC did not include a standardized model of how to administrate RBAC [17]. Later, an approach called *Administrative RBAC* (ARBAC) [50] was introduced, extending the core RBAC systems by elements enabling the management of permissions of administrative users for changing roles, users, permissions and the relations amongst them. As research made progress, more sophisticated approaches were introduced, e.g. trying to derive user-role assignments automatically based on attributes of the new user [2]. This approach is called *Rule-Based RBAC*, as the automatic role-assignments depend on rules defined in advance by an administration authority. For this purpose, users are provided with a certain set of attributes describing them. Based on this attributes, as well as on given role constraints, roles are dynamically assigned or revoked from users.

### RBAC extensions

As aforementioned, the basic version of RBAC did not include any administration concepts, as well as it was not able to consider contextual information when deriving access decisions [17]. Research focused partly on extending RBAC by functionalities needed in practice.

As an example, *GEO-RBAC* [4] was introduced to help RBAC systems gain awareness of spatial conditions. Motivation behind this are use cases, where access decisions depend on the current location of the requestor, e.g. a certain highly-protected document might only be viewed if the user demanding access to it currently resides in a secured room, in order to prevent information being stolen by unauthorized persons around. For this purpose, roles get attached a geographic position or boundary. The result is referred to as *spatial role*. Making use of Core RBACs session system, roles get only activated within a session if the current position provided by the requesting user complies with the defined geographic position of the role. Obviously, using spatial roles is not mandatory, as not all access decisions depend on geographic constraints.

Similar to GEO-RBAC, *Generalized Temporal RBAC* (GTRBAC) [25] was designed to enable access control based on temporal conditions. These are implemented via *temporal constraints* that may be defined upon roles as well as on user-role and role-permission assignments. Examples of constraints and the covered use cases, respectively, are:

- Periodicity constraints - activation of RBAC elements during a certain time interval
- Duration constraints - activation for a maximum period of time at once

A rather special extension of RBAC is proposed by Byun and Li [5], implementing "purpose based access control [..] in relational database systems". Utilizing role attributes, a subset of users of the role can be derived for which, provided that the stated access purpose within the attribute matches all conditions, access is granted to the desired database objects. However, this requires the corresponding database objects to be labelled with a set of purposes the use is allowed for.

### 2.1.5 Attribute-Based Access Control

So far, all presented access control models relied on identifying information regarding both requestor and resource for deriving access control decisions, e.g. a user asking for access to a certain information has to authenticate using his credentials, followed by a lookup in the ACL attached to the requested information. These models may be subsumed under the term *Identity-Based Access Control* (IBAC). A drawback of this approach is, that subjects and objects involved in access control have to be known in advance or at least some configuration effort has to be taken when they first appear in the information system.

In contrast, Attribute-Based Access Control (ABAC) focuses on attributes describing the current context of an authorization request rather than depending on identity. Based on these contextual conditions and a set of predefined business rules an authorization decision is derived. Thus, ABAC is also referred to as a *Context-Based Access Control*

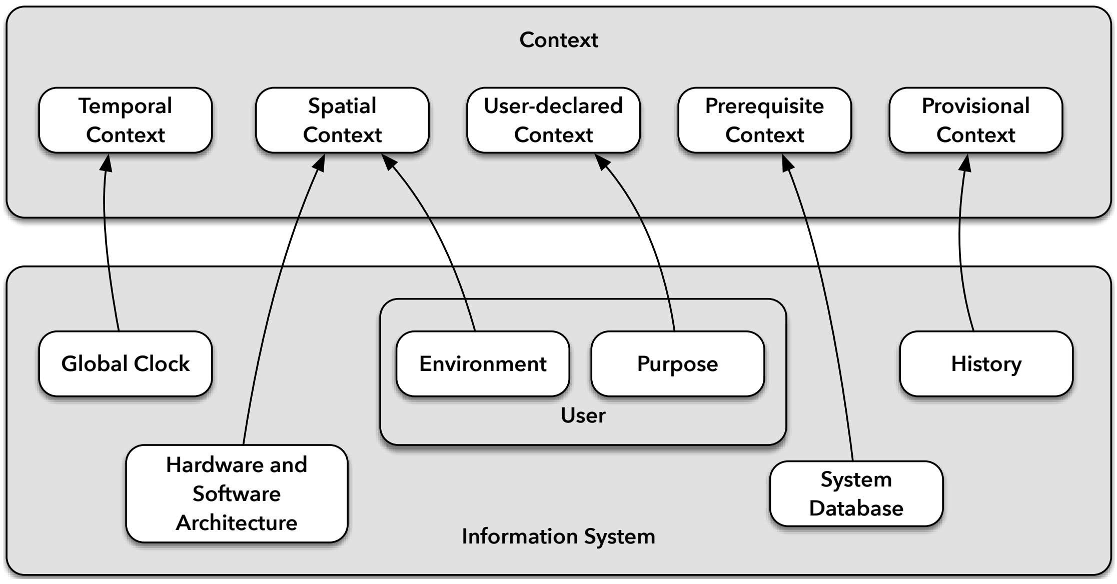


Figure 2.2: OrBAC contexts, adopted from [8]

model [20]. Other synonyms used in research literature are *Policy-Based Access Control* [40] and *Rule-Based Access Control* [44]. Although definitions of all these terms may vary, they are all based on the concept of ABAC and are therefore summarized under this term within the scope of this work.

### Early approaches towards ABAC

Access control models solely based on IBAC turned out to lack on expressiveness concerning contextual conditions possibly affecting authorization decisions [8]. Hence, research started to move towards ABAC gradually by extending existing models with single concrete contextual capabilities. As already mentioned in Section 2.1.4, focus was on developing extensions for the well-known RBAC model, starting with temporal and location-based attributes attached to core elements of RBAC. Cuppens and Cuppens-Boulahia went even further and defined a whole framework of contextual considerations applicable within a single security policy named *Organization-Based Access Control* (OrBAC) [8]. The listing of supported contexts is depicted in Figure 2.2 and gives a good overall view on aspects with possible relevance to access control systems as well as the sources the information will be taken from. *Temporal and spatial contexts* refer to time-based and location-based constraints [8]. An example for temporal limitation is that access to a certain resource is only granted during office hours. Spatial contexts can describe both physical and logical locations. Concrete physical locations might be an office room with special security precautions, while logical locations could refer to a concrete local area network the request is sent from. *User-declared contexts* provide a means of stating for which purpose the authorization is needed, consequently this information has to be provided by the requestor. Based on concrete permissions, a user might or might not be allowed to gain access to a resource for the stated purpose. *Prerequisite contexts* enable to provide general rules that must be fulfilled for deriving a positive authorization decision. Most likely the data necessary for evaluating these rules will come from a global database containing all the application data. Finally, *provisional contexts* are used to express obligations that can either involve actions that must be undertaken before or after the requested authorization can be granted.

### Core ABAC in Detail

Concerning the basic entities, ABAC uses similar terms as the access control models introduced above. *Subjects* (e.g. users) represent active entities of the system requesting access to *objects* (e.g. resources of any kind: files, services, small chunks of information, etc.) [23, 49]. In addition, the *environment* is considered an entity with relevance to authorization decisions. The most-important elements of ABAC are called *attributes*, which can be attached to all of these entities. Crucial prerequisite to all types of attributes is, that they are well-defined in terms of possible expressions and hence easy to evaluate and compare against.

*Subject attributes* are used to describe the user asking for access to a system. Examples include name, department within a company or the job title. As literally every information can be attached, subject attributes are capable of expressing all the necessary information used for enforcing the previously introduced access control models. ABAC could be mapped to DAC using identifying attributes. MAC could be achieved using a “clearance” subject attribute, whereas RBAC could be implemented by using role names as attributes.

*Object attributes* are used to attach information to resources necessary for access control. For example, files can be provided with title, author or creation date attributes. Again, well-known concepts such as security classifications or compartments could be used to provide enough information for meaningful access control decisions.

*Environment attributes* contain descriptions on environmental conditions, that can neither be attached to subjects nor to objects. Examples could be current time, location-based information or current threat-levels within or around the system under protection. These attributes will also be considered during evaluation time of an authorization request.

There is no recommendation or obligation regarding the location of attribute storage. They might either be attached to the relevant entities directly, or get centrally managed using an attribute store, for example using a database.

### ABAC policies

Concrete access control policies are implemented by defining rules for accessing resources based on the above-mentioned attributes [23]. These rules are formulated using a set of Boolean expressions subsequently leading to a Boolean result reflecting the access decision. A comprehensive access control policy should contain rules for every object to be secured by the access control system. A simple example for a rule restricting access to the turnover report of a company could be defined as follows:

```
attr( env, CURRENT_TIME ) between 8 and 17      and
attr( subject, DEPARTMENT ) = 'Sales'           and
attr( subject, CLEARANCE ) >= 'confidential'    and
attr( subject, YEARS_IN_COMPANY ) > 5
```

Let *attr* denote a relation taking an entity and the name of an attribute, returning the concrete attribute value attached to the entity. Provided that *subject* and *env* contain the concrete requestor and the object describing the current environmental conditions, the rule would read as follows:

> *"Grant access during office hours only if the requestor works in the sales department and earned a clearance level of at least 'confidential' and has been working in the company for more than five years."*

As attributes are free to be defined, ABAC policies are a very expressive mechanism for enforcing access control. In fact, the expressiveness is only limited by the capabilities of the programming language chosen for implementation. Furthermore, existing policies allow for the addition of new users without any further effort regarding access control. That is, access control can even be defined for unanticipated users.

### ABAC architecture

ABAC comes with a proposed system architecture, providing all the modules necessary for deploying access control at enterprise level. Figure 2.3 depicts the elements comprising the system. Mandatory elements are a *Policy Enforcement Point* (PEP) and a *Policy Decision Point* (PDP).

The PEP acts as an intermediary between the subject and the resource. In large IT environments several PEPs may be deployed across the system. Each access request has to be intercepted, prepared for and subsequently sent to the PDP. The PEP thereupon waits for a response which can either contain a positive or a negative access decision. Based on this decision, the request is either redirected to the resource or rejected resulting in an error message delivered to the requestor.

The PDP marks a central module with responsibility of deciding whether an access request should be granted or rejected. As mentioned before, the PDP is contacted by a PEP containing information on the subject, resource and type of access requested. After being entrusted with such a request, it gathers all the necessary information to come to an authorization decision. First of all, applicable policies concerning the requested resource are retrieved from a central *policy storage*. Next, to be able to evaluate the rules gathered in this way, all the necessary attributes are retrieved from the *Policy*

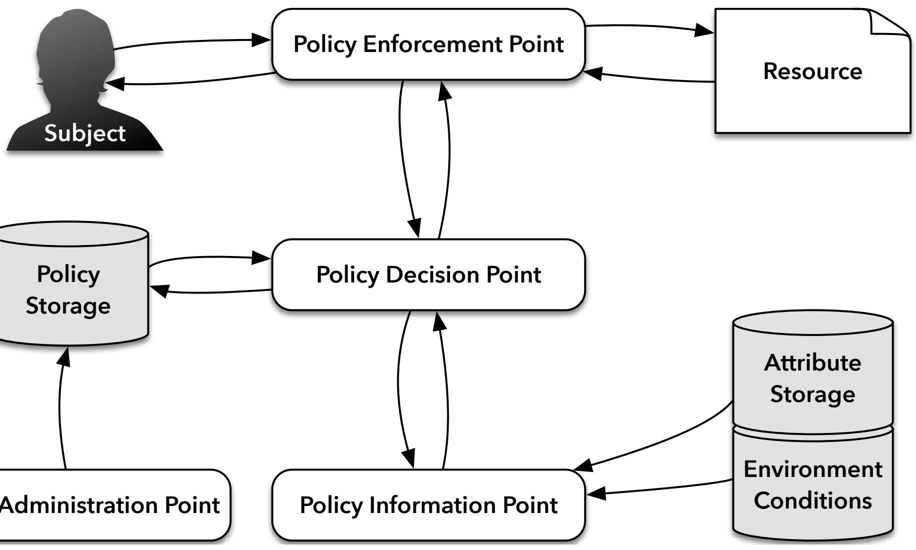


Figure 2.3: ABAC architecture, adopted from [23]

*Information Point* (PIP). If multiple rules or readily derived access decisions lead to a conflicting state, the PDP has to resolve the conflicts to come to an unambiguous decision. Finally, the result is transferred back to the PEP.

Key responsibility of the PIP is the retrieval of all the data, respectively attribute values, necessary for the PDP to evaluate business rules. Data sources can be various, attribute values might be stored in relational databases or key value stores, whereas environmental conditions could even be derived on-the-fly.

Finally, a *Policy Administration Point* (PAP) has to be deployed. This module serves for managing all the security policies containing the business rules and therefore addresses an important tool for security officers and system administrators. Thus, the PAP provides means for analysing, adding, changing and deleting security rules and policies in the policy repository. In addition, development-related capabilities are inherent to any PAP, such as testing or debugging the defined policies.

From a more organizational point of view, so-called *Attribute Authorities* (AA) are responsible for well-defined attribute specifications [23]. It is possible to set up one ore more AAs within one system. Usually they are divided into Subject AAs, Object AAs and Environment AAs. However, authority can also be distributed among business responsibilities, e.g. letting HR define all the HR-related attributes. For means of inter-organizational security-enforced communication common attribute definitions have to be agreed on, or at least automatic mappings between differing definitions should be possible.

To provide a common way of implementing ABAC, a de facto standard called *eXtensible Access Control Markup Language* (XACML) was adopted by the OASIS standards organization[1]. We will discuss this standard in detail in Section 2.4.

## 2.2 Foundations of Fine-Grained Data Access Control

Evolving database technology has enabled applications to store and process practically uncountable amounts of person-related data [5]. The implementation of access control mechanisms aims at protecting sensitive data from being accessed by unauthorized users or user groups. Usually, RDBMS are utilized for storing all the application data in tabular form. Built-in access control in most RDBMS is traditionally implemented only at relation level, that is, access can only be granted for tables or views as a whole [22]. This level of access control is often referred to as *coarse-grained.*

Due to the lack of support of more fine-grained authorization mechanisms at database level, solution developers were forced to implement access control at application level [51, 41], which brings up several shortcomings, which are discussed in the remainder of this paragraph. Obviously, access control enforcement has to be woven into each of the user interfaces providing access to information under protection. On one hand, that leads to high effort, as many different application modules have to be implemented and maintained. For example, changes to the applications access control policy have to be propagated to all code locations implementing access control enforcement. In addition, systems provide a higher risk of being vulnerable to data breach attempts, the more data access points they provide. In other words, bigger applications involve a bigger risk of exposing interfaces which are poorly implemented in terms of data access control. Even worse, when application code is responsible for enforcing access control, it is necessary to provide access to the database using a fully-privileged user capable of reading or even modifying all of the sensitive data [41]. Consequently, a single point of security policy enforcement is necessary. It is therefore recommendable letting the database take care of this step.

Conventional RDBMS, such as Oracle Database, do not come with built-in assistance for FGDAC enforcement [29]. Thus research focused on proposing approaches to overcome the lack of support, leading to a variety of possible solutions all of them coming with different strengths and weaknesses [51, 43, 41, 19, 6, 1, 44]. These solutions can be roughly grouped into *static views*, *parameterized views*, *extension of the SQL standard* and *query rewriting*, which are described in detail and illustrated using the *EMPLOYEES* example in the following sections.

### 2.2.1 Problem description

As an introductory example, consider the definition of a common *EMPLOYEE* relation within a database (see Table 2.1). According to the possibilities of coarse-grained access control, users of the surrounding system could only be granted access to the table as a whole or could get no access at all, which is very restrictive, though. In the following we

[1] `https://www.oasis-open.org/`

Table 2.1: DB table *EMPLOYEE*

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
| 1 | Jane | Doe | Sales | Head Of Sales | 4200 |
| 2 | Max | Power | Sales | Sales Clerk | 1800 |
| 3 | Frank | Wright | Sales | Sales Clerk | 2100 |
| 4 | John | Hancock | Accounting | Head Of Accounting | 4500 |
| 5 | Sandra | Brown | Accounting | Accountant | 2200 |
| 6 | Linda | Roberts | IT | Developer | 2400 |

Table 2.2: DB table *EMPLOYEE* viewed by Max Power

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
|  | Jane | Doe | Sales | Head Of Sales |  |
|  | *Max* | *Power* | *Sales* | *Sales Clerk* | *1800* |
|  | Frank | Wright | Sales | Sales Clerk |  |

Table 2.3: DB table *EMPLOYEE* viewed by John Hancock

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
|  | *John* | *Hancock* | *Accounting* | *Head Of Accounting* | *4500* |
|  | Sandra | Brown | Accounting | Accountant | 2200 |

Table 2.4: DB table *EMPLOYEE* viewed by Linda Roberts

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
| 1 | Jane | Doe | Sales | Head Of Sales |  |
| 2 | Max | Power | Sales | Sales Clerk |  |
| 3 | Frank | Wright | Sales | Sales Clerk |  |
| 4 | John | Hancock | Accounting | Head Of Accounting |  |
| 5 | Sandra | Brown | Accounting | Accountant |  |
| *6* | *Linda* | *Roberts* | *IT* | *Developer* | *2400* |

provide a list of possible real-world access control constraints, that might be necessary to preserve privacy, if all users (equal to all records in *EMPLOYEES*) are allowed to access at least parts of the table:

(C1) Users are only allowed to view information on employees of their own department, except for members of the IT department, which may view the records of all employees company-wide.

(C2) Users are only allowed to view their own salary, except for department heads, which may view the salary of all employees of their department.

(C3) Users are not allowed to view their ID, except for members of the IT department, which may view the ID of all employees company-wide.

For implementing the above-mentioned constraints, FGDAC is necessary. FGDAC refers to managing authorizations each on row, column and even cell level. Constraint (C1) needs row-level access control, as some rows (in particular: rows of employees belonging to another department) have to be filtered. Constraint (C3) needs column-level access control, as the ID column has to be omitted for all non-IT employees. Finally, constraint (C2) needs cell-level access control, as both filtering and projection are necessary to ensure proper compliance. The effects of applying the constraints to a selection of all authorized data from the viewpoint of different employees are depicted in Tables 2.2, 2.3 and 2.4.

### 2.2.2 Static views

Primitive approaches utilized statically defined database views for preparing data in different granularity levels [1]. These views were either designed to meet the permissions of one single user, or if possible, at least of a whole group of users. Obviously this approach is hardly scalable, as the number of views needed is directly related to the number of system users. Additionally, the selection of which view to present to which user has to be implemented in application code, which just as much leads to high maintenance effort. According to the *EMPLOYEES* relation introduced above, a static authorization view for user *Max Power* could be implemented as depicted in Listing 2.1. Notice that the results of querying the view correspond to those presented in Table 2.2.

Listing 2.1: View on *EMPLOYEES* for Max Power

```
create view EMPLOYEES_MaxPower as
select -- Max Power is not allowed to see IDs
       null           as ID,
       FIRSTNAME      as FIRSTNAME,
       LASTNAME       as LASTNAME,
       DEPT           as DEPT,
       POSITION       as POSITION,
       -- Max Power may only see his own salary
       case
         when ID = 2
         then SAL
       end            as SAL
  from EMPLOYEES
 -- Max Power may see all employees of his own department
 where DEPT = ( select DEPT
                  from EMPLOYEES
                 where ID = 2 );
```

For achieving comprehensive access control, (1) each of the defined employees must be provided with its own view and (2) no direct access to the *EMPLOYEES* table may be implemented at application level [1]. Especially (2) lead to serious security problems, if users of malicious intent manage to get access to the underlying tables containing all the unfiltered raw data.

### 2.2.3 Parameterized views

To address the issue of having to manage countless different static views and all the places in code querying them, *parameterized views* were proposed [41]. Although Rizvi et al. used the notion of parameterized views in their model, they did not propose a technical solution. Views should be able to be equipped with abstract parameters at design time, which are used within the SQL statement to refine the results. When requesting query results, the concrete parameter values have to be provided. Although there has been no proposal for such a concept to the SQL standard so far, a sample implementation considering the *EMPLOYEES* example could look as depicted in Listing 2.2. Note that the presented code snippet will not work in any existing RDBMS as it is supposed that views can be equipped with parameters of the form ( PARAM_NAME in data_type [, ...] ) in the header, where PARAM_NAME can be used in the view query.

Listing 2.2: Parameterized View on *EMPLOYEES*

```
create view EMPLOYEES_ByUserID( USER_ID in number ) as
select -- only members of IT may view IDs
       case
         when 'IT' = ( select DEPT
                         from EMPLOYEES
                        where ID = USER_ID )
         then ID
       end              as ID,
       FIRSTNAME        as FIRSTNAME,
       LASTNAME         as LASTNAME,
       DEPT             as DEPT,
       POSITION         as POSITION,
       case
         -- users may only view their own salary
         when ID = USER_ID
         then SAL

         -- except for department heads, which may view
         -- the salary of all employees of their department
         when ( DEPT, 'Head of' ) = ( select DEPT,
                                             left( POSITION, 7 )
                                        from EMPLOYEES
                                       where ID = USER_ID )
         then SAL
       end              as SAL
  from EMPLOYEES
 -- users may only see employees of their own department
 where DEPT = ( select DEPT
                    from EMPLOYEES
                   where ID = USER_ID )
 -- except for members of IT
    or 'IT' = ( select DEPT
                    from EMPLOYEES
                   where ID = USER_ID );
```

**Table functions** are a means of implementing parameterized views embedded into some popular RDBMS. Oracle provides the concept of *pipelined table functions* [21], which are capable of returning collection results that in turn can be used in the SQL context similar to querying views or tables. Similar to traditional stored functions, pipelined table functions can be provided with input parameters. The interface of the resulting collection can optionally be predefined using user-defined types. Finally, the tabular information has to be "piped" row by row to the calling unit. Utilizing the *EMPLOYEES* example, Listing 2.2 shows (1) the definition of the interface of the resulting collection, (2) the implementation of the pipelined table function and finally (3) how to query the function from within a traditional SQL statement.

Listing 2.3: Oracle pipelined function on *EMPLOYEES*

```
-- (1) define interface of resulting collection
create type EMPLOYEE_OT as object (
  ID          number,
  FIRSTNAME   varchar2(256),
  LASTNAME    varchar2(256),
  DEPT        varchar2(256),
  POSITION    varchar2(256),
  SAL         number
);
create type EMPLOYEES_TT as table of EMPLOYEE_OT;


-- (2) implement pipelined table function
create function EMPLOYEES_ByUserID( USER_ID in number )
return EMPLOYEES_TT pipelined
is
begin

  for rec in ( /* use same query as in Listing 2.2 */ ) loop
    pipe row (
      EMPLOYEE_OT (
        rec.ID,
        rec.FIRSTNAME,
        rec.LASTNAME,
        rec.DEPT,
        rec.POSITION,
        rec.SAL ) );
  end loop;

end EMPLOYEES_ByUserID;


-- (3) use pipelined function:
--     query EMPLOYEES for user Max Power (ID = 2)
select *
  from table( EMPLOYEES_ByUserID( 2 ) );
```

Other RDBMS provide similar functionalities, including *table functions* in PostgreSQL [13] and *table-valued user-defined functions* in MS SQL Server [38].

### 2.2.4 Extension of the SQL language

As mentioned in Section 2.2.3, some of the proposed approaches rely on extending the SQL language by constructs supporting FGDAC enforcement. As an example, in [1], Agrawal et al. introduce the notion of *restrictions*. These can be considered FGDAC-enabled versions of the existing more coarse-grained *grant* statements. While traditional grant statements can only be used to define authorizations on relation level (e.g. tables or views), restrictions provide syntactical possibilities for defining each row-level, column-level and even cell-level authorizations. The straight-forward syntax of the proposed language constructs is depicted in Listing 2.4.

