UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Intelligent Enforcement of Fine-Grained Access Control Policies for SQL Queries

Máster Interuniversitario en Métodos Formales en Ingeniería Informática

Autor: NGUYEN, PHUOC BAO, HOANG

Tutor: LARA JARAMILLO, JUAN DE
Tutor: GARCIA CLAVEL, MANUEL

Departamento de Ingeniería Informática

Septiembre, 2021

(This page intentionally left blank)

Master Thesis

Intelligent Enforcement of Fine-Grained Access Control Policies for SQL Queries

Author: Hoang Nguyen Phuoc Bao

1st supervisor: Prof. Dr. Juan de Lara
2nd supervisor: Assoc. Prof. Dr. Manuel Clavel

A thesis submitted in fulfillment of the requirements for the Inter-Master Degree on Formal Methods in Computer Science and Engineering at Universidad Autónoma, Complutense and Politécnica de Madrid

September 3, 2021

Acknowledgements

This thesis brings my study at the Inter-Master of Formal Methods in Computer Science and Engineering to an end. I would like to take this opportunity to send my gratitude to everyone that has supported me during this period.

First and foremost, I would like to thank Manuel Clavel, for everything he has done for me: for supporting my decision of taking this master and helping me through the enrollment process. During the time of this thesis, I thank Manuel for every lesson he has taught.

Secondly, I would like to thank Juan, for being my thesis advisor, having meetings with me and giving feedback to improve the thesis artifacts, and for helping me with all the administrative stuff throughout the year.

Thirdly, I would like to thank all of my classmates and lecturers for their understanding and encouragement. To Victor and Roland, thanks for helping me to keep up with the program at the beginning. To Matias and Sajid, for the excellent teamwork spirit you put up during the group project.

Finally, I thank my family, my friends and colleagues who have always been there for me. Cảm ơn ông ngoại. Cảm ơn ba Bảo, mẹ Hương và Minh. Cảm ơn chị Trang đã giúp em ghi danh. Cảm ơn mọi người rất nhiều!

Abstract

Recently, we proposed a model-driven methodology to support fine-grained access control (FGAC) at the database level. More specifically, we defined a model transformation function that inputs SQL queries and generates so-called security-aware SQL stored-procedures. As part of the proposal, we developed an application prototype, called SQL Security Injector (SQLSI). In a nutshell, given an FGAC policy $\mathcal{S}$, a user u, with role r, and a query q, SQLSI automatically generates a stored-procedure sp, such that: if the user u is authorized, according to the FGAC policy $\mathcal{S}$, to execute the query q, then calling the stored-procedure sp will return the same result as executing the query q; otherwise, calling the stored-procedure sp will signal an error.

As expected, there is a performance overhead when executing an (unsecured) SQL query via the corresponding (secured) stored-procedure generated by SQLSI. The reason is clear: FGAC policies require performing authorization checks on the current state of the system, which, in the case of executing SQL queries, will translate into performing authorization checks at execution-time on the database. SQLSI takes care of generating these checks and makes sure that they are called at execution-time when a protected resource is accessed. There are cases, however, where these authorization checks are unnecessary, and, therefore, the performance overhead can and should be avoided. For example: when the database integrity constraints guarantee that these checks will always be successful; or, when the current state of the database guarantees that these checks will be successful in this state.

In this thesis, I propose to develop a formal, model-based methodology for enforcing FGAC policies when executing SQL queries in a smart, efficient way. First of all, I identify situations in which performing authorization checks when executing SQL queries seem unnecessary, based on the invariants of the underlying data model, or based on the known properties of the given scenario, or based on the known properties of the arguments of the given query. Secondly, I formally prove that performing authorization checks when executing SQL queries in these situations is indeed unnecessary. Thirdly, I develop a tool for detecting unnecessary authorization checks when executing SQL queries.

Contents

Chapter 1

Introduction

Software Engineering is the science of engineering software systems [39]. For decades, new programming languages have been developed and new software development methodologies have been proposed, all with the goal of increasing software's reliability, maintainability, and cost-efficiency. One characteristic of a well-engineered software system, which one cannot take for granted, is security. *How to engineer a secure software?*— this is a longstanding question that is drawing more and more attention from the public in recent years.

1.1 Model-Driven Engineering, Model-Driven Security, SecureUML

To engineer a *secure* software, one promising approach is Model-Driven Engineering (MDE) [14], which is a software development methodology that focuses on creating *models* of different views of a system. These models can be created using either domain-specific or general-purpose modeling languages, like the Unified Modelling Language (UML) [50, 51]. Moreover, exceed the scope of documentation, in MDE, system artifacts, like executable code and configuration data, can be automatically generated from these models using either code-generators or transformation tools, like the Xtext Framework [13] or the Epsilon Generation Language [46].

As far as security and reliability are concerned, Model-Driven Security (MDS) [11] is a specialization of MDE for developing *secure* systems. In contrast to the traditional approaches in which security is classified as a non-functional requirement, MDS promotes *security-by-design* as it integrates security into the software design

process at the model level. Informally in MDS, designers (or modelers) specify system models along with their security requirements. This approach, on the one hand, allows security-related artifacts, such as, the access control infrastructures [12], to be automatically generated; and on the other hand, opens room for formal reasoning about the security aspects of the system, for example, analyzing the security policies [9, 10].

SecureUML [32] is the 'de facto' modeling language used in MDS for specifying *fine-grained access control* (FGAC) policies. These are policies that depend not only on static information, namely the assignments of users and permissions to roles, but also on dynamic information, namely the satisfaction of *authorization constraints* by the current state of the system.

> For example, consider a simple eStudent Management System, which consists of students and lecturers. In this system, a typical FGAC policy is: each lecturer can only access the record of its own students; and moreover, to make it more secure: the records can only be accessed in the working time.

1.2 Enforcing FGAC policies on relational database

Recently, we proposed a model-based *characterization* of FGAC *authorization* for SQL queries [2] and developed a model-driven approach for *enforcing* FGAC policies when executing SQL queries [3].

Our approach in [3] consists of defining a function SecQuery() that, given an FGAC policy $\mathcal{S}$ and an SQL select-statement q, generates an SQL stored-procedure SecQuery$(\mathcal{S}, q)$, such that: if a user u, with role r, is authorized, according to $\mathcal{S}$, to execute q, then calling SecQuery$(\mathcal{S}, q)$ with the user u and role r as parameters, i.e. SecQuery$(\mathcal{S}, q)(u, r)$, returns the same result that when u executes q; otherwise, if the user u, with role r, is not authorized, according to $\mathcal{S}$, to execute q, then calling SecQuery$(\mathcal{S}, q)(u, r)$ signals an error. Informally, we can say that SecQuery$(\mathcal{S}, q)$ is the *secured* version of the query q with respect to the FGAC policy $\mathcal{S}$, or that SecQuery$(\mathcal{S}, q)$ *secures* the query q with respect to the FGAC policy $\mathcal{S}$.

In a nutshell, the stored-procedure SecQuery($\mathcal{S}, q$) implements the *authorization checks* that are required to comply with the policy $\mathcal{S}$ when executing the query q. These authorization checks were defined in our model-based characterization of FGAC *authorization* for SQL queries [2]. As mentioned before, FGAC policies depend on the satisfaction of authorization constraints by the current state of the system. Thus, unavoidably, executing the aforementioned FGAC authorization checks causes a performance overhead at execution-time, which will be greater or lesser depending on the "complexity" of the underlying security policy.

> For example, consider the eStudent Management System and the FGAC policy as before: *each lecturer can read the records of its own students*. When a user l, with the lecturer role, attempting to read the record of all students; then according to the given policy, the database system must check, for every student s, whether l is authorized to read the record of s. Again, these computations cannot be pre-computed, and must be executed in the database at execution-time, i.e. every time any lecturer l attempts to read the record of all students.

As an extension to the work presented in [3], during the first semester of the Master studies, I reported on some preliminary experiments that highlighted this execution-time performance issue of the "secured" stored-procedures generated by the function SecQuery() with respect to the execution time of the "unsecured" query. This recent work has been accepted and published in the Springer Nature Computer Science Journal, Volume 2, Issue 5, September 2021 [4]. As part of the future work in this extended version, we have proposed to develop a formal methodology for *optimizing* those "secured" stored-procedures.

> Now, in addition to what has been described in the last example, consider the following database integrity constraint that has been observed: *every lecturer teaches every student*. Indeed, with this newly remark, if it holds for the current database state, then in this state, for every user l, with the lecturer role, and for every student s, l is a lecturer of s, and hence, l is authorized to read s. This leads to the fact that the authorization check for reading the record of the students, in this database state, becomes *unnecessary* as it will always return satisfied, for any input pair of lecturer and student.

In this thesis, I propose a formal, model-based methodology for *optimizing* the stored-procedures generated by the function SecQuery(). Basically, this methodology

consists of "removing" from the stored-procedures generated by SecQuery() those authorization checks that can be proven to be *unnecessary* in a given execution context. To perform these proofs, I propose to use Satisfiability Modulo Theories (SMT) Solvers [8]. As part of the work presented here, I have developed an open-source tool, called `FGAC-Optimizer`, that supports our model-driven methodology for detecting unnecessary FGAC authorization checks. Last but not the least, I showcase the usage of this tool by conducting a non-trivial case study and evaluating its outcome.

Organization The rest of the thesis is organized as follows. In Chapter 2, I review some preliminary knowledge and introduce the running example that will be used throughout the thesis. Next, in Chapter 3, I provide the basic context of my previous work, including the definition of data models, object models and security models for modeling fine-grained access control policies. Also, I recall the important remarks of my model-driven approach for enforcing FGAC policies for SQL queries and describe the performance overhead that comes with it. Then, in Chapter 4, I define formally the methodology for eliminating unnecessary authorization checks. For Chapter 5, I showcase my methodology by proving some cases in which the authorization checks are unnecessary. In Chapter 6, I introduce the tool support and the typical use-case of our methodology. In Chapter 7, I evaluate the tool usage by revisiting the case study in Chapter 5. Finally, in Chapter 8, I discuss the related work, and in Chapter 9, I discuss some limitations of this approach, conclude with some remarks and propose the future work.

Chapter 2

Background

In this chapter, we first give a brief introduction about the Structure Query Language (SQL) and the Role-based Access Control (RBAC) in relational database management systems (RDBMS). Then, we introduce the running example that will be used throughout the thesis. Finally, we introduce the Object Constraint Language (OCL) [36], which is the language used for specifying authorization constraints in our security models.

2.1 Structure Query Language

The Structure Query Language (SQL) is a special-purpose programming language designed for managing data in relational databases [49]. Originally based upon relational algebra and tuple relational calculus, its scope includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is to a great extent a declarative language, it also includes procedural elements. In particular, the procedural extensions to SQL support stored procedures which are routines (like a subprogram in a regular computing language, possibly with loops) that are stored in the database. In these stored-procedures, the temporary tables, which are tables that created and exists temporarily, are particularly useful when one needs to store temporarily a number of records for the next querying/-computing steps. Nowadays, major commercial RDBMS support SQL as a standard language. Specifically, in this thesis, we chose to work with MySQL database management system (MySQL for short).

2.2 Role-Based Access Control vs. Fine-Grained Access Control

The Role-Based Access Control (RBAC) is a security mechanism to assign rights for accessing resources to users via the concept of roles [26]. The RBAC was initially proposed in [25], then its formalization was defined in [47] and finally standardized in [48, 28]. Since then, the RBAC is widely used in most commercial relational database systems [42, 44, 41, 43]. Traditionally, in a database-centric application, using RBAC, users may be assigned to specific roles depending on their responsibilities in that application. Each role can be seen as a collection of permissions and each permission is a restriction of by which actions (e.g. `INSERT`, `SELECT`, `UPDATE`, `DELETE`) and on which resources can be acted, usually on the table- or attribute-level. Furthermore, the RBAC allows roles to be organized in a hierarchy, in which a role can inherit permissions of its children roles.

The Fine-Grained Access Control (FGAC), on the other hand, is restricting access on a finer granularity, i.e. on the row- and cell-level. Moreover, the FGAC allows to define permissions, also known as authorization constraints, based on the current system state. For example, consider the FGAC policy in the previous chapter, with lecturer l is attempting to access the record of student s, in such case, that authorization check needs to inspect the current system state, to check who is currently the students of l and that the access is operated in working hours. Unfortunately, the major commercial RDBMS does not natively support FGAC [34, 38, 24, 52]. As a consequence, enforcing FGAC policies has been performed at the application layer. Although the following opinion deserves a longer discussion, about the importance of supporting FGAC at the database level, we basically agree with [30]:

> "Fine-grained access control [on databases] has traditionally been performed at the level of application programs. However, implementing security at the application level makes management of authorization quite difficult, in addition to presenting a large surface area for attackers —any breach of security at the application level exposes the entire database to damage, Since every part of the application has complete access to the data belonging to every application user."

2.3 Running Example

Consider a simple university data model, namely `Uni`, in Figure 2.1. It consists of two classes, `Student` and `Lecturer`, with one association, `Enrollment`, between them. `Student` and `Lecturer` have attributes `name`, `email`, and `age`. [1] The class `Student` represents the students of a university, with their names, emails, and ages. The class `Lecturer` represents the lecturers of a university, with their names, emails, and ages. The association `Enrollment` represents the links between the students and their lecturers. A student may have none or many lecturers, they are his `lecturers`. And a lecturer may have none or many students, they are his `students`.

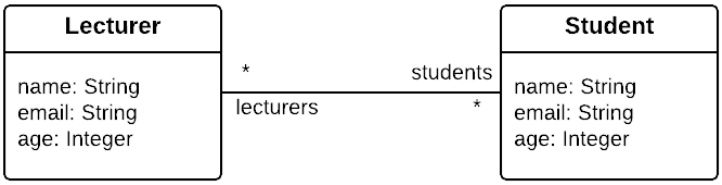


Figure 2.1: UML diagram: Simple University model

2.4 Object Constraint Language

Object Constraint Language (OCL) [36] is a language for specifying constraints and queries using a textual notation. It is a part of the Unified Modeling Language (UML): in Version 1.1 Specification [50], the OCL appears as the standard for specifying invariants, pre- and post-conditions; however, as in Version 2.0 Specification [51], the OCL has been assigned for a broader use, including usage in the definition of specific domain metamodels, model transformation, model testing and validation.

OCL is a strongly-typed language: each expression has either a primitive type, a class type, a tuple type, or a collection type. Collections can be sets, bags, ordered sets and sequences, and can be parametrized by any type, including other collection types. The language provides standard operators on primitive types, tuples, and

[1] Typically when designing this system, one creates a super-class, for example `Person` class, to store the common attributes for `Student` and `Lecturer` class (`name`, `email`, and `age`). However, since our FGAC data model definition, which will be later described in Section 3.1, currently does not support generalization, we intend not to create such a super-class.

collections. Every OCL expression is written in the context of a data model (the so-called *contextual* model).

- For objects, OCL provides a notational style similar to that of object-oriented languages: a dot-operator to access the value of an attribute of the object, or the collection of objects linked with another object at the end of an association. For example, suppose that the contextual model includes a class c with an attribute *at* and an association-end *ase*. Then, if o is an object of the class c, in the given data model instance, the expression $o\,.\,at$ refers to the value of the attribute *at* of the object o, and $o\,.\,ase$ refers to the collection of objects linked to the object o at the association-end *ase*.

- For collections, OCL provides an `allInstances`-operator to collect all objects of a specific class and an arrow-operator "$\rightarrow$" to either access a property of the collection or to iterate over the collection and perform some actions. For example, suppose that the contextual model includes a class c. Then, c.`allInstances()` represents the collection of all objects in class c. Now, suppose that *source* represents a collection. Then, *source*$\rightarrow$`size()` returns the size of this collection, *source*$\rightarrow$`isEmpty()` returns whether this collection is empty, *source*$\rightarrow$`forAll(`$v|$*body*`)` iterates over this collection and checks whether all elements v in this collection satisfy the property stated in *body*, *source*$\rightarrow$`exists(`$v|$*body*`)` iterates over this collection and checks whether there exists at least one element v in this collection satisfies the property stated in *body* and *source*$\rightarrow$`includes(`o`)` iterates over this collection and checks whether object o is included.

Finally, to represent *undefinedness*, OCL provides two constants: null and invalid. Intuitively, null represents an unknown or undefined value, whereas invalid represents an error or an exception.

Example 1. *Consider the university model above as the underlying data model:*

- *To know the number of students, in OCL, one can express as follows:*

  $\underbrace{\texttt{Student.allInstances()}}_{(1)}\underbrace{\rightarrow\texttt{size()}}_{(2)}$

  in which (1) *is a subexpression that applies the* `allInstances`*-operator on the* `Student` *class, and* (2) *is an arrow-operator that applies the size property on* (1).

- *To know whether there is a student that is taught by no lecturer, in OCL, one can express as follows:*

  $$\underbrace{\texttt{Student.allInstances()}}_{(1)}\underbrace{\texttt{→exists}}_{(2)}\texttt{(s|}\underbrace{\texttt{s.lecturers→ size() = 0}}_{(3)}\texttt{)}$$

  in which (1) *is as above, and* (2) *is an operator that iterates over* (1) *and checks whether any student* `s` *has a number of lecturers that is equal to* 0 — *i.e.* `s` *has no lecturer — which is precisely defined in* (3)*.*

- *To know whether every lecturer teaches every student, in OCL, one can express as follows:*

  $$\underbrace{\texttt{Student.allInstances()}}_{(1)}\underbrace{\texttt{→forAll}}_{(2)}\texttt{(s|}$$
  $$\underbrace{\texttt{Lecturer.allInstances()}}_{(3)}\underbrace{\texttt{→forAll}}_{(4)}\texttt{(s|}\underbrace{\texttt{s.lecturers → includes(l)}}_{(5)}\texttt{))}$$

  in which (1) *and* (3) *are sub-expressions that apply the* `allInstances`*-operator on the* `Student` *and* `Lecturer` *class, respectively;* (2) *is an operator that iterates over* (1) *and checks whether for every student* `s` *in* (1)*,* `s` *satisfies that, for every lecturer* `l` *in* (3)*,* `l` *is a lecturer of* `s`*, which is precisely defined in* (5)*.*

△

In what follows, we use the following notation. Let $\mathcal{D}$ be a data model. Then, $\text{Exp}(\mathcal{D})$ denotes the set of OCL expressions whose contextual model is $\mathcal{D}$. Let $\mathcal{O}$ be an instance of $\mathcal{D}$, and let e be an OCL expression in $\text{Exp}(\mathcal{D})$. Then, $\text{Eval}(\mathcal{O}, e)$ denotes the result of *evaluating* e in $\mathcal{O}$, according to the semantics of OCL.

Chapter 3

Previous work

In [3], we proposed a *model-driven* approach for enforcing FGAC policies for SQL queries. This means, in particular, that in our approach the FGAC policies are specified using *models* and that the corresponding policy-enforcement artifacts are *generated* from these models. In our approach, for modeling FGAC policies, we use SecureUML [32], which uses OCL for specifying authorization constraints.

3.1 Modeling FGAC policies

SecureUML [32] is an extension of Role-Based Access Control (RBAC) [27]. In RBAC, permissions are assigned to roles, and roles are assigned to users. However, in SecureUML one can model access control decisions that depend on two kinds of information: namely, static information, i.e., the assignments of users and permissions to roles; and dynamic information, i.e., the satisfaction of *authorization constraints* by the current state of the system.

SecureUML leaves open the nature of the protected *resources*, i.e., whether these resources are data, business objects, processes, controller states, etc. and, consequently, the nature of the corresponding controlled *actions*. These are to be declared in a so-called SecureUML dialect. Particularly, in [4] we model the data to be protected using *classes* and *associations*, and we consider the *read*-actions on these class attributes and association-ends as the actions to be controlled. Finally, we model authorization constraints using OCL boolean expressions.

In this section, we recall the notions of data model, objects model, and security model that we use for modeling fine-grained access control policies.

3.1.1 Data models and object models

Data models specify the *resources* to be protected. Object models (also called scenarios) are instances of data models.

Definition 1. *Let $\mathcal{T}$ be a set of predefined types. A* data model $\mathcal{D}$ *is a tuple $\langle C, AT, AS\rangle$, where:*

- *C is a set of* classes *c.*
- *AT is a set of* attributes *at, $at = \langle atn, c, t\rangle$, where: atn is name of the attribute; c is the class of the attribute; and t is the type of the values of the attribute, with either $t \in \mathcal{T}$ or $t \in C$.*
- *AS is a set of* associations *as, $as = \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$, where: asn is the name of the association; ase_{l} and ase_{r} are the ends of the association as; c_{l} is the class of the objects at the association-end ase_{l}; and c_{r} is the class of the objects at the association-end ase_{r}.*

Without loss of generality, we assume that every class and every association has a unique name, and that, in each class, every attribute also has a unique name.

Example 2. *Consider the* `Uni` *data model in Subsection 2.3, it can be formally defined as follows:*

$$
\begin{aligned}
C = {} & \{\texttt{Student}, \texttt{Lecturer}\},\\
AT = {} & \{\langle \texttt{name}, \texttt{Student}, \mathrm{String}\rangle, \langle \texttt{age}, \texttt{Student}, \mathrm{Integer}\rangle,\\
& \langle \texttt{email}, \texttt{Student}, \mathrm{String}\rangle, \langle \texttt{name}, \texttt{Lecturer}, \mathrm{String}\rangle,\\
& \langle \texttt{age}, \texttt{Lecturer}, \mathrm{Integer}\rangle, \langle \texttt{email}, \texttt{Lecturer}, \mathrm{String}\rangle\}\\
AS = {} & \{\langle \texttt{Enrollment}, \texttt{students}, \texttt{Student}, \texttt{lecturers}, \texttt{Lecturer}\rangle\}
\end{aligned}
$$

For the sake of simplicity, in what follows, we denote by `Student : name` *the attribute* $\langle$`name`, `Student`, String$\rangle$, `Student : age` *the attribute* $\langle$`age`, `Student`, Integer$\rangle$, *and* `Enrollment` *the association* $\langle$`Enrollment`, `students`, `Student`, `lecturers`, `Lecturer`$\rangle$. △

Definition 2. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. An* object model $\mathcal{O}$ *of $\mathcal{D}$ is a tuple $\langle OC, OAT, OAS\rangle$ where:*

- *OC is a set of objects o, $o = \langle oi, c\rangle$, where: oi is the identifier of the object o, and $c \in C$ is the class of the object o.*

- *OAT is a set of* attribute values *atv*, $atv = \langle\langle atn,\ c,\ t\rangle,\ \langle oi,\ c\rangle,\ vl\rangle$, *where:* $\langle atn, c, t\rangle \in AT$, $\langle oi, c\rangle \in OC$, *and vl is a value of the type t.*
- *OAS is a set of* association links *asl*, $asl = \langle\langle asn, ase_{\mathrm{l}},\ c_{\mathrm{l}},\ ase_{\mathrm{r}},\ c_{\mathrm{r}}\rangle,\ \langle oi_{\mathrm{l}},\ c_{\mathrm{l}}\rangle,\ \langle oi_{\mathrm{r}},\ c_{\mathrm{r}}\rangle\rangle$, *where:* $\langle asn, ase_{\mathrm{l}},\ c_{\mathrm{l}},\ ase_{\mathrm{r}},\ c_{\mathrm{r}}\rangle \in AS$, $\langle oi_{\mathrm{l}}, c_{\mathrm{l}}\rangle \in OC$, *and* $\langle oi_{\mathrm{r}}, c_{\mathrm{r}}\rangle \in OC$.

Without loss of generality, we assume that every object has a unique identifier and that the object identifier is of type Integer.

Example 3. *Consider the scenario where there is only one student*—Hoang*, and two lecturers*—Juan *and* Manuel*, with the appropriate age and email. Furthermore, only* Manuel *is teaching* Hoang. *Assuming the name of the object is its identification, this object model can be formally defined as follows:*

$$
\begin{aligned}
OC &= \{\langle \text{Hoang}, \texttt{Student}\rangle, \langle \text{Juan}, \texttt{Lecturer}\rangle, \langle \text{Manuel}, \texttt{Lecturer}\rangle\},\\
OAT &= \{\langle \texttt{Student} : \texttt{name}, \langle \text{Hoang}, \texttt{Student}\rangle, \text{Hoang}\rangle,\\
&\quad\ \langle \texttt{Student} : \texttt{age}, \langle \text{Hoang}, \texttt{Student}\rangle, 25\rangle,\\
&\quad\ \ldots\\
&\quad\ \}\\
OAS &= \{\langle \texttt{Enrollment}, \langle \text{Hoang}, \texttt{Student}\rangle, \langle \text{Manuel}, \texttt{Lecturer}\rangle\rangle\}
\end{aligned}
$$

△

3.1.2 FGAC security models

As described in the previous section, FGAC security models specify fine-grained access control policies for executing *actions* on protected resources. In this section, we recall the *actions* whose execution can be controlled, in our approach, by FGAC policies. Then, we recall the definition of FGAC security models, and their *semantics* i.e., the actions that are authorized to be executed for which users, with which roles, and under which conditions.

In our approach, the notion of role is defined such that: (i) it is associated with a class, i.e. each object of this class is considered as a user, and that (ii) every user can have at most one role. [1] In what follows, we extend the definition of FGAC data model by adding the definition of users-provider class.

[1] A user may have no role. According in our definition, however, this user will not be authorized to access any resource.

Definition 3. *Let $\mathcal{D}$ be a data model, $\mathcal{D} = \langle C, AT, AS \rangle$. Then, we denote by* $\mathrm{Users}(C)$ *the users-provider class of $\mathcal{D}$.*

Next, we define the notion of *read*-actions.

Definition 4. *Let $\mathcal{D}$ be a data model, $\mathcal{D} = \langle C, AT, AS \rangle$. Then,* $\mathrm{Act}(\mathcal{D})$ *denotes the following set of read-actions:*

- *For every attribute $at \in AT$,* $\mathrm{read}(at) \in \mathrm{Act}(\mathcal{D})$.
- *For every association $as \in AS$,* $\mathrm{read}(as) \in \mathrm{Act}(\mathcal{D})$.

Finally, we define our FGAC security model.

Definition 5. *Let $\mathcal{D}$ be a data model. Then, a* security model $\mathcal{S}$ *for $\mathcal{D}$ is a tuple $\mathcal{S} = \langle R, \mathrm{auth} \rangle$, where: R is a set of roles, and* $\mathrm{auth} : R \times \mathrm{Act(D)} \rightarrow \mathrm{Exp}(\mathcal{D})$ *is a function that assigns to each role $r \in R$ and each action $a \in \mathrm{Act}(\mathcal{D})$ an authorization constraint $e \in \mathrm{Exp}(\mathcal{D})$.*

In our approach for modeling fine-grained access control policies, we consider authorization constraints whose satisfaction depends on information related to: (i) the users who are attempting to perform a read-action; (ii) the objects whose attributes are attempted to be read; (iii) the objects between which the links are attempted to be read. By convention, the users referred to in (i) are denoted by the keyword `caller`; the objects referred to in (ii) are denoted by the keyword `self`; and the objects referred to in (iii) are denoted by using as keywords the corresponding association-ends.

Example 4. *Consider the data model in Subsection 2.3, let* `Lecturer` *be the user class, assume that there are two roles, namely:* `Admin` *and* `Lecturer`*, and that the user with the role* `Admin` *can always read any student's age but the user with the role* `Lecturer` *can only read the age of the students whom it teaches. This security model can be formally defined as follows:*

$$R = \{\texttt{Admin}, \texttt{Lecturer}\},$$
$$\mathrm{auth}(\texttt{Admin}, \texttt{Student} : \texttt{age}) = \texttt{true}$$
$$\mathrm{auth}(\texttt{Lecturer}, \texttt{Student} : \texttt{age}) = \texttt{caller.students} \rightarrow \texttt{includes}(\texttt{self})$$

△

Definition 6. *Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = \langle R, \mathrm{auth} \rangle$ be an FGAC security model for $\mathcal{D}$. Let $\mathcal{O} = \langle OC, OAT, OAS \rangle$ be an object model of $\mathcal{D}$. Then,*

- *A user u with role $r \in R$ is authorized, according to $\mathcal{S}$, to read the value of an attribute $at = \langle atn, c, t \rangle$, $at \in AT$, of an object o, $o \in OC$, if and only if:*

$$\text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(at))[\underline{\texttt{self}} \leftarrow o; \underline{\texttt{caller}} \leftarrow u]) = \texttt{true}.$$

- *A user u with role $r \in R$ is authorized, according to $\mathcal{S}$, to read whether an association $as = \langle asn, ase_{\text{l}}, c_{\text{l}}, ase_{\text{r}}, c_{\text{r}} \rangle$, $as \in AS$, links two objects o_{l} and o_{r}, $o_{\text{l}} \in OC$ and $o_{\text{r}} \in OC$, if and only if:*

$$\text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(as))[\underline{\texttt{ase}_{\text{l}}} \leftarrow o_{\text{l}}; \underline{\texttt{ase}_{\text{r}}} \leftarrow o_{\text{r}}; \underline{\texttt{caller}} \leftarrow u]) = \texttt{true}.$$

Example 5. *Consider the* `Uni` *data model in Subsection 2.3, with the object model $\mathcal{O}$ in Example 3 and the FGAC security model in Example 4, let* `Lecturer` *be the user class. Suppose* Juan *and* Manuel *have the role* `Lecturer`*, we say that:*

- Juan *is not authorized, according to the security model in Example 4, to read the age of student* Hoang*, since* Hoang *is not his student, i.e.*

$$\begin{aligned}
&\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Student} : \texttt{age})) \begin{bmatrix} \underline{\texttt{self}} \leftarrow \text{Hoang} \\ \underline{\texttt{caller}} \leftarrow \text{Juan} \end{bmatrix}) \\
&= \text{Eval}(\mathcal{O}, \underline{\texttt{caller}}.\texttt{students} \rightarrow \texttt{includes}(\underline{\texttt{self}}) \begin{bmatrix} \underline{\texttt{self}} \leftarrow \text{Hoang} \\ \underline{\texttt{caller}} \leftarrow \text{Juan} \end{bmatrix}) \\
&= \text{Eval}(\mathcal{O}, \text{Juan}.\texttt{students} \rightarrow \texttt{includes}(\text{Hoang})) \\
&= \texttt{false}.
\end{aligned}$$

- *On the other hand,* Manuel *is authorized, according to the security model in Example 4, to read the age of* Hoang*, since* Hoang *is his student, i.e.*

$$\begin{aligned}
&\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Student} : \texttt{age})) \begin{bmatrix} \underline{\texttt{self}} \leftarrow \text{Hoang} \\ \underline{\texttt{caller}} \leftarrow \text{Manuel} \end{bmatrix}) \\
&= \text{Eval}(\mathcal{O}, \underline{\texttt{caller}}.\texttt{students} \rightarrow \texttt{includes}(\underline{\texttt{self}}) \begin{bmatrix} \underline{\texttt{self}} \leftarrow \text{Hoang} \\ \underline{\texttt{caller}} \leftarrow \text{Manuel} \end{bmatrix}) \\
&= \text{Eval}(\mathcal{O}, \text{Manuel}.\texttt{students} \rightarrow \texttt{includes}(\text{Hoang})) \\
&= \texttt{true}.
\end{aligned}$$

△

In the next section, if the FGAC security model is not clear from the context, then it will be passed as an extra argument to the function auth().

3.2 Enforcing FGAC security model for SQL queries

In this section, we recall our approach for *secury-ing* SQL queries, describe the overview of our implementation of this approach and highlight the performance penalty incurred if we *plainly* apply this implementation.

3.2.1 Secure SQL queries

In [2], we defined the conditions for a user u, with role r, to be authorized to execute an SQL query q according to FGAC security model $\mathcal{S}$. Then, in [3] , we proposed an approach for *enforcing* these conditions when executing SQL queries. Our approach consists in defining a function SecQuery() that, given an FGAC security model $\mathcal{S}$ and an SQL query q, it generates an SQL stored-procedure SecQuery($\mathcal{S}, q$) that implements the *authorization checks* that are required to comply with the policy $\mathcal{S}$ when executing the query q.

More specifically, the stored-procedure SecQuery($\mathcal{S}, q$) takes two arguments, `caller` and `role`, representing respectively, the user executing the query q and the role of this user when executing this query. The body of stored-procedure SecQuery($\mathcal{S}, q$) comprises a list of temporary tables, corresponding to the list of conditions that need to be satisfied for the user `caller`, with the role `role`, to be authorized to execute the query q, according to $\mathcal{S}$. The definition of each temporary table is such that, when attempting to create the table, if the corresponding condition is not satisfied, then an error will be signalled and the table will not be created. If all temporary tables can be successfully created, then the stored-procedure SecQuery($\mathcal{S}, q$) will simply execute q; otherwise, if any of the temporary tables cannot be created, then an error will be signalled. The reason for using temporary tables is to prevent the SQL optimizer from "skipping" (by rewriting the corresponding sub-queries) the authorization checks that SecQuery() generates to guarantee that queries are executed *securely*.