Listing 2.4: Syntax of *restriction* command, adopted from [1]

```
create restriction restrictionName
on tableX
for authName1 [ except authName2 ]
( (   ( to columns columnNameList )
    | ( to rows [ where searchCondition ] )
    | ( to cells ( columnNameList
                   [ where searchCondition ] )+ )
  )
  [ for purpose purposeList ]
  [ for recipient recipientList ]
)+
commandRestriction;
```

A restriction may be applied to a combination of table (*tableX*) and authorized subjects (*authName1*), which can be concrete users or roles. Optionally, *authName2* can be used to define an exceptional subject, to which the restriction shall not be applied. Below, the concrete data restrictions may be defined. Column-level restrictions have to be provided with a *columnNameList*. Row-level restrictions are implemented via a custom where clause to be specified. Consequently, cell-level restrictions may be defined using a *columnNameList* and again a where clause, which may contain a correlated sub query accessing data from other cells of the row. Additionally, allowed purposes (*purposeList*) for querying the data as well as concrete recipients (*recipientList*) may be defined. Finally, *commandRestriction* completes the statement by stating to whether access type the restriction has to be applied, e.g. *select*, *delete*, *insert* or *update*.

Applied to our *EMPLOYEES* example, the restriction describing all the authorizations for user Max Power could look as depicted in Listing 2.5.

Listing 2.5: Example restriction on *EMPLOYEES* for user Max Power

```
create restriction R_EMP_MAX_POWER
on EMPLOYEES
for user     MAXPOWER
to columns FIRSTNAME,
             LASTNAME,
             DEPT,
             POSITION
to rows      where DEPT = ( select DEPT
                              from EMPLOYEES
                             where ID = 2 )
to cells     SAL where ID = 2
restricting access to select;
```

### 2.2.5 Query rewriting

Agrawal et al. further define how the enforcement of the restrictions defined using above new language constructs can be implemented. By generating an ad-hoc view on the underlying database table, all restrictions are implemented. To be more exact, a sample algorithm is provided that applies all the where clauses, column projections and hiding of cell values resulting in a concrete SQL query to be executed instead of the originally requested query. This process is referred to as *query rewriting* and has been proposed multiple times in the research area of FGDAC. Figure 2.4 depicts the architectural basics necessary for enforcement using query rewriting, considering *Q* as the original query issued by the requestor and *Q'* as the translated query rewritten according to the applied access control policies retrieved from a generally available policy store.

In [43], Shi et al. introduce their model of cell-level FGDAC enforcement including a practical approach using query rewriting, based on the previous work of LeFevre et al. [26]. Core elements are so-called *restricted objects*, which comprise a relation (i.e. a database table) and a policy function, applying filters to all attributes of the relation. Filters can be used either to explicitly allow (*"AllowedFilter"*) or deny (*"ProhibitedFilter"*) access. That is, both open and closed access control policies are supported. In case of conflicting filters (i.e. access is defined to be both allowed and denied), prohibitions are promoted to ensure security. The proposed concrete implementation of the query rewriting algorithm makes use of the *CASE* condition inherent to SQL. In an ad-hoc view, each attribute of a restricted object is only returned to the requestor, if there applies at least one "AllowedFilter" and no "ProhibitedFilter", otherwise *NULL* is returned. Both Shi et al. and LeFevre et al. focus strongly on cell-level FGDAC, putting row-level FGDAC aside.

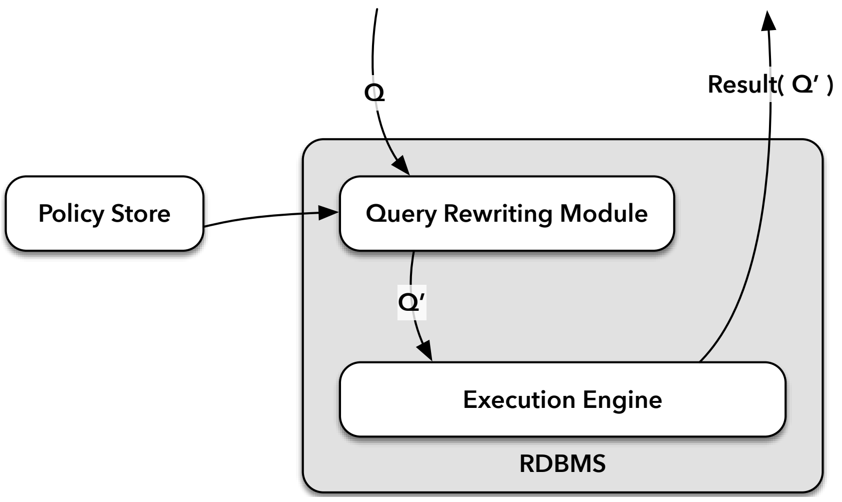


Figure 2.4: Concept of query rewriting, adopted from [7]

At least the latter work states that rows consisting of *NULL* values only after applying all cell-level filters can be omitted as a whole.

To give a practical example of how cell-level FGDAC can be implemented using query rewriting, suppose that every column C used within a *SELECT* query is replaced by a *CASE* condition of the following form:

```
case
  when AccessAllowed( 'C', context ) then C
                                      else null
end as C
```

It is supposed that *AccessAllowed( columnName, context )* denotes a function holding awareness of all information necessary to derive an access decision, taking the name of the queried column and contextual information only available at the time of query execution as input and returning either *TRUE* or *FALSE* for positive and negative decisions, respectively. The implementation applies all the access control policies - it could thus access additional contextual information such as the user currently logged in or even query other database tables for retrieving identity based or attribute based authorization information. Note that, for more complex systems, the column name alone is not sufficient for identifying the object an access decision is needed for. That is, it will likely be necessary to provide the table name or even the name of the database scheme for a unique identification. Moreover, the interface of the *AccessAllowed()* function has to be defined in a way, that all information that will ever be needed to hand over to the decision algorithm, can be provided as an input parameter. Application-specific design is therefore crucial and inevitable, unless more dynamic query rewriting approaches are investigated or the decision algorithm is woven directly into the SQL query without the detour via functional encapsulation.

**VPD [39]** is a well-known implementation of FGDAC using query rewriting in one of the most commonly used commercial RDBMS, *Oracle Database 12c*[2]. It is based on *PL/SQL*, the proprietary procedural programming language inherent to the database system. The core elements of FGDAC enforcement using VPD are called *policies*, which at least consist of a unique policy name, the DB object needing protection (specified by its table or view name) and a user-defined function written in PL/SQL returning the restrictions to apply. Basically, VPD adds all the restrictions applicable to an object to the *WHERE* clause of the SQL requesting access. The function has to provide an interface as follows: two input parameters describing both schema (object location) and name of the object, the function result has to be a string containing the restrictions. These restrictions are called *predicates* and must be formulated as valid Boolean expressions. If more than one policy function is applied to a single object, all predicates are logically conjuncted, i.e. they are put together using the *AND* operator.

VPD supports both row-level and cell-level security policies. Row-level access control is, as already mentioned above, achieved by adding all the applicable predicates to the WHERE clause of the submitted query. Predicates can also be applied to columns. By default, the whole row is removed from the result set, if at least one cell value is restricted. It is possible to explicitly specify that restricted cell values should be masked with the *NULL* value, though. If so, all the rows not restricted by any other possible *WHERE* clause are returned. If all values of a row are empty, the row is returned anyway. VPD does not natively support custom masking for restricted cell values other than using the *NULL* value, e.g. just displaying the last 3 digits of a credit card number.

Coming back to our *EMPLOYEES* example, Listing 2.6 shows how a PL/SQL function is defined for use as a VPD policy function. We assume that a context of name *ctx_emp* has already been defined, taking care of storing the employee ID of the user currently logged in. The stored ID can be retrieved via the predefined function *sys_context()* and may be used for gathering additional information necessary for building the predicate, which finally is returned back as a string. Note that the result must contain a syntactically valid Boolean expression, usable in the WHERE clause of any SQL query.

The listing contains the implementation for constraint (C1) as defined in Section 2.2.1. The remaining constraints (C2) and (C3) have to be defined according to this example implementation.

Listing 2.6: Define VPD policy functions

```
create function filter_dept
  ( schema in varchar2, owner in varchar2 )
  return varchar2
is
  dept EMPLOYEES.DEPT%type;
begin
```

[2]VPD is part of the paid version (*"Enterprise Edition"*). However, Oracle provides a free pre-installed virtual machine for testing purposes at http://www.oracle.com/technetwork/community/developer-vm/index.html

```
  -- ensure that no data is exposed by accident in case of
  -- missing context information by returning a predicate
  -- evaluating to a contradiction
  if sys_context( 'ctx_emp', 'ID' ) is null then
    return '1=0';
  end if;

  -- collect additional information necessary for
  -- defining the predicate, based on the employee ID
  -- stored in the context
  select DEPT
    into dept
    from EMPLOYEES
   where ID = sys_context( 'ctx_emp', 'ID' );

  -- return predicate string
  -- for the DEPT row-level restriction
  return '( DEPT = ''' || dept || '''' ||
         ' or ' ||
         '''IT'' = ''' || dept || ''' )';
end filter_dept;

-- [...] define two more policy functions
-- filter_sal() and filter_id() accordingly [...]
```

Listing 2.7 shows how the predefined system package *DBMS_RLS* allows to install new FGDAC policies using the *add_policy()* function. By default, filter functions are appended to the WHERE clause enforcing row-level security. If explicitly stated using the *sec_relevant_cols* and *sec_relevant_cols_opt* parameters, policy functions may be used for applying cell-level restrictions. In addition, the package contains functions for altering, dropping, enabling, disabling and grouping policies.

Listing 2.7: Add VPD policies using the sys package DBMS_RLS

```
-- use filter_dept() for row-level access control
dbms_rls.add_policy(
  object_name     => 'EMPLOYEES',
  policy_name     => 'employees_filter_dept',
  policy_function => 'filter_dept'
);

-- use filter_sal() for cell-level access control
dbms_rls.add_policy(
  object_name           => 'EMPLOYEES',
  policy_name           => 'employees_filter_sal',
  policy_function       => 'filter_sal',
  sec_relevant_cols     => 'SAL',
  sec_relevant_cols_opt => dbms_rls.ALL_ROWS
);
```

```
-- [...] add policy for filtering IDs accordingly [...]
```

Finally, Listing 2.8 shows how the actual query is built. At session initialization, the employee ID has to be set using the predefined *set_context* procedure. This could for example be implemented using a logon trigger at database level, reading the operating system user from the system environment and setting the corresponding employee ID in the context. Note that we implemented the policy function in Listing 2.6 in a restrictive way, that nothing is returned querying the *EMPLOYEES* table unless the context is initialized properly.

Listing 2.8: Set context and query table

```
-- query EMPLOYEES table without initializing context
SQL> select *
  2     from EMPLOYEES;

no rows selected

-- set employee ID for user currently logged in
-- for example, use ID of employee Max Power
SQL> begin
  2     dbms_session.set_context( 'ctx_emp', 'ID', 2 );
  3   end;
  4   /

-- again: query EMPLOYEES table without WHERE clause:
-- VPD automatically applies all policy functions
SQL> select *
  2     from EMPLOYEES;

ID FIRSTNAME LASTNAME DEPT   POSITION        SAL
-- --------- -------- ------ --------------- ------
    Jane      Doe      Sales  Head of Sales
    Max       Power    Sales  Sales Clerk     1800
    Frank     Wright   Sales  Sales Clerk
```

If full data access to a restricted object is necessary for administration or development purposes, one can either connect as the predefined, fully-equipped *SYS* database user or grant the EXEMPT ACCESS POLICY privilege to any trusted administration user, to bypass FGDAC enforcement. Obviously, granting of this privilege should be handled with care.

## 2.3 Dynamic Fine-Grained Data Access Control: Introducing the Secure SQL Server

All solutions discussed in Section 2.2 are dependent on knowing each the database structure and the policies to apply in advance. However, there exist application scenarios, where both of these prerequisites can not be fulfilled.

Such is the case in the emerging research field on SNGB [35], which aims at creating a system of structured, machine-readable law. SNBG involves democratic creation, enactment and modification of machine-readable access control policies to system-relevant data during runtime, as well as democratic creation and modification of the underlying database structure - likewise, during runtime. Accordingly, database access policies in SNBG can neither be defined nor abstracted (e.g. in form of templates) at design-time. In SNBG, the absence of administrative super users, who could corrupt the democratically created policies when implementing them in the system, is of crucial importance.

There is a prototype implementation, SecSQL, providing the above-mentioned functionalities. In the following, SecSQL's features of interest with respect to the scope of this work, will be discussed.

### 2.3.1 Functional requirements

> "The challenge we are facing ... is how to appropriately design the technical infrastructure that would make it possible to store and retrieve such data without human moderation, while maintaining the possibility to moderate at design-time unpredictable read- and write-access requests to this data through advanced, at design-time unpredictable rules." [35]

#### Electronic registries

The core of SecSQL consists of electronic registries, containing information that can be roughly categorized into 3 types. First, business data needs to be stored. In the concrete problem domain, the data describes everything related to public governments, including, for example, citizen's data, law and bureaucratic work flows. Second, rules need to be maintained that govern the way business data can be accessed, i.e. selected, inserted, updated or deleted. Finally, the technical structure, that stores both business and governance data, needs to be maintained.

#### Electronic Legal Acts

Rules are called *Electronic Legal Acts* (ELA). Incoming requests of any type are parsed and checked for the applicability of one ore more ELAs. Similar to the concept of ABAC (see Section 2.1.5), ELAs can refer both to business data and contextual information, such as the identity of the user currently logged in, or other environmental conditions.

### Collaborative decision-making

All parts of electronic registries can be created and changed only by means of *collaborative decision-making*. The intention of this restriction is to reduce or even eliminate the need for supervising administration. In addition, this process of decision-making is governed by the rules described above, too. That is, rules are self-administrative, collaborators are capable of changing the conditions of collaboration at runtime. The process of decision-making is defined as follows. First, users of the system are entitled to propose changes to any part of the electronic registry. This step is comparable to a parliamentary group handing in a new draft law. After that, all members of the decision-making body are required to declare their decision on the pending proposal, similar to a parliamentary voting. Finally, if the proposal was collaboratively accepted, action is undertaken to execute to proposed changes to the system. In the comparative real-world parliament example, the proposed draft law would get enacted.

### Fine-Grained Data Access Control

Access to both data and structure of the electronic registries needs to be governed at finest granularity. That is, even the smallest chunk of data can be restricted from being accessed, as described in more detail in Section 2.2. In addition, access control needs to evolve hand in hand with structures changing due to the process of collaborative decision-making. Hence, dFGDAC is introduced, describing that even properties and enforcement of (fine-grained) access control can be changed during runtime without intervention of a system administrator.

## 2.3.2 Approaching Dynamic Fine-Grained Data Access Control

Neither of the approaches to enforce FGDAC discussed in Section 2.2 are feasible for use in the SecSQL application. Parameterized views, SQL language extension as proposed in [6] and query rewriting as suggested in [26] are, each on their own, not capable of providing the dynamism and sustainability necessary [35].

As a result, the concept of *cascading projections* is proposed. Figure 2.5 depicts a schematic illustration of the concept. It involves the iterative application of access control restrictions to an incoming request, until only the data survives, the requesting user is allowed to access. The restrictions applied can either emerge from current context dependencies (i.e. the requesting user, current day of time, etc.) or the queried data itself. The resulting set of data is called *ad-hoc virtual view.*

Listing 2.9: Example query masked using cascading projection

```
select *
  from ( select *
           from (
                  /* original query begin */
                  select id,
                         sal
```

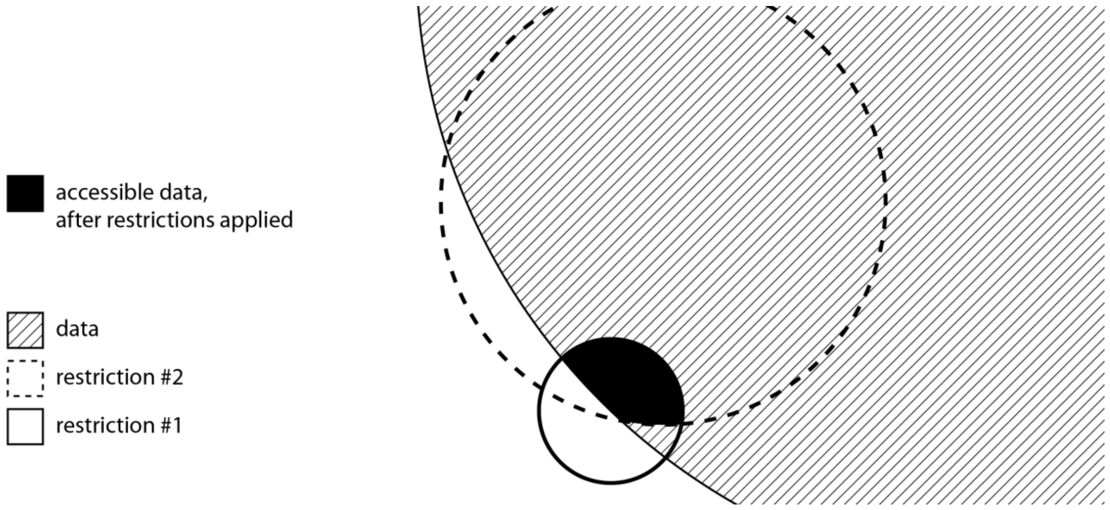


Figure 2.5: Cascading projections, schematic illustration from [35]

```
                      dept
                 from employees
               /* original query end */ )

          where /* restriction #2 */ )
 where /* restriction #1 */
```

Practically speaking, given SQL queries are sub-queried and attached with WHERE clauses without changing the structure of the result set. Finally, the original query is executed on the prepared ad-hoc virtual view returning only the permitted data. Following this approach, the dFGDAC system is capable of dynamically attaching an unlimited extensible list of restrictions to each query. With small technical adaptions, the approach works for all types of SQL statements, including DML statements (INSERT, UPDATE, DELETE).

Listing 2.9 shows an example application of two restrictions as cascading projections to a query attempt on a table containing employee's data. Although two different filter clauses are applied, the structure of the result set expected by the requestor remains unchanged.

### 2.3.3 Implementation of the Secure SQL Server

In the following sections, the current reference implementation of the SecSQL Server is described, concerning architecture, information flow and additionally interesting implementation details.

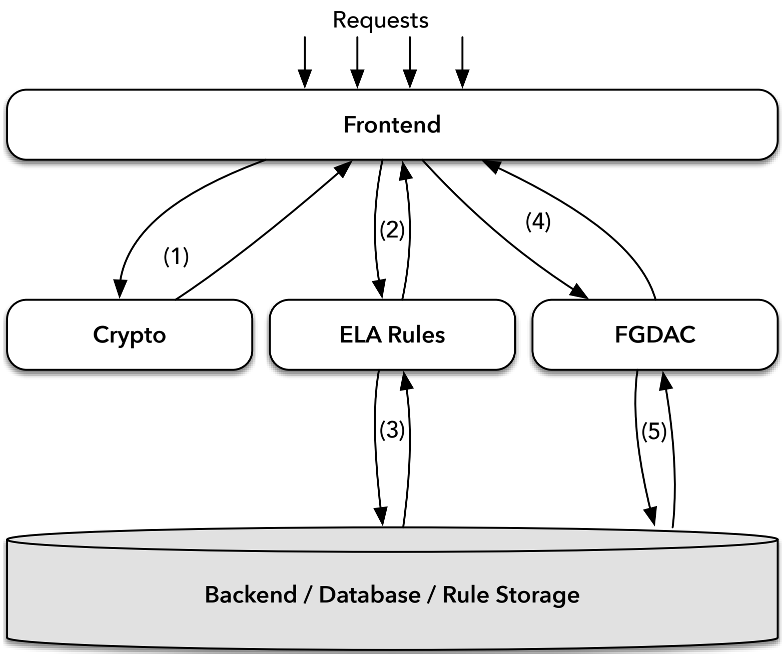


Figure 2.6: SecSQL Server modules and information flow, adopted from [35]

### Overview of architecture and information flow

The SecSQL Server is logically divided into five main modules. Figure 2.6 gives an overview of the architecture and the internal process and information flows.

The *front end module* listens for incoming requests and coordinates the complete work flow it takes to process the request properly. Prior to dispatching messages to all the modules, contextual information is attached that is necessary for further internal processing of the request. Example context information can be the authentication of the requestor as well as time and location the incoming message originated from.

The *crypto module* is the first internal module called by the frontend module when a new request arrives (1). It takes care of verifying the authenticity and integrity of the original message. Validation of the attached signature is undertaken. In addition, the identity of the requestor is determined and verified and attached to the context of the request for further use in later steps of the processing.

The *ELA rules engine* is called by the frontend module after successful processing of the crypto module (2). It is responsible for analysis of the SQL query contained in the request and the derivation of applicable ELA rules. To this end, an ELA rules repository is queried (3), which can reside in a relational database in the backend. Potentially numerous of different ELA repositories could exist, residing in arbitrary locations over the network, using arbitrary storage mechanisms apart from relational databases, e.g.

XML files. The ELA rules engine retrieves all rules from all locations and returns the result for further processing.

In a next step, the *FGDAC module* is provided with the incoming request, the previously determined context information and the set of applicable ELA rules (4). Based on the given information, the requested SQL query is being rewritten according to the dFGDAC approach utilizing cascading projections described in Section 2.3.2. In the following, the rewritten SQL statement is queried against the backend SQL database (5). The result of the SQL query contains access-control-enforced data only and is returned to the frontend module, which in turn formulates the response to the requesting client.

The *backend database* stores each business application data and access-control rules. It provides an SQL-queryable interface to the ELA rules engine and the FGDAC module.

### Implementation technology

The reference implementation SecSQL Server is developed using the *Microsoft .NET Framework*. The modules described in the previous section are implemented in a loosely coupled way and dynamically bound together at runtime to reach maximum extensibility and interchangeability. In theory, additional or replacing components can be added over time without changing or even corrupting the overall system. The backend of the reference implementation uses a MySQL database.

### Request and response format

Requests issued by the client to the frontend module as well as responses sent back to the caller use the *JavaScript Object Notation* (JSON) format. The fixed message format is depicted in Figures 2.7 and 2.8. Incoming requests contain the *SQL* query either entered manually in a minimal text based client or assembled by some assistive graphical user interface. In addition, a digital signature of the query is attached in the field *Pkcs7* to enable the system to perform integrity checks on the issued query. Outgoing responses to the client comprise the actually (rewritten) executed SQL, the originally issued SQL as well as the results of the query. Beyond that, meta data on the results of the query processing is attached.

### Query processing

Allowed SQL query types are SELECT, INSERT and UPDATE as well as the MySQL-specific SHOW command to present definitions and structure of database objects. In addition, a custom command called *SX.ENACT* is exposed. SX.ENACT can be used to enact proposed changes to the database, accepted by collaborative decision-making, as described in Section 2.3.1. It contains no mechanism to prevent malicious queries, as such would violate the basic principle of neutrality in SNBG. As a result, it is theoretically possible for the decision-making body to agree on the execution of statements that would be able to destroy the complete system.

```
{
  SQL:      /* content of the request expressed in SQL */,
  Pkcs7:    /* Base64 encoded Pkcs7 signature of the SQL */,
  Comment:  /* informative human-legible description of the request */
}
```

Figure 2.7: Structure of incoming JSON requests [35]

```
{
  Results:
  [{
    ExecutedSQL:    /* the transformed request */,
    RequestedSQL:   /* the original request */,
    Rows:
    [{
      Name:         /* the name of the column/attribute */,
      Value:        /* the stored value */
    }]
  }],
  Feedback:         /* feedback message, e.g. in case of error */,
  GenerationDate:   /* execution date */,
  OK:               /* flag denoting success of the execution */
}
```

Figure 2.8: Structure of JSON responses [35]

All statements are subject to dFGDAC. SELECT statements only return rows where not a single requested attribute is restricted, using cell-level access control, from being accessed. The reason is to prevent the partial exposure of data, a possible state occurring in systems, whose FGDAC set-up is referred to as *The Truman Model* [41]. dFGDAC enforcement for INSERT and UPDATE statements requires a workaround making use of the MySQL built-in *DUAL* virtual relation. Selecting this relation as basis for INSERT and UPDATE statements makes it possible to prepare values, that need to be updated or inserted, in variables, and equip the set of data to be changed with WHERE-clauses reflecting ELA rules applicable to the request.

### ELA storage

ELAs (access rules) are, just as data and structure of the system itself, subject to collaborative decision-making. For this reason, a predefined data structure for storing both positive and negative authorizations is necessary, catering for a default solution to the at design-time unpredictable data access rules. In the SecSQL system, they are referred to as *Permissions* and *Restrictions*, respectively. The *Permissions* table contains the identification of the described database object (either table or column) and the permitted SQL statement type. Rows in the *Restrictions* table are directly related to a corresponding entry in the *Permissions* table and additionally comprise a potentially unlimited number of WHERE clauses, defined in SQL language, restricting all requests of the defined SQL statement type to the corresponding database object.

## 2.4 XACML: A standardized access control policy language

XACML is an XML based language catering for a standardized way of defining ABAC policies. In addition, it comes with a complete proposal concerning both architecture and data flow necessary for implementing an access control system. However, system engineers can freely decide on the degree of accordance with the proposed concepts and cherry-pick the components and concepts tailored to their needs.

XACML 1.0 was first introduced as a standard by the OASIS[3] standards organization in 2003, followed by version 2.0 in 2005. Currently, the latest version is XACML 3.0, announced as a standard in 2013 [15].

The following sections describe the basics of the XACML policy language, usage of policies and architectural foundations.

### 2.4.1 XACML policies

Similar to the access control models introduced in the previous sections, central elements of XACML policies are *subjects*, *resources* and *rules*. Those terms refer to requestors, objects to be accessed and granted permissions (either positive or negative), respectively. Furthermore, *attributes* are used to describe all elements involved in an access control decision. These could either be simply used for identifying the involved actors or more generally for describing classes of affected elements. By default, attributes can be attached to subjects, resources and the *environment* surrounding the access control system, although the model is capable of using customized additional categories.

#### XACML policy language constructs

In order to express and group access control policies, the basic elements *rule*, *policy* and *policy set* are used. Figure 2.9 gives an overview on these elements. Policy sets may comprise multiple policies or even other policy sets. Policies consist of a non-empty set of rules.

XACML rules provide the basis for access control decisions by defining permissions and prohibitions at finest granularity. The *target* element of a rule is used to specify, to what kind of access requests the rule is applicable. It makes therefore use of a logical expression, defined over the attributes describing the current request. As rules obligatorily have to be contained in a policy element, the rule target can be omitted. If so, the target

[3]Organization for the Advancement of Structured Information Standards, `https://www.oasis-open.org/`

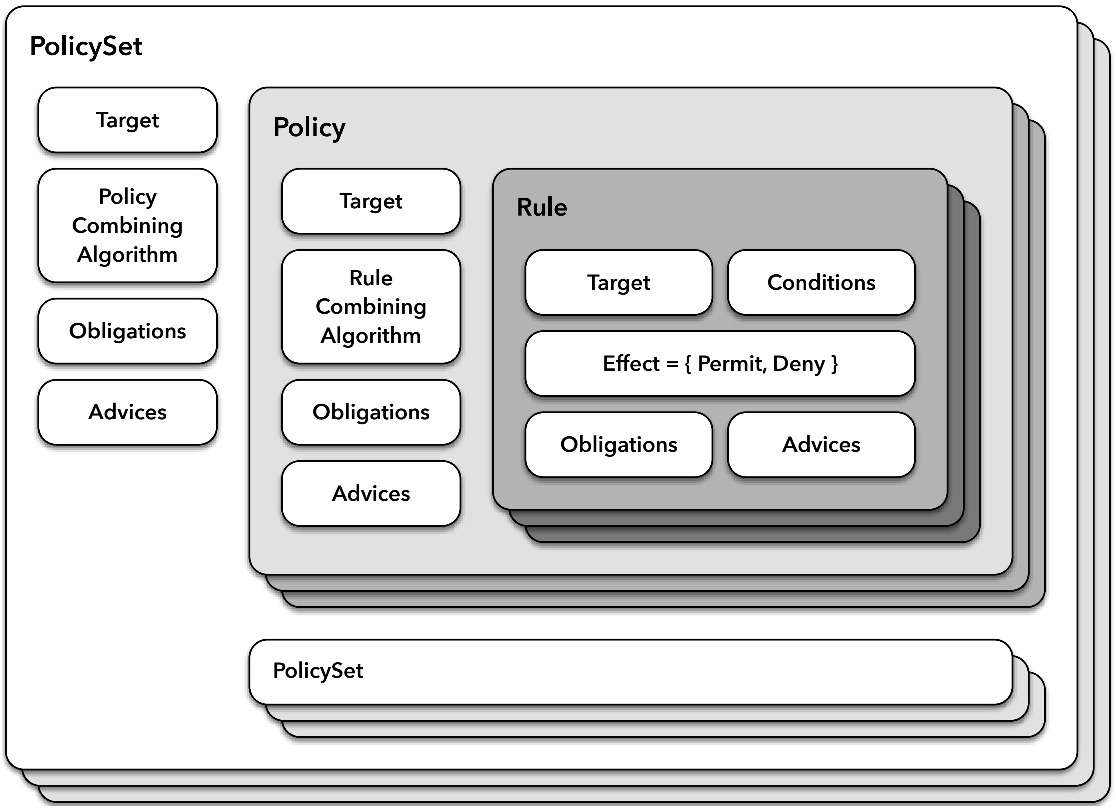


Figure 2.9: Core language constructs of XACML policy language, adopted from [18]

of the containing policy is derived. Rule *conditions* are used to further refine the set of applicable requests by providing more capabilities such as more detailed functions or comparability of multiple attributes. The main difference between targets and conditions is that targets can only be used for expressing static comparisons, i.e. only the left-hand side of a comparison can hold a variable attribute value. Conditions, however, can be utilized to do a comparison of two variable attribute values. The *effect* of a rule simply describes whether the fulfilment of all target and condition expressions lead to a result of either *Permit* or *Deny*. Optionally, *obligations* may be attached to a rule, describing actions that *must* be undertaken by the requestor along with an access decision. For example, the requesting system could be obliged to log any access attempt on confidential resources to an audit trail. Likewise, *advices* can be attached to such an access decision of which execution by the requestor is optional.

Similar to rules, applicability of policies is described using the *target* element. Here, targets can either be defined explicitly by the policy writer or may be computed by the targets of the rules it contains. To this end, either the intersection or the union of all sub rules targets can be used. Obviously, selection of one of these approaches has to be considered carefully, to achieve correct sets of applicable policies. Obligations and advices are used equivalently at policy level and rule level. As policies may comprise more than one applicable rule, there exists the possibility that several conflicting access decisions can be derived for a single authorization request. For this purpose, *rule combining algorithms* have to be defined, which take care of deciding what decision has to be returned to the requestor. Combining algorithms will be discussed in detail in the next section.

Finally, policy sets comprise multiple policies or even sub policy sets. Again, combination algorithms are necessary to resolve conflicts in authorization decisions of the contained policies and rules. In addition, targets, obligations and advices may be defined as well.

### Combining algorithms

Combining algorithms are part of both policy and policy set elements. They are used to compute one single overall decision per authorization request, mapping possible several decisions into a single unambiguous result. As an extension, combining algorithms can be provided with parameters affecting the behaviour of the decision algorithm. The list of possible results of a rule, policy or combining algorithm evaluation contains the values *Permit*, *Deny*, *NotApplicable* and *Indeterminate*. *NotApplicable* is returned if no target was matched in any rule or policy. *Indeterminate* is the result of an erroneous rule or policy evaluation. It is extended by the information, whether the applicable but failed rule or policy would have evaluated to *Permit*, *Deny* or possibly one of both, which is denoted as *Indeterminate(P)*, *Indeterminate(D)* or *Indeterminate(DP)*, respectively.

In the following, some built-in combining algorithms are listed and described. Combining algorithms are identified using a URN within the *RuleCombiningAlgID* and *PolicyCombiningAlgID* elements of the described policy and policy set, respectively. System engineers are enabled to implement custom algorithms beyond those included by default.

Listing 2.10: Example policy combining algorithm URNs

```
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:deny-
    overrides
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:permit-
    overrides
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first-
    applicable
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:only-one-
    applicable
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:ordered-deny-
    overrides
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:ordered-
    permit-overrides
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:deny-unless-
    permit
```

```
urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:permit-unless
    -deny
```

**Deny-overrides** describes a rather restrictive combining algorithm. As can easily be recognized by its name, any single *Deny* decision applicable to the current request overrides all other decisions, even if there are several rules returning *Permit* as a result. *NotApplicable* and *Indeterminate* decisions are possible.

**Permit-overrides** obviously implements the complete opposite of *Deny-overrides.* As long as there is one single *Permit* decision derived, the overall decision is *Permit* as well. *NotApplicable* and *Indeterminate* decisions are possible.

**Deny-unless-Permit and Permit-unless-Deny** are combining algorithms guaranteeing that the overall result is either *Permit* or *Deny*, even if there is no rule or policy applicable at all or the evaluation of possibly applicable rules fails. Opposing to the first two algorithms, a result of *NotApplicable* or *Indeterminate* is not possible, which in particular is useful at the topmost level of a policy tree in order to ensure the returning of an unambiguous authorization decision back to the requestor. *Deny-unless-Permit* evaluates in favour of positive access decisions, that is, presence of a single *Permit* decision is sufficient for returning the same as an overall result. In every other case, the overall result is *Deny.* Consequently, *Permit-unless-Deny* prioritizes negative access decisions.

**First-applicable** describes an algorithm with consideration of the order of rules within a policy / policies within a policy set. The first applicable rule evaluating to a concrete *Permit* or *Deny* decision will halt the algorithm and the result will be returned. If no applicable rule was found, *NotApplicable* is returned. If an error occurs while evaluating a rule, the corresponding *Indeterminate* state is returned.

**Only-one-applicable** is used to retrieve a completely unambiguous authorization decision. If more than one policy is applicable for the current request, *Indeterminate* is returned. If exactly one policy is applicable, the result of the found policy is returned. If no policy is applicable, *NotApplicable* is returned.

### Example XACML policy

Listing 2.11 shows a minimalist example XACML policy containing one single rule. The policy governs the access to the attribute "LASTNAME" of an employee record. Consider that full qualifiers and namespaces of XML elements, attributes and attribute values are omitted for the sake of simplicity.

In Line 1, the policy is provided with a name and "deny-unless-permit" as rule combining algorithm. Starting with Line 2, the target is set using given attributes defining the intent of access (*action-id* = *VIEW*) and the specific requested resource

(*resource.attributeName* = *LASTNAME*). The definition of the single rule contained starts in Line 20. By setting the effect to “Permit”, a successful evaluation of the attached condition will lead to a positive access decision. The condition makes use of the built-in function “string-equal”, taking the parameters “subject.employee.dept” and “resource.employee.dept”, describing department affiliation of the user requesting access and those of the employee record being queried, respectively. As mentioned above, conditions need to be chosen over targets if two variable attribute values need to be compared, as target clauses are only capable of comparing one single variable value with a static literal.

Given the whole definition of the policy, the meaning reads as follows: “If the requestor wants to view the LASTNAME attribute of an employee record: Permit access, if the requestor is affiliated with the same department as the employee being described by the queried record. Otherwise, deny access.”.

Listing 2.11: Example XACML policy: ViewEmployeeLastName

```
 1 <Policy PolicyId="ViewEmployeeLastName" RuleCombiningAlgId="deny-unless-permit">
 2   <Target>
 3     <AnyOf>
 4       <AllOf>
 5         <Match MatchId="string-equal">
 6           <AttributeValue DataType="string">VIEW</AttributeValue>
 7           <AttributeDesignator AttributeId="action-id"
 8                                DataType="string"
 9                                Category="action"/>
10         </Match>
11         <Match MatchId="string-equal">
12           <AttributeValue DataType="string">LASTNAME</AttributeValue>
13           <AttributeDesignator AttributeId="resource.attributeName"
14                                DataType="string"
15                                Category="resource"/>
16         </Match>
17       </AllOf>
18     </AnyOf>
19   </Target>
20   <Rule Effect="Permit" RuleId="ViewEmployeeLastName_SameDept">
21     <Target />
22     <Condition>
23       <Apply FunctionId="any-of-any">
24         <Function FunctionId="string-equal"/>
25         <AttributeDesignator AttributeId="subject.employee.dept"
26                              DataType="string"
27                              Category="access-subject"/>
28         <AttributeDesignator AttributeId="resource.employee.dept"
29                              DataType="string"
30                              Category="resource"/>
31       </Apply>
32     </Condition>
33   </Rule>
34 </Policy>
```

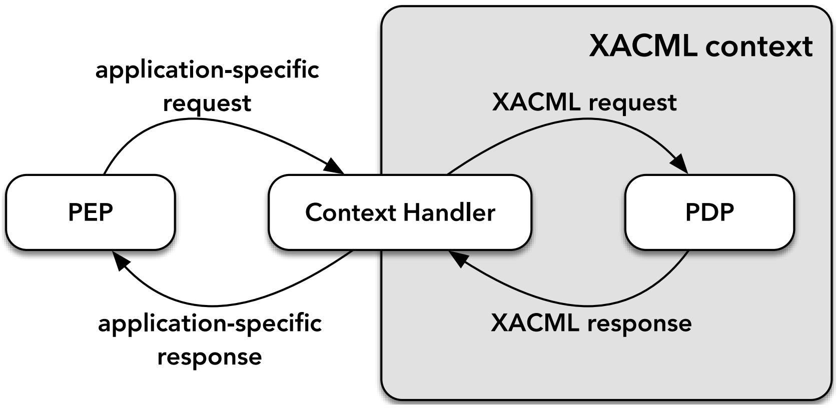


Figure 2.10: XACML context, adopted from [15]

### 2.4.2 XACML architecture and data flow

The system architecture recommended in the XACML 3.0 standard largely complies with the proposed ABAC system architecture already introduced in Section 2.1.5. Additionally, XACML introduces a *context handler* component (see Figure 2.10), taking care of transforming incoming requests from application-specific communication formats, issued by an application-specific PEP, to XACML authorization requests. These requests are forwarded to a PDP that thereupon returns a XACML authorization response. Subsequently, the context handler is responsible for converting the decision back to the application-specific format and notifying the PEP. Thus, the application environment in need of access control support by a XACML system is completely encapsulated from XACML-specific technologies. In turn, PDPs are not forced to retrieve XACML policies applicable to a specific request from a pure XML based XACML policy store. By definition of the standard, the policies could just as well be stored in and retrieved from a conventional RDBMS or other policy sources.

#### XACML request format

XACML requests comprise a list of attributes describing the intention of the access control request. Attached attributes can be categorized into the following default categories:

- **access-subject**
  attributes describing the access subject, i.e. the user requesting access to some query-able information

- **resource**
  attributes describing the resource, i.e. identification or description of the information, the requestor wants to query

- **action**
  attributes describing the action, i.e. the intended use, e.g. *read* some chunk of information, *update* a specific record

- **environment**
  attributes describing the environment, i.e. external circumstances of the access control request, e.g. spatiotemporal information

Listing 2.12 depicts an example XACML request containing attributes of the categories *access-subject* (from Line 5), *action* (from Line 13) and *resource* (from Line 22). The subject is identified via her *subject.employee.id*. The requested resource is described using the *resource.employee.id* attribute and the specific desired detailed information, defined via *resource.attributeName*. Finally, the intended action is described using the attribute *action-id*.

The meaning of the access control request reads as follows: “I am employee with ID=1 and I want to view the LASTNAME attribute of employee with ID=2”.

Listing 2.12: Example XACML request: view LASTNAME of Max Power

```
1  <Request ReturnPolicyIdList="false" CombinedDecision="false">
2
3     <!-- attributes describing access subject:
4          employee id = 1 -->
5     <Attributes Category="access-subject">
6        <Attribute AttributeId="subject.employee.id" IncludeInResult="false">
7           <AttributeValue DataType="string">1</AttributeValue>
8        </Attribute>
9     </Attributes>
10
11    <!-- attributes describing desired action:
12         action-id = VIEW -->
13    <Attributes Category="action">
14       <Attribute AttributeId="action-id">
15          <AttributeValue DataType="string">VIEW</AttributeValue>
16       </Attribute>
17    </Attributes>
18
19    <!-- attributes describing requested resource:
20         employee id = 2,
21         attributeName = LASTNAME -->
22    <Attributes Category="resource">
23       <Attribute AttributeId="resource.employee.id" IncludeInResult="false">
24          <AttributeValue DataType="string">2</AttributeValue>
25       </Attribute>
26       <Attribute AttributeId="resource.attributeName" IncludeInResult="false">
27          <AttributeValue DataType="string">LASTNAME</AttributeValue>
28       </Attribute>
29    </Attributes>
30
31 </Request>
```

**MultiRequests** As of XACML 3.0, support for submitting multiple access control requests within one single message was introduced [15]. Therefore a reduction of message exchange can be achieved, reducing the overall time of network latency. Using the *CombinedDecision* attribute of the request element, the requestor can either ask the PDP for several single decisions on each contained request or for a combined overall decision. In the latter case, the overall decision can only be *Permit* if every single contained request results in a *Permit* decision.

Listing 2.13 shows an example XACML MultiRequest similar to the single request presented above. It contains access requests for two different attributes of the desired employee's record.

Similar to the previous example, the request contains attributes describing the access subject and the desired action, in Lines 5 and 13, respectively. Note that each of the <Attributes> elements now contains an *id* attribute. This ID is going to be used later for referencing the attributes in concrete requests. In addition to the requested LASTNAME attribute (Line 22), we now have another resource attributes section describing access intention on the SAL attribute of employee with ID=2 (Line 34). Again, the <Attributes> elements are equipped with identifiers for the purpose of being referenced.

Finally, the *MultiRequests* section contains one *RequestReference* section per request to be contained in the message. By adding references to attributes defined above, using the attributes *id*'s, each request is equipped with the necessary attribute values. The first request reference in Line 50 generates exactly the same request as the single request described in Listing 2.13. Additionally, the request reference in Line 61 adds a request for accessing the SAL column of employee with ID=2. As the *CombinedDecision* attribute in the request element is set to false, the response of the PDP will contain 2 single decisions on each of the requests. Setting the *CombinedDecision* attribute to true would lead to a positive access control decision only if both requests result in a *Permit* decision.

Listing 2.13: Example XACML MultiRequest: view LASTNAME and SAL of Max Power

```
1  <Request ReturnPolicyIdList="false" CombinedDecision="false">
2
3     <!-- attributes describing access subject:
4          employee id = 1 -->
5     <Attributes Category="access-subject" id="ID-accessSubjectAttributes">
6        <Attribute AttributeId="subject.employee.id" IncludeInResult="false">
7           <AttributeValue DataType="string">1</AttributeValue>
8        </Attribute>
9     </Attributes>
10
11    <!-- attributes describing desired action:
12         action-id = VIEW -->
13    <Attributes Category="action" id="ID-actionAttributes">
14       <Attribute AttributeId="action-id">
15          <AttributeValue DataType="string">VIEW</AttributeValue>
16       </Attribute>
17    </Attributes>
18
19    <!-- attributes describing requested resource:
20         employee id = 2,
21         attributeName = LASTNAME -->
22    <Attributes Category="resource" id="ID-resourceAttributesLASTNAME2">
23       <Attribute AttributeId="resource.employee.id" IncludeInResult="false">
```

```
            <AttributeValue DataType="string">2</AttributeValue>
        </Attribute>
        <Attribute AttributeId="resource.attributeName" IncludeInResult="false">
            <AttributeValue DataType="string">LASTNAME</AttributeValue>
        </Attribute>
    </Attributes>

    <!-- attributes describing requested resource:
         employee id = 2,
         attributeName = SAL -->
    <Attributes Category="resource" id="ID-resourceAttributesSAL2">
        <Attribute AttributeId="resource.employee.id" IncludeInResult="false">
            <AttributeValue DataType="string">2</AttributeValue>
        </Attribute>
        <Attribute AttributeId="resource.attributeName" IncludeInResult="false">
            <AttributeValue DataType="string">SAL</AttributeValue>
        </Attribute>
    </Attributes>

    <MultiRequests>

        <!-- request comprising following attributes:
             access-subject: employee id = 1
             action:         action-id = VIEW
             resource:       employee id = 2
                             attributeName = LASTNAME -->
        <RequestReference>
            <AttributesReference ReferenceId="ID-accessSubjectAttributes" />
            <AttributesReference ReferenceId="ID-actionAttributes" />
            <AttributesReference ReferenceId="ID-resourceAttributesLASTNAME2" />
        </RequestReference>

        <!-- request comprising following attributes:
             access-subject: employee id = 1
             action:         action-id = VIEW
             resource:       employee id = 2
                             attributeName = SAL -->
        <RequestReference>
            <AttributesReference ReferenceId="ID-accessSubjectAttributes" />
            <AttributesReference ReferenceId="ID-actionAttributes" />
            <AttributesReference ReferenceId="ID-resourceAttributesSAL2" />
        </RequestReference>

    </MultiRequests>

</Request>
```

### XACML response format

The format of a XACML response message contains a single *decision* node containing one of the following allowed values: *Permit*, *Deny*, *NotApplicable*, *Indeterminate*.

Listing 2.14: Example XACML response: view LASTNAME of Max Power

```
<Response>
    <Result>
        <Decision>Permit</Decision>
    </Result>
</Response>
```

## 2.5 Summary

In this chapter, basic concepts and the evolutionary process of access control are presented. Low-level models, DAC and MAC, are followed by first standardization approaches in the form of RBAC. In its core, RBAC supports the concepts of users, roles, sessions and operations on objects. ABAC utilizes attributes describing requestors and accessed objects rather than definite identifiers, which leads to less maintenance intensive access control systems. FGDAC enables access control on database tables each on table, row, column and cell level. Proposed approaches to implement FGDAC are static views, parameterized views, the extension of the SQL language and query rewriting. The following section presents SecSQL as an example of an existing dFGDAC implementation. Core feature is the democratic creation, enactment and modification of machine-readable access policies to system-relevant data during runtime. Human administrators and super-users are avoided. In the final section, XACML is presented. XACML provides means for standardized definition, exchange and enforcement of ABAC policies.

CHAPTER 3

# A prototype dFGDAC system utilizing Oracle Virtual Private Database

In this chapter, a new prototype is proposed trying to replicate the dFGDAC features of SecSQL introduced in Section 2.3. The prototype relies on the capabilities of Oracle Database, in particular on the fine-grained query rewriting mechanism of VPD. The functional requirements are inherited from the model solution and the research field of SNBG.

The foundation for the proposed technical prototype is Oracle Database's built-in feature VPD, as introduced in Section 2.2.5. VPD offers a query-rewriting approach as a solution to the problem of FGDAC. Simply put, issued SQL queries to the database get dynamically equipped with additional WHERE clauses filtering the queried database table for permitted data. These filters can either be applied to whole records or single columns of the queried table, providing row-level and cell-level access control, respectively.

The remainder of this chapter is organized as follows. Firstly, the basic data model for providing dFGDAC is presented in Section 3.1. In Section 3.2 the concept of Proxy Views is introduced. Section 3.3 discusses the component providing means of collaborative decision-making. Finally, an overview of the prototype web-based user interface is given in Section 3.4.