The definition of the function SecQuery() is included in Appendix B. It assumes that data models and object models are implemented in SQL following specific mappings, which are included in Appendix A.

3.2.2 The SQLSI use-case

As part of previous work presented in [4], we developed an application, namely SQLSI, based on the definition of SecQuery(). Figure 3.1 describes the typical use-

case of the SQLSI tool to enforce FGAC policies on a database-centric application. In particular,

1. the modeller defines (or derives) the data-model $\mathcal{D}$ from the application database, defines a FGAC security model $\mathcal{S}$ and collects all "unsecured" SQL queries Q that will be issued in this application,

2. then, the modeller inputs $\mathcal{D}$, $\mathcal{S}$ and Q into the SQLSI tool, which will generate the set of SQL authorization functions corresponding to $\mathcal{S}$; moreover, for every query $q \in Q$, based on function SecQuery(), the SQLSI tool generates a "secured" stored-procedure SecQuery($\mathcal{S}, q$),

3. finally, the modeller takes these newly generated artifacts and sources them into the application database. Furthermore, whenever a query $q \in Q$ is issued, the modeller replaces it by calling the corresponding stored-procedure SecQuery($\mathcal{S}, q$) with proper user and role.

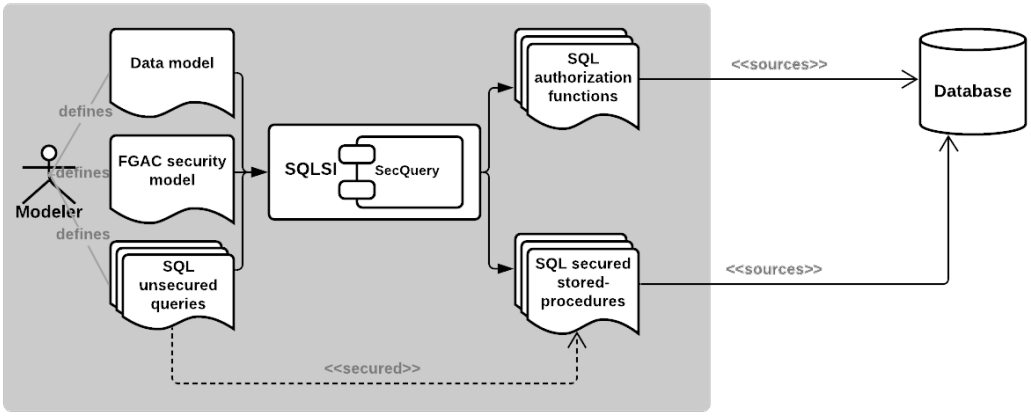


Figure 3.1: The SQLSI use case

3.2.3 Execution-time overhead for secure SQL queries

As mentioned above, fine-grained access control policies depend not only on static information, namely the assignments of users and permissions to roles, but also on dynamic information, namely the satisfaction of authorization constraints on the current state of the system. Unavoidably, executing FGAC-related authorization

checks will cause a performance overhead, greater or lesser depending on the "size" of the database and the "complexity" of the authorization checks. We recall here the experiments reported in [4] about the performance-overhead incurred when executing *securely* queries by calling the corresponding stored-procedures generated by the function SecQuery().

A. Experimental setup
The experiments were conducted on a MySQL Community Server (version 8.0.16) running on a computer with Intel(R) Core(TM), 1.60GHz, and 8 GB RAM. For each experiment, the execution-time reported corresponds to the arithmetic mean of 10 different executions.

B. Data model
The experiments consider the `Uni` data model in Subsection 2.3. Furthermore, let `Lecturer` be the users-provider class.

C. Object models
The experiments consider scenarios with equal number of students and lecturers, where every student is a student of every lecturer. More specifically, for $n > 2$, `Uni`(n) denotes an instance of the data model `Uni` such that: there are exactly n students and n lecturers; students and lecturers have unique names; every lecturer has every student as his/her student, so that the number of enrollments is exactly n^2. Moreover, the experiments consider three distinguished lecturers for all the scenarios `Uni`(n): namely, Trang, Michel and Vinh. They also assume that in all the scenarios no other lecturer is older than Michel.

D. FGAC security models
The experiments consider the following FGAC security models: [2]

- `Sec#1`. There is only one role, namely `Admin`. The policy contains the following clauses: (i) *an admin can know the age of any student*; and (ii) *an admin can know the students of any lecturer*. This policy can be modelled in SecureUML as follows:

  roles = {`Admin`}

[2]For the interested readers, the SQL implementation of these FGAC security models can be found in Appendix E.

auth(Admin, read(Enrollment)) = true
auth(Admin, read(Student:age)) = true

- Sec#2. There is only one role, namely Lecturer. The policy contains the following clauses: (i) *a lecturer can know the age of any student, if no other lecturer is older than he/she is*; and (ii) *a lecturer can know the students of any lecturer, if no other lecturer is older than he/she is*. This policy can be modelled in SecureUML as follows:

  roles = {Lecturer}
  auth(Lecturer, read(Student : age))
  = Lecturer.allInstances()→ select(l|l.age > caller.age)→ isEmpty()
  auth(Lecturer, read(Enrollment))
  = Lecturer.allInstances()→ select(l|l.age > caller.age)→ isEmpty()

- Sec#3. There is only one role, namely Lecturer. The policy contains the following clauses: (i) *a lecturer can know the age of any student, if the student is his/her student*; and (ii) *a lecturer can know the students of any lecturer, if the student is his/her student*. This policy can be modelled in SecureUML as follows:

  roles = {Lecturer}
  auth(Lecturer, read(Student:age)) = caller.students → includes(self)
  auth(Lecturer, read(Enrollment)) = caller.students → includes(students)

E. SQL Queries

The experiments consider the queries Query#1 and Query#2 shown in Figure 3.2, which return, respectively, the number of students whose age is greater than 18, and the number of enrollments.

Query#1	SELECT COUNT(*) FROM Student WHERE age > 18
Query#2	SELECT COUNT(students) from Enrollment

Figure 3.2: Experiments: Queries 1–2.

F. Experimental Results
Here we conduct an experiment on the execution-time of the original query compared to its security-aware stored-procedure, under different configurations. In particular, under

- Security policy `Sec#1`, user: Trang, role: `Admin`.
- Security policy `Sec#2`, user: Michel, role: `Lecturer`.
- Security policy `Sec#3`, user: Vinh, role: `Lecturer`.

Note that, in the original queries, there is no security enforcement. Note also that, in all three configurations, the user with the given role is authorized, according to the given security policy, to execute the secured stored-procedures.

When discussing the experiments, unavoidably, we must make reference to functions — SecQuery(), AuthFunc(), and AuthFuncRole() — whose formal definitions are given in Appendix B. Still, we hope that the informal explanation given below is sufficient for understanding the main outcome, for our present purpose, of these experiments: namely, that *plainly* executing the stored procedures generated by SecQuery() may cause a non-negligibly performance-overhead.

`Query#1.`
According to the definition of SecQuery() in Appendix B, for a policy $\mathcal{S} \in \{\texttt{Sec\#}i \mid 1 \leq i \leq 3\}$, the body of SecQuery($\mathcal{S}$, `Query#1`) contains the following statement: [3]

```
CREATE TEMPORARY TABLE ⌜TempTable(age > 18)⌝ AS (
    SELECT * FROM Student
    WHERE CASE ⌜AuthFunc(S, age)⌝ (Student_id, caller, role)
      WHEN 1 THEN age ELSE throw_error() END as age > 18
);
```

Notice that, to create the temporary table ⌜TempTable(`age` > 18)⌝, for every tuple contained in the table `Student`, the function ⌜AuthFunc($\mathcal{S}$, `age`)⌝ is called. Logically

[3]For the interested readers, the complete SQL implementation of this secured stored-procedure can be found in Appendix E.

then, as shown in Figure 3.3, the execution-time for SecQuery($\mathcal{S}$, `Query#1`) increases depending on the "size" of the table `Student`.

Notice also that, according to the definition of SecQuery(), depending on the role r of the caller, for every student in table `Student`, the function $\ulcorner$AuthFunc($\mathcal{S}$, `age`)$\urcorner$ calls the function $\ulcorner$AuthFuncRole($\mathcal{S}$, `age`, r)$\urcorner$, which in turn calls the function map(auth($\mathcal{S}$, r, read(`age`))), which returns the query in SQL that implements the authorization constraint auth($\mathcal{S}, r,$ read(`age`)). Therefore, the execution-time for SecQuery($\mathcal{S}$, `Query#1`) depends also on the "complexity" of the SQL implementation of the authorization constraint auth($\mathcal{S}, r,$ read(`age`)), since this query will be executed for every student in the table `Student`. In particular, in the experiments reported in [4], the authorization constraint

$$\text{auth}(\texttt{Sec\#3}, \texttt{Lecturer}, \text{read}(\texttt{age})) = \underline{\texttt{caller}}.\texttt{students} \rightarrow \texttt{includes}(\underline{\texttt{self}})$$

is implemented as follows:

```
EXISTS ( SELECT 1 FROM Enrollment e
  WHERE e.lecturers = caller                    (3.1)
    AND e.students = self )
```

Then, in the case of the scenario `Uni`(10^3), when executing the stored-procedure

$$\ulcorner\text{SecQuery}(\texttt{Sec\#3}, \texttt{Query\#1})\urcorner(\text{Vinh}, \texttt{Lecturer}), \tag{3.2}$$

the Query (3.1) will be executed 10^3 times, each time with `caller` replaced by Vinh and `self` replaced by a different student in the table `Student`. Notice also that, each time the Query (3.1) is executed, the clause

```
WHERE e.lecturers = caller
  AND e.students = self
```

will search in a table `Enrollment` that contains 10^6 rows. Not surprisingly, as shown in Figure 3.3, the execution of the (secured) stored-procedure depicted in equation (3.2) in the scenario `Uni`(10^3) takes around 2.5 seconds more than the execution of the (unsecured) query `Query#1`.

`Query#2`.

According to the definition of SecQuery() in Appendix B, for a policy $\mathcal{S} \in \{\texttt{Sec\#}i \mid 1 \leq i \leq 3\}$, the body of $\ulcorner$SecQuery($\mathcal{S}$, `Query#2`)$\urcorner$ contains the following create-statements: [4]

[4]For the interested readers, the complete SQL implementation of this secured stored-procedure can be found in Appendix E.

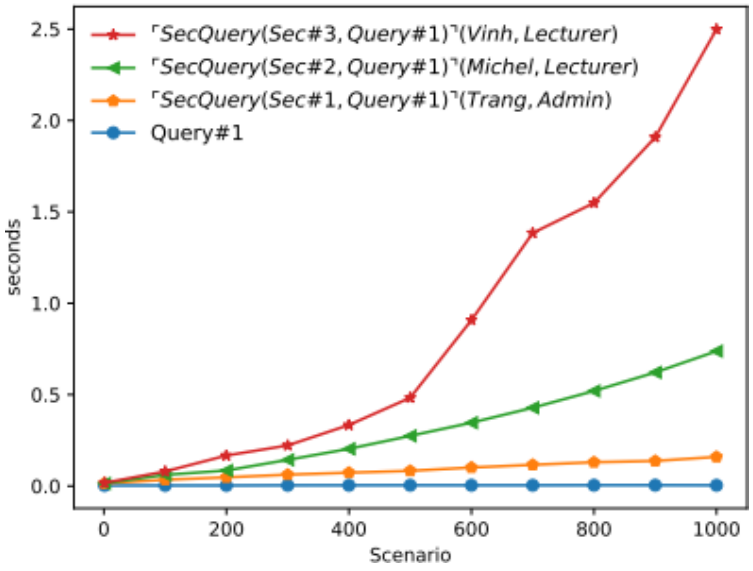


Figure 3.3: `Query#1` experiments. This shows the execution-time (measured in seconds) of `Query#1` with its secured version in different `Uni`(n) scenarios, under different security models `Sec#1`, `Sec#2` and `Sec#3` with the user and role as described above.

```
CREATE TEMPORARY TABLE ⌜TempTable(True)⌝ AS (
  SELECT Student_id AS students, Lecturer_id AS lecturers
  FROM Student, Lecturer
);

CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
  SELECT * FROM ⌜TempTable(True)⌝
  WHERE CASE ⌜AuthFunc(S, Enrollment)⌝ (students,
        lecturers, caller, role)
    WHEN 1 THEN TRUE ELSE throw_error() END as students
);
```

Notice that, to create the table ⌜TempTable(`students`)⌝, for every tuple contained in the table ⌜TempTable(`True`)⌝, which happens to be the Cartesian product of the tables `Student` and `Lecturer`, the function ⌜AuthFunc($\mathcal{S}$, `Enrollment`)⌝ is called. Logically then, as shown in Figure 3.4, the execution-time for SecQuery($\mathcal{S}$, `Query#2`) increases depending on the "size" of the tables `Student` and `Lecturer`.

Notice also that, according to the definition of SecQuery(), depending on the role r of the caller, for every pair student-lecturer contained in the temporary table ⌜TempTable(`True`)⌝, the function ⌜AuthFunc($\mathcal{S}$, `Enrollment`)⌝ calls ⌜AuthFuncRole

($\mathcal{S}$,Enrollment, r)$\urcorner$, which in turn calls the function map(auth($\mathcal{S}, r,$ read(Enrollment))), which returns the query in SQL that implements the authorization constraint auth($\mathcal{S}, r,$ read(Enrollment)). Therefore, the execution-time for SecQuery ($\mathcal{S}$, Query#2) depends also on the "complexity" of the SQL implementation of the authorization constraint auth($\mathcal{S}, r,$ read(Enrollment)), since this query will be executed for every pair student-lecturer in the Cartesian product of the tables Student and Lecturer. In particular, in the experiments reported in [4], the authorization constraint

auth(Sec#3, Lecturer, read(Enrollment))
= caller.students → includes(students)

is implemented as follows:

```
EXISTS ( SELECT 1 FROM Enrollment e
  WHERE e.lecturers = caller                    (3.3)
    AND e.students = students )
```

Then, in the case of the scenario Uni(10^3), when executing the stored-procedure

$$\ulcorner \text{SecQuery}(\texttt{Sec\#3}, \texttt{Query\#2})\urcorner(\texttt{Vinh}, \texttt{Lecturer}) \tag{3.4}$$

the Query (3.3) will be executed 10^6 times, each time with caller replaced by Vinh and students replaced by a student in a different pair student-lecturer in the table Enrollment. Notice also that, each time the Query (3.3) is executed, the clause

```
WHERE e.lecturers = caller
  AND e.students = students
```

will search in a table Enrollment that contains 10^6 rows. Not surprisingly, as shown in Figure 3.4, the execution of the (secured) stored-procedure depicted in equation (3.4) in the scenario Uni(10^3) takes around 8000 seconds more than the execution of the (unsecured) query Query#2.

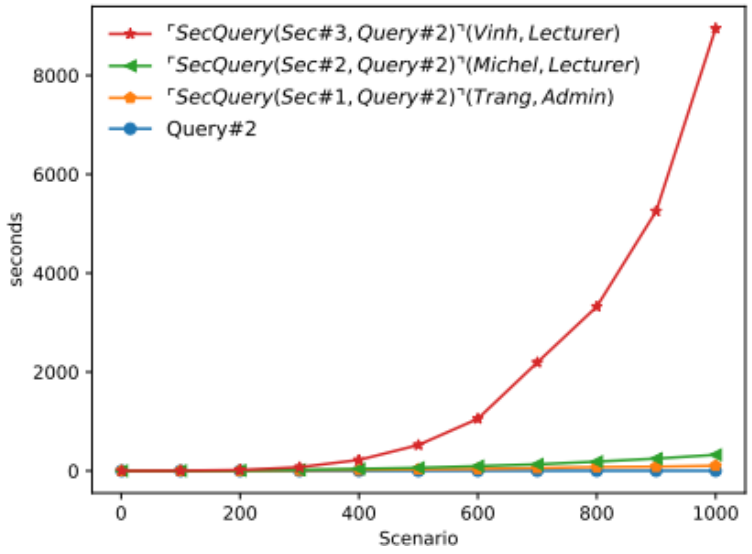


Figure 3.4: `Query#2` experiments. This shows the execution-time (measured in seconds) of `Query#2` with its secured version in different `Uni`(n) scenarios, under different security models `Sec#1`, `Sec#2` and `Sec#3` with the user and role as described above.

Enforcing FGAC policies for SQL queries implies performing authorization checks at execution-time. As the experiments above shown, this enforcement comes with the significant loss in performance. Notice that, there are, however, situations in which (some of) these authorization checks are in fact *unnecessary*. For example, in the experiments reported, for the case of the policy `Sec#3`, if any lecturer attempts to execute `Query#1`, it is unnecessary to perform the corresponding authorization checks (because every student is a student of every lecturer). Similarly, in the case of the policy `Sec#2`, if the lecturer Michel attempts to execute `Query#2`, it is unnecessary to perform the corresponding authorization checks (because no other lecturer is older than Michel). With this in mind, in the next chapter, we present our proposal for intelligent enforcement of FGAC policies.

Chapter 4

Intelligently enforcing FGAC policies for SQL queries

In this chapter we present a model-based methodology for *optimizing* the approach proposed in [3, 4]. In a nutshell, the idea is the following: the function SecQuery() implements the authorization checks by using case-expressions; if these checks (i) can be proved to be trivial, or if they (ii) can be proved to be satisfied given the invariants of the underlying data model, or if they (iii) can be proved to be satisfied given the properties of the objects involved in the authorization request, then the corresponding case-expressions are unnecessary.

4.1 General approach

Recall from Figure 3.1 the use-case for enforcing FGAC policies on database-centric application, the SQLSI tool automatically generates the secured stored-procedures. Here, in Figure 4.1 we extend the use-case with the additional *processing* step of optimizing these generated secured stored-procedures as follows:

1. for each stored-procedure, for each case-expression, the modeller attempts to prove the unncessity of this case-expression:
    - if the corresponding authorization check of this case-expression can be proven to be trivial (as introduced in (i)), then the case-expression can be replaced by the original expression like in the unsecured query.
    - if the corresponding authorization check of this case-expression can be proven to be satisfied given the invariants of the underlying data model

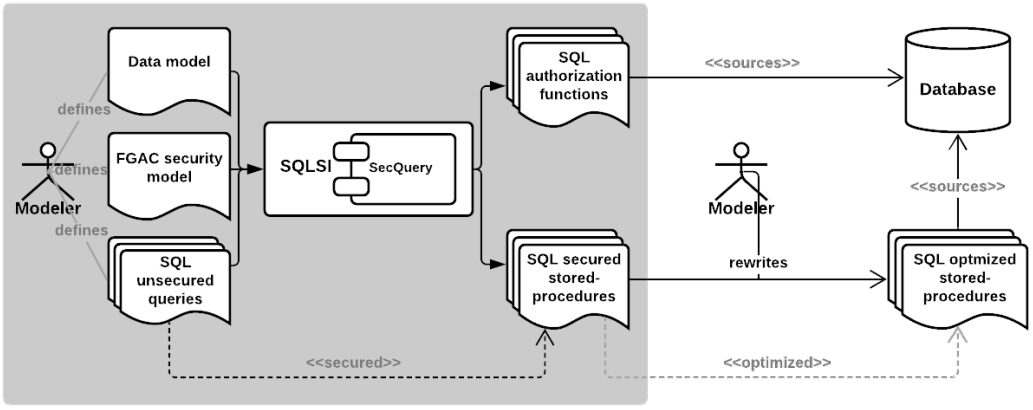


Figure 4.1: The extension of the SQLSI use case. In this extension, instead of sourcing the generated stored-procedures, the modeller performs an additional processing step, optimizing these stored-procedures and then sourcing the optimized ones.

(as introduced in (ii)) or given the properties of the objects involved in the authorization request (as introduced in (iii)), then the modeller can manually rewrite the secured stored-procedure in a way that it makes use of this new information (for example, using an SQL if-then-else statement). Note that, in this case, the invariants or the properties are not automatically derived from the case-statements but rather are introduced in an ad-hoc way by the modeller.

- otherwise, if it cannot be proven to be unnecessary, then we must keep the case-statement as is.

2. after the methodology is applied, for each secured stored-procedure, we obtain an "*optimized-y-secured*" stored procedure. Then, instead of sourcing the SQLSI generated stored-procedure as described in the last use-case, here, the modeller sources the newly rewritten one into the application database.

In the rest of this chapter we introduce step-by-step our approach for proving (i)–(iii) using many-sorted first-order logic (MSFOL) theorem-proving tools, in particular, SMT-solvers. Our approach is based on mappings that have been previously proposed: namely, a mapping from data models to MSFOL theories; a mapping from object models to MSFOL interpretations; a mapping from OCL boolean expressions to MSFOL formulae; a mapping from data models to SQL database schema; and

a mapping from object models to SQL database instances. Below we recall these mappings along with their key properties. Our approach also assumes that the implementation in SQL of the OCL authorization constraints is *correct*, in a sense that will be also formally defined below.

4.2 Different mappings and preliminary remarks

4.2.1 From data models to MSFOL theories

In [19] Dania and Clavel defined a mapping from data models to MSFOL theories. Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. In what follows we denote by map($\mathcal{D}$) the MSFOL theory corresponding to $\mathcal{D}$. In a nutshell, this mapping contains:

- The sort Classifier, representing objects in an instance of $\mathcal{D}$ and two constant symbols, nullClassifier and invalClassifier of sort Classifier, representing null and invalid objects, respectively. In addition, an axiom constraining that nullClassifier and invalClassifier must have different interpretations.
- For every predefined type $t \in \mathcal{T}$, we create a sort t and two constant symbols, namely null$t$ and inval$t$, representing null and invalid value of type t, respectively. In addition, an axiom constraining that null$t$ and inval$t$ must have different interpretations.
- For each class $c \in C$, a unary predicate c : Classifier $\rightarrow$ Bool, representing the definition of c-object in an instance of $\mathcal{D}$. In addition, axioms constraining that nullClassifier and invalClassifier are not of type c and that an object of type c are not of other class types.
- For each attribute $at \in AT$, $at = \langle atn, c, t \rangle$, a function atn_c : Classifier $\rightarrow$ t, representing the values of the attribute at in the objects of an instance of $\mathcal{D}$. In addition, axioms constraining that there is no value of at in the nullClassifier and invalClassifier and that for every valid object of class c, the value of at in that object cannot be invalid.
- For each association $as \in AS$, $as = \langle asn, ase_{\mathrm{l}}, c_l, ase_{\mathrm{r}}, c_r \rangle$, a binary predicate asn : Classifier $\times$ Classifier $\rightarrow$ Bool, representing the definition of association links as between objects in an instance of $\mathcal{D}$. In addition, an axiom constraining that for every link of association as, the left- and the right-end object must be of type c_l and c_r, respectively.

Example 6. *Consider the* `Uni` *data model in Subsection 2.3, which is formally defined in Example 2. Then the mapping generates a MSFOL theory that contains:* [1]

- *The sorts* Classifier, Int *and* String, *to represent objects, integer values and string values, respectively.*
- *The constant* nullClassifier, invalClassifier *and an axiom constraining that these two constants have different interpretation (and similarly, for sort* Int *and* String*):*

$$\text{nullClassifier} \neq \text{invalClassifier}$$

- *For class* `Lecturer`, *the predicate* $\text{Lecturer}(x : \text{Classifier})$ *and two axioms constraining that the* nullClassifier *and* invalClassifier *are not of type* `Lecturer` *(and similarly, for class* `Student`*). In addition, an axiom constraining that a* `Lecturer` *object, cannot be a* `Student` *object (and analogously, for class* `Student`*).*

$$\begin{aligned}
&\text{Lecturer}(\text{nullClassifier}) = \bot \\
&\text{Lecturer}(\text{invalClassifier}) = \bot \\
&\forall x : \text{Classifier}.\ \text{Lecturer}(x) \implies \neg\ \text{Student}(x)
\end{aligned}$$

- *For attribute* `age` *of* `Lecturer`, *the function*

$$\text{age\_Lecturer}(x : \text{Classifier}) : \text{Int}$$

and three axioms constraining that (i) it is invalid to get the age of a null object, (ii) it is invalid to get the age of an invalid object and (iii) the age of a lecturer cannot be invalid (and similarly, for other attributes, for other classes).

$$\begin{aligned}
&\text{age_Lecturer}(\text{nullClassifier}) = \text{invalInt} \\
&\text{age_Lecturer}(\text{invalClassifier}) = \text{invalInt} \\
&\forall x : \text{Classifier}.\ \text{Lecturer}(x) \implies \text{age_Lecturer}(x) \neq \text{invalInt}
\end{aligned}$$

- *For association* `Enrollment`, *the binary predicate*

$$\text{Enrollment}(x : \text{Classifier}, y : \text{Classifier})$$

[1] For the interested readers, the complete theory, written in SMT-LIB language, is depicted in Listing F.1.

and an axiom constraining the type of two association-ends.

$$\forall x : \text{Classifier}.\ \forall y : \text{Classifier}.\\ \text{Enrollment}(x, y) \implies \text{Lecturer}(x) \wedge \text{Student}(y)$$

△

4.2.2 From object models to MSFOL interpretations

Let $\mathcal{D}$ be a data model. Let $\mathcal{O}$ be an object model of $\mathcal{D}$. In what follows we denote by $\text{map}(\mathcal{O})$ the MSFOL interpretation of the theory $\text{map}(\mathcal{D})$ that corresponds to the object model $\mathcal{O}$ according to this mapping.

Example 7. *Consider the object model $\mathcal{O}$ defined in Example 3, denote by $\simeq$ the infix notation of our interpretation function, then* $\text{map}(\mathcal{O})$ *consists of:*

- *The set of* Classifier *objects:* $\{\text{nullClassifier}, \text{invalClassifier}, \text{Hoang}, \text{Juan}, \text{Manuel}\}$. *Moreover,* $\text{nullInt} \simeq -1$, $\text{invalInt} \simeq 0$, $\text{nullString} \simeq$ "A" *and* $\text{invalString} \simeq$ "" *(empty string),*
- *The functions and predicates:*
  - $\text{Lecturer}(x) \simeq \begin{cases} \text{true}, & \texttt{if x = Juan or x = Manuel} \\ \text{false}, & \texttt{otherwise} \end{cases}$,
  - $\text{Student}(x) \simeq \begin{cases} \text{true}, & \texttt{if x = Hoang} \\ \text{false}, & \texttt{otherwise} \end{cases}$,
  - $\text{age\_Student}(x) \simeq \begin{cases} 25 & \texttt{if}, \ \text{x = Hoang} \\ 0, & \texttt{otherwise} \end{cases}$,
  - *. . . (and other attribute functions),*
  - $\text{Enrollment}(x, y) \simeq \begin{cases} \text{true}, & \texttt{if x = Manuel and y = Hoang} \\ \text{false}, & \texttt{otherwise} \end{cases}$.

△

4.2.3 From OCL boolean expressions to MSFOL formulae

In [19] Dania and Clavel also defined a mapping $\text{map}_{\text{true}}()$ from OCL boolean expressions to MSFOL formulae. In particular, let $\mathcal{D}$ be a data model, $\mathcal{O}$ be an object

model of $\mathcal{D}$, and $exp \in \mathrm{Exp}(\mathcal{D})$ be a boolean expression. Then, the following holds:

$$\begin{gathered} \mathrm{map}(\mathcal{O}) \models \mathrm{map}_{\mathrm{true}}(exp) \\ \Updownarrow \\ \mathrm{Eval}(\mathcal{O}, exp) = \texttt{true} \end{gathered}$$

Note that the mapping $\mathrm{map}_{\mathrm{true}}()$ includes an auxiliary mapping called $\mathrm{map}_{\mathrm{def}}()$ for generating additional formulae and constraints (if any) needed by $\mathrm{map}_{\mathrm{true}}()$. More specifically,

1. if the OCL expression *exp* contains a literal *lit* of type t as subexpression, then $\mathrm{map}_{\mathrm{true}}(exp)$ includes an additional constraint, generated by $\mathrm{map}_{\mathrm{def}}(exp)$, stating that the interpretation of the null value and invalid value of t differ from *lit*.

2. if the OCL expression *exp* contains a non-boolean expression exp' as subexpression, then $\mathrm{map}_{\mathrm{true}}(exp)$ includes a predicate for exp' (called temp), and an additional formula, generated by $\mathrm{map}_{\mathrm{def}}(exp)$, defines the meaning of newly created predicate.

In what follows, unless explicitly stated, applying $\mathrm{map}_{\mathrm{true}}()$ on an OCL expression involves calling $\mathrm{map}_{\mathrm{def}}(exp)$.

Example 8. *Consider the* `Uni` *data model in Subsection 2.3, given the OCL expression exp:*

$\texttt{Student.allInstances}() \rightarrow \texttt{select}(\texttt{s}|\texttt{s.age} \geq \texttt{19}) \rightarrow \texttt{isEmpty}()$

Then, $\mathrm{map}_{\mathrm{true}}(exp)$ *generates the following:* [2]

- *Note that,* $\texttt{Student.allInstances}() \rightarrow \texttt{select}(\texttt{s}|\texttt{s.age} \geq \texttt{19})$ *is a subexpression in exp, then* $\mathrm{map}_{\mathrm{def}}(exp')$ *includes a predicate* $\mathrm{temp}(x : \mathrm{Classifier})$ *and generates the following formula to define* $\mathrm{temp}()$*:*

$$\begin{aligned} \forall s : \mathrm{Classifier}.\ \mathrm{temp}(s) \iff & \mathrm{Student}(s) \wedge (\mathrm{age\_Student}(s) \geq 19) \\ & \wedge \neg\ (\mathrm{age\_Student}(s) = \mathrm{nullInt} \\ & \quad \vee s = \mathrm{nullClassifier} \vee s = \mathrm{invalClassifier}) \end{aligned}$$

  follows by $\mathrm{map}_{\mathrm{true}}(exp)$*:*

$$\forall x : \mathrm{Classifier}.\ \neg\ \mathrm{temp}(x) \tag{4.1}$$

[2] For the interested readers, the complete theory generated for this OCL expression, written in SMT-LIB language, is depicted in Listing F.2.

- *Note also that, since* 19 *is an integer literal, then* $\text{map}_{\text{def}}(exp)$ *includes the constraints:*

$$\text{nullInt} \neq 19 \wedge \text{invalInt} \neq 19$$

Now, consider the object model $\mathcal{O}$ *defined in Example 3.*

- *In this object model, there is only one Student, namely* Hoang *with the age of* 25*. Therefore, the result of evaluating exp in* $\mathcal{O}$ *is* false*, i.e.* $\text{Eval}(\mathcal{O}, exp) = \text{false}$*.*

- *On the other hand, consider the interpretation of this object model shown in Example 6. In this interpretation, there are 5* Classifier *objects in total, namely* nullClassifier*,* invalClassifier*,* Hoang*,* Juan *and* Manuel*. Since* Student(Hoang) = true*,* age_Student(Hoang) = 25*,* 25 ≠ nullInt*,* Hoang ≠ nullClassifier *and* Hoang ≠ invalClassifier*, we obtain that* temp(Hoang) = true*. Indeed, this contradicts with the axiom in* (4.1)*, therefore, the interpretation of* O *cannot satisfy the formulae in* $\text{map}_{\text{true}}(exp)$*, i.e.* $\text{map}(\mathcal{O}) \not\models \text{map}_{\text{true}}(exp)$*.*

△

Next, the following remark is a corollary of the $\text{map}_{\text{true}}()$ definition.