## 3.1 Implementing Fine-Grained Data Access Control: the VPD schema

The necessary FGDAC functionalities are grouped in a module called "the VPD schema". This schema comprises the following components. A data model for storing authorization

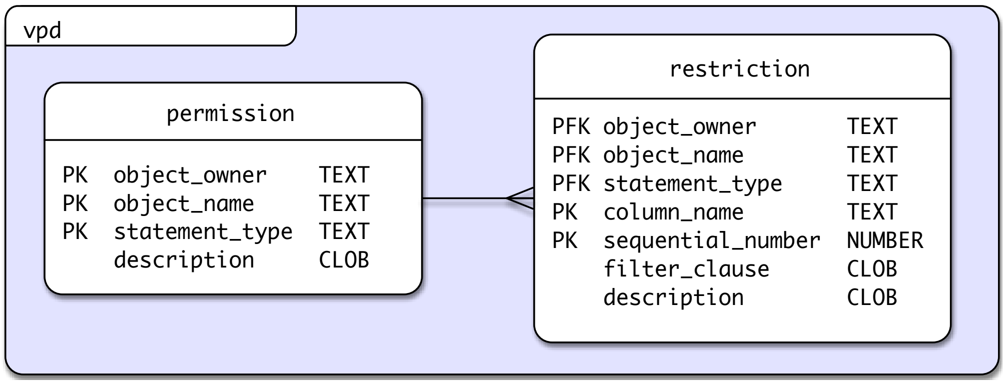


Figure 3.1: Data model of the VPD schema

data, a component providing context awareness to the access control system and means for defining and maintaining VPD policies. The components are described in detail in the following sections.

### 3.1.1 A simple data model for positive and negative authorizations

To implement dynamic access control using VPD, the access rules need to be maintained in a dedicated data structure. Figure 3.1 depicts the data model designed for use with our prototype implementation. Two tables, PERMISSION and RESTRICTION are necessary to store both positive and negative authorizations.

The intended usage of the former table is as follows: Each and every database table, that needs to be exposed to the public, shall be represented by one record. Tables lacking a corresponding entry in PERMISSION can not be accessed. The exposed table is identified using the columns OBJECT_OWNER and OBJECT_NAME. The column STATEMENT_TYPE is used to define the type of action, for which permission on the given table is granted. The allowed values include the standard SQL query types SELECT, INSERT, UPDATE and DELETE as well as the application specific permission types USER_ADD_PROPOSAL, USER_VOTE, USER_ENACT and PROPOSAL_ENACT, which will be discussed in Section 3.3. Finally, the DESCRIPTION column can be used to optionally add a comment or justification for the presence of the given permission.

The RESTRICTION table is related to the PERMISSION table via the columns OBJECT_OWNER, OBJECT_NAME and STATEMENT_TYPE, establishing a one-to-many relationship. Using the COLUMN_NAME column, it can take restrictions defined either for whole rows (using an asterisk as column name) or for single columns, using the name of the column the restriction should be applied to. As part of the primary key of the table, the SEQUENTIAL_NUMBER column allows the application of an unlimited number of restrictions per column.

The definition of the restriction is done by providing an SQL statement to populate the FILTER_CLAUSE column. The provided statement must return a boolean result, indicating whether the restricted row/column should be visible within the result of an arbitrary SQL statement, queried against the exposed table. Filter clauses may utilize the full expressiveness of the SQL language, including simple predicate comparisons and set operations. As predicates, the definer may refer to the column names of the restricted table, or even sub-select other information from the database. Context-aware restrictions are possible using a wildcard mechanism. The system provides automatic integration of the user ID currently logged-in to the system at the time of query execution. For this purpose, the user may be referenced via his citizen identification (referring to the parliament scenario from SNBG) within the filter clause using the wildcard @CITIZEN_ID. Information on wildcards and the context awareness module is described in more detail in Section 3.1.2. Finally, equivalent to the PERMISSION table, the RESTRICTION table contains a column DESCRIPTION allowing to optionally add a comment on the stored restriction.

By default, there is different behaviour of access control enforcement for row-level and column-level access control, concerning the need for explicit governance of access rules. On the one hand, general table access needs to be enabled explicitly by entering a corresponding record in the PERMISSION table. In other words, row-level security follows a pessimistic access control approach. On the other hand, provided that an entry in the PERMISSION table exists, all the columns of the table are exposed to the public by default. That means that column-level access control follows an optimistic approach. Restricting access to single columns of a table generally exposed to the public needs to be defined explicitly by entering a corresponding record to the RESTRICTION table.

Table 3.1 shows an example entry of the RESTRICTION table, providing a row-level (COLUMN_NAME = *) restriction for selecting (STATEMENT_TYPE = SELECT) the table SOVEREIGN.CITIZEN (columns OBJECT_OWNER and OBJECT_NAME). The FILTER_CLAUSE column contains an SQL predicate making use of the @CITIZEN_ID wildcard to retrieve the citizen identification of the logged-in user at SQL execution time. The clause also includes a sub-select retrieving additional data necessary for the access control decision. Finally, the DESCRIPTION column shows a textual description of the restriction's intention.

### 3.1.2 Providing context awareness to the access control system

To be capable of providing environment-attribute dependent access control, a context awareness module is necessary. The proposed solution utilizes Oracle Database's built-in feature "Application Contexts". Application contexts provide means of storing application-specific, session-dependent contextual information in key-value pairs. For further organization, contexts may be categorized using the concept of namespaces.

The context awareness module comprises the following components. (1) a key-value based storage for context information, implemented using Oracle application contexts; (2) an access layer consisting of the functions `context.set` and `context.get` for storing into

Table 3.1: Example entry of table RESTRICTION

| | |
|---|---|
| **OBJECT_OWNER** | SOVEREIGN |
| **OBJECT_NAME** | CITIZEN |
| **STATEMENT_TYPE** | SELECT |
| **COLUMN_NAME** | * |
| **FILTER_CLAUSE** | `citizen_id = @citizen_id or ( select position from sovereign.citizen where citizen_id = @citizen_id ) = 'official'` |
| **DESCRIPTION** | except for officials, citizens may only view their own record |

The filter clause as printed:

```
citizen_id = @citizen_id
or
( select position
    from sovereign.citizen
   where citizen_id = @citizen_id ) = 'official'
```

and retrieving key-value pairs from the context, respectively; (3) a database logon trigger taking care of proper initialization of the context every time a user logs in to the system; (4) a stored function, used by the VPD query rewriting engine, dedicated to replace all wildcards in restriction's filter clauses with concrete values from the context, taking an abstract filter clause as an input parameter and returning a concrete, SQL-executable filter clause.

Figure 3.2 gives a combined overview of both architecture of the context module and the process of query rewriting using context information. When a user logs in to the system, a database logon trigger is fired (1). The logon trigger determines the technical identification of the user attempting to log in and tries to establish a logical connection between the technical user and the real world citizen, the user is associated to. To this end, the CITIZEN table is queried for the citizen identification (2). Subsequently, the retrieved CITIZEN_ID is stored to the application context using context.set. In addition, a small number of environment attributes is determined from Oracle Database's meta information and provided to the context. The predefined environment attributes include the name of the client's host machine ("CLIENT_HOST"), the IP address the request originiated from ("CLIENT_IP_ADDRESS"), the operating system user of the client ("CLIENT_OSUSER") and the time the client has logged in ("CLIENT_LOGON_TIME") (3). As soon as the logged in user attempts to query the CITIZEN table, or any other table containing sensitive business data, the VPD query rewriting engine intercepts the request (4). The engine determines all restrictions applicable to the request and queries the context for necessary environment information using context.get (5). In a next step, the query is rewritten utilizing the wildcard replacement component and executed against the desired business data tables. Finally, the results are propagated the same path back to the user (6).

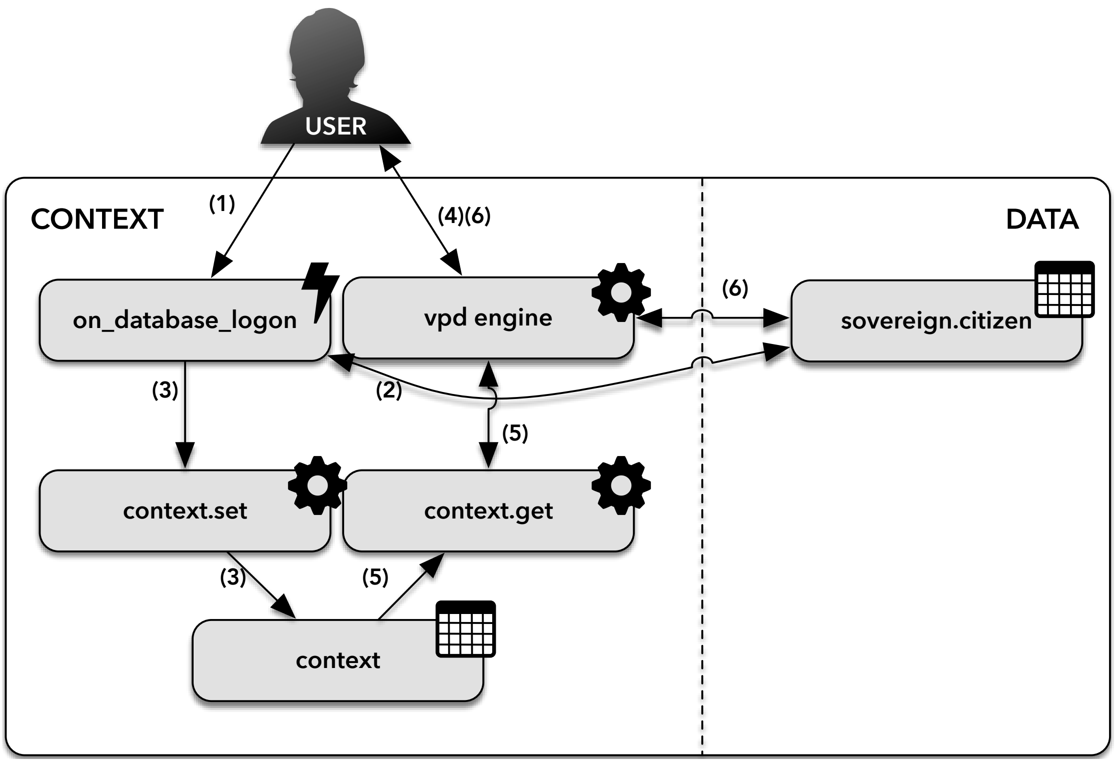


Figure 3.2: Overview of context awareness model

### 3.1.3 Definition of VPD policy functions

As already described in Section 2.2.5, Oracle VPD utilizes user-defined stored functions written in PL/SQL to return restrictions applicable to a specific query. Basically, VPD adds all the restrictions to the *WHERE* clause of the issued SQL query. The stored function has to provide an interface as follows: two input parameters of data type VARCHAR2 describing both schema (object owner) and name of the object; the function result has to be a string containing a boolean expression. The function is automatically called by the internal VPD engine on query execution and dynamically provided with the necessary input parameters describing the queried database object. For computation of the result it is allowed to make use of the complete expressiveness of PL/SQL, i.e. provide constant predicates, use the input parameters, use the application context or other session based information, or even query additional data from database objects. If more than one policy function is applied to a single database object, all returned predicates are logically conjuncted, i.e. they are put together using the *AND* operator.

### Row-level access control

Setting up policy functions for dynamic row-level access control is possible using little coding effort. It is sufficient to define one generic policy function and four wrappers for each supported SQL statement type, that is SELECT, INSERT, UPDATE and DELETE. The reason for that becomes clear if the way, policies are connected to policy functions, is explained. In Listing 3.1, the call to add a new row-level access control policy to the CITIZEN table is depicted. Several parameters of the call can be automatically populated with the values stored in the RESTRICTION table defined in Section 3.1.1, including OBJECT_SCHEMA, OBJECT_NAME and STATEMENT_TYPES. Although VPD allows to define a single policy function for all statement types by simply enumerating the desired statement types comma separated, the prototype system forces the usage to separate functions per statement type, to allow separate restriction behaviour based on the statement type. For example, SELECT access might be defined less restrictive than DELETE access on some business table. The POLICY_FUNCTION parameter is provided with the name of the statement type specific stored function, e.g. FILTER_ROW_SELECT for SELECT access.

Listing 3.1: Adding a new row-level VPD policy

```
DBMS_RLS.ADD_POLICY(
  object_schema    => 'SOVEREIGN',
  object_name      => 'CITIZEN',
  policy_name      => 'POL_SOV_CIT_ROW_LEVEL',
  function_schema  => 'VPD',
  policy_function  => 'POLICY.FILTER_ROW_SELECT',
  statement_types  => 'SELECT'
);
```

That leads to the definition of the four statement type specific policy functions for row-level access control: *FILTER_ROW_SELECT*, *FILTER_ROW_INSERT*, *FILTER_ROW_UPDATE* and *FILTER_ROW_DELETE*. All functions call a generic function FILTER_ROW, taking the owner and name of the restricted object and the statement type, to be able to return separate restriction predicates based on the type of access request.

The generic function's behaviour is explained using pseudocode in Listing 3.2. In Line 3, the application context is queried for the technical user currently logged in. If the user matches the owner of the queried object, a tautology can be returned indicating that no data filtering needs to be established. Owners always enjoy unrestricted access to their object's data. In Line 7, application context is queried for the citizen identification of the user currently logged in. If the citizen ID is not set properly, a contradictious boolean clause is returned, preventing unauthorized citizens from accessing potentially restricted data. Line 11 contains querying the PERMISSION table for the existence of a permission for the given combination of OBJECT_OWNER, OBJECT_NAME and STATEMENT_TYPE. If the query returns an empty result, meaning that no permission is present, again a contradictious boolean clause is returned. Starting from Line 18, the algorithm queries the RESTRICTION table and iterates through the set of returned restrictions. In each iteration, the concrete filter clause is generated by calling the wildcard replacement component of the context awareness module, providing the abstract filter clause of the iterated restriction. Subsequently, the filter clause is added to the resulting filter predicate using logical conjunction. Finally, in Line 30, the resulting filter predicate is checked for any found restrictions. The function returns the generated filter predicate or a tautology, if applicable restrictions were found or there are no restrictions present, respectively.

Listing 3.2: Behaviour of FILTER_ROW

```
 1 FILTER_ROW( objectOwner, objectName, statementType )
 2
 3    currentTechnicalUser = context.get( "user" )
 4    if currentTechnicalUser = objectOwner
 5       return true
 6
 7    citizenID = context.get( "citizen_id" )
 8    if citizenID = ""
 9       return false
10
11    permission = table.select( "PERMISSION",
12                               objectOwner,
13                               objectName,
14                               statementType )
15    if permission = ""
16       return false
17
18    predicate = ""
19    restrictions = table.select( "RESTRICTION",
20                                 objectOwner,
21                                 objectName,
22                                 statementType )
23    for each restriction in restrictions
24       if predicate ≠ ""
25          predicate = predicate + " AND"
26       abstractFilterClause = restriction.filterClause
27       concreteFilterClause = context.replaceWildcards( abstractFilterClause )
28       predicate = predicate + concreteFilterClause
29
30    if predicate = ""
31       return true
32    else
33       return predicate
```

### Column-level access control

Definition of policy functions for dynamic column-level access control requires more effort than for row-level access control. This is due to the fact, that it is not possible to dynamically retrieve the column name, the filter predicate should be generated for, during execution of the policy function. As described above, VPD policy functions need to be defined using a fixed interface, including only owner and table/view name of the currently queried object as input parameters. To be capable of dynamically retrieving the name of the affected column from within the function, an additional parameter in the function interface is necessary, providing the column name. This aspect was discussed in a online post in Oracle's official discussion board [10], leading to the finding, that dynamism to the extent necessary is not provided in the default feature set of Oracle VPD, which requires the invention of a workaround.

This workaround involves the dynamic creation of one single policy function per column-level restriction. This dynamic creation needs to be triggered by the emergence of a new column-level restriction, as this involves a change in the set of column-level access control rules. The name of the column under protection needs to be compiled hard-coded into the definition of the policy function. Listing 3.3 shows the behaviour of a custom, dynamically compiled column-level policy function. To ensure the creation of a large number of custom policy functions without running into naming conflicts, a random hash value is added to the name of the function. In contrast to the row-level policy functions described above, column-level functions are never reused for protection of more columns than the column it was originally dedicated to at compile time. For that reason, the mandatory input parameters OBJECT_OWNER and OBJECT_NAME are completely ignored in the code section of the function, instead, only hard-coded values are used to retrieve the applicable access constraints. For example, in Line 3 and the following, the owner of the protected object is known at compile time. Thus, the second operand of the comparison can be hard-coded to the function. Checking the presence of a valid citizen identification, starting from Line 7, stays the same with respect to the behaviour of FILTER_ROW. In turn, a permission check is omitted (Line 11) due to the optimistic approach of column-level access control defined in Section 3.1.1. The procedure of retrieving restrictions and wildcard replacements to the filter clauses essentially does not differ either, with one exception: For the selection of all applicable restrictions from the RESTRICTION table only hard-coded values are used. Additionally, the hard-coded column name is included in the query (starting from Line 13). Finally, the return part of the function remains unchanged compared to the FILTER_ROW algorithm (Line 26).

Listing 3.3: Behaviour of FILTER_COLUMN_<HASH>

```
1 FILTER_COLUMN_<HASH>( objectOwner , objectName )
2
3    currentTechnicalUser = context.get( "user" )
4    if currentTechnicalUser = "Hard−coded object owner"
5       return true
6
```

```
    citizenID = context.get( "citizen_id" )
    if citizenID = ""
        return false

    /* no permission check necessary: optimistic access control */

    predicate = ""
    restrictions = table.select( "RESTRICTION",
                                 "Hard-coded object owner",
                                 "Hard-coded object name",
                                 "Hard-coded statement type",
                                 "Hard-coded column name" )
    for each restriction in restrictions
        if predicate ≠ ""
            predicate = predicate + " AND"
        abstractFilterClause = restriction.filterClause
        concreteFilterClause = context.replaceWildcards( abstractFilterClause )
        predicate = predicate + concreteFilterClause

    if predicate = ""
        return true
    else
        return predicate
```

Adding column level policies to the VPD engine requires additional parameterization. An example call is depicted in Listing 3.4. The parameters OBJECT_SCHEMA, OBJECT_NAME, STATEMENT_TYPES and SEC_RELEVANT_COLS can be populated with the corresponding values from the RESTRICTIONS entry.

Note that the SEC_RELEVANT_COLS parameter corresponds to the COLUMN_NAME column. The custom policy function introduced above needs to be compiled upfront, the connection between the function and the policy is established by providing the randomly generated function name to the POLICY_FUNCTION parameter. The last parameter SEC_RELEVANT_COLS_OPT is used to express, that rows containing restricted columns need to be returned regardless of the existence of an actual hidden column value. Instead, NULL will be returned as the value of the hidden column. Omitting the parameter would lead to filtering of the complete row, if at least one column value is hidden.

Listing 3.4: Adding a new column-level VPD policy

```
DBMS_RLS.ADD_POLICY(
  object_schema         => 'SOVEREIGN',
  object_name           => 'CITIZEN',
  policy_name           => 'POL_SOV_CIT_DOB_COLUMN_LEVEL',
  function_schema       => 'VPD',
  policy_function       => 'POLICY.FILTER_COLUMN_<HASH>',
  statement_types       => 'SELECT',
  sec_relevant_cols     => 'DATE_OF_BIRTH',
  sec_relevant_cols_opt => dbms_rls.all_rows
);
```

### 3.1.4 Maintaining VPD policies

Effective enforcement of access constraints defined in the FGDAC data model is undertaken by the built-in VPD engine of Oracle Database. To ensure a comprehensive and accurate implementation of all the permissions and restrictions, maintenance of VPD policies is important - the applied policies need to comply with the given permission and restriction meta data at any time. This requires several components. Firstly, a component taking care of enforcing all the defined access constraints is necessary. Also, a mechanism needs to be established for cleaning up policies which are no longer required. Finally, those two program sections need to be triggered every time either the structure of the database or access constraints change.

#### Adding new policies

The first component is implemented using a predefined function called *ENFORCE_POLICIES*. The behaviour of this function is depicted in Listing 3.5. The function takes three input parameters used to identify the object, for which FGDAC needs to be enforced. First, all restrictions applicable to the given object are retrieved from the RESTRICTION table (Line 3). Additionally, in Line 7, all statement types for which there is no corresponding entry for the given object in the PERMISSION table at all, are determined. As a consequence of defining row-level access control using a pessimistic approach in Section 3.1.1, implicit restrictions need to be established for all objects lacking explicit permission for any of the default SQL statement types. In the following, the set of determined restrictions is iterated and processed depending on the level of access control. Row-level restrictions (starting from Line 12), are recognized by an asterisk symbol instead of a real column name. First, the applicable, pre-defined row-level policy function is determined based on the statement type of the iterated restriction. Then, a new VPD policy is added, providing the determined function name and the values identifying the object and the desired statement type from the restriction. The steps for processing column-level restrictions are depicted starting from Line 19. First of all, a new random, unique and yet unused name for the necessary custom policy function needs to be derived. In a second step, the custom policy function is dynamically created. As input to the dynamic creation the following parameters are necessary: the previously derived function name, the object and column identification and the statement type. Finally, a new VPD policy is added, providing the same parameters.

Listing 3.5: Behaviour of ENFORCE_POLICIES

```
1  ENFORCE_POLICIES( objectOwner, objectName, columnName )
2
3     restrictions = table.select( "RESTRICTION",
4                                  objectOwner,
5                                  objectName,
                                       columnName )
    restrictions = restrictions +
                   table.select( "ALL_OBJECTS without any permission" )

    for each restriction in restrictions

        if restriction.columnName = "*"
            functionName = "FILTER_ROW_" + restriction.statementType
            vpd.add_policy( restriction.objectOwner,
                            restriction.objectName,
                            functionName,
                            restriction.statementType )

        else
            functionName = "FILTER_COLUMN_" + hash()
            create_policy_function( functionName,
                                    restriction.objectOwner,
                                    restriction.objectName,
                                    restriction.statementType,
                                    restriction.columnName )
            vpd.add_policy( restriction.objectOwner,
                            restriction.objectName,
                            functionName,
                            restriction.statementType,
                            restriction.columnName )
```

### Cleaning up outdated policies

Restrictions on database objects may become obsolete at some time during the usage of the prototype application. In order to maintain a clean and tidy system state, outdated policies and all related system components need to be removed in this case.

To this end, a function called CLEANUP_POLICIES is provided. The behaviour of this function is depicted in Listing 3.6. The function's interface provides the same input parameters as ENFORCE_POLICIES does. First, all the policies effective for the given object identification are determined. In a next step, the determined policies are iterated through. As step one of each iteration, the VPD policy, identified by it's unique policy name, is dropped. If the dropped policy provided column-level access control, the custom policy function created during addition of the policy is dropped as well. The reusable policy functions for row-level access control are dropped at no time.

Listing 3.6: Behaviour of CLEANUP_POLICIES

```
CLEANUP_POLICIES( objectOwner, objectName, columnName )

    policies = vpd.get_policies( objectOwner,
                                 objectName,
                                 columnName )

    for each policy in policies
        drop_policy( policy.policyName )
        if policy.columnName ≠ "*"
            drop_function( policy.policyFunction )
```

### Triggering policy maintenance

As stated in the introductory paragraph of this section, policy maintenance needs to be triggered every time either the structure of the database or access constraints change. To this end, several triggers on different events potentially occurring during the usage of the prototype application are installed.

To take care of changing database structures, Oracle allows for the definition of triggers AFTER CREATE ON DATABASE and BEFORE DROP ON DATABASE. Every time one of these events occurs, the database calls a stored function associated to the event allowing for custom code to be executed as a result of the change. The implementation of these two events contains calls to ENFORCE_POLICIES and CLEANUP_POLICIES, respectively.