Remark 1. *Let* $\mathcal{D} = \langle C, AT, AS \rangle$ *be a data model. Let* $\mathcal{S} = \langle R, \text{auth} \rangle$ *be an FGAC security model for* $\mathcal{D}$*. Let* $\mathcal{O} = \langle OC, OAT, OAS \rangle$ *be an object model of* $\mathcal{D}$*. Let* $r \in R$ *be a role in* $\mathcal{S}$*. Let* $\text{users}(C) \in C$ *be the* users-provider *class in* $\mathcal{D}$*.*
Let $at = \langle atn, c, t \rangle$ *be an attribute of* $\mathcal{D}$*. Let* $u = \langle oi, \text{users(C)} \rangle \in OC$ *be an object in* $\mathcal{O}$*. Let* $w = \langle oi, c \rangle \in OC$ *be an object in* $\mathcal{O}$*. Then,*

$$\text{map}(\mathcal{O})[\underline{\texttt{self}} \mapsto w; \underline{\texttt{caller}} \mapsto u] \models \text{map}_{\text{true}}(\text{auth}(r, \text{read}(at)))$$
$$\Updownarrow$$
$$\text{Eval}(\mathcal{O}, \text{auth}(r, \text{read}(at))[\underline{\texttt{self}} \leftarrow w; \underline{\texttt{caller}} \leftarrow u]) = \texttt{true}$$

where $\text{map}(\mathcal{O})[\underline{\texttt{self}} \mapsto w; \underline{\texttt{caller}} \mapsto u]$ *denotes the interpretation* $\text{map}(\mathcal{O})$ *extended with the assignments of the objects w and u to the variables* self *and* caller*, respectively; and* $\text{auth}(r, \text{read}(at)[\underline{\texttt{self}} \leftarrow w; \underline{\texttt{caller}} \leftarrow u])$ *denotes the expression* $\text{auth}(r, \text{read}(at))$ *after substituting the variables* self *and* caller *by the objects w and u, respectively.*
Similarly, let $as = \langle asn, ase_{\text{l}}, c_{\text{l}}, ase_{\text{r}}, c_{\text{r}} \rangle$ *be an association of* $\mathcal{D}$*. Let* $u = \langle oi, \text{users(C)} \rangle \in OC$ *be an object in* $\mathcal{O}$*. Let* $w_l = \langle o_l, c_l \rangle \in OC$ *and* $w_r = \langle o_r, c_r \rangle \in OC$ *be objects*

in $\mathcal{O}$*. Then,*

$$\begin{array}{c} \mathrm{map}(\mathcal{O})[\underline{ase_{\mathrm{l}}} \mapsto w_l; \underline{ase_{\mathrm{l}}} \mapsto w_r; \underline{\texttt{caller}} \mapsto u] \models \mathrm{map}_{\mathrm{true}}(\mathrm{auth}(r, \mathrm{read}(as))) \\ \Updownarrow \\ \mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(at))[\underline{ase_{\mathrm{l}}} \leftarrow w_l; \underline{ase_{\mathrm{r}}} \leftarrow w_r; \underline{\texttt{caller}} \leftarrow u]) = \texttt{true} \end{array}$$

where, as before, $\mathrm{map}(\mathcal{O})[\underline{ase_{\mathrm{l}}} \mapsto w_l; \underline{ase_{\mathrm{r}}} \mapsto w_r; \underline{\texttt{caller}} \mapsto u]$ *denotes the interpretation* $\mathrm{map}(\mathcal{O})$ *extended with the assignments of the objects* l*,* r*, and* u *to the variables* $\underline{ase_{\mathrm{l}}}$*,* $\underline{ase_{\mathrm{r}}}$*, and* $\underline{\texttt{caller}}$*, respectively; and* $\mathrm{auth}(r, \mathrm{read}(as)[\underline{ase_{\mathrm{l}}} \leftarrow w_l; \underline{ase_{\mathrm{r}}} \leftarrow w_r; \underline{\texttt{caller}} \leftarrow u])$ *denotes the expression* $\mathrm{auth}(r, \mathrm{read}(as))$ *after substituting the variables* $\underline{ase_{\mathrm{l}}}$*,* $\underline{ase_{\mathrm{r}}}$*, and* $\underline{\texttt{caller}}$ *by the objects* w_l*,* w_r*, and* u*, respectively.*

In the last remark, we establish a connection between the result of evaluating an authorization constraint *exp* in FGAC security model on a scenario (i.e. an object model $\mathcal{O}$ with a user and a class-object/association-link to be read) to the satisfiability problem of the interpretation of $\mathrm{map}(\mathcal{O})$ on the generated MSFOL formulae of $\mathrm{map}_{\mathrm{true}}(exp)$.

As mentioned before, the SecQuery() implements FGAC authorization constraints by using case-expressions. Logically, to securely eliminate the *unnecessary* case-expression, we need to prove that *the evaluation of the corresponding authorization constraint is trivially true.* The following remark, which is a corollary of Remark 1, is to formally prove the aforementioned by reducing it into a satisfiability problem.

Remark 2. *Let* $\mathcal{D} = \langle C, AT, AS \rangle$ *be a data model. Let* $\mathcal{S} = \langle R, \mathrm{auth} \rangle$ *be a security model for* $\mathcal{D}$*. Let* $\mathrm{users}(C) \in C$ *be the* users-provider *class of* $\mathcal{D}$*. Let* $r \in R$ *be a role. Let* $at = \langle atn, c, t \rangle$ *be an attribute of* $\mathcal{D}$*. Then, for every object model* $\mathcal{O} = \langle OC, OAT, OAS \rangle$ *of* $\mathcal{D}$*, for every object* $\langle self, c \rangle \in OC$ *and for every object* $\langle caller, \mathrm{users}(C) \rangle \in OC$ *it holds that:*

$$\mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(at))[\underline{\texttt{self}} \leftarrow self; \underline{\texttt{caller}} \leftarrow caller]) = \texttt{true}$$

if and only if the following MSFOL theory is unsatisfiable*:*

$$\begin{array}{l} \mathrm{map}(\mathcal{D}) \cup \mathrm{map}(\underline{\texttt{self}}, c) \cup \mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C)) \\ \quad \cup\ \mathrm{map}_{\mathrm{def}}(\mathrm{auth}(r, \mathrm{read}(at))) \cup \neg\ \mathrm{map}_{\mathrm{true}}(\mathrm{auth}(r, \mathrm{read}(at))) \end{array}$$

where $\mathrm{map}(\underline{\texttt{self}}, c)$ *and* $\mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C))$ *simply add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the constant symbols* $\underline{\texttt{self}}$ *and* $\underline{\texttt{caller}}$*, with sorts* c *and* $\mathrm{users}(C)$*, respectively.*

Similarly, let $as = \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}} \rangle$ *be an association of* $\mathcal{D}$*. Then, for every object model* $\mathcal{O} = \langle OC, OAT, OAS \rangle$ *of* $\mathcal{D}$*, for every object* $\langle o_l, c_l \rangle \in OC$*,* $\langle o_r, c_r \rangle \in OC$ *and for every caller* $\in \mathrm{users(C)}$*, it holds that:*

$$\mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(as))[\underline{\texttt{ase}_{\mathrm{l}}} \leftarrow o_{\mathrm{l}}; \underline{\texttt{ase}_{\mathrm{r}}} \leftarrow o_{\mathrm{r}}; \texttt{caller} \leftarrow \underline{\texttt{caller}}]) = \texttt{true}$$

if and only if the following MSFOL theory is unsatisfiable*:*

$$\begin{aligned} &\mathrm{map}(\mathcal{D}) \cup \mathrm{map}(\underline{\texttt{ase}_{\mathrm{l}}}, c_{\mathrm{l}}) \cup \mathrm{map}(\underline{\texttt{ase}_{\mathrm{r}}}, c_{\mathrm{r}}) \cup \mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C)) \\ &\quad \cup \mathrm{map}_{\mathrm{def}}(\mathrm{auth}(r, \mathrm{read}(as))) \cup \neg\, \mathrm{map}_{\mathrm{true}}(\mathrm{auth}(r, \mathrm{read}(as))) \end{aligned}$$

where $\mathrm{map}(\underline{\texttt{ase}_{\mathrm{l}}}, c_{\mathrm{l}})$*,* $\mathrm{map}(\underline{\texttt{ase}_{\mathrm{r}}}, c_{\mathrm{r}})$ *and* $\mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C))$ *simply add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the constant symbols* $\underline{\texttt{ase}_{\mathrm{l}}}$*,* $\underline{\texttt{ase}_{\mathrm{r}}}$*,* $\underline{\texttt{caller}}$*, with sorts* c_{l}*,* c_{r}*,* $\mathrm{users}(C)$*, respectively.*

The previous remark is key in our methodology to prove (i) an authorization check is trivial. Here, we extend our remark above to prove (ii) an authorization check is satisfied given a data invariant and (iii) an authorization check is satisfied given the properties of the objects involved in the authorization request. Since data invariants and objects' properties can be expressed using OCL boolean expressions, we then append the generated formulae of that OCL expression to the MSFOL theory generated by Remark 2.

Remark 3. *Let* $\mathcal{D} = \langle C, AT, AS \rangle$ *be a data model. Let* $\mathcal{S} = \langle R, \mathrm{auth} \rangle$ *be a security model for* $\mathcal{D}$*. Let* $\mathrm{users}(C) \in C$ *be the users provider-class of* $\mathcal{D}$*. Let* $r \in R$ *be a role. Let* $exp \in \mathrm{Exp}(\mathcal{D})$ *be an OCL boolean expression.*
Let $at = \langle atn, c, t \rangle \in AT$ *be an attribute of* $\mathcal{D}$*. Then, for every object model* $\mathcal{O} = \langle OC, OAT, OAS \rangle$ *of* $\mathcal{D}$ *such that the evaluation of exp returns* `true`*, i.e.*

$$\mathrm{Eval}(\mathcal{O}, exp[\underline{\texttt{self}} \leftarrow self; \underline{\texttt{caller}} \leftarrow caller]) = \texttt{true},$$

for every object $\langle self, c \rangle \in OC$ *and for every object* $\langle caller, \mathrm{users}(C) \rangle \in OC$ *it holds that:*

$$\mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(at))[\underline{\texttt{self}} \leftarrow self; \underline{\texttt{caller}} \leftarrow caller]) = \texttt{true}$$

if and only if the following MSFOL theory is unsatisfiable*:*

$$\begin{aligned} &\mathrm{map}(\mathcal{D}) \cup \mathrm{map}(\underline{\texttt{self}}, c) \cup \mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C)) \\ &\quad \cup \mathrm{map}_{\mathrm{def}}(exp) \cup \mathrm{map}_{\mathrm{true}}(exp) \\ &\quad \cup \mathrm{map}_{\mathrm{def}}(\mathrm{auth}(r, \mathrm{read}(at))) \cup \neg\, \mathrm{map}_{\mathrm{true}}(\mathrm{auth}(r, \mathrm{read}(at))) \end{aligned}$$

where as before, $\mathrm{map}(\underline{\texttt{self}}, c)$ *and* $\mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C))$ *simply add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the constant symbols* $\underline{\texttt{self}}$ *and* $\underline{\texttt{caller}}$*, with sorts* c *and* $\mathrm{users}(C)$*, respectively; and* $\mathrm{map}_{\mathrm{def}}(exp)$ *and* $\mathrm{map}_{\mathrm{true}}(exp)$ *then add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the predicates/formulae generates by mapping exp to MSFOL.*
Similarly, let $as = \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}} \rangle \in AS$ *be an association of* $\mathcal{D}$*. Then, for every object model* $\mathcal{O} = \langle OC, OAT, OAS \rangle$ *of* $\mathcal{D}$ *such that the evaluation of exp returns* `true`*, i.e.*

$$\mathrm{Eval}(\mathcal{O}, exp[\underline{\texttt{ase}_{\mathrm{l}}} \leftarrow o_{\mathrm{l}}; \underline{\texttt{ase}_{\mathrm{r}}} \leftarrow o_{\mathrm{r}}; \texttt{caller} \leftarrow \underline{\texttt{caller}}]) = \texttt{true},$$

for every object $\langle o_l, c_l \rangle \in OC$*,* $\langle o_r, c_r \rangle \in OC$ *and for every caller* $\in \mathrm{users(C)}$*, it holds that:*

$$\mathrm{Eval}(\mathcal{O}, \mathrm{auth}(r, \mathrm{read}(as))[\underline{\texttt{ase}_{\mathrm{l}}} \leftarrow o_{\mathrm{l}}; \underline{\texttt{ase}_{\mathrm{r}}} \leftarrow o_{\mathrm{r}}; \texttt{caller} \leftarrow \underline{\texttt{caller}}]) = \texttt{true}$$

if and only if the following MSFOL theory is unsatisfiable*:*

$$\begin{aligned} &\mathrm{map}(\mathcal{D}) \cup \mathrm{map}(\underline{\texttt{ase}_{\mathrm{l}}}, c_{\mathrm{l}}) \cup \mathrm{map}(\underline{\texttt{ase}_{\mathrm{r}}}, c_{\mathrm{r}}) \cup \mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C)) \\ &\quad \cup\ \mathrm{map}_{\mathrm{def}}(exp) \cup \mathrm{map}_{\mathrm{true}}(exp) \\ &\quad \cup\ \mathrm{map}_{\mathrm{def}}(\mathrm{auth}(r, \mathrm{read}(as))) \cup \neg\ \mathrm{map}_{\mathrm{true}}(\mathrm{auth}(r, \mathrm{read}(as))) \end{aligned}$$

where as before, $\mathrm{map}(\underline{\texttt{ase}_{\mathrm{l}}}, c_{\mathrm{l}})$*,* $\mathrm{map}(\underline{\texttt{ase}_{\mathrm{r}}}, c_{\mathrm{r}})$*,* $\mathrm{map}(\underline{\texttt{ase}_{\mathrm{l}}}, \underline{\texttt{ase}_{\mathrm{r}}}, as)$ *and* $\mathrm{map}(\underline{\texttt{caller}}, \mathrm{users}(C))$ *simply add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the constant symbols* $\underline{\texttt{ase}_{\mathrm{l}}}$*,* $\underline{\texttt{ase}_{\mathrm{r}}}$*,* $\underline{\texttt{caller}}$*, with sorts* c_{l}*,* c_{r}*,* $\mathrm{users}(C)$*, and the association link as between* $\underline{\texttt{ase}_{\mathrm{l}}}$ *and* $\underline{\texttt{ase}_{\mathrm{r}}}$*, respectively; and* $\mathrm{map}_{\mathrm{def}}(exp)$ *and* $\mathrm{map}_{\mathrm{true}}(exp)$ *then add to the MSFOL theory* $\mathrm{map}(\mathcal{D})$ *the predicates/formulae generates by mapping exp to MSFOL.*

4.2.4 From data models to SQL database schema

In [3] we defined a mapping from data models to SQL database schema. Let $\mathcal{D}$ be a data model. In what follows we denote by $\overline{\mathcal{D}}$ the SQL database schema corresponding to $\mathcal{D}$ according to this mapping. The definition of this mapping is recalled in Appendix A. [3]

Example 9. *Consider the* `Uni` *data model in Subsection 2.3, the Listing below shows the description of the SQL database schema of* `Uni`*, according to the mapping from data model to SQL database schema:*

[3]For the sake of illustration, readers can find Appendix E the example SQL database schemata of the data model in Subsection 2.3.

```
mysql> describe Student;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| Student_id | varchar(100) | NO   | PRI | NULL    |       |
| name       | varchar(100) | YES  |     | NULL    |       |
| age        | int(11)      | YES  |     | NULL    |       |
| email      | varchar(100) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+

mysql> describe Lecturer;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| Lecturer_id | varchar(100) | NO   | PRI | NULL    |       |
| name        | varchar(100) | YES  |     | NULL    |       |
| age         | int(11)      | YES  |     | NULL    |       |
| email       | varchar(100) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+

mysql> describe Enrollment;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| lecturers | varchar(100) | YES  | MUL | NULL    |       |
| students  | varchar(100) | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

where `Student_id` *is the primary key and* `name`, `age`, `email` *are the attributes of the table* `Student`. *Similarly,* `Lecturer_id` *is the primary key and* `name`, `age`, `email` *are the attributes of the table* `Lecturer`. *And finally,* `students` *and* `lecturers` *are the foreign keys refers to the primary keys of the table* `Student` *and* `Lecturer`, *respectively.* △

4.2.5 From object models to SQL database instances

In [3] we also defined a mapping from object models to SQL database instances. Let $\mathcal{D}$ be a data model. Let $\mathcal{O}$ be an object model of $\mathcal{D}$. In what follows we denote by

$\overline{\mathcal{O}}$ the instance of the database schema $\mathcal{D}$ that corresponds to $\mathcal{O}$ according to this mapping. The definition of this mapping is recalled in Appendix A.

Example 10. *Consider the object model in Example 3, the Listing below depicts the corresponding database state.*