In addition, it is possible to establish database triggers notifying the system about changing table contents. To ensure correct maintenance of access control policies on changing access rules, a row-level change trigger AFTER INSERT OR UPDATE OR DELETE on table RESTRICTION is defined and implemented. The behaviour of the triggered function is depicted in Listing 3.7. Row-level triggers provide the entire record of values of both the old row (before the change) and the new row (after the change) to the context of the trigger function. This enables the developer to react to specific data changes, by referring to the context variables oldRow and newRow. In case that the trigger reacted to an UPDATE or DELETE action on the RESTRICTION table, the oldRow record will be filled with the values valid before the action. As such an action implies that an access rule has changed or even was deleted, the CLEANUP_POLICIES function needs to be called with the values of the oldRow record (Line 3). Line 8 and the following define the reaction on an INSERT or UPDATE statement. Again, CLEANUP_POLICIES needs to be called if it was not already done for the same parameters in the previous Section. Finally, the ENFORCE_POLICIES function needs to be called in any case other than a performed DELETE action, which can be recognized by a non-empty newRow record.

Listing 3.7: Behaviour of ON_RESTRICTION_CHANGE

```
1  ON_RESTRICTION_CHANGE( oldRow, newRow )
2
3      if oldRow ≠ EMPTY
4          CLEANUP_POLICIES( oldRow.objectOwner,
5                            oldRow.objectName,
6                            oldRow.columnName )
7
8      if newRow ≠ EMPTY AND newRow ≠ oldRow
9          CLEANUP_POLICIES( newRow.objectOwner,
                                newRow.objectName,
                                newRow.columnName )

    if newRow ≠ EMPTY
        ENFORCE_POLICIES( newRow.objectOwner,
                          newRow.objectName,
                          newRow.columnName )
```

After execution of the ON_RESTRICTION_CHANGE trigger function the access control system has successfully reacted to a change of the access control data model and (re-)enforced all necessary VPD policies.

## 3.2 Enabling complex dynamic access control policies using Proxy Views

Due to a number of technical limitations of VPD policies, implementing access control using VPD is not possible in the dynamism needed for covering the desired dFGDAC features. To this end, a workaround is presented in order to overcome the shortcomings of VPD. The following sections comprise the description of the mentioned limitations as well as a solution approach.

### 3.2.1 Limitations of VPD policies

Oracle VPD includes technical drawbacks, that limit the practical usefulness of the technology in the area of dFGDAC. In the following, two problems arisen during the prototyping phase of the proposed solution, are presented and described using examples.

#### Circular VPD policies

Consider the EMPLOYEE table introduced in Section 2.2.1. An exemplary fine-grained access constraint on the SAL column could be defined as follows. “Employees are only allowed to view their own salary, except for department heads, which may view the salary of all employees of their department." Consequently, an example implementation of the policy function used for masking the column could look like the one in Listing 3.8.

Listing 3.8: Example 1: policy function FILTER_SALARY

```
function FILTER_SALARY(
  object_owner  in varchar2,
  object_name   in varchar2
) return varchar2
is
  nEmployeeID varchar2(16) := util.context.get( 'employee_id' );
begin

  return 'id = ' || nEmployeeID || '
```

```
or
( ( select position from EMPLOYEE where id = ' || nEmployeeID || ' )
  =
  ''Head of '' || dept
)';
end FILTER_SALARY;
```

Note that the complete filter predicate is written in one single dynamic SQL string, utilizing the application context for retrieval and correct application of the employee identification. Thus, the information, whether the employee currently logged in is a department head, is retrieved from a sub-SELECT, in turn querying the EMPLOYEE table. This leads to the following question: How should the access control enforcement engine handle subquerying of the same table, that is subject to access control, from within the generated filter predicate? VPD's answer is just as simple as to throw an exception during SQL execution time:

```
ORA-28113: policy predicate has error
 [...]
*Cause:    Policy function generates invalid predicate.
*Action:   Review the trace file for detailed error information.
```

Inspecting the session's trace file as suggested by the error message leads to the following information:

```
 Error information for ORA-28113:
 [...]
ORA-28108: circular security policies detected
*Cause:    Policies for the same object reference each other.
*Action:   Drop the policies
```

Obviously, but not proven by any official documentation of Oracle Database, columns used within VPD predicates are in turn subject to access control. In the worst case, this fact can lead to circular dependencies of VPD predicates. The problem was discussed in an online post in Oracle's official discussion board [11]. One proposed solution approach was rejected because it included the on-design-time static preparation of information necessary during access constraint evaluation time, which resulted in incompatibility with the dynamism requirement. The discussion also came up with a promising approach including the usage of separate database views for access control enforcement, rather than directly restricting access to the table itself.

### Masked columns as access control criterion

There is another limiting characteristic of the Oracle VPD enforcement engine. It unveils when columns, that are already restricted by a valid VPD predicate, need to be considered in another VPD predicate. As an example, two access constraints to the EMPLOYEE table are defined as follows: (1) "No one can view salaries" and (2) "Job positions of employees can only be viewed if the salary of the respective employee does not exceed

4000". Consequently, access constraint (1) restricts access to data that is necessary to successfully evaluate access constraint (2). The definition of two corresponding policy functions is depicted in Listing 3.9.

Listing 3.9: Example 2: policy functions FILTER_SALARY and FILTER_BIG_EARNERS

```
function FILTER_SALARY(
  object_owner in varchar2,
  object_name  in varchar2
) return varchar2
is
begin
  return '1=0';
end FILTER_SALARY;

function FILTER_BIG_EARNERS(
  object_owner in varchar2,
  object_name in varchar2
) return varchar2
is
begin
  return 'sal < 4000';
end FILTER_BIG_EARNERS;
```

As introduced in Section 2.2.5, the two policy functions defined above are applied using the dbms_rls.ADD_POLICY function. By linking the functions to the corresponding columns SAL and POSITION, the expected behaviour should be achieved. In reality however, adding the two policy functions in random order causes indeterminate results. The two possible sequences of policy additions and their concrete effect on querying the EMPLOYEE table are depicted in Tables 3.2 and 3.3. Values masked by the VPD engine are plotted crossed out and in gray color. By first adding the restriction on the SAL column, followed by adding the restriction on the POSITION column, the query returns exactly the expected results: no salaries are exposed, the job position of the two "big earners", (Jane Doe=4200, John Hancock=4500) are masked. Compared with this, the results of querying the table after switching the order of policy addition, are unexpected and undesired. Now every single value of the POSITION column is masked. The fact that the ordering of policy addition obviously matters, is a show stopper to the dynamism needed in the proposed prototype application, as it is not possible to control the order of arising access constraints at runtime.

It was not possible to find an explanation for the undeterministic behaviour in official product documentation or other official publications of Oracle Database. The problem was also discussed in an online post in Oracle's official discussion board [12]. None of the participants could justify the behaviour. However, the problem was only observed for column-level policies. Together with the circularity problem discussed above, this is the reason for the idea of a new concept including database views in the access control

Table 3.2: Example query of the EMPLOYEE table
order of policy addition: 1. FILTER_SAL, 2. FILTER_BIG_EARNERS

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
| 1 | Jane | Doe | Sales | ~~Head Of Sales~~ | ~~4200~~ |
| 2 | Max | Power | Sales | Sales Clerk | ~~1800~~ |
| 3 | Frank | Wright | Sales | Sales Clerk | ~~2100~~ |
| 4 | John | Hancock | Accounting | ~~Head Of Accounting~~ | ~~4500~~ |
| 5 | Sandra | Brown | Accounting | Accountant | ~~2200~~ |
| 6 | Linda | Roberts | IT | Developer | ~~2400~~ |

Table 3.3: Example query of the EMPLOYEE table
order of policy addition: 1. FILTER_BIG_EARNERS, 2. FILTER_SAL

| ID | FIRSTNAME | LASTNAME | DEPT | POSITION | SAL |
|---|---|---|---|---|---|
| 1 | Jane | Doe | Sales | ~~Head Of Sales~~ | ~~4200~~ |
| 2 | Max | Power | Sales | ~~Sales Clerk~~ | ~~1800~~ |
| 3 | Frank | Wright | Sales | ~~Sales Clerk~~ | ~~2100~~ |
| 4 | John | Hancock | Accounting | ~~Head Of Accounting~~ | ~~4500~~ |
| 5 | Sandra | Brown | Accounting | ~~Accountant~~ | ~~2200~~ |
| 6 | Linda | Roberts | IT | ~~Developer~~ | ~~2400~~ |

enforcement process. The concrete implementation of this approach will be discussed in the following sections.

### 3.2.2 Structural elements of the Proxy View concept

To provide a solution for the problems arising from usage of Oracle VPD for dFGDAC, the concept of "Proxy Views" is introduced. It involves the creation and maintenance of a set of database views on concrete database objects subject to access control, for the purpose of allowing comprehensive definition of VPD filter predicates. Each protected database table is wrapped by an exclusive database view. An example is depicted in Listing 3.10. The definition of the view simply involves the automatic determination of a unique view name and the unrestricted selection of the underlying database table (in the example, the definition of a Proxy View on the EMPLOYEE table is provided).

Listing 3.10: Proxy View on table EMPLOYEE

```
create view proxy.PVIEW_<HASH>
as
select *
  from EMPLOYEE;
```

As a result, it is possible to set up the following process upon emergence of new database objects:

1. a new database table is created
2. creation of the table is recognized by the system
3. a new Proxy View is created
4. an association from the table to the Proxy View is stored in a maintenance table
5. all access constraints on the concrete database table are actually installed on top of the Proxy View using Oracle VPD
6. all VPD policies have unrestricted access to all concrete tables from within the filter predicates, as no policies are installed on top of actual database tables
7. all queries of protected database tables are rewritten so that they actually query the corresponding Proxy Views; the rewriting module makes use of the association information stored in the Proxy View maintenance table
8. direct access to protected database tables is restricted in order to prevent potential bypassing of the access control system

The advantages gained by establishing the proposed Proxy View system regarding the two problems discussed in the previous section are as follows. The problem of circular VPD policies does not occur as there is the possibility to define filter predicates using references to the data of the original table rather than the actually protected Proxy View. Using subquerying of the protected database table is now fully supported without the described VPD limitation, as the original table is never subject to physical VPD protection. The same applies to the problem regarding already masked columns for usage as access control criterion. As subquerying is now supported, all additional data necessary for determining the applicable filter predicates can be retrieved from the original table, which from VPD's point of view, is fully accessible. It has to be acknowledged that it is still possible to provoke the problems mentioned with the proposed Proxy View system in effect, hence maintenance of dynamic access control policies still needs to be evaluated carefully.

To simplify access to the information, which Proxy View is associated to which concrete protected database object, a maintenance table called PROXY_VIEW is introduced. An example record of the table is depicted in Table 3.4. One record contains the full qualified database names of both the protected database table (ORIGIN_OWNER and ORIGIN_NAME) and the associated Proxy View (PROXY_OWNER and PROXY_NAME). In addition, informative meta data on the association is stored (OBJECT_TYPE of the protected database object and CREATE_DATE of the association).

Table 3.4: Example entry of table PROXY_VIEW

| | |
|---|---|
| **ORIGIN_OWNER** | EXAMPLE |
| **ORIGIN_NAME** | EMPLOYEE |
| **PROXY_OWNER** | PROXY |
| **PROXY_NAME** | PVIEW_<HASH> |
| **OBJECT_TYPE** | TABLE |
| **CREATE_DATE** | 2017-12-04 |

In addition, a simple interface for retrieving stored Proxy View associations is providing, utilizing a stored function called proxy.GET. Listing 3.11 provides the definition of the function's behaviour. The function is primarily used by the rewriting module taking care of exchanging all names of protected database tables in issued SQL queries with their respective Proxy View.

Listing 3.11: Behaviour of proxy.GET

```
proxy.GET( originOwner, originName )

    proxy = table.select( "PROXY_VIEW",
                          originOwner,
                          originName )
    return proxy
```

## 3.2.3 Triggering Proxy View generation

Maintenance of the Proxy View system needs to be triggered every time the structure of the database changes. To this end, Oracle allows for the definition of triggers AFTER CREATE ON DATABASE and BEFORE DROP ON DATABASE. Every time one of these events occurs, the database calls a stored function associated to the event allowing for custom code to be executed as a result of the change.

As a result, a stored function called ON_TABLE_CREATION is introduced. The behaviour of this function is depicted in Listing 3.12. Firstly, a new unused and unique name for the new Proxy View needs to be generated. In a next step, the new Proxy View needs to be generated according to the method introduced in Listing 3.10. Subsequently, the association between the protected database object and the newly generated Proxy View is saved to the PROXY_VIEW table. Finally, row-level security is activated immediately by calling the ENFORCE_POLICIES function as introduced in Section 3.1.4. From this point in time, all queries on the database table need to be rewritten to actually query the respective Proxy View to ensure comprehensive protection of the table's contents.

Listing 3.12: Behaviour of ON_TABLE_CREATION

```
ON_TABLE_CREATION( objectOwner, objectName, objectType )

    [...]

    proxyOwner  = "PROXY"
    proxyName   = "PVIEW_" + hash()

    create_proxy_view( proxyOwner,
                       proxyName,
                       objectOwner,
                       objectName,
                       objectType )

    table.insert( "PROXY_VIEW",
                  proxyOwner,
                  proxyName,
                  objectOwner,
                  objectName,
                  objectType )

    enforce_policies( proxyOwner,
                      proxyName,
                      "*" )
```

To maintain a clean system state, outdated Proxy Views need to be removed properly. To this end, another function called ON_TABLE_DROP is triggered every time a protected table is being dropped from the system. The function comprises the following tasks:

1. determine the corresponding Proxy View by the name of the dropped object
2. drop the determined Proxy View
3. remove the association from the PROXY_VIEW table

## 3.3 Collaborative decision-making: the sovereign schema

One central idea of the proposed dFGDAC prototype is the possibility to change the system during runtime by means of collaborative decision-making. Users are in charge of proposing and deciding on alterations of the system democratically. To this end, a module called “the sovereign schema” is provided. In the following sections both data model and a callable interface of the sovereign schema are presented.

### 3.3.1 Democratic processes modelled in a simple data structure

The basic data structures enabling the democratic administration of the prototype system are preinstalled and hold information on both users and the change process of the system. The three included database tables are CITIZEN, PROPOSAL and DECISION. Figure 3.3 depicts their definitions and associations among each other.

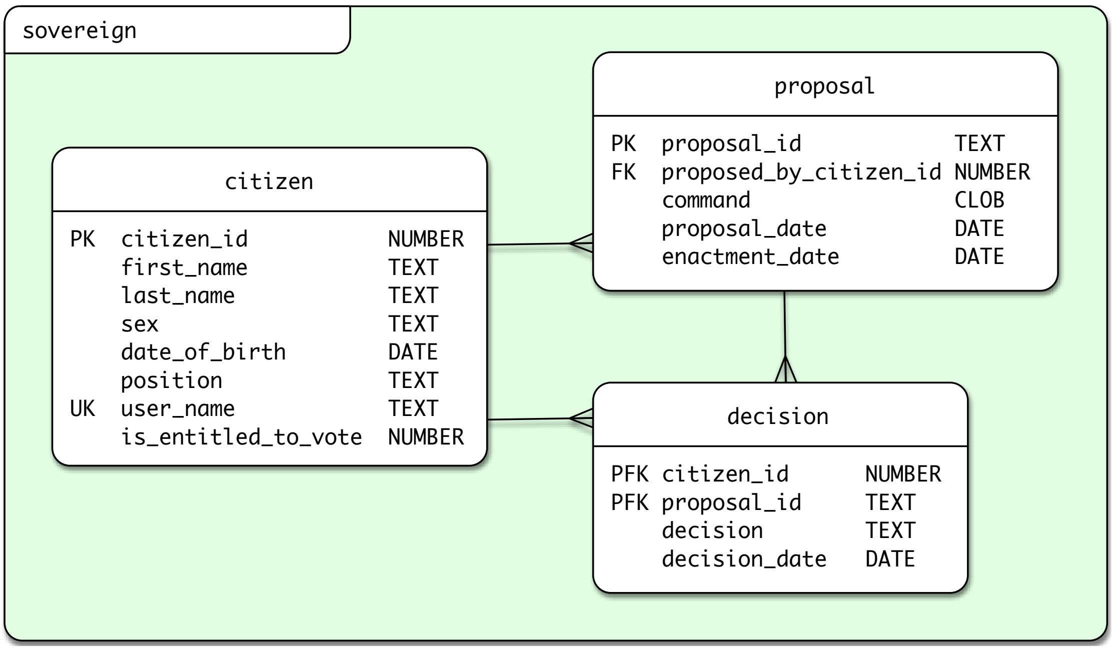


Figure 3.3: Data model of the SOVEREIGN schema

The users of the prototype system are referred to as “citizens”. There is a corresponding table entry in the CITIZEN table for every user, including the name, a unique identification and other personalized and application-specific static data. Users are allowed to propose changes to the prototype system in the form of SQL statements to be executed in the system’s database. To this end, users need to insert a corresponding record into the PROPOSAL table, containing a proposal identification and the SQL statement desired to be executed. In addition, administrative meta data is stored automatically by the responsible insert function. This includes the user, that inserted the proposal, the date of proposal and the date when, if at all, the proposal got enacted. If a proposal gets enacted, the proposed SQL statement is executed in the system’s database. Users of the application, i.e. citizens, are entitled to vote on any proposal made by any citizen. Voting results are stored in the DECISION table, which contains references to the citizen giving his vote, the affected proposal and the decision of the citizen. The decision can be one of “in favor of” or “against”.

For the scope of this work it is assumed, that every user is capable of writing syntactically correct and meaningful SQL statements. For future evolutions of the prototype system it is conceived that proposed SQL statements are generated by some high-level user interface.

Access to the three basic tables of the SOVEREIGN schema is not granted directly to the users of the system. Instead, well-defined interfaces for all possible actions in the context of democratic changes to the system are provided. Exposed actions and their behaviour are described in more detail in the following section.

### 3.3.2 An interface for participating in enactment

The interface for executing democratic actions in the SOVEREIGN schema exposes three crucial procedures. The design aims at simplicity, naturalness and practicability. In Listing 3.13, the PL/SQL based interface definition is depicted.

First of all, the procedure ADD_PROPOSAL provides means for the caller to bring in new proposals to change the system. It takes a mandatory textual PROPOSAL_ID and the COMMAND written in SQL, that should be executed on enactment of the proposal. In a preliminary step, the procedure dynamically checks whether execution is allowed in the current application context. If not, execution is aborted immediately throwing an exception indicating that permission is denied. Otherwise, a new record utilizing the given parameters and the citizen identification retrieved from the application context is inserted to the PROPOSAL table.

The VOTE procedure technically behaves in a similar way. It does the same dynamic execution permission check upfront and, in case of successful execution, leads to a new record being inserted to the DECISION table. The procedure can be used to hand in a decision on a pending proposal. The PROPOSAL_ID parameter identifies the affected proposal, whereas the DECISION parameter carries one of the two allowed values: “in favor of” or “against”.

Listing 3.13: Main procedures: ADD_PROPOSAL, VOTE and ENACT

```
-- inserts the given @proposal_id including @command
-- into the proposal table
procedure add_proposal( proposal_id in varchar2, command in clob );

-- inserts the given @decision on @proposal_id
-- into the decision table
procedure vote( proposal_id in varchar2, decision in varchar2 );

-- enacts given @proposal_id if enactable
-- or all enactable proposals if no @proposal_id is given
procedure enact( proposal_id in varchar2 );
```

Finally, the ENACT procedure is provided as the key component of the sovereign schema. Its behaviour is depicted in Listing 3.14. The procedure takes the identification of the proposal that is desired to be enacted. In a first step, similar to the two functions mentioned earlier, a dynamic execution permission check is done by retrieving and applying the application-specific restriction “USER_ENACT” (starting from Line 3). Subsequently, the current enactment criterion in place is dynamically retrieved utilizing the application-specific restriction “PROPOSAL_ENACT”, applied to the given proposal identification (starting from Line 11). Finally, if all checks are passed successfully, the command associated with the proposal is executed and the proposal is set to be successfully enacted (starting from Line 21). Otherwise, an exception corresponding to the type of the failed check is thrown to the caller. Other exceptions occurring during the execution of any of the three procedures introduced are not handled internally but rather get handed over uncaught to the caller's responsibility. That is, also errors during execution of the proposed SQL command are thrown to the calling user interface.

Listing 3.14: Behaviour of ENACT

```
1  ENACT( proposalID )
2
3       /* check if execution of procedure ENACT is allowed */
4      executionCriterion = table.select( "RESTRICTION",
5                                         "SOVEREIGN",
6                                         "ENACTMENT",
7                                         "USER_ENACT" )
8      if executionAllowed( executionCriterion ) = false
9        raise "PERMISSION DENIED"
10
11     /* check if proposal meets enactment criterion */
12     enactmentCriterion = table.select( "RESTRICTION",
13                                        "SOVEREIGN",
14                                        "ENACTMENT",
15                                        "PROPOSAL_ENACT" )
16     if enactmentAllowed( proposalID,
17                          enactmentCriterion ) = false
18       raise "ENACTMENT CRITERION NOT MET"
19
20
21     /* execute COMMAND and set ENACTMENT_DATE */
22     proposal = table.select( "PROPOSAL",
23                              proposalID )
24     execute( proposal.command )
25     table.update( "PROPOSAL",
26                   proposalID,
27                   enactmentDate = currentTime() )
```

### Initial access constraints

The VPD data model contains each four pre-installed permission and restrictions types, governing authorizations on executing democratic actions. In addition to the standard SQL operations SELECT, INSERT, UPDATE and DELETE new application-specific actions are introduced in the STATEMENT_TYPE column. The list of application-specific statement types comprises USER_ADD_PROPOSAL, USER_VOTE, USER_ENACT and PROPOSAL_ENACT. Each of the democratic actions mentioned above starts with a permission check, making use of the FILTER_CLAUSE of the corresponding application-specific statement type.

Listing 3.15 shows the filter clause that is applied for checking execution permission of all three democratic actions; the initial filter clause is the same for all three actions. It contains a check, whether the user currently logged-in, identified by her @CITIZEN_ID, has the vote entitlement property set to 1 in the corresponding static data record in the CITIZEN table. Technically, there is no difference to traditional restrictions in the same table. The result of the provided filter clause needs to contain a boolean expression.

Listing 3.15: Initial access constraints for execution of ADD_PROPOSAL, VOTE and ENACT

```
1 = ( select nvl( is_entitled_to_vote , 0 )
        from dual
        left join sovereign.citizen
          on citizen_id = @citizen_id )
```

Listing 3.16 shows the initial enactment criterion. The implementation of the ENACT procedure expects the filter clause to return a boolean expression indicating, whether a given yet un-enacted proposal, identified by its PROPOSAL_ID, meets the enactment criterion. Initially, the criterion is defined as follows: Enactment is permitted, if the proposal was not yet enacted and the number of votes in favor of the proposal is equal to the total number of citizens entitled to vote. In other words, proposals need 100 per cent acceptance by all citizens entitled to vote in order to get enacted. In addition, to prevent duplicate executions of proposed SQL queries, already enacted proposals are excluded once and for all from the enactment process.

Listing 3.16: Initial enactment criterion

```
proposal_id in
  ( select proposal_id
      from sovereign.proposal p
     where p.enactment_date is null
       and ( select count(*)
               from sovereign.decision d
              where d.proposal_id = p.proposal_id
                and d.decision    = 'in favor of' )
           =
           ( select count(*)
               from sovereign.citizen
              where is_entitled_to_vote = 1 ) )
```

Note that the initial filter clauses for the application-specific restrictions, as presented above, can be subject to democratic change just like any other part of the application. That is, each enactment criterion and the execution permissions on the procedures providing access to democratic actions can in turn be changed by means of democratic enactment during runtime of the prototype application. For example, it is conceivable that the initial enactment criterion of 100 per cent acceptance is weakened soon after a temporary setup phase of the application in order to facilitate democratic decision-making.

## 3.4 The parliament scenario: a prototype web-based user interface

In order to provide a simple user interface to test the implemented prototype application, a minimal web-based client is presented. The client is developed using the free server-side interpreting script language PHP[1] and deployed to a local Apache web server[2] running a PHP 7 interpreter.

The user interface comprises an input section and a result section. The input section contains a multi-line text input field, where SQL statements can be entered, that need to be queried against the prototype application. Permitted statement types are the standard DML types SELECT, INSERT, UPDATE, DELETE and CALL. The latter one can be used to call stored functions and procedures to be executed in the form "CALL myFunction()". The primary intention is to make it possible to call the procedures implementing democratic actions as introduced in the previous section. Other types of statements are forbidden, such as the DDL (Data Definition Language) commands CREATE, DROP and ALTER as well as Oracle PL/SQL's anonymous blocks (BEGIN ... END). These commands are only allowed to be issued utilizing the democratic enactment system.

In addition, there is a dropdown field containing a selection of all available technical database users. Prior to hitting the "execute" button, the database user needs to be chosen, which should be utilized to establish the connection to the database. The scope of this work excludes the implementation of a session-based login mechanism securing the web-based user interface. Each of the database user entries corresponds to a record in the CITIZEN table, which in turn is used to establish the association from the technical database user to the access permissions the associated citizen is granted. Technically, this is implemented using a database login trigger, which is capable of determining the correct citizen based on the database user and setting the citizen identification in the application context properly, making it available to the subsequently carried-out access control enforcement mechanism.

In a next step, the user interface is responsible for the rewriting of the entered database query in a way, that all database objects mentioned in the query are replaced by their corresponding proxy views. Remember that this is necessary, as all the fine-grained access constraints are enforced on top of the proxy views only, and the concrete objects are fully restricted from being accessed. To this end, all associations from real objects to proxy views are queried from the database. This needs to be done using an administrative application user with permissions to query the PROXY_VIEW table. Subsequently, the SQL query string is parsed for the occurrence of database object names and simply string-replaced with the corresponding proxy view names. The integration or implementation of a dedicated technical SQL parser and query rewriting engine is considered out of scope of

[1]PHP: Hypertext Preprocessor `https://www.php.net/`

[2]Apache HTTP Server Project `https://httpd.apache.org/`

this work. Finally, the successfully rewritten SQL query is executed, subject to dynamic enforcement of all fine-grained data access constraints defined for the citizen logged-in.

The result section contains a status bar and an optional result table. In every case, the status bar is used to provide meaningful meta-information on the outcome of the query attempt. Success messages are displayed in green, while error messages are displayed in red. In case of exceptions, regardless of them being generated by the database core or the parliament application itself, technical error messages and the error stack are displayed in the status bar. If applicable to the statement type of the entered SQL, additional information is provided, such as the number of returned rows by a SELECT statement, or the number of affected rows by statements that successfully manipulated data. In case of a SELECT statement, the tabular results of the SQL query are displayed beneath the status bar.

Figure 3.4 shows a screenshot of the web-based user interface. In the depicted screenshot, a SELECT statement on the CITIZEN table has been executed successfully in the database with the fine-grained access permissions of the database user “max”, which corresponds to the citizen record with ID=“3”, “Max Power”. The test scenario of this example defined the following column-level access constraint: “Citizens may view only their own date of birth”. As a result, the SQL query, when executed with the privileges of database user “max”, hides the values of the DATE_OF_BIRTH column of all the other citizen records.

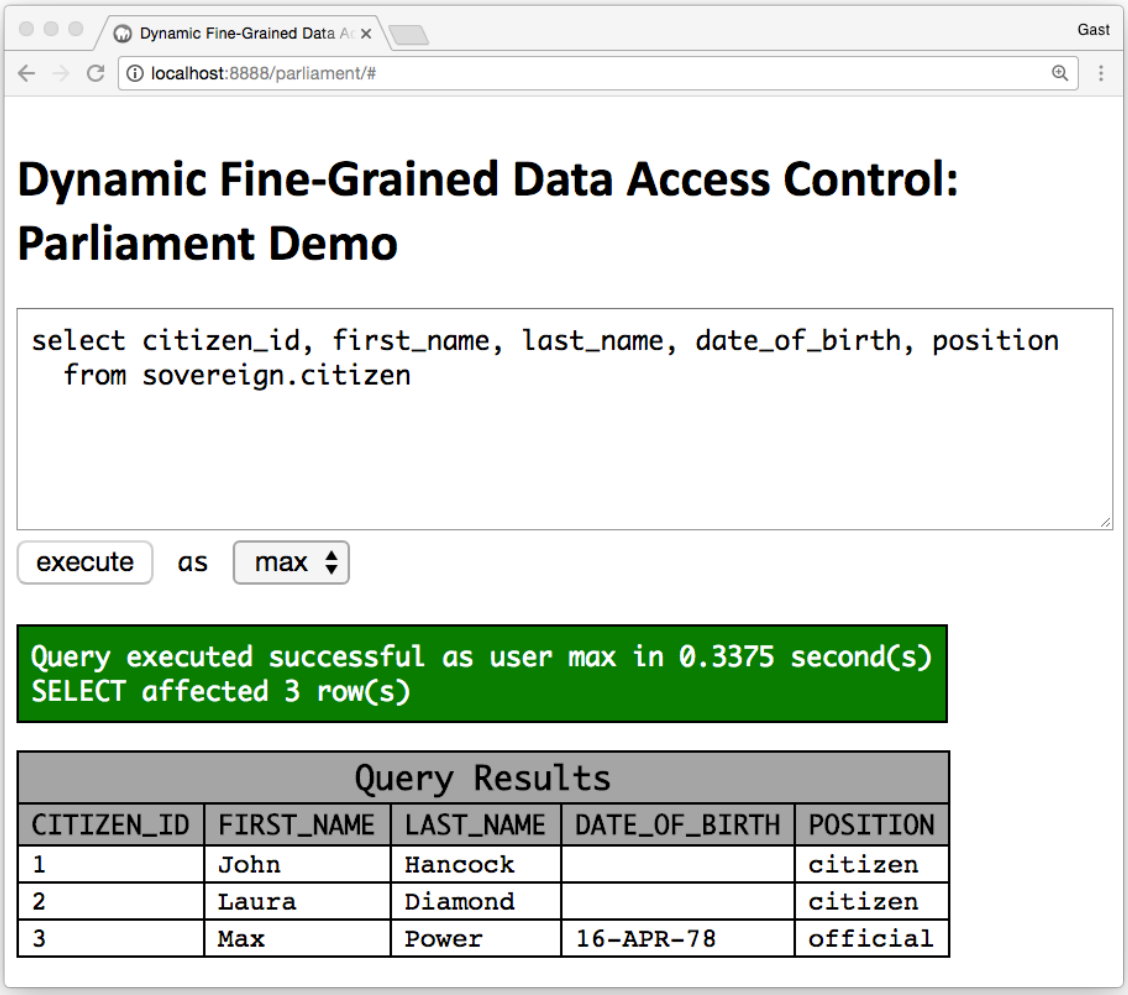
## 3.5 Summary

This chapter presents a new prototype dFGDAC system making use of the Oracle VPD query rewriting technology. The prototype aims at replicating the features of SecSQL. Main components are the VPD schema, the sovereign schema and a web-based user interface.

The VPD schema implements the functional requirements of a FGDAC system. Authorization data is stored in a dedicated data structure. Along with a context awareness module, means for defining and maintaining VPD policies are provided. Technical limitations of VPD require the definition of proxy views, a workaround for implementing the dynamism needed for covering the desired dFGDAC features.

The sovereign schema enables users of the system to propose and decide on alterations of both database structure and access control in a democratic manner. A simple data structure stores information on citizens and their decisions on proposals. The PL/SQL based interface to the sovereign functionality comprises the functions ADD_PROPOSAL, VOTE and ENACT.

A prototype web-based user interface written in PHP allows for issuing SQL requests to the database. Supported actions are DML requests and calling stored procedures. dFGDAC enforced query results are presented to the user immediately.



## Dynamic Fine-Grained Data Access Control: Parliament Demo

```
select citizen_id, first_name, last_name, date_of_birth, position
  from sovereign.citizen
```

```
Query executed successful as user max in 0.3375 second(s)
SELECT affected 3 row(s)
```

Query Results

| CITIZEN_ID | FIRST_NAME | LAST_NAME | DATE_OF_BIRTH | POSITION |
|---|---|---|---|---|
| 1 | John | Hancock | | citizen |
| 2 | Laura | Diamond | | citizen |
| 3 | Max | Power | 16-APR-78 | official |

Figure 3.4: Screenshot of the prototype web-client

CHAPTER 4

# Approaching standardized access policy definition using XACML

So far, only proprietary technologies were discussed as enablers for implementing a dFGDAC system. Neither SecSQL Server, nor the dFGDAC system utilizing Oracle VPD proposed in Chapter 3 contain remarkable approaches of standardization. However, for the intended application scenario of Open Data it is desirable to establish compatibility and interoperability with similar systems. To this end, XACML is evaluated in terms of possible integration with a dFGDAC system.

The following sections are organized as follows. In Section 4.1, an overview of high-level XACML implementations is provided. Section 4.2 presents approaches for direct integration of XACML policies with relational databases. Finally, all the approaches are discussed with respect to their applicability in a dFGDAC scenario in Section 4.3.

## 4.1 High-level XACML implementations

The non-exhaustive list of languages, XACML implementations are available for, contains Java and .NET. Research and concrete hands-on testing of different implementations and APIs showed that only a small number of projects is still supported or under development.

The most promising looking solution, the *Axiomatics Policy Server*[1] is available for both Java and .NET platforms. The product is distributed under a commercial license only and not freely available for testing purposes, neither for a limited amount of time, nor with a limited feature set. For this reason, detailed hands-on testing of this product is not possible in the scope of this work.

[1]`https://www.axiomatics.com`

From the remaining running Open Source XACML API projects the *AT&T XACML 3.0 Implementation*[2] was chosen for performing hands-on testing of a XACML implementation in a high level programming language. Other high-level implementations mentioned in the literature are *SunXACML* and *XEngine* [24].

### 4.1.1 The AT&T XACML 3.0 implementation

The reference implementation provided by AT&T supports the latest XACML 3.0 standard. Crucial implemented components are the XACML Core and XACML Multiple Decision Profile, which allows the PDP to process multiple decisions within one single XACML request issued by the PEP. In addition, it is possible to define custom PIPs which can be used by the PDP to retrieve missing attribute information necessary for deriving decisions.

The steps for developing an example XACML application using the *AT&T XACML 3.0 Implementation* are as follows. First of all, a new *PDPEngine* needs to be created. A new instance can be retrieved using the *PDPEngineFactory*, providing a file containing properties necessary to initialize the PDP. The most important property is the file location of the XACML policies to be enforced. In addition, PIP engines can be defined using the engine type, one of CSV, LDAP or JDBC, and the access parameters, that is, file name of the information point or database connection credentials. Having a *PDPEngine* instance created successfully is sufficient to start issuing requests. To this end, it is necessary to gather all attributes describing subject, resource and action in maps and attach them to a new *StdRequest* object. By calling the *decide* method of the PDP engine handing over the request object, a decision is derived by the PDP and immediately returned represented by a new *Response* object. From this object the decision can be extracted, one of *PERMIT*, *DENY*, *INDETERMINATE* or *NOT APPLICABLE.*

In order to implement attribute-based FGDAC on relational database tables it is necessary to derive a decision for every single cell of the result set. To this end, an implementing XACML policy needs to define rules for accessing the single chunks of data based on the requestors identity, name of the accessed column and contextual information, e.g. values of other columns in the same row. XACML-enforced SQL query execution makes it necessary to fully retrieve the unrestricted data from the database in a first step and then request access control decisions from the PDP for each of the result cells. The PEP then needs to filter or mask out all the cell values for which a *DENY* decision was returned by the PDP. This leads to R times C XACML requests, with R being the number of rows and C the number of columns returned by the SQL query, respectively. Access control decision times increase with the size of the result set.

[2]`https://github.com/att/XACML`

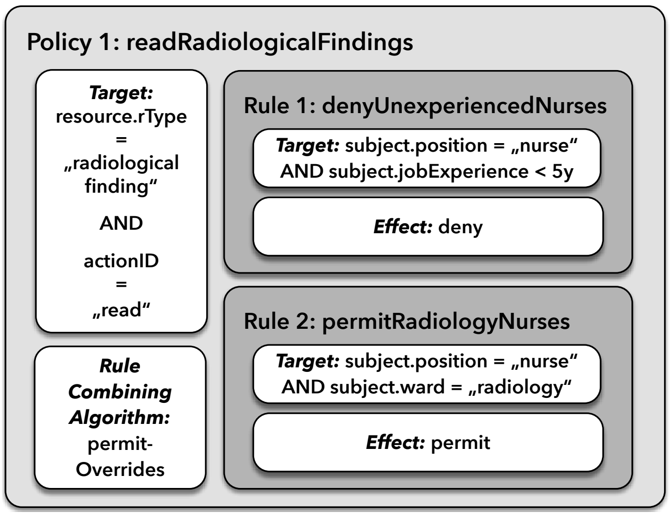

Policy 1: readRadiologicalFindings
Target: resource.rType = „radiological finding" AND actionID = „read"
Rule Combining Algorithm: permit-Overrides
Rule 1: denyUnexperiencedNurses
Target: subject.position = „nurse" AND subject.jobExperience < 5y
Effect: deny
Rule 2: permitRadiologyNurses
Target: subject.position = „nurse" AND subject.ward = „radiology"
Effect: permit

Figure 4.1: Example policy: hospital

## 4.2 Integrating XACML policies with relational databases

A second approach for obtaining standardization using XACML needs to be considered. Instead of using an implementation of a XACML PDP in a high-level programming language, access control policies described in XACML can be directly weaved into the RDBMS. However, there are no suggestions on implementing database access control enforcement given in the core specification of XACML. Thus, research has to devise solutions for making standardized XML based access control policies available from within RDBMS. Jahid et al. propose compilation of XACML policies into native database access control lists [24]. In contrast, El-Aziz and Kannan introduce an algorithm for creating and populating relational authorization tables reflecting the contents of a given XACML policy file [14].

For the sake of describing both approaches in a practical way in the following sections, an example XACML policy describing an imaginary hospital scenario is introduced in Figure 4.1. *Policy 1* targets read access requests for all resources of type "radiological finding". *Rule 1* states that nurses with job experience of less than five years are denied access to radiological findings. Additionally, *Rule 2* expresses that nurses of the radiology ward are permitted to access such findings. Assuming that *Nurse Betty* has been working at the radiology ward for three years, she would be granted access to any radiological finding due to the *PermitOverrides* combining algorithm defined for *Policy 1*, as *Rule 2* returns *Permit* as a result. If the rule combining algorithm would be changed to *DenyOverrides*, *Nurse Betty* would be restricted from access, as *Rule 1* returns a *Deny* decision.

Additionally, a simplified example of XACML code is depicted in Listing 4.1. The rule *denyUnexperiencedNurses* is described using the XML attributes *RuleId* and *Effect*. Moreover, sub-elements are used to describe the target clauses.

Listing 4.1: Example XACML code describing rule *denyUnexperiencedNurses*

```
<Rule RuleId="denyUnexperiencedNurses" Effect="Deny">
    <Target>
        <AnyOf>
            <AllOf>
                <Match MatchId="string-equal">
                    <AttributeValue DataType="string">nurse</AttributeValue>
                    <AttributeDesignator AttributeId="subject.position"
                                         DataType="string"
                                         Category="access-subject" />
                </Match>
                <Match MatchId="integer-less-than">
                    <AttributeValue DataType="integer">5</AttributeValue>
                    <AttributeDesignator AttributeId="subject.jobExperience"
                                         DataType="integer"
                                         Category="access-subject" />
                </Match>
            </AllOf>
        </AnyOf>
    </Target>
</Rule>
```

### 4.2.1 Compiling XACML policies into access control lists

Jahid et al. state that decision time is not satisfactory when implementing database access control using standard XACML PDPs, such as SunXACML [24]. Therefore an approach called *MyABDAC* is proposed, providing compilation of high-level XACML policies into low-level database access control lists.

The basic architecture of *MyABDAC* is depicted in Figure 4.2. The *Policy Compilation Engine* comprises the following modules. The *Policy Parsing Module* parses the contents of a given XACML policy file and creates a tree representing the rules. Subsequently, the *User and Resource Extraction Module* queries the underlying database for all the values of attributes used in any of the previously parsed policies, e.g. the contents of a table describing all the employees of a company, which are the users of the protected application. The retrieved attribute values are attached directly to the rules they are referred from. In a further step, the *Conflict Discovery and Resolution Module* checks for the presence of possibly conflicting rule decisions and applies resolution strategies such as correct ordering, merging and combination of rules. Finally, the *ACL Building Module* is responsible for translating the tree established in the previous steps into common *GRANT* and *REVOKE* statements. As a result, each database resource affected by the access control policy is provided with a complete list of users that are allowed to access it. For this purpose, common databases internally manage access control lists.

Considering the hospital example policy introduced in Section 4.2, database tables containing the information on nurses and radiological findings could be defined as described in Tables 4.1 and 4.2. The **Personnel** table contains data on all the users of

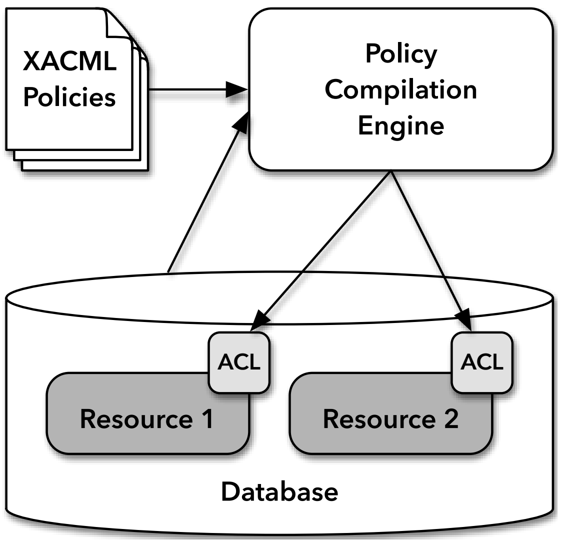


Figure 4.2: Basic architecture of *MyABDAC*, adopted from [24]

Table 4.1: DB table **Personnel**

| **Name** | **Position** | **Ward** | **Experience** |
|---|---|---|---|
| Betty | nurse | radiology | 3y |
| Steve | nurse | emergency | 8y |
| Tom | secretary | office | 10y |

Table 4.2: DB table **RadiologicalFindings**

| **ID** | **Physician** | **Results** | **...** |
|---|---|---|---|
| 27 | Frank | ... | ... |
| 33 | John | ... | ... |

the hospital's management software. The *User and Resource Extraction Module* queries the table for all the information necessary to translate the abstract rules defined in the policy to applicabilities by replacing abstract attribute variables with their concrete values.

Listing 4.2: GRANT/REVOKE statements for table **RadiologicalFindings**

```
grant select on RadiologicalFindings
             to Betty;
revoke select on RadiologicalFindings
              from Steve;
revoke select on RadiologicalFindings
              from Tom;
```

As a result, access control to the **RadiologicalFindings** table is achieved by genera-

Table 4.3: DB table **Policy** for hospital example

| **PolicyID** | **RuleCombiningAlg** | **...** |
|---|---|---|
| readRadiologicalFindings | permitOverrides | ... |

Table 4.4: DB table **Rule** for hospital example

| **RuleID** | **Effect** | **RuleTargetID_FK** | **PolicyID_FK** | **...** |
|---|---|---|---|---|
| denyUnexperiencedNurses | deny | targetDenyUN | readRadiologicalFindings | ... |
| permitRadiologyNurses | permit | targetPermitRN | readRadiologicalFindings | ... |

Table 4.5: DB table **RuleTargetSubject** for hospital example

| **AttributeID** | **MatchOperator** | **AttributeValue** | **RuleTargetID_FK** | **...** |
|---|---|---|---|---|
| subject.position | equals | nurse | targetDenyUN | ... |
| subject.jobExperience | less-than | 5 | targetDenyUN | ... |
| subject.position | equals | nurse | targetPermitRN | ... |
| subject.ward | equals | radiology | targetPermitRN | ... |

tion and execution of the statements by the *ACL Building Module*, as depicted in Listing 4.2. As mentioned above, *Nurse Betty* is granted read access due to her employment in the radiology ward. *Nurse Steve* is not granted access because he works in the emergency ward, even though he would have sufficient job experience. Tom is not granted access because he is not employed as a nurse.

As both policies and the underlying user and resource data may change over time, recompilation of the ACLs has to be considered to maintain consistency. To address performance issues, the system keeps administrative information in memory to react to changes of the access control rules as economically as possible. That is, ACLs are only changed for database objects that are actually affected by any change in the environment, e.g. a change of any user's static data or change of a specific access control policy.

A comparison shows that the proposed system outperforms a prototype implementation with the existing *SunXACML* framework, providing response 6 times faster during runtime. In addition, it is stated that the solution is "reasonably faster" than *XEngine*, another technology providing functionalities of a XACML PDP.

### 4.2.2 Mapping XACML policies to relational tables

In [14], El-Aziz and Kannan argue that the structure of XACML is complex and maintaining policies requires profound knowledge of the language constructs. Therefore an approach is introduced, aiming at minimization of complexity by migrating existing XACML policies to the relational paradigm.

The proposed algorithm parses a given XACML policy file top-down and creates database tables corresponding to the contents of the file. If a policy set element is

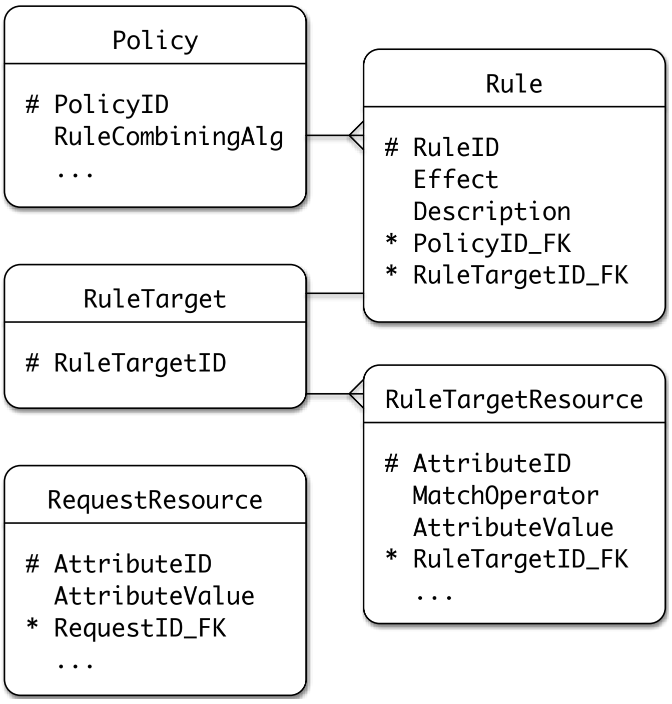


Figure 4.3: Example database structure as created by algorithm proposed in [14]

found for the first time, a **PolicySet** table is created and populated with the element's information. The same procedure is applied for policy and rule elements, likewise. Figure 4.3 shows a minimal example of the created database structure. Selection and renaming of the original columns is done for the sake of comprehensibility. To address the hierarchical structure of the file, foreign keys pointing from tables of lower levels to higher levels are established. For example, the **Rule** table includes a **PolicyID_FK** column storing the ID of the policy it is contained in. In addition, target elements are mapped to one table each per element they occur in (target elements may occur in policy sets, policies and rules). Concrete target matching criteria are stored in tables reflecting their attribute category. For instance, a **RuleTargetResource** table contains the values necessary for matching the resource attribute of a rule target. That is, it defines all the resources a specific rule is applicable to.

Consider the hospital example policy introduced in Section 4.2. Tables 4.3, 4.4 and 4.5 depict an excerpt of the result of applying the algorithm described above for mapping the hospital policy to relational tables. Firstly, a **Policy** table is created and filled with a data set representing the *readRadiologicalFindings* policy. Amongst others, the policy identifier and the chosen rule combining algorithm are stored. Another table, **Rule**, is used to represent both rules the policy contains. The rule-policy relation is established using the **PolicyID_FK** column containing the policy ID *readRadiologicalFindings*. The targets of the rules are referenced via their **RuleTargetID's**, *targetDenyUN* and *targetPermitRN*, respectively. Furthermore, both identifiers and effects of the rules are stored. Finally, the **RuleTargetSubject** table is used for representing the concrete attribute-based target matching information for the rules. Both rules contain two Boolean clauses that are logically ANDed, e.g. *subject.position equals nurse AND subject.jobExperience less-than 5* to match the target of the *denyUnexperiencedNurses* rule. Note that the **MatchOperator** column is not contained in the original system which is only capable of doing simple equality comparisons. It is rather added in this paper to gain the expressiveness necessary for implementing arithmetic comparisons, e.g. less-than or greater-than. Moreover, the original system does not provide a solution for choosing the method of logically connecting multiple clauses used within one target, that is, whether conjunction or disjunction should be used.

In the next step, incoming access requests are stored to the database in a similar way. Each subject, resource, environment and action attributes provided by the request context are mapped to specific request attribute tables corresponding to their categorization. For instance, attributes describing the resource requested are stored in the **RequestResource** table.

Listing 4.3: Search for rules applicable to a specific requested resource

```
select Rule.Effect
  from RequestResource,
       RuleTargetResource,
       Rule
 where RequestResource.RequestID_FK = 1
   and RequestResource.AttributeID =
       RuleTargetResource.AttributeID
   and RequestResource.AttributeValue =
       RuleTargetResource.AttributeValue
   and RuleTargetResource.RuleTargetID_FK =
       Rule.RuleTargetID_FK;
```

Access control enforcement can be implemented by building SQL queries matching policy targets to the concrete values provided in the access request and thereby deriving applicable rules containing either a *permit* or *deny* decision. A minimal example SQL query reflecting an access request is provided in Listing 4.3. The meaning of the query reads as follows: "Find all rules applicable to the resource specified in request 1 and return the value of the **Effect** column". In other words, the rule's target resource attribute must match the resource attribute in the request with ID = 1. The issuer of the access control query is responsible for correct interpretation of the query results.

### 4.2.3 Comparison of approaches

**Access control enforcement** In the simple policy mapping approach, access control enforcement has to be implemented by the database developer making use of the stored, fine-grained authorization data. In contrast, the policy compilation approach hands this responsibility over to the existing, built-in access control system of the database.

**Primary objectives** The mapping approach is limited to proposing an algorithm for making the contents of a XACML policy available to the database. Additionally, a small example for how access control enforcement can be handled is added. In turn, compilation into native database ACL aims at reducing the effort of access control for the developer of a business application as much as possible.

**XACML Language Support** Neither of the two papers explicitly provide any acknowledgements in terms of unsupported XACML language constructs. However, the mapping approach does not provide solutions for implementing target clauses containing multiple Boolean expressions. Moreover, arithmetic comparisons are not supported. In contrast, the compilation approach explicitly mentions arithmetic comparisons in an example policy.

**Evaluation** El-Aziz and Kannan do not provide extensive information on evaluation efforts of their mapping approach. Although an example application implemented using Java and MS Access is mentioned, there are no further details given on performance measures. Jahid et al. comprehensively describe a performance comparison with related implementations, namely SunXACML and XEngine. Results claim that the proposed compilation approach outperforms SunXACML by 600% and is “reasonably faster than XEngine”.

**Practicability** In the mapping approach, little details are given on the intention of the solution. One possible use could be as a XACML preprocessor for subsequent applications not capable of self-employed parsing of raw XACML policies. The compilation approach comes with detailed solution description as well as discussion on performance, security issues and expressiveness of the proposed solution. As recompilation of ACLs has to be triggered when the underlying attributes change, the model is better suited to applications with less dynamic data in terms of change frequency.

**Compatibility** Both solutions involve an algorithm for parsing a given XACML policy and building a desired representation. While in the policy mapping approach the parsing algorithm directly advances to the end-product, the compilation approach uses a similar algorithm for constructing an intermediate result. This intermediate representation could also easily be constructed from the information stored in the tables created by the mapping algorithm.

## 4.3 Discussion: XACML as an enabler for standardization in dFGDAC

In order to discuss the applicability of the XACML standard in our prototype dFGDAC system it is necessary to briefly recall the functional requirements. The most important features are FGDAC, collaborative decision making and dynamic changes of both data model, business data and access control data. It is conceivable that XACML is capable of providing means for the maintenance of access control policies. This could possibly cover the required aspects of FGDAC.

First let us consider the approach of utilizing a high-level XACML implementation as discussed in Section 4.1. Unfortunately there is a show stopper for this approach. The language paradigms of Java and SQL are not compatible in terms of the dynamism and expressiveness necessary for covering the requirements. The FGDAC component in the prototype solution is capable of providing the full expressiveness of the SQL language, as it is possible to define access control policies even at the level of SQL subquerying. In contrast, the inspected AT&T XACML Java implementation only provides minimal support for dynamically retrieving additional authorization data necessary for implementing ABAC. The number of attributes that is likely to be needed during request evaluation needs to be known in advance, as the data sources for the PIP must be defined at application design time. In addition, access control decisions are expensive, as authorizations can only be queried per result cell of the issued SQL query. This leads to R times C necessary XACML requests, with R being the number of rows and C the number of columns in the SQL result set.

The policy compilation approach in the database integration section brings one major disadvantage when applied to an Oracle Database system: the static access privilege system of Oracle is not capable of defining authorization on row- or cell-level. For that reason it is not possible to use the approach for FGDAC with the standard access control system. Although it would be feasible to implement compilation to VPD policies to solve this issue, it is not possible to enhance the situation due to the (again) lack of compatibility and expressiveness of XACML policies compared to the possibilities of SQL subquerying. Another factor worth mentioning is the effort of dynamically maintaining a consistent state reflecting changes to authorization data. Even efficient re-compilation approaches would likely be able to provide not more than eventual consistency.

It is conceivable that the policy mapping approach could be used as a kind of import mechanism. In a scenario including the interchange of authorization data between two similar dFGDAC systems it would be necessary to decide on a common data format. Provided that a way can be found to map the expressive access constraints of our prototype VPD application into XACML, a reverse mapping mechanism could be used to export authorization data from a source system in order to transfer the data to a destination system. However, further research is necessary to find a way to establish compatibility and to overcome the differences in expressiveness. Summarizing the findings it can be stated that neither of the evaluated approaches qualifies for a reasonable integration of XACML as a policy language in our proposed dFGDAC prototype.

## 4.4 Summary

In this chapter, the XACML language is evaluated as an enabler of standardized access control policy maintenance in the proposed dFGDAC system. To this end, three approaches are presented and discussed.

The AT&T XACML 3.0 Java framework is an example of a high-level XACML implementation. It supports static definition of attributes needed for evaluating access control enforced SQL queries. Decisions need to be derived separately for each cell of a SQL's result set, which makes this solution expensive for big result sets.

Another approach is the compilation of high-level XACML policies into low-level database access control lists. Dynamic changes of database structure, users and policies require ongoing recompilation in order to maintain consistency. The approach is not compatible with the coarse grained access control system of Oracle database.

Finally, mapping XACML policies to relational tables is discussed. XACML policies files are parsed and transferred to a SQL-queryable hierarchical data structure. Accordingly, incoming access requests are stored to relational tables which makes it possible to derive access decisions by joining request data with policy data.

The chapter finishes with a comparison and discussion of the three approaches in terms of access control enforcement, primary objectives, XACML language support, evaluation, practicability, compatibility and quality as an enabler for standardization in the proposed dFGDAC system.

CHAPTER 5

# Discussion of dFGDAC solutions

So far we have introduced two prototype implementations of a dFGDAC system. This chapter contains detailed discussions of the evaluated solutions. To establish comparability, performance indicators are identified by considering both functional and non-functional features necessary for implementing a dFGDAC system. These indicators are presented in Section 5.1. The following Sections 5.2 and 5.3 contain a detailed by-feature discussion of SecSQL and the Oracle dFGDAC prototype, respectively. Finally, findings of the discussions are compared in a clear and compact tabular form in Section 5.4.

## 5.1 Performance indicators

To gain an overview on the discussed technologies, a descriptive comparative analysis is performed. Thus, characteristics and features necessary for enabling dFGDAC in the problem domain of SNBG are evaluated. The investigation covers both functional and non-functional performance characteristics. The detailed attributes are further categorized into sub-groups, which are listed in the following.

- Functional attributes
  - Access control mechanism
    - Granularity
    - Support for ABAC
    - Access control enforcement
  - Self-Administrability
    - Support for dynamic data model changes
    - Support for dynamic access control changes

- Non-functional attributes
  - Performance
    - Access control decision times
    - Access control updating times
  - Development enablers
    - Development effort
    - Documentation of technology
    - Licensing
  - Miscellaneous attributes
    - Security
    - Interoperability
    - Extensibility

## 5.2 Discussion of Secure SQL Server

In this section, the results of the descriptive comparative analysis of the SecSQL system are presented. Findings are grouped by the performance indicators identified previously. Each paragraph, additionally grouped into categories, contains the descriptions of one single performance indicator.

### 5.2.1 Functional attributes

#### Access control mechanism

**Granularity** The access control mechanism provides full-featured FGDAC. As dynamic SQL clauses are used to restrict the results of incoming queries, everything that can be expressed using SQL is potentially expressible as an access constraint.

Restricting access is enabled both on different object levels and different statement types. Regarding object levels, authorizations and restrictions can be defined each on relation level, row level or cell level. It should be noted that although cell-level access control is possible in general, only rows are accessible, for which every single requested column is accessible, too. Regarding statement types, access control can be established on SELECT, INSERT, UPDATE and DELETE statements. The MySQL-specific SHOW command, used to present definitions and structure of database objects, is not restricted at all. As a result, every user of the application is allowed to view the entire database structure.

**Support for ABAC** ABAC requires access to all security-related information in the context of a current request. Each of the three main categories of security-related attributes, (1) subject attributes, (2) resource attributes and (3) environment attributes, are accessible during access control enforcement.

Subject attributes can be identified and additionally retrieved from the backend database using the requestor identification stored in the context of any request. Resource attributes belong to the requested database objects and can easily be attached to the restriction clause. Environment attributes can be retrieved from the system context, examples are time and location the incoming request originated from.

**Access control enforcement** Enforcement of the access control rules is executed using the approach of query rewriting. The rewriting algorithm is implemented in .NET using a component specialized on rewriting of MySQL queries. There is no detailed technical information on how the algorithm works internally.

To enforce row-level access control, the approach of cascading projections, as already discussed in detail in Section 2.3.2, is applied. Column-level access control, as mentioned above, is carried out in a strict manner: only rows will be exposed to the requestor, where all desired columns are accessible. The access rules are retrieved from the ELA storage, which comprises tables for *Permissions* and *Restrictions*, the latter which comprises a number of restricting SQL-WHERE clauses.

### Self-Administrability

**Support for dynamic data model changes** The system perfectly implements the requirement of self-administrability by its approach of collaborative decision-making. Users of the system are generally allowed to change the structure of the system, with the following restriction: the decision for a change needs to be reached by collaboration of a defined group of users of the system. In addition, the conditions for a change decision must be met.

Execution of a request for structural changes is carried out by the virtual function SX.ENACT. The function first retrieves the condition definitions and checks if any constraints are violated. If this is not the case, the proposed changes are applied and taken into effect immediately. SX.ENACT allows arbitrary, SQL-expressible changes to the system. It does not take any effort in preventing changes of malicious intent.

**Support for dynamic access control changes** Similar to the support for data model changes, access control can be adapted dynamically to new requirements during runtime. Again, the approach of collaborative decision-making is used to determine changes to the access control system.

The internal representation of the system itself is the only fixed, hardcoded component in SecSQL. The tables *Permissions* and *Restrictions* are used to store positive and negative authorizations, respectively. The latter are described by a potentially unlimited number of WHERE clauses, defined in SQL language. By changing the contents of those two tables, the access control enforcement mechanism immediately applies new policies.

### 5.2.2 Non-functional attributes

#### Performance

**Access control decision times** Neither of the known papers describing SecSQL contains technical evaluation regarding latency in the query process caused by access control enforcement.

**Access control updating times** Neither of the known papers describing SecSQL contains technical evaluation regarding updates of access control rules. As it is sufficient to modify or add few database rows to change the behaviour of access control enforcement, it can be presumed that the delay is negligibly short.

#### Development enablers

**Development effort** Information on the concrete development effort of SecSQL could not be found in the system's documentation. As well-established technologies are used, development and evolution is presumably straight-forward. Especially the wide variety of available .NET-components (e.g. the used query rewriting engine) enables developers to work at a highly efficient level.

**Documentation of technology** Both the .NET framework and the MySQL database comprise extensive online and offline documentation, as well as official product support. In addition, the broad usage of the technologies lead to the emergence of large communities enabling developers to get fast answers on concrete problem statements, including large numbers of tutorials, books and discussion boards.

**Licensing** The basic .NET framework is free of charge. Costs may arise if additional commercial components need to be integrated. The most common development environment, Visual Studio, needs to be purchased using a commercial license. However, there exist numerous alternatives free of charge.

MySQL is available using two different licensing schemes. Commercial products not willing to publish its source code need to purchase a commercial license. Open source products using MySQL can use the database for free.

#### Miscellaneous attributes

**Security** One of the most important security aspects in the context of SNBG is *fair non-repudiable communication.* That is, message transmission needs to be performed ensuring full integrity including the identity of the requestor. To this end, messages contain Base64-encoded digital signatures of the desired access request. Further efforts on security are subject to future research.

**Interoperability** Although the core of SecSQL is written in a proprietary technology, the system comes with good characteristics regarding interoperability. The usage of dynamic binding enables operations with changing components. As SQL was chosen as language for requests, underlying data structures can be exchanged easily. SQL is a widely used standardized query language; every data storage exposing an SQL-queryable interface can potentially be used for retrieving both business data and access rules.

**Extensibility** The .NET solution is built upon a modular structure. Architectural components (e.g. the query rewriting engine, the crypto module) are determined and bound dynamically. Thus, preparations were undertaken to simplify extension of the system or replacement of existing modules.

## 5.3 Discussion of the Oracle dFGDAC prototype

In this section, the results of the descriptive comparative analysis of the proposed Oracle dFGDAC prototype are presented. Findings are grouped by the performance indicators identified previously. Each paragraph, additionally grouped into categories, contains the descriptions of one single performance indicator.

### 5.3.1 Functional attributes

#### Access control mechanism

**Granularity** The implemented dFGDAC prototype provides full support for FGDAC. Utilizing the VPD feature of Oracle Database leads to inheritance of all capabilities of the access control mechanism regarding granularity. By using the proxy view concept some drawbacks of the VPD engine can be wiped out to gain full expressiveness in access constraint definition utilizing SQL statements.

All access constraints are defined either on row level or on column level. Definitions of both levels fit into one single data structure by using either the asterisk symbol or the concrete column name for mapping access constraints to entire rows or single columns, respectively. Technically, row level access control removes rows as a whole from the result set. In contrast, column level access control only masks restricted column values with NULL values. It is possible to obtain result sets solely consisting of NULL values, if all selected columns are masked but there is no row level constraint in place. Column level access control provides means to be extended to cell level access control, as access constraint definitions can refer to data of the same record easily.

Access constraints are not applied recursively within the step of access control enforcement, leading to the advantage, that data already restricted by other rules can still be used inside other access rules. In other words, definers of restrictions can always rely on the fact, that all necessary data is available at any time of access constraint evaluation.

The four default SQL statement types SELECT, INSERT, UPDATE and DELETE can be equipped with separate access constraints. SELECT constraints work in the background and quietly lead to automatically restricted result sets. Violated INSERT constraints lead to a technical exception thrown internally by the VPD engine and directly handed back to the user interface. The exception uses the internal Oracle error code "ORA-28115: policy with check option violation". UPDATE and DELETE constraints behave similar to SELECT constraints. Attempts to updating or deleting restricted data quietly fails and leads to the data simply not being modified.

**Support for ABAC** By using SQL for definition of access constraints, the full expressiveness of the language is made available to the access control mechanism. For that reason it is easy to implement ABAC to its full extent. The three main attribute categories of ABAC, namely subject attributes, resource attributes and environment attributes, are supported as described in the following.

Subject attributes always relate to the user trying to access information. With the citizen identification being stored in the application context, all subject information necessary for evaluating access constraints can be additionally queried from the database. Resource attributes are fully accessible, as all resources are stored in the application database in SQL-queryable relational tables. By default, a small number of environment attributes is provided to the access control mechanism. These are determined at the time a user logs in to the application database and stored in the application context by the prototype's context awareness module. The predefined environment attributes include the name of the client's host machine, the IP address the request originiated from, the operating system user of the client and the time the client has logged in.

**Access control enforcement** Enforcement of access control is handled by a combination of the internal Oracle VPD engine and a rudimentary query rewriting module located in the prototype application's user interface.

VPD also makes use of the query rewriting method to provide access control enforcement. It is activated automatically each time any query is issued to the database, there is no responsibility for the application developer to make sure enforcement works. However, the process of query rewriting is entirely hidden from the database user issuing requests. Besides that, there is no official documentation on how the VPD algorithm internally works in detail. SQL queries get dynamically and implicitly equipped with additional conditions filtering the requested database tables for permitted data only. In case of row level restrictions, the rewritten query is most probably equipped with an extra WHERE clause containing the defined filter clauses. In case of column level restrictions it can be assumed, that the algorithm follows an approach similar to the cascading projections approach introduced in the SecSQL section above.

The introduced concept of proxy views involves the necessity for some preparatory processing steps on issued SQL queries. That is, all references to database objects need to be replaced by references to their corresponding proxy views in order to enable correct access constraint evaluation by the VPD engine. This steps need to be undertaken before the query reaches the database, that is, it is executed within the prototype application's frontend module.

#### Self-Administrability

Dynamic changes of structures within the application database are subject to the system of collaborative decision-making. Changes to both contents and structure need to be agreed on by a defined group of users of the application, all of which is possible during runtime of the prototype application.

The sovereign schema introduced in Section 3.3 provides interfaces for proposing database changes in the form of SQL statements, voting on the proposal and finally "enacting" them by executing the proposed statements in the database. The crucial functionality of this mechanism is provided by the ENACT procedure. It does both checks on execution permission (for the user requesting execution) and enactment permission (whether the enactment criterion is met). With the ENACT procedure it is possible to execute arbitrary types of SQL statements, including all types of statements defined in the Data Definition Language, e.g. CREATE, DROP and ALTER. In addition, it provides means to prevent proposed changes being applied repeatedly.

#### Support for dynamic access control changes

Similar to changes of the database structure, changes of data access constraints are possible only utilizing the implemented feature of collaborative decision-making.

The prototype application continuously listens to data changes in the PERMISSION and RESTRICTION tables and immediately reacts by recompiling and re-establishing VPD access policies in order to comply with the authorization data maintained by the users of the system. Furthermore there are triggers reacting on changes to the data structures as well, making sure that newly created tables are subject to protection instantly after they emerged in the system. The components ENFORCE_POLICIES and CLEANUP_POLICIES, as introduced in Section 3.1.4 take care of maintaining a proper system state by i.e. dropping unnecessary security policies or outdated proxy views.

### 5.3.2 Non-functional attributes

#### Performance

**Access control decision times** Access control is enforced automatically by Oracle's VPD feature on query execution time. Poor overall query performance can be caused by poor access control decision times. The latter depend on several parameters.

First of all, number and complexity of restrictions applicable to an issued request needs to be considered. Computation of concrete WHERE clauses takes linear time with respect to the number of applicable restrictions. In addition, physical and logical organization of the queried table is relevant for access control performance. Table columns used in access control restrictions without any indexation or comparable tuning mechanism are causing less efficient SQL queries after the rewriting phase of VPD. Finally, as it is the case for non-protected systems too, the amount of data within a queried table is of crucial relevance for query performance, which needs to be considered along with table organization, type and frequency of data access.

**Access control updating times** The update process of access constraints is enforced each time the content of the PERMISSION or RESTRICTION tables is changed. Temporal delay is divided into two actions undertaken by the system. First, it takes a (most probably) insignificant amount of time until the trigger listening on changes to the content of the two tables fires. Second, the trigger needs to create a new job in the database's job queue responsible for cleaning up existing, possibly outdated access policies and enforcing new access policies. It is necessary to run these tasks in a separate job because it is not possible to execute DDL code directly from within a trigger's source code. The latter is necessary for cleaning up unused policies.

There is a significant time delay from adding a job to Oracle Databases job queue to its successful execution. The delay depends both on (1) configuration of the job system regarding the polling frequency to check for new jobs and (2) the number of jobs issued at the same time, i.e. a possible blocking time due to competing jobs.

### Development enablers

**Development effort** Development is divided into two different technologies. Firstly, the prototype web-based user interface uses a PHP script consisting of approximately 500 lines of code, most of it necessary for design considerations. It comes with basic frontend functionality only, which makes it easy to maintain or even to be substituted as a whole by a new component using a different technology, provided that it allows for connecting to and querying Oracle Databases.

Secondly, the more complex application backend is provided using Oracle Database, including its feature VPD for supporting FGDAC. The core solution comprises approximately 1500 lines of code. Although this is also a manageable amount of source code, it comes with considerably more complexity compared to the web-based user interface. For instance, overcoming the technical drawbacks of VPD policies with respect to the usage in the domain of dynamic access control, as described in Section 3.2.1, took days of time, effort and endurance. On the other hand, rapid development of dynamic row-level access control is encouraged by the VPD system.

**Documentation of technology** Oracle provides extensive online documentation on all of its products, including Oracle Database in general [31] and its inherent feature Oracle

VPD [39]. In addition, there is an active online community consisting of approximately 500.000 active users [32]. Among others, the community provides technical articles, best-practices and discussions on all Oracle related issues. Finally, Oracle provides technical support on a contractual basis, including personal assistance by Oracle professionals as well as update and patch supply for software products [33].

Similar to Oracle, PHP provides extensive technical online documentation [36], covering, among others, guidance on installation, the PHP language itself and best practices. Unlike Oracle community, the PHP community is spread all over the WWW including uncountable amounts of technical discussion forums.

**Licensing** Oracle database is available in two different editions, Standard and Enterprise [30]. Standard edition covers only a reduced set of features compared to the Enterprise edition. As the feature Oracle VPD is only available in Enterprise edition, purchasing the latter is necessary for commercial use of our prototype application. Prices depend on the type of licensing (per user or per processor) and the edition. Enterprise edition is usually approximately three times as expensive as Standard edition. However, Oracle provides pre-installed VirtualBox virtual machine images including Oracle Database Enterprise edition for research and testing purposes, free-of-charge [34].

PHP 7, used for the web-based user interface, is distributed under an Open Source license and may be used free-of-charge for both commercial and non-commercial use [37].

#### Miscellaneous attributes

**Security** Technical security issues of the proposed prototype solution are not subject to the scope of this work. Future efforts need to address communication between the web-based frontend module and the database application backend.

**Interoperability** The proposed application backend heavily relies on proprietary technology. There is no chance of adding components not developed using technology provided by Oracle. The only exception is the web-based user interface, which could be replaced by any component written in any UI technology capable of issuing SQL queries to a Oracle Database.

**Extensibility** Provided that development sticks to the proprietary technology of Oracle, the proposed solution is highly extensible. The imperative programming language PL/SQL is Turing complete. In addition, PL/SQL allows for the integration of Java components, which potentially enables the system to make use of this high level programming language and all of its power.

## 5.4 Comparison of the discussed technologies

In the previous sections the performance indicators, introduced at the beginning of this chapter, were summarized extensively and in great detail with respect to the technologies under comparison. In the following, the results are displayed in a compact and clear form. Tables 5.1 and 5.2 depict an overview of all functional and non-functional attributes of the considered technologies, respectively.

Table 5.1: Comparison of functional attributes

| Attribute | SecSQL | Oracle dFGDAC prototype |
|---|---|---|
| AC granularity | full FGDAC on all SQL statement types; expressiveness of SQL; entire row restricted if at least one column is restricted | full FGDAC on all SQL statement types; expressiveness of SQL; AC features inherited from VPD, drawbacks wiped out using proxy views; no custom column masking possible |
| AC ABAC Support | subject attributes via requestor ID; resource attributes by referencing queried DB objects; environment attributes available from the system context (e.g. time and location of the request) | subject attributes via citizen ID from the application context; resource attributes by referencing queried DB objects; environment attributes inherited from default Oracle session information (e.g. OS user of the client, IP address of the request) |
| AC enforcement | query rewriting algorithm using .NET component; approach of cascading projections; rules are retrieved from ELA storage containing permissions and restrictions | combination of rudimentary query rewriting and VPD; no official documentation on the internal working method of VPD; references to DB objects are replaced by proxy views; access to new data structures is restricted by default |
| Dynamic model changes | structure changes possible via collaborative decision making; function SX.ENACT allows changes that are expressible using SQL; malicious changes possible | structure changes possible via collaborative decision making; proposals can include everything that is expressible in SQL; votings and enactment using the sovereign schema |
| Dynamic AC changes | access constraint changes possible via collaborative decision making; positive and negative authorization described using SQL WHERE clauses | access constraint changes possible via collaborative decision making; data changes to the PERMISSION and RESTRICTION cause recompilation of VPD policies |

Table 5.2: Comparison of non-functional attributes

| Attribute | SecSQL | Oracle dFGDAC prototype |
|---|---|---|
| Performance: AC decision times | detailed information missing | decision times depend on number and complexity of restrictions (linear effort), organization of queried tables and amount of data stored |
| Performance: AC updating times | detailed information missing; presumably short delays | temporal delay depending on trigger reaction time and configuration of utilized internal job system |
| Development effort | detailed information missing; presumably no overflowing efforts, as well-established technologies are used | simple PHP script (approx. 500 LOCs), more complex Oracle backend (approx. 1500 LOCs); tricky technical drawbacks of Oracle VPD |
| Documentation | extensive documentation for both .NET and MySQL database; large, active online communities; official product support | extensive documentation for both PHP and Oracle Database; active online communities providing discussions an best practices; official product support |
| Licensing | .NET is free of charge, commercial components available; free use of MySQL for OpenSource products, chargeable otherwise | PHP 7 under OpenSource license; Oracle Enterprise edition including VPD feature is expensive, prices depend on type of licensing (per user or per processor); free pre-installed VMs available for testing purposes |
| Security | efforts on fair non-repudiable communication via Base64-encoded signature; open questions for future research | out of scope of this work; open questions for future research |
| Interoperability | dynamic binding is used, operations with changing components possible; data backend can be provided by any technology offering an SQL interface | proprietary technology provided by Oracle, little chance for interaction with components of other technology; exception: frontend module can be written in any language with SQL support |
| Extensibility | prerequisites for extensions created by using modular .NET structure and dynamic binding | highly extensible via PL/SQL; integration of Java components possible |

## 5.5 Summary

This chapter summarizes the features of SecSQL and the proposed Oracle VPD prototype. To establish comparability, performance indicators are identified. Functional performance indicators can be grouped into attributes describing the access control mechanism and the support of self-administrability. Non-functional performance indicators comprise performance, development enablers and miscellaneous other attributes. The following sections discuss the technologies under comparison with regard to the performance indicators. Finally, the results of the discussion are compared in tabular form, again grouped into functional and non-functional performance indicators.

CHAPTER 6

# Conclusion and Future Work

In this thesis several aspects of access control are discussed. While the first chapters introduce concepts and evolution of access control, the majority of the contents focuses on FGDAC. To this end, two dFGDAC systems are discussed and compared. Additionally, the possibility for approaching standardization using XACML is investigated.

The remainder of this chapter is organized as follows. In Section 6.1, general observations along with the topic of dFGDAC are described. Section 6.2 discusses the findings of this thesis. Finally, an outlook on future research is presented in Section 6.3.

## 6.1 General observations

Evolving database technology has enabled applications to store and process practically uncountable amounts of person-related data [5]. One way of storing this data is to use RDBMS, such as Oracle Database. The underlying relational data model allows for a well-structured organization of the information [35]. Depending on the application scenario, smaller or bigger chunks of the overall stored data are sensitive and therefore need to be protected from unauthorized access [5].

Classical relational databases, such as Oracle Database, allow only for a coarse-grained definition of data access control. The most detailed level in this form of access control is the relation/table level. In contrast, FGDAC caters for more detailed authorization constraints. Row-level, column-level and cell-level access control are used to clearly separate allowed from prohibited access to a very detailed extent. Earlier approaches relied on defining rules solely based on the identity of the person or process requesting access. In the 2000s, ABAC emerged as a mechanism for achieving less maintenance-intensive access control systems [23]. ABAC systems prefer the utilization of contextual data rather than identifying data for deriving access decisions.

The concept of dFGDAC comprises the features of both FGDAC and ABAC. In addition, the important aspect of dynamism is added. Constraints of the access control system are not known in advance and can change over the entire life cycle of a dFGDAC application. Even changes to the underlying data structure are not limited to application design time any more. This is the case in the research field of SNBG [35]. The goal is to create structured, machine-readable law under aspects of collaborative decision making. To be concrete, an application scenario is introduced, that allows citizens of a fictive country to collaboratively propose and enact changes to an Open Data system in form of SQL statements. Both structural changes and modifications of the access control mechanism are possible via democratic voting.

## 6.2 Discussion of findings

SecSQL is a prototype implementation of an SNBG system introduced in by Paulin [35]. It covers full support for FGDAC and ABAC. Enforcement of access control is done by a .NET component using the approach of cascading projections. In this step, rules are dynamically retrieved from a persistent rule storage. Dynamic changes to both data model and access constraints are possible using collaborative decision making. Change requests can include anything that is expressible using SQL. The prototype contains miscellaneous additional non-functional features in the areas of security, interoperability and extensibility.

In this work, an Oracle dFGDAC prototype is introduced trying to cover the same feature set, defined by the requirements of SNBG, as SecSQL provides. Most of the functionality is implemented in Oracle PL/SQL except for a small client frontend written in PHP. It was possible to reach full support of FGDAC and ABAC utilizing the VPD feature of Oracle Database. To overcome some drawbacks of VPD limiting the expressiveness of possible access constraints, the concept of Proxy Views was introduced. Proxy Views enable the system to evaluate access requests correctly. Dynamic changes to both data structure and access constraints are possible using a system of collaborative decision making. Users of the application can issue change requests in the provided web-based frontend and subsequently vote on these requests. If a request has reached the necessary approvals, the contained change in form of a SQL statement is executed. An (uncounted) small delay of changing access constraints needs to be accepted due to the complex system of polling database jobs and triggers.

Parts of the efforts on the work were spent to reach the goal of adding standardization aspects to the prototype system. To this end, the integration of XACML, an XML based standard for definition and exchange of access control policies, was targeted. Two approaches were evaluated, and both were considered not applicable for use in the Oracle dFGDAC prototype. Utilizing a high-level XACML implementation in Java fails due to the lack of compatibility between the language paradigms of XACML and SQL. Although technical compatibility between Java and Oracle Database could be reached, the expressiveness of access constraints would suffer from the fact that a direct mapping between the concepts of XACML rules and SQL subqueries is not possible in any conceivable case. The same problem applies to the second approach containing the integration of XACML rules directly to the database system.

The main research question this work aims to answer is: Is it possible to build a dFGDAC system in the application scenario of SNBG utilizing Oracle VPD? Basically the answer is yes, including all the strengths and weaknesses discussed in the scope of this work. A subordinate question is: Is it possible to add aspects of standardization by integrating XACML as language for definition of fine-grained access constraints? This question can be answered with no - the paradigms of SQL-based FGDAC and XACML ABAC policy definition are hard to integrate.

## 6.3 Future work

The proposed Oracle dFGDAC prototype provides a feature set similar to the compared SecSQL system. However, both prototypes use completely different technologies to implement the desired system. To this end, it is questionable how the outcome of this work can be used to gain synergy by combining strengths of both approaches. It is subject to future research to find possible compatibilities to add value to the research field of SNBG.

Focusing on the proposed Oracle dFGDAC prototype, there is a number of future work that needs to be done to further improve the system. The most crucial improvement is the integration of a dedicated query rewriting engine in the interface of the frontend and backend components. The current approach just aims at supporting standard cases and contains known issues that lead to a possibly high rate of errors.

Furthermore, enhancements of the rudimentary web-based frontend need to be implemented. The current user interface only provides the possibility to directly enter and send SQL commands to the dFGDAC system. Future versions could implement some sort of query builder in order to make a step towards better usability for laypersons as intended users of the application.

Focusing on security issues, two more enhancements are still left to do. The current implementation does not contain any secured authentication step at all. For testing purposes only a dropdown field for selecting the desired database connection (including credentials) is provided. In a future version, a session-based login mechanism securing the web-based user interface should be introduced. Besides that, considerations regarding a secure communication between the frontend module and the database application backend need to be undertaken.

# Bibliography

[1] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. “Extending Relational Database Systems to Automatically Enforce Privacy Policies”. In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan.* Ed. by Karl Aberer, Michael J. Franklin, and Shojiro Nishio. IEEE Computer Society, 2005, pp. 1013–1022. ISBN: 0-7695-2285-8. DOI: 10.1109/ICDE.2005.64. URL: http://dx.doi.org/10.1109/ICDE.2005.64.

[2] Mohammad A. Al-Kahtani and Ravi S. Sandhu. “A Model for Attribute-Based User-Role Assignment”. In: *18th Annual Computer Security Applications Conference (ACSAC 2002), 9-13 December 2002, Las Vegas, NV, USA.* IEEE Computer Society, 2002, pp. 353–362. ISBN: 0-7695-1828-1. DOI: 10.1109/CSAC.2002.1176307. URL: http://dx.doi.org/10.1109/CSAC.2002.1176307.

[3] Vijay Alturi and David F. Ferraiolo. “Role-Based Access Control”. In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, pp. 1053–1055. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_829. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5_829.

[4] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. “GEO-RBAC: a spatially aware RBAC”. In: *SACMAT 2005, 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden, June 1-3, 2005, Proceedings.* Ed. by Elena Ferrari and Gail-Joon Ahn. ACM, 2005, pp. 29–37. ISBN: 1-59593-045-0. DOI: 10.1145/1063979.1063985. URL: http://doi.acm.org/10.1145/1063979.1063985.

[5] Ji-Won Byun and Ninghui Li. “Purpose based access control for privacy protection in relational database systems”. In: *VLDB J.* 17.4 (2008), pp. 603–619. DOI: 10.1007/s00778-006-0023-0. URL: http://dx.doi.org/10.1007/s00778-006-0023-0.

[6] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. “Fine Grained Authorization Through Predicated Grants”. In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007.* Ed. by Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K.

Sellis. IEEE Computer Society, 2007, pp. 1174–1183. ISBN: 1-4244-0802-4. DOI: 10.1109/ICDE.2007.368976. URL: http://dx.doi.org/10.1109/ICDE.2007.368976.

[7] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. "Database Access Control and Privacy: Is there a common ground?" In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 2011, pp. 96–103. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper11.pdf.

[8] Frédéric Cuppens and Nora Cuppens-Boulahia. "Modeling contextual security policies". In: *Int. J. Inf. Sec.* 7.4 (2008), pp. 285–305. DOI: 10.1007/s10207-007-0051-9. URL: http://dx.doi.org/10.1007/s10207-007-0051-9.

[9] Frédéric Cuppens and Nora Cuppens-Boulahia. "Multilevel Security Policies". In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, pp. 811–812. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_824. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5_824.

[10] Oracle Developer Community Forum Discussion. *Database Security General - Dynamic VPD policies - works for row-level security, but not for column masking?* last visited on 2017-11-24. URL: https://community.oracle.com/thread/3995566.

[11] Oracle Developer Community Forum Discussion. *Database Security General - VPD - get rid of circular references.* last visited on 2017-11-24. URL: https://community.oracle.com/thread/3994346.

[12] Oracle Developer Community Forum Discussion. *Database Security General - VPD - use already masked column as predicate in another policy function.* last visited on 2017-11-24. URL: https://community.oracle.com/thread/3963300.

[13] PostgreSQL 7.3.21 Documentation. *Table Functions.* last visited on 2016-08-12. URL: https://www.postgresql.org/docs/7.3/static/xfunc-tablefunctions.html.

[14] Abd El-Aziz Abd El-Aziz and Arputharaj Kannan. "XML access control: mapping XACML policies to relational database tables". In: *Int. Arab J. Inf. Technol.* 11.6 (2014), pp. 532–539. URL: http://ccis2k.org/iajit/?option=com_content&amp;task=blogcategory&amp;id=94&amp;Itemid=364.

[15] *eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard.* 22 January 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[16] David Ferraiolo and Richard Kuhn. "Role-Based Access Control". In: *Proceedings of the NIST-NSA National (USA) Computer Security Conference.* 1992, pp. 554–563.

[17] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control.* 2nd ed. Artech House computer security series. Boston, Mass. [u.a.]: Artech House, 2007. ISBN: 1-58053-370-1.

[18] David F. Ferraiolo, Ramaswamy Chandramouli, Rick Kuhn, and Vincent C. Hu. "Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC)". In: *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control, ABAC@CODASPY 2016, New Orleans, Louisiana, USA, March 11, 2016.* Ed. by Elisa Bertino, Ravi Sandhu, and Ram Krishnan. ACM, 2016, pp. 13–24. ISBN: 978-1-4503-4079-3. DOI: 10.1145/2875491.2875496. URL: http://doi.acm.org/10.1145/2875491.2875496.

[19] Stefano Franzoni, Pietro Mazzoleni, and Stefano Valtolina. "Towards a Fine-Grained Access Control Model and Mechanisms for Semantic Databases". In: *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 2007, pp. 993–1000. ISBN: 0-7695-2924-0. DOI: 10.1109/ICWS.2007.176. URL: http://dx.doi.org/10.1109/ICWS.2007.176.

[20] Alban Gabillon. "Web Access Control Strategies". In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, pp. 1368–1371. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_664. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5_664.

[21] Oracle Database Data Cartridge Developer's Guide. *Using Pipelined and Parallel Table Functions.* last visited on 2016-08-12. URL: https://docs.oracle.com/database/121/ADDCI/pipe_paral_tbl.htm.

[22] Raju Halder and Agostino Cortesi. "Observation-based Fine Grained Access Control for Relational Databases". In: *ICSOFT 2010 - Proceedings of the Fifth International Conference on Software and Data Technologies, Volume 1, Athens, Greece, July 22-24, 2010.* Ed. by José A. Moinhos Cordeiro, Maria Virvou, and Boris Shishkov. SciTePress, 2010, pp. 254–265. ISBN: 978-989-8425-22-5. URL: http://www.dsi.unive.it/~cortesi/paperi/icsoft10_a.pdf.

[23] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* National Institute of Standards and Technology. 2014. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf (visited on 07/27/2016).

[24] Sonia Jahid, Carl A. Gunter, Imranul Hoque, and Hamed Okhravi. "MyABDAC: compiling XACML policies for attribute-based database access control". In: *First ACM Conference on Data and Application Security and Privacy, CODASPY 2011, San Antonio, TX, USA, February 21-23, 2011, Proceedings.* Ed. by Ravi S. Sandhu and Elisa Bertino. ACM, 2011, pp. 97–108. ISBN: 978-1-4503-0466-5.

DOI: 10.1145/1943513.1943528. URL: http://doi.acm.org/10.1145/1943513.1943528.

[25] James Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. “A Generalized Temporal Role-Based Access Control Model”. In: *IEEE Trans. Knowl. Data Eng.* 17.1 (2005), pp. 4–23. DOI: 10.1109/TKDE.2005.1. URL: http://dx.doi.org/10.1109/TKDE.2005.1.

[26] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. “Limiting Disclosure in Hippocratic Databases”. In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 108–119. ISBN: 0-12-088469-0. URL: http://www.vldb.org/conf/2004/RS3P3.PDF.

[27] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. “Evolutionary Inference of Attribute-Based Access Control Policies”. In: *Evolutionary Multi-Criterion Optimization - 8th International Conference, EMO 2015, Guimarães, Portugal, March 29 -April 1, 2015. Proceedings, Part I*. Ed. by António Gaspar-Cunha, Carlos Henggeler Antunes, and Carlos A. Coello Coello. Vol. 9018. Lecture Notes in Computer Science. Springer, 2015, pp. 351–365. ISBN: 978-3-319-15933-1. DOI: 10.1007/978-3-319-15934-8_24. URL: https://doi.org/10.1007/978-3-319-15934-8_24.

[28] LouAnna Notargiacomo. “Role-based access control in ORACLE7 and Trusted ORACLE7”. In: *Proceedings of the First ACM Workshop on Role-Based Access Control, RBAC 1995, Gaithersburg, MD, USA, November 30 - December 2, 1995*. Ed. by Charles E. Youman, Ravi S. Sandhu, and Edward J. Coyne. ACM, 1995. ISBN: 0-89791-759-6. DOI: 10.1145/270152.270185. URL: http://doi.acm.org/10.1145/270152.270185.

[29] Lars E. Olson, Carl A. Gunter, William R. Cook, and Marianne Winslett. “Implementing Reflective Access Control in SQL”. In: *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*. Ed. by Ehud Gudes and Jaideep Vaidya. Vol. 5645. Lecture Notes in Computer Science. Springer, 2009, pp. 17–32. ISBN: 978-3-642-03006-2. DOI: 10.1007/978-3-642-03007-9_2. URL: http://dx.doi.org/10.1007/978-3-642-03007-9_2.

[30] Oracle. *Global Pricing and Licensing Welcome Page*. last visited on 2018-01-13. URL: https://www.oracle.com/corporate/pricing/index.html.

[31] Oracle. *Oracle Database Online Documentation 12c Release 1 (12.1)*. last visited on 2018-01-13. URL: https://docs.oracle.com/database/121.

[32] Oracle. *Oracle Developer Community Welcome Page*. last visited on 2018-01-13. URL: https://community.oracle.com.

[33] Oracle. *Oracle Support Welcome Page*. last visited on 2018-01-13. URL: https://support.oracle.com.

[34] Oracle. *Pre-Built Developer VMs (for Oracle VM VirtualBox)*. last visited on 2018-01-13. URL: http://www.oracle.com/technetwork/community/developer-vm/index.html.

[35] Alois Paulin. "Towards a sustainable system for non-bureaucratic government : doctor of science thesis". dissertation. University of Maribor, 2015. URL: http://cobiss6.izum.si/scripts/cobiss?command=DISPLAY&base=99999&rid=18746390&fmt=11&lani=en.

[36] php.net. *PHP English Manual*. last visited on 2018-01-13. URL: http://php.net/manual/en/.

[37] php.net. *PHP Licensing*. last visited on 2018-01-13. URL: http://php.net/license.

[38] Microsoft TechNet Library SQL Server 2008 R2. *Table-Valued User-Defined Functions*. last visited on 2016-08-12. URL: https://technet.microsoft.com/en-us/library/ms191165(v=sql.105).aspx.

[39] Oracle Database Online Documentation 12c Release 1 (12.1) / Database Administration / Database Security Guide. *Using Oracle Virtual Private Database to Control Data Access*. last visited on 2016-08-22. URL: https://docs.oracle.com/database/121/DBSEG/vpd.htm.

[40] Erik Rissanen, David Brossard, and Adriaan Slabbert. "Distributed Access Control Management - A XACML-Based Approach". In: *Service-Oriented Computing, 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24-27, 2009. Proceedings*. Ed. by Luciano Baresi, Chi-Hung Chi, and Jun Suzuki. Vol. 5900. Lecture Notes in Computer Science. 2009, pp. 639–640. ISBN: 978-3-642-10382-7. DOI: 10.1007/978-3-642-10383-4_47. URL: http://dx.doi.org/10.1007/978-3-642-10383-4_47.

[41] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. "Extending Query Rewriting Techniques for Fine-Grained Access Control". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. Ed. by Gerhard Weikum, Arnd Christian König, and Stefan Deßloch. ACM, 2004, pp. 551–562. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007631. URL: http://doi.acm.org/10.1145/1007568.1007631.

[42] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. "Role-Based Access Control Models". In: *IEEE Computer* 29.2 (1996), pp. 38–47. DOI: 10.1109/2.485845. URL: http://dx.doi.org/10.1109/2.485845.

[43] Jie Shi and Hong Zhu. "A fine-grained access control model for relational databases". In: *Journal of Zhejiang University - Science C* 11.8 (2010), pp. 575–586. DOI: 10.1631/jzus.C0910466. URL: http://dx.doi.org/10.1631/jzus.C0910466.

[44] Scott D. Stoller. “Trust Management and Trust Negotiation in an Extension of SQL”. In: *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*. Ed. by Christos Kaklamanis and Flemming Nielson. Vol. 5474. Lecture Notes in Computer Science. Springer, 2008, pp. 186–200. ISBN: 978-3-642-00944-0. DOI: 10.1007/978-3-642-00945-7_12. URL: http://dx.doi.org/10.1007/978-3-642-00945-7_12.

[45] Henk C. A. van Tilborg and Sushil Jajodia, eds. *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5.

[46] Sabrina De Capitani di Vimercati. “Discretionary Access Control Policies (DAC)”. In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, pp. 356–358. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_817. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5_817.

[47] Sabrina De Capitani di Vimercati and Pierangela Samarati. “Mandatory Access Control Policy (MAC)”. In: *Encyclopedia of Cryptography and Security, 2nd Ed.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer, 2011, p. 758. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_822. URL: http://dx.doi.org/10.1007/978-1-4419-5906-5_822.

[48] Zhongyuan Xu and Scott D. Stoller. “Mining Attribute-Based Access Control Policies from Logs”. In: *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*. Ed. by Vijay Atluri and Günther Pernul. Vol. 8566. Lecture Notes in Computer Science. Springer, 2014, pp. 276–291. ISBN: 978-3-662-43935-7. DOI: 10.1007/978-3-662-43936-4_18. URL: http://dx.doi.org/10.1007/978-3-662-43936-4_18.

[49] Eric Yuan and Jin Tong. “Attributed Based Access Control (ABAC) for Web Services”. In: *2005 IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA*. IEEE Computer Society, 2005, pp. 561–569. ISBN: 0-7695-2409-5. DOI: 10.1109/ICWS.2005.25. URL: http://dx.doi.org/10.1109/ICWS.2005.25.

[50] Yue Zhang and James B. D. Joshi. “Administration Model for RBAC”. In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. Springer US, 2009, p. 58. ISBN: 978-0-387-35544-3. DOI: 10.1007/978-0-387-39940-9_1507. URL: http://dx.doi.org/10.1007/978-0-387-39940-9_1507.

[51] Hong Zhu and Kevin Lü. “Fine-Grained Access Control for Database Management Systems”. In: *Data Management. Data, Data Everywhere, 24th British National Conference on Databases, BNCOD 24, Glasgow, UK, July 3-5, 2007, Proceedings*. Ed. by Richard Cooper and Jessie B. Kennedy. Vol. 4587. Lecture Notes in Computer Science. Springer, 2007, pp. 215–223. ISBN: 978-3-540-73389-8. DOI: 10.1007/978-

3-540-73390-4_24. URL: http://dx.doi.org/10.1007/978-3-540-73390-4_24.