```
mysql> SELECT * FROM Student;
+------------+-------+------+-------------------+
| Student_id | name  | age  | email             |
+------------+-------+------+-------------------+
| Hoang      | Hoang |   25 | Hoang@student.com |
+------------+-------+------+-------------------+

mysql> SELECT * FROM Lecturer;
+-------------+--------+------+---------------------+
| Lecturer_id | name   | age  | email               |
+-------------+--------+------+---------------------+
| Juan        | Juan   | NULL | Juan@lecturer.com   |
| Manuel      | Manuel | NULL | Manuel@lecturer.com |
+-------------+--------+------+---------------------+

mysql> SELECT * FROM Enrollment;
+-----------+----------+
| lecturers | students |
+-----------+----------+
| Manuel    | Hoang    |
+-----------+----------+
```

where there is one tuple in table `Student`*, representing student* Hoang*. Similarly, there are two tuples in table* `Lecturer`*, representing lecturer* Juan *and* Manuel*, respectively. Finally, there is one tuple in table* `Enrollment`*, representing the association link between* Manuel *and* Hoang*.* △

Remark 4. *Our mapping from object models to SQL database instances has an inverse mapping. Let $\mathcal{D}$ be a data model. Let $\mathcal{Y}$ be an instance of the database schema $\mathcal{D}$. In what follows we denote by $\underline{\mathcal{Y}}$ the object model of $\mathcal{D}$ that corresponds to the database instance $\mathcal{Y}$ according to this inverse mapping.*

4.2.6 From OCL boolean expressions to SQL queries

In [35] we proposed a mapping from OCL expressions to SQL queries. However, in our methodology, we do not assume that authorization constraints are implemented in SQL using this mapping. In fact, in terms of execution-time efficiency, the manual implementations (i.e. written by experts) typically perform better than the ones automatically generated by our mapping, as depicted in [17]. In whatever way the implementation is done, our methodology assumes that this implementation is *correct* in the following sense:

Definition 7. *Let $\mathcal{D}$ be a data model. Let $\overline{\mathcal{D}}$ be an SQL implementation of $\mathcal{D}$. Let $exp \in \mathrm{Exp}(\mathcal{D})$ be a boolean expression. Let qry be an SQL $\overline{\mathcal{D}}$-query. Denote by* `TRUE` *the SQL-value for* true. *We say that qry is a* correct implementation *of exp if and only if:*

- *For any object model $\mathcal{O}$ of $\mathcal{D}$, and any valid assignment σ of objects in $\mathcal{O}$ to the free-variables in exp, the following holds:*

$$\mathrm{Eval}(\mathcal{O}, exp[\sigma]) = \texttt{true} \iff \mathrm{Exec}_{\overline{\sigma}}(\overline{\mathcal{O}}, qry) = \texttt{TRUE}.$$

  where $exp[\sigma]$ is the OCL expression that results from substituting the free-variables in exp using the assignment σ; and $\mathrm{Exec}_{\overline{\sigma}}(\overline{\mathcal{O}}, qry)$ denotes the execution of the query qry in the database instance $\overline{\mathcal{O}}$ of $\overline{\mathcal{D}}$ within an execution-context where, for each assignment $v \to o$ in σ, the variable v has been declared and set to the value $\overline{o}$.

- *For any database instance $\mathcal{Y}$ of $\overline{\mathcal{D}}$, and any valid execution-context τ, the following holds:*

$$\mathrm{Exec}_{\tau}(\mathcal{Y}, qry) = \texttt{TRUE} \iff \mathrm{Eval}(\underline{\mathcal{Y}}, exp[\underline{\tau}]) = \texttt{true}.$$

  where $\mathrm{Exec}_{\tau}(\mathcal{Y}, qry)$ denotes the execution of the query qry in the database instance $\mathcal{Y}$ within the execution-context τ; and $exp[\underline{\tau}]$ denotes the OCL expression that results from substituting the free-variables in exp using the following assignment: each variable v that is declared in τ is assigned to the object $\underline{t}$ in $\underline{\mathcal{Y}}$ that corresponds to the value t in $\mathcal{Y}$ to which the variable v is set in $\underline{\tau}$.

Example 11. *Consider the* `Uni` *data model in Subsection 2.3 and its corresponding SQL database schemata in Example 9. Given an OCL boolean expression exp:*

$$\underline{\texttt{caller}}\texttt{.students} \to \texttt{includes}(\underline{\texttt{self}})$$

and an SQL select-statement qry:

```
SELECT EXISTS ( SELECT 1 FROM Enrollment e
  WHERE e.lecturers = caller AND e.students = self )
```

we say that qry correctly *implements exp with respect to Definition 7. Consider now the object model $\mathcal{O}$ in Example 3 and its corresponding SQL database state $\overline{\mathcal{O}}$ in Example 10.*

- *Let* $\sigma = [\underline{\texttt{self}} \leftarrow \text{Hoang}; \underline{\texttt{caller}} \leftarrow \text{Manuel}]$ *be our assignment function. Then,* $\overline{\sigma} = [\underline{\texttt{self}} \mapsto \overline{\text{Hoang}}; \underline{\texttt{caller}} \mapsto \overline{\text{Manuel}}]$*, where* $\overline{\text{Hoang}}$ *and* $\overline{\text{Manuel}}$ *are the primary keys of the tuple representing* Hoang *in the* `Student` *table and* Manuel *in the* `Lecturer` *table, respectively. As shown in Example 5,* $\text{Eval}(\mathcal{O}, exp[\sigma]) =$ `true`*. Furthermore,* $\text{Exec}_{\overline{\sigma}}(\overline{\mathcal{O}}, qry) =$ `TRUE`*:*

```
mysql> SELECT EXISTS (SELECT 1 FROM Enrollment e
    -> WHERE e.lecturers = 'Manuel'
    -> AND e.students = 'Hoang') AS result;
+--------+
| result |
+--------+
|      1 |
+--------+
```

- *Otherwise, let* $\sigma = [\underline{\texttt{self}} \leftarrow \text{Hoang}; \underline{\texttt{caller}} \leftarrow \text{Juan}]$ *be our assignment function. Then,* $\overline{\sigma} = [\underline{\texttt{self}} \mapsto \overline{\text{Hoang}}; \underline{\texttt{caller}} \mapsto \overline{\text{Juan}}]$*, where* $\overline{\text{Hoang}}$ *is as above and* $\overline{\text{Juan}}$ *is the primary key of the tuple representing* Juan *in the* `Lecturer` *table. As shown in Example 5,* $\text{Eval}(\mathcal{O}, exp[\sigma]) =$ `false`*. Furthermore,* $\text{Exec}_{\overline{\sigma}}(\overline{\mathcal{O}}, qry) =$ `FALSE`*:*

```
mysql> SELECT EXISTS (SELECT 1 FROM Enrollment e
    -> WHERE e.lecturers = 'Juan'
    -> AND e.students = 'Hoang') AS result;
+--------+
| result |
+--------+
|      0 |
+--------+
```

△

4.3 Reducing execution-time overhead: Case expressions

The function SecQuery() implements the authorization checks by using case-expressions. More specifically, the function SecQuery() uses the functions SecAtt() and SecAs() to *wrap*, respectively, any access to a protected attribute at or to a protected association as into a case-expression. The value of this case expression is a call to a function AuthFunc() that implements those authorization checks required for accessing the corresponding attribute or association. If the result of this function-call is `TRUE`, then the case-expression will return the requested resource; otherwise, it will signal an error. In what follows, $\ulcorner \text{AuthFunc}(\mathcal{S}, at) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an attribute at; whereas $\ulcorner \text{AuthFunc}(\mathcal{S}, as) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an association as. When the argument $\mathcal{S}$ is clear from the context, it may be omitted.

The functions SecAtt() and SecAs() use the functions AuthFunc() and AuthFuncRole() to check that the access to a specific protected resource is authorized. For each protected resource, the required authorization checks depend on the role of the user attempting to access this resource. Accordingly, for each role, the function AuthFunc() calls a function AuthFuncRole() that implements the authorization checks required for a user with that role to access a specific protected resource. In what follows, $\ulcorner \text{AuthFuncRole}(\mathcal{S}, at, r) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$, an attribute at, and a role r; whereas $\ulcorner \text{AuthFuncRole}(\mathcal{S}, as, r) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$, an association as. Again, when the argument $\mathcal{S}$ is clear from the context, we may omit it.

The function AuthFuncRole() implements the authorization constraints associated with the permission for users of a given role for executing a given read-action on a specific resource. There are many different ways of implementing in SQL an OCL authorization constraint. The definition of the function AuthFuncRole() only assumes that there exists a function map() that, for each OCL constraint of interest, it returns a *correct* implementation in SQL of this constraint, in the precise sense of Definition 7.

The following remark makes explicit the relationship between the functions AuthFunc() and AuthFuncRole().

Remark 5. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be a security model for $\mathcal{D}$. Let* $\text{users}(C) \in C$ *be the users provider-class in $\mathcal{D}$. Let $r \in R$ be a role in $\mathcal{S}$. Let $\mathcal{Y}$ be an instance of the database $\overline{\mathcal{D}}$ and denote by* `TRUE` *the SQL-value for* true.
Let $at = \langle atn, c, t\rangle$ be an attribute in $\mathcal{D}$. Let self be a key-value identifying a row in the table $\overline{c}$ in $\mathcal{Y}$. Let caller be a key-value identifying a row in the table $\overline{\text{users}}(C)$ in $\mathcal{Y}$. Then, the following holds:

$$\begin{gathered}\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFunc}(at) \urcorner(\mathit{self}, \mathit{caller}, r)) = \texttt{TRUE} \\ \Updownarrow \\ \text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFuncRole}(at, r) \urcorner(\mathit{self}, \mathit{caller})) = \texttt{TRUE}.\end{gathered}$$

Similarly, let $as = \langle asn, ase_{\text{l}}, c_{\text{l}}, ase_{\text{r}}, c_{\text{r}}\rangle$ be an association in $\mathcal{D}$. Let ase_{l} and ase_{r} be key-values identifying, respectively, rows in the tables $\overline{c}_{\text{l}}$ and $\overline{c}_{\text{r}}$ in $\mathcal{Y}$. Let caller be a key-value identifying a row in the table $\overline{\text{users}}(C)$ in $\mathcal{Y}$. Then, the following holds:

$$\begin{gathered}\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFunc}(as) \urcorner(ase_{\text{l}}, ase_{\text{r}}, \mathit{caller}, r)) = \texttt{TRUE} \\ \Updownarrow \\ \text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFuncRole}(as, r) \urcorner(ase_{\text{l}}, ase_{\text{r}}, \mathit{caller})) = \texttt{TRUE}.\end{gathered}$$

The following remark makes explicit the relationship between the function AuthFuncRole() and the function map().

Remark 6. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be a security model for $\mathcal{D}$. Let* $\text{users}(C) \in C$ *be the users provider-class in $\mathcal{D}$. Let $r \in R$ be a role in $\mathcal{S}$. Let* map() *be a correct implementation of the authorization constraints in $\mathcal{S}$. Let $\mathcal{Y}$ be an instance of the database $\overline{\mathcal{D}}$ and denote by* `TRUE` *the SQL-value for* true.

Let $at = \langle atn, c, t\rangle$ be an attribute in $\mathcal{D}$. Let self be a key-value identifying a row in the table $\overline{c}$ in $\mathcal{Y}$. Let caller be a key-value identifying a row in the table $\overline{\text{users}}(C)$ in $\mathcal{Y}$. Then, the following holds:

$$\begin{aligned}&\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFuncRole}(at, r) \urcorner(\mathit{self}, \mathit{caller}) = \texttt{TRUE} \\ &\Updownarrow \\ &\text{Exec}_{\tau}(\mathcal{Y}, \text{map}(\text{auth}(r, \text{read}(at)))) = \texttt{TRUE} \\ &\Updownarrow \text{ (by Definition 7)} \\ &\text{Eval}(\underline{\mathcal{Y}}, \text{auth}(r, \text{read}(at))[\underline{\tau}]) = \texttt{true}.\end{aligned}$$

where τ denotes the execution context, and the variables $\underline{\texttt{self}}$ *and* $\underline{\texttt{caller}}$ *have been declared and set, respectively, to the key-values self and caller.*
Similarly, let $as = \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$ be an association in $\mathcal{D}$. Let $\mathcal{Y}$ be an instance of the database $\overline{\mathcal{D}}$. Let ase_{l} and ase_{r} be a key identifying a row in the table $\overline{c}_{\mathrm{l}}$ and $\overline{c}_{\mathrm{r}}$ in $\mathcal{Y}$. Let caller be a key identifying a row in the table $\overline{\mathrm{users}}(C)$ in $\mathcal{Y}$. Then, the following holds:

$$
\begin{array}{l}
\mathrm{Exec}(\mathcal{Y}, \ulcorner \mathrm{AuthFuncRole}(as, r) \urcorner(ase_{\mathrm{l}}, ase_{\mathrm{r}}, caller) = \texttt{TRUE} \\
\Updownarrow \\
\mathrm{Exec}_{\tau}(\mathcal{Y}, \mathrm{map}(\mathrm{auth}(r, \mathrm{read}(as)))) = \texttt{TRUE} \\
\Updownarrow \text{ (by Definition 7)} \\
\mathrm{Eval}(\underline{\mathcal{Y}}, \mathrm{auth}(r, \mathrm{read}(as))[\underline{\tau}]) = \texttt{true}.
\end{array}
$$

where τ denotes the execution context, and the variables $\underline{\texttt{ase}}_{\mathrm{l}}$, $\underline{\texttt{ase}}_{\mathrm{r}}$, *and* $\underline{\texttt{caller}}$ *have been declared and set, respectively, to the key-values ase_{l}, ase_{r}, and caller.*

Summary
In conclusion, to securely eliminate a case-expressions generated by the function SecQuery(), it is enough to prove that

> (a) the execution of the corresponding authFunc() call will always return TRUE (in any instances of the given database schema).

Furthermore, by the definition of SecQuery(), calling authFunc() involves calling authFuncRole(), which in turns calls the SQL *correct* implementation of the corresponding authorization constraint in the FGAC security model. By Remarks 5–6 and the Definition 7, in order to prove (a) is enough to prove that

> (b) the OCL authorization constraint under consideration will always evaluate to true (in any scenario of the UML/OCL data model corresponding to the given database schema).

By Remarks 1–3, in order to prove (b) is enough to prove that

> (c) it is unsatisfiable the MSFOL theory that results from adding the negation of the formulae returned by applying the mapping $\mathrm{map}_{\mathrm{true}}()$ to the authorization constraint under consideration to the theory returned by applying the mapping map() to the UML/OCL data model corresponding to the given database schema.

As shown in our case study below, to prove (c) we can use SMT solvers.

4.4 Reducing execution-time overhead: Temporary tables

The function SecQuery() implements authorization checks by using case-expressions. These case-expressions are executed within create-statements that generate temporary tables. The reason for using temporary tables (instead of sub-queries), is to prevent the SQL optimizer for "skipping" (by rewriting the corresponding sub-queries) the authorization checks generated by SecQuery().

The following remarks are corollaries of the definition of the functions SecAtt() and SecAs(), and provide a (secure) approach for replacing with the original sub-queries the temporary tables generated by the function SecQuery(), when these tables are proven to be unnecessary. Logically, to allow the SQL optimizer to do its job, whenever "secure", sub-queries should be favoured over temporary tables.

Notice that, based on the remarks below, we can follow the same approach described before (for eliminating unnecessary case-expressions) to prove using SMT solvers that a temporary table generated by the function SecQuery() can be securely replaced with the original sub-query.

Remark 7. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be a security model for $\mathcal{D}$. Let* $\text{users}(C) \in C$ *be the* users-provider *class in $\mathcal{D}$. Let $c \in C$ be a class in $\mathcal{D}$. Let $\mathcal{Y}$ be an instance of the database $\overline{\mathcal{D}}$. Given an SQL query:*

$$\texttt{SELECT} * \texttt{FROM}\ c\ \texttt{WHERE}\ \text{SecAtt}(\mathcal{S}, \mathit{exp})$$

Suppose that, for every attribute $\langle atn, c, t\rangle \in AT$ occurring in exp, every role $r \in R$, every key-value self identifying a row in the table c in $\mathcal{Y}$, and every key-value caller identifying a row in the table $\overline{\text{users}}(C)$ in $\mathcal{Y}$, it holds that:

$$\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFunc}(at) \urcorner(\mathit{self}, \mathit{caller}, r)) = \texttt{TRUE}$$

Then, it holds that:

$$\begin{aligned}&\text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ c\ \texttt{WHERE}\ \text{SecAtt}(\mathcal{S}, \mathit{exp}))\\ &= \text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ c\ \texttt{WHERE}\ \mathit{exp})\end{aligned}$$

Remark 8. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be a security model for $\mathcal{D}$. Let* $\text{users}(C) \in C$ *be the* users-provider *class in $\mathcal{D}$. Let $c \in C$ be a class in $\mathcal{D}$. Let $\mathcal{Y}$ be an instance of the database $\overline{\mathcal{D}}$. Let* SubSelect_c *be an execution result that contains tuples of c-object. Given an SQL query:*

$$\texttt{SELECT} * \texttt{FROM}\ \text{SubSelect}_c\ \texttt{WHERE}\ \text{SecAtt}(\mathcal{S}, exp)$$

Suppose that, for every attribute $\langle atn, c, t\rangle \in AT$ *occurring in* exp*, every role* $r \in R$*, every key-value self identifying a row in the returned subselect* SubSelect_c*, and every key-value caller identifying a row in the table* $\overline{\text{users}}(C)$ *in* $\mathcal{Y}$*, it holds that:*

$$\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFunc}(at) \urcorner(self, caller, r)) = \texttt{TRUE}$$

Then, it holds that:

$$\begin{aligned}&\text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ \text{SubSelect}_c\ \texttt{WHERE}\ \text{SecAtt}(\mathcal{S}, exp))\\ &= \text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ \text{SubSelect}_c\ \texttt{WHERE}\ exp)\end{aligned}$$

Remark 9. *Let* $\mathcal{D} = \langle C, AT, AS\rangle$ *be a data model. Let* $\mathcal{S} = \langle R, \text{auth}\rangle$ *be a security model for* $\mathcal{D}$*. Let* $\text{users}(C) \in C$ *be the* users-provider *class in* $\mathcal{D}$*. Let* $\mathcal{Y}$ *be an instance of the database* $\overline{\mathcal{D}}$*. Let* $c_l, c_r \in C$ *be classes in* $\mathcal{D}$*. Let* o_l *and* o_r *be key-values identifying, respectively, rows in the tables* c_l *and* c_r *in* $\mathcal{Y}$ *Let caller be a key-value identifying a row in the table* $\overline{\text{users}}(C)$ *in* $\mathcal{Y}$*. Let* CartProd *(cartesian product) be the returned results of executing the select statement:*

$$\texttt{SELECT}\ c_\text{l}\texttt{\_id},\ c_\text{r}\texttt{\_id}\ \texttt{FROM}\ c_\text{l},\ c_\text{r}\ \texttt{WHERE}\ exp$$

where exp is an SQL boolean statement that may contain the attributes in c_l *and* c_r*. Suppose that, for every association* $as = \langle asn, ase_\text{l}, c_\text{l}, ase_\text{r}, c_\text{r}\rangle$*,* $as \in AS$*, given an SQL query:*

$$\texttt{SELECT} * \texttt{FROM}\ \text{CartProd}\ \texttt{WHERE}\ \text{SecAs}(\mathcal{S}, as)$$

For every returned tuple $\langle o_l, o_r\rangle \in \text{CartProd}$*, every role* $r \in R$*, and every caller in the table* $\overline{\text{users}}(C)$ *in* $\mathcal{Y}$*, it holds that*

$$\text{Exec}(\mathcal{Y}, \ulcorner \text{AuthFunc}(as) \urcorner(o_l, o_r, caller, r)) = \texttt{TRUE}$$

Then, it holds that:

$$\begin{aligned}&\text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ \text{CartProd}\ \texttt{WHERE}\ \text{SecAs}(\mathcal{S}, as))\\ &= \text{Exec}(\mathcal{Y}, \texttt{SELECT} * \texttt{FROM}\ \text{CartProd})\end{aligned}$$

In the next chapter, we provide non-trivial examples in which the generated stored-procedures can be optimized. By applying these remarks, we formally prove that, indeed, the case-expressions in those stored-procedures are unnecessary.

Chapter 5

Case Study

In this chapter, we conduct a case study for our methodology described above. Here, we revisit the two experiments reported in Subsection 3.2.3 and apply the approach introduced in Section 4.3 to identify the unnecessary checks and optimize the stored-procedures generated by the function SecQuery() and report on the results. For the sake of convenience, we recall briefly the experiment setup in Subsection 3.2.3:

- The data model introduced in Subsection 2.3,
- The scenarios `Uni`(n), for $n > 2$, in which: there are exactly n students and n lectures; students and lectures have unique names; every lecturer has every student as his/her student; there are three distinguished lecturers, namely: `Trang`, `Michel`, and `Vinh`; and no other lecturer is older than `Michel`,
- Three different FGAC security models, namely: `Sec#1`, `Sec#2`, and `Sec#3`. In particular: [1]
  - `Sec#1` contains the following clauses: *an admin can know the age of any student*; and *an admin can know the students of any lecturer*.
  - `Sec#2` contains the following clauses: *a lecturer can know the age of any student, if he/she is the oldest lecturer*; and *a lecturer can know the students of any lecturer, if he/she is the oldest lecturer*.
  - `Sec#3` contains the following clauses: *a lecturer can know the age of any student, if the student is his/her student*; and *a lecturer can know the students of any lecturer, if the student is his/her student*.

[1] For interested readers, the SQL implementation of these FGAC security models can be found in Appendix E.

- Here we consider the three different SQL queries, `Query#1`, `Query#2` and the new `Query#3`, which return, respectively, *the number of students whose age is greater than 18*, *the number of enrollments* and *the average age of students of a current user*.

`Query#1`	`SELECT COUNT(*) FROM Student WHERE age > 18`
`Query#2`	`SELECT COUNT(students) FROM Enrollment`

`Query#3`	`SELECT AVG(age) FROM Student`
	`JOIN (SELECT students FROM Enrollment`
	`  WHERE lecturers = caller) AS TEMP`
	`ON Student_id = students`

Figure 5.1: Experiments: Queries 1–3.

In the following sections, we apply our methodology to optimize the generated stored-procedures in four different configurations.

5.1 First example: Trivial authorization constraints

In this first example, consider the following configuration:

Data model:	`Uni` (in Subsection 2.3)
User class:	`Lecturer`
Scenarios:	`Uni`(n), for $n \geq 2$
Security policy:	`Sec#1`
Role:	There is only one role, namely `Admin`
Query:	`Query#1`

- In this case, we recall the corresponding authorization constraint, i.e. *An admin can know the age of any student*:

$$\text{auth}(\texttt{Admin}, \text{read}(\texttt{Student} : \texttt{age})) = \texttt{true}$$

  Denote by *auth* the above OCL authorization constraint.

- We extend the signature with the symbol constants for caller and self, and the corresponding axioms. In this case: caller is a lecturer, and self is a student.

Notice that the following theory is *unsatisfiable*:

$$\begin{aligned}\text{map}(\texttt{Uni}) \cup \neg\ \text{map}_{\text{true}}(auth) \\ \cup\ \text{map}(\underline{\texttt{caller}}, \texttt{Lecturer}) \cup \text{map}(\underline{\texttt{self}}, \texttt{Student})\end{aligned}$$

Therefore, following Remark 2, we can prove that for every object model $\mathcal{O}$ of Uni data model, for every object *self* of class Student and for every user *caller* of user class Lecturer:

$$\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Admin}, \text{read}(\texttt{Student} : \texttt{age}))[\underline{\sigma}]) = \texttt{true}$$

where $\underline{\sigma} = [\underline{\texttt{caller}} \leftarrow \textit{caller}; \underline{\texttt{self}} \leftarrow \textit{self}]$. Then, following Remark 6, we can prove that for every database instance $\mathcal{Y}$ of $\overline{\texttt{Uni}}$ database schema, given the corresponding execution context σ:

$$\text{Exec}_{\sigma}(\mathcal{Y}, \text{map}(\text{auth}(\texttt{Admin}, \text{read}(\texttt{Student} : \texttt{age})))) = \texttt{TRUE}$$

Finally, recall the snippet body of the stored-procedure generated by SecQuery(Sec#3,Query#2) (depicted in Figure 3.2.3) that contains the authorization check:

```
CREATE TEMPORARY TABLE ⌜TempTable(age > 18)⌝ AS (
    SELECT * FROM Student
    WHERE CASE ⌜AuthFunc(S, age)⌝ (Student_id, caller, role)
      WHEN 1 THEN age ELSE throw_error() END as age > 18
);
```

following Remark 7, it can be *optimized* as follows:

```
CREATE TEMPORARY TABLE ⌜TempTable(age > 18)⌝ AS (
    SELECT * FROM Student WHERE age > 18
);
```

Remarks: Applying the methodology described before, we can in fact prove that, in this case where a user has role `Admin`, the case-statement can be securely removed. [2]

5.2 Second example: Data invariants

In this second example, consider the following configuration:

Data model:	`Uni` (in Subsection 2.3)
User class:	`Lecturer`
Scenarios:	`Uni`(n), for $n \geq 2$
Security policy:	`Sec#3`
Role:	There is only one role, namely `Lecturer`
Query:	`Query#2`

- Firstly, we consider the relevant invariant of the given scenarios. In this case: *Every lecturer has every student as his/her student.*

  $$\texttt{Lecturer.allInstances}() \rightarrow \texttt{forAll}(\texttt{l}|$$
  $$\quad \texttt{Student.allInstances}() \rightarrow \texttt{forAll}(\texttt{s}|\texttt{l.students} \rightarrow \texttt{includes}(\texttt{s})))$$

  Denote by *inv* the above OCL invariant. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(inv)$ (as in this case, there is no formula/axiom returned by $\text{map}_{\text{def}}(inv)$).

- Secondly, we recall the corresponding authorization constraint. In this case: *A lecturer can know the students of any lecturer, if the student is his/her student.*

  $$\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment})) =$$
  $$\underline{\texttt{caller}}\texttt{.students} \rightarrow \texttt{exists}(\texttt{s} \mid \texttt{s} = \underline{\texttt{students}})$$

  Denote by *auth* the above OCL authorization constraint. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(auth)$ (as in this case, there is no formula/axiom returned by $\text{map}_{\text{def}}(auth)$).

- We extend the signature with the symbol constants for $\underline{\texttt{caller}}$, $\underline{\texttt{students}}$ and $\underline{\texttt{lecturers}}$, and the corresponding axioms. In this case: $\underline{\texttt{caller}}$, $\underline{\texttt{lecturers}}$ are lecturers, and $\underline{\texttt{students}}$ is a student.

[2] For interested readers, the complete SQL implementation of this secured stored-procedure as well as the optimized version can be found in Appendix E.

Notice that the following theory is *unsatisfiable*:

$$\begin{aligned}\text{map}(\texttt{Uni}) \cup \neg\ & \text{map}_{\text{true}}(\mathit{auth}) \\ \cup\ & \text{map}(\underline{\texttt{caller}}, \texttt{Lecturer}) \\ \cup\ & \text{map}(\underline{\texttt{students}}, \texttt{Student}) \cup \text{map}(\underline{\texttt{lecturers}}, \texttt{Lecturer}) \\ \cup\ & \text{map}_{\text{true}}(\mathit{inv})\end{aligned}$$

Therefore, following Remark 3, we can prove that for every object model $\mathcal{O}$ of `Uni` data model that satisfies the integrity constraint *inv*, for every object *student* of class `Student`, *lecturer* of class `Lecturer`, and for every user *caller* of user class `Lecturer`:

$$\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment}))[\underline{\sigma}]) = \texttt{true}$$

where $\underline{\sigma} = [\underline{\texttt{caller}} \leftarrow \mathit{caller}; \underline{\texttt{students}} \leftarrow \mathit{student}; \underline{\texttt{lecturers}} \leftarrow \mathit{lecturer}]$. Then, following Remark 6, we can prove that for every database instance $\mathcal{Y}$ of $\overline{\texttt{Uni}}$ database schema that satisfies map(*inv*), given the corresponding execution context σ:

$$\text{Exec}_{\sigma}(\mathcal{Y}, \text{map}(\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment})))) = \texttt{TRUE}$$

Finally, recall the snippet body of the stored-procedure generated by SecQuery(`Sec#3`,`Query#2`) (depicted in Figure 3.2.3) that contains the authorization check:

```
CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
  SELECT * FROM ⌜TempTable(True)⌝
  WHERE CASE ⌜AuthFunc(S, Enrollment)⌝ (students,
        lecturers, caller, role)
    WHEN 1 THEN TRUE ELSE throw_error() END as students
);
```

following Remark 9, it can be *optimized* as follows:

```
IF (map(inv))
THEN
  CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
    SELECT * FROM ⌜TempTable(True)⌝
  );
ELSE
  CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
    SELECT * FROM ⌜TempTable(True)⌝
    WHERE CASE ⌜AuthFunc(S, Enrollment)⌝ (students,
          lecturers, caller, role)
      WHEN 1 THEN TRUE ELSE throw_error() END as students
  );
END IF;
```

Remarks: Applying the methodology described before, we can in fact prove that, in this case where (i) the user has the role `Lecturer` and (ii) the invariant *every student is a student of every lecturer* holds, the case-statement can be securely removed. Notice that the case-statement cannot be removed, however, for the case of the policies `Sec#2`. Neither can it be removed if the invariant does not hold. [3]

5.3 Third example: User properties

In this third example, consider the following configuration:

Data model:	`Uni` (in Subsection 2.3
User:	Michel
Scenarios:	`Uni`(n), for $n \geq 2$
Security policy:	`Sec#2`
Role:	This is only one role, namely `Lecturer`
Query:	`Query#2`

[3] In general, the invariant can not be taken for granted and must be proved by formulating the invariant using again the OCL expressions to SQL statements map() function introduced in Subsection 4.2.6 and the idea of *correct implementations* of OCL queries/invariants. For the interested readers, the complete SQL implementation of this secured stored-procedure as well as the optimized version can be found in Appendix E.

- We recall the corresponding authorization constraint. In this case: *A lecturer can know the age of any student, if no other lecturer is older than he/she is.*

$$\begin{aligned} &\mathit{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{Student} : \texttt{age})) = \\ &\quad \texttt{Lecturer.allInstances}() \rightarrow \texttt{select}(\texttt{l} | \texttt{l.age} > \underline{\texttt{caller}}.\texttt{age}) \\ &\qquad \rightarrow \texttt{isEmpty}(). \end{aligned}$$

  Denote by *auth* the above OCL authorization constraint. More specifically, we state the formulae returned by $\mathrm{map}_{\mathrm{true}}(\mathit{auth})$ (as in this case, there is no formula/axiom returned by $\mathrm{map}_{\mathrm{def}}(\mathit{auth})$).

- We extend the signature with the symbol constants for $\underline{\texttt{caller}}$, $\underline{\texttt{students}}$ and $\underline{\texttt{lecturers}}$, and the corresponding axioms. In this case: $\underline{\texttt{caller}}$, $\underline{\texttt{lecturers}}$ are lecturers, and $\underline{\texttt{students}}$ is a student.

- Furthermore, we acknowledge that the caller, Michel, is the oldest lecturer. This property can be manually written as an OCL expression:

$$\texttt{Lecturer.allInstances()} \rightarrow \texttt{forAll(l|l.age} \leq \underline{\texttt{caller}}.\texttt{age})$$

  Denote by *prop* the above OCL $\underline{\texttt{caller}}$–property. More specifically, we state the formulae returned by $\mathrm{map}_{\mathrm{true}}(\mathit{prop})$ (as in this case, there is no formula/axiom returned by $\mathrm{map}_{\mathrm{def}}(\mathit{prop})$).

Notice that the following theory is *unsatisfiable*:

$$\begin{aligned} \mathrm{map}(\texttt{Uni}) \cup \neg\ & \mathrm{map}_{\mathrm{true}}(\mathit{auth}) \\ \cup\ & \mathrm{map}(\underline{\texttt{students}}, \texttt{Student}) \cup \mathrm{map}(\underline{\texttt{lecturers}}, \texttt{Lecturer}) \\ \cup\ & \mathrm{map}_{\mathrm{true}}(\mathit{prop}) \end{aligned}$$

Therefore, following Remark 3, we can prove that for every object model $\mathcal{O}$ of `Uni` data model, for every object *student* of class `Student`, *lecturer* of class `Lecturer`, and for any user *caller* of user class `Lecturer` that satisfies the property of *being an oldest lecturer*:

$$\mathrm{Eval}(\mathcal{O}, \mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{Enrollment}))[\underline{\sigma}]) = \texttt{true} \tag{5.1}$$

where $\underline{\sigma} = [\underline{\texttt{caller}} \leftarrow \mathit{caller}; \underline{\texttt{students}} \leftarrow \mathit{student}; \underline{\texttt{lecturers}} \leftarrow \mathit{lecturer}]$. Then, following Remark 6, we can prove that for every database instance $\mathcal{Y}$ of $\overline{\texttt{Uni}}$ database schema, given the corresponding execution context σ:

$$\mathrm{Exec}_{\sigma}(\mathcal{Y}, \mathrm{map}(\mathrm{auth}(\texttt{Lecturer}, \mathrm{read}(\texttt{Enrollment})))) = \texttt{TRUE} \tag{5.2}$$

Under the assumption that the property holds, we can eliminate unnecessary authorization checks in SecQuery(`Sec#2`, `Query#2`). Recall the snippet body of the stored-procedure generated by SecQuery(`Sec#2`,`Query#2`) (depicted in Figure 3.2.3) that contains the authorization check:

```
CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
  SELECT * FROM ⌜TempTable(True)⌝
  WHERE CASE ⌜AuthFunc(S, Enrollment)⌝ (students,
        lecturers, caller, role)
    WHEN 1 THEN TRUE ELSE throw_error() END as students
);
```

can be *optimized* as follows:

```
IF (map(prop))
THEN
  CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
    SELECT * FROM ⌜TempTable(True)⌝
  );
ELSE
  CREATE TEMPORARY TABLE ⌜TempTable(students)⌝ AS (
    SELECT * FROM ⌜TempTable(True)⌝
    WHERE CASE ⌜AuthFunc(S, Enrollment)⌝ (students,
          lecturers, caller, role)
      WHEN 1 THEN TRUE ELSE throw_error() END as students
  );
END IF;
```

Remarks: Applying the methodology described before, we can in fact prove that, in this case where (i) the user has the role `Lecturer` and (ii) the user satisfies the property of *being the oldest lecturer*, the case-statement can be securely removed. Notice that the case-statement cannot be removed, however, for the case of the policies `Sec#3`. Neither can it be removed for any user that is not the oldest lecturer. [4]

[4]For the interested readers, the complete SQL implementation of this secured stored-procedure as well as the optimized version can be found in Appendix E.

5.4 Fourth example: Object properties

In this fourth and final example, consider the following configuration:

Data model:	`Uni`
User class:	`Lecturer`
Scenarios:	`Uni`(n), for $n \geq 2$
Security policy:	`Sec#3`
Role:	`Lecturer`
Query:	`Query#3`

To begin with, we show the create-statements generated by the function call SecQuery(`Sec#3`,`Query#3`):

```
CREATE TEMPORARY TABLE ⌜TempTable(lecturers = caller)⌝ AS (
  SELECT Student_id AS students, Lecturer_id AS lecturers
  FROM Student, Lecturer
  WHERE Lecturer_id = caller;
);

CREATE TEMPORARY TABLE ⌜TempTable(students, lecturers)⌝ AS (
  SELECT *
  FROM ⌜TempTable(lecturers = caller)⌝
  WHERE CASE ⌜AuthFunc(S, Enrolment)⌝ (students, lecturers,
    caller, role) WHEN 1 THEN TRUE
    ELSE throw_error() END as students
);

CREATE TEMPORARY TABLE ⌜TempTable(Student_id = students)⌝ AS (
  SELECT *
  FROM Student
  JOIN ⌜TempTable(students, lecturers)⌝
  ON Student_id = students
);

CREATE TEMPORARY TABLE ⌜TempTable(age)⌝ AS (
    SELECT CASE ⌜AuthFunc(S, age)⌝ (Student_id, caller, role)
      WHEN 1 THEN age ELSE throw_error() END as age
    FROM ⌜TempTable(Student_id = students)⌝
);
```

Consider the first case-statement in the temporary table $\ulcorner$TempTable(`students`, `lecturers`)$\urcorner$:

- Firstly, we consider the relevant invariant of the given scenarios. In this case: *Every lecturer has every student as his/her student.*

  $$\texttt{Lecturer.allInstances}() \rightarrow \texttt{forAll}(\texttt{l}|\ \texttt{Student.allInstances}() \rightarrow \texttt{forAll}(\texttt{s}|\texttt{l.students} \rightarrow \texttt{includes}(\texttt{s})))$$

  Denote by *inv* the above OCL invariant. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(inv)$ (as in this case, there is no formula/axiom returned by $\text{map}_{\text{def}}(inv)$).

- We recall the corresponding authorization constraint. In this case: *A lecturer can know the students of any lecturer, if the student is his/her student.*

  $$\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment})) = \underline{\texttt{caller}}\texttt{.students} \rightarrow \texttt{exists}(\texttt{s} \mid \texttt{s} = \underline{\texttt{students}})$$

  Denote by *auth* the above OCL authorization constraint. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(auth)$ (as in this case, there is no formula/axiom returned by $\text{map}_{\text{def}}(auth)$).

- We extend the signature with the symbol constants for $\underline{\texttt{caller}}$, $\underline{\texttt{students}}$ and $\underline{\texttt{lecturers}}$, and the corresponding axioms. In this case: $\underline{\texttt{caller}}$, $\underline{\texttt{lecturers}}$ are lecturers, and $\underline{\texttt{students}}$ is a student.

Notice that the following theory is *unsatisfiable*:

$$\begin{aligned}\text{map}(\texttt{Uni}) \cup \neg\ & \text{map}_{\text{true}}(auth) \\ \cup\ & \text{map}(\underline{\texttt{caller}}, \texttt{Lecturer}) \\ \cup\ & \text{map}(\underline{\texttt{students}}, \texttt{Student}) \cup \text{map}(\underline{\texttt{lecturers}}, \texttt{Lecturer}) \\ \cup\ & \text{map}_{\text{true}}(inv)\end{aligned}$$

Therefore, following Remark 3, we can prove that for every object model $\mathcal{O}$ of `Uni` data model that satisfies *inv*, for every object *student* of class `Student`, *lecturer* of class `Lecturer`, and for any user *caller* of user class `Lecturer`:

$$\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment}))[\underline{\sigma}]) = \texttt{true}$$

where $\underline{\sigma} = [\underline{\texttt{caller}} \leftarrow \mathit{caller}; \underline{\texttt{students}} \leftarrow \mathit{student}; \underline{\texttt{lecturers}} \leftarrow \mathit{lecturer}]$. Then, following Remark 6, we can prove that for every database instance $\mathcal{Y}$ of $\overline{\texttt{Uni}}$ database schema that satisfies map(inv), given the execution context σ:

$$\text{Exec}_\sigma(\mathcal{Y}, \text{map}(\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Enrollment})))) = \texttt{TRUE}$$

Finally, following Remark 9, we can eliminate this unnecessary authorization check. As a result, the temporary table $\ulcorner\text{TempTable}(\texttt{students}, \texttt{lecturers})\urcorner$ can be rewritten as follows:

```
IF (map(inv))
THEN
  CREATE TEMPORARY TABLE ⌜TempTable(students, lecturers)⌝ AS (
    SELECT * FROM ⌜TempTable(lecturers = caller)⌝
  );
ELSE
    CREATE TEMPORARY TABLE ⌜TempTable(students, lecturers)⌝ AS (
    SELECT *
    FROM ⌜TempTable(lecturers = caller)⌝
    WHERE CASE ⌜AuthFunc(S, Enrolment)⌝ (students, lecturers,
      caller, role) WHEN 1 THEN TRUE
    ELSE throw_error() END as students
  );
END IF;
```

Moreover, consider second case-statement in temporary table $\ulcorner\text{TempTable}(\texttt{age})\urcorner$:

- We recall the corresponding authorization constraint, i.e. *A lecturer can know the age of any student, if the student is his/her student*:

  $$\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Student} : \texttt{age})) = \\ \underline{\texttt{caller}}.\texttt{students} \rightarrow \texttt{exists}(\texttt{s} \mid \texttt{s} = \underline{\texttt{students}})$$

  Denote by *auth* the above OCL authorization constraint. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(\mathit{auth})$ (as in this case, there is no formula/axiom returned by $\text{map}_{\text{def}}(\mathit{auth})$).

- We extend the signature with the constants for caller and self, and the corresponding axioms. In this case: caller is a lecturer, and self is a student.
- Furthermore, we acknowledge that the temporary table ⌜TempTable(Student_id = students)⌝ only contains the students of the caller. This property can be manually written as an OCL expression: [5]

  $$\underline{\texttt{caller}}.\texttt{students} \rightarrow \texttt{includes}(\underline{\texttt{self}})$$

  Denote by *prop* the above OCL students–property. More specifically, we state the formulae returned by $\text{map}_{\text{true}}(prop)$ (as in this case, there is no formulae/axiom returned by $\text{map}_{\text{def}}(prop)$).

Notice that the following theory is *unsatisfiable*:

$$\begin{aligned}\text{map}(\texttt{Uni}) \cup \neg\ \text{map}_{\text{true}}(auth) \\ \cup\ \text{map}(\underline{\texttt{caller}}, \texttt{Lecturer}) \cup \text{map}(\underline{\texttt{self}}, \texttt{Student}) \\ \cup\ \text{map}_{\text{true}}(prop)\end{aligned}$$

Therefore, following Remark 3, we can prove that for every object model $\mathcal{O}$ of Uni data model, for every object *student* of class Student and for every user *caller* of user class Lecturer:

$$\text{Eval}(\mathcal{O}, \text{auth}(\texttt{Admin}, \text{read}(\texttt{Student} : \texttt{age}))[\underline{\sigma}]) = \texttt{true} \tag{5.3}$$

where $\underline{\sigma} = [\underline{\texttt{caller}} \leftarrow caller; \underline{\texttt{self}} \leftarrow student]$. Then, following Remark 6, we can prove that for every database instance $\mathcal{Y}$ of $\overline{\texttt{Uni}}$ database schema, given the execution context σ:

$$\text{Exec}_{\sigma}(\mathcal{Y}, \text{map}(\text{auth}(\texttt{Lecturer}, \text{read}(\texttt{Student} : \texttt{age})))) = \texttt{TRUE} \tag{5.4}$$

Finally, following Remark 8, we can eliminate this unnecessary authorization check. As a result, the temporary table ⌜AuthFunc($\mathcal{S}$, age)⌝ can be rewritten as follows:

```
CREATE TEMPORARY TABLE ⌜TempTable(age)⌝ AS (
    SELECT age
    FROM ⌜TempTable(Student_id = students)⌝
);
```

[5] Note that, these so-called "coincidental" properties are manually written by the modeller. One interesting question arises as: *Can these properties be automatically derived?*. However, due to the time limit, we leave it as part of future work.

Remarks Applying the methodology described before, we can in fact prove that, in this case where the user has the role `Lecturer`, the case-statements can be securely removed. Notice that neither of the case-statements cannot be removed, however, for the case of the policies `Sec#2`. [6]

[6]For interested readers, the complete SQL implementation of this secured stored-procedure as well as the optimized versions can be found in Appendix E.

Chapter 6

Tool support

In this thesis, we have shown that, in fact, some of these authorization checks are unnecessary and hence can be removed from the stored-procedure to optimize the execution performance. As a proof of concept, we have implemented a prototype to check for the necessity of the authorization checks based on the formal approach described in Chapter 4. In what follows, we will denote this tool by the name `FGAC-Optimizer`.

In this chapter, we introduce the `FGAC-Optimizer` tool, then describe its typical use-case scenario.

6.1 The `FGAC-Optimizer` tool

The `FGAC-Optimizer` tool is a command-line application implemented using general-purpose programming languages, namely Java and Python. In general, the `FGAC-Optimizer` tool accepts a `JSON` configuration file as input and performs two tasks: firstly, it generates the corresponding *many-sorted* first-order logic theory, written in SMT-LIB language (version 2.0) [6], and then uses an SMT solver of choice to determine whether the above theory is satisfiable. [1] The detail implementation of this tool can be found on the GitHub repository at `https://github.com/npbhoang/FGAC-Optimizer`.

[1] The SMT-LIB is an international initiative, coordinated by the "gu-ru" of the SMT community, with the aim of facilitating research and development in SMT [6]. One of the main contributions of the SMT-LIB is to define a common standard input language for SMT-solvers, called SMT-LIB language.

Input configuration The input configuration stores the vital information for the `FGAC-Optimizer` tool to generate the theory. More specifically, the available setting variables are:

- `DataModel`: Thel filename containing the data model, in `JSON`-format. [2]
- `Invariants`: The OCL invariants that hold in the data model (for example, from the last chapter, *every lecturer teaches every student*), in text format.
- `SecurityModel`: The filename containing the security model, in `JSON`-format. [3]
- `Role`: The considered role, in text format.
- `Resource`: The target property to be read, it may be either an attribute of a class or an association.
  - for the former case, a `JSON`-object consists of two fields, namely `entity` and `attribute` containing the class name and the attribute to be read, respectively.
  - for the latter case, a `JSON`-object consists of one field, namely `association` containing the association name to be read.
- `Properties`: The OCL expressions represent the properties of the user or the object to be read (for example, *the user is the oldest lecturer*), in text format.
- `Solvers`: The SMT solvers of choice (these solvers must support the SMT-LIB language).

Listing 6.1 displays a sample input configuration for checking the necessity of the authorization check described in Example 5.1. In this example, the end-user would like to use CVC4 solver to check for the necessity of the authorization check when an user with a role `Admin` attempting to read `Enrollment` association links, in `Uni` data model, according to the security model `Sec#1`.

[2]For the interested readers, the definition of data model in JSON representation is included in Appendix C.

[3]For the interested readers, the definition of FGAC security model in JSON representation is included in Appendix D.

Listing 6.1: A sample configuration input for the FGAC-Optimizer tool

```
{
    "DataModel": "Uni",
    "SecurityModel": "Sec#1",
    "Role": "Admin",
    "Resource": {
        "Association": "Enrollment"
    },
    "Solvers": ["CVC4"]
}
```

Generating MSFOL theories Firstly, the FGAC-Optimizer takes the input configuration and generates the corresponding *many-sorted* first order logic theory. Figure 6.1 describes the design of this feature, at the component level. This part is implemented using Java and essentially consists of five main components:

- DMParser handles the parsing of data models from JSON representation to Java objects.
- SMParser handles the parsing of security models from JSON representation to Java objects.
- OCLParser handles the parsing of OCL expressions from string to Java objects.
- DM2MSFOL implements the function map(), generating the MSFOL theory of the data model.
- OCL2MSFOL implements the function $\text{map}_{\text{true}}()$ and its auxiliary functions, generating the MSFOL formulae from the OCL expressions. [4]

Solving MSFOL theories Secondly, the FGAC-Optimizer tool uses SMT-solvers to solve the generated MSFOL theory. The result value can only be either SAT (satisfiable), UNSAT (unsatisfiable) or UNKNOWN:

[4]Note that, for this proof of concept, we currently support $\text{map}_{\text{true}}()$ only for the subset of the OCL that the OCLParser is able to parse. The supported subset description can be found at our repository at https://github.com/npbhoang/FGAC-Optimizer/wiki/.

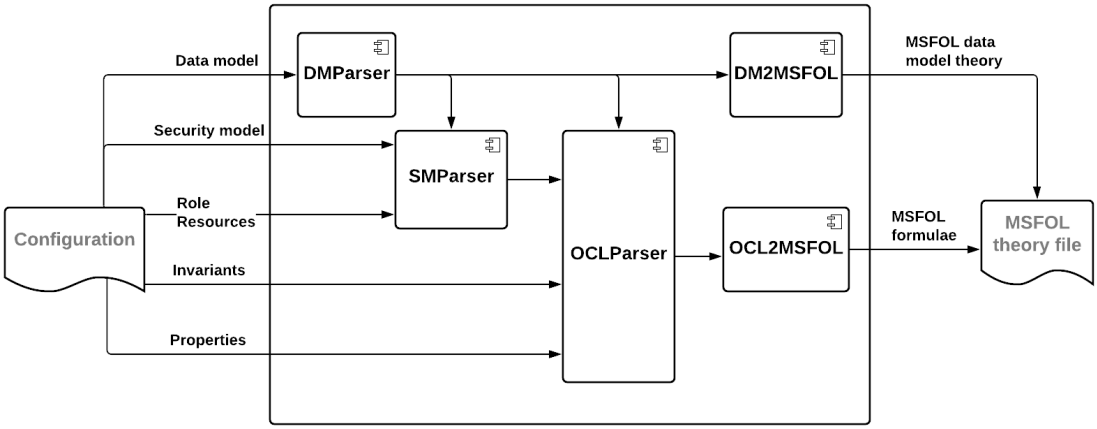


Figure 6.1: The `FGAC-Optimizer` component diagram

- if the result returns `SAT`, then there exists an instance (or a model) where an user u, with the given `Role`, is not authorized to read the given `Resource` of some objects. In this case, the authorization check cannot be removed.
- if the result returns `UNSAT`, then there exists no instance (model) where an user u, with the given `Role`, is not authorized to read the given `Resource` of some objects. In this case, the authorization check is unnecessary and hence, can be removed.
- otherwise, if the result returns `UNKNOWN`, then it remains unknown whether such instance (model) exists. In this case, the authorization check cannot be removed.

6.2 The SQLSI use-case (extended)

In [3], we proposed a model-driven approach to support enforcing fine-grained access control at the database level. As part of our work presented in [4], we have implemented a transformation tool, called `SQLSI`, that automatically rewrites normal SQL queries into stored-procedures which include the authorization checks. In this thesis, we propose a model-driven methodology to *optimize* the generated stored-procedures. And as part of the work presented here, we include a prototype to support our methodology, called `FGAC-Optimizer`.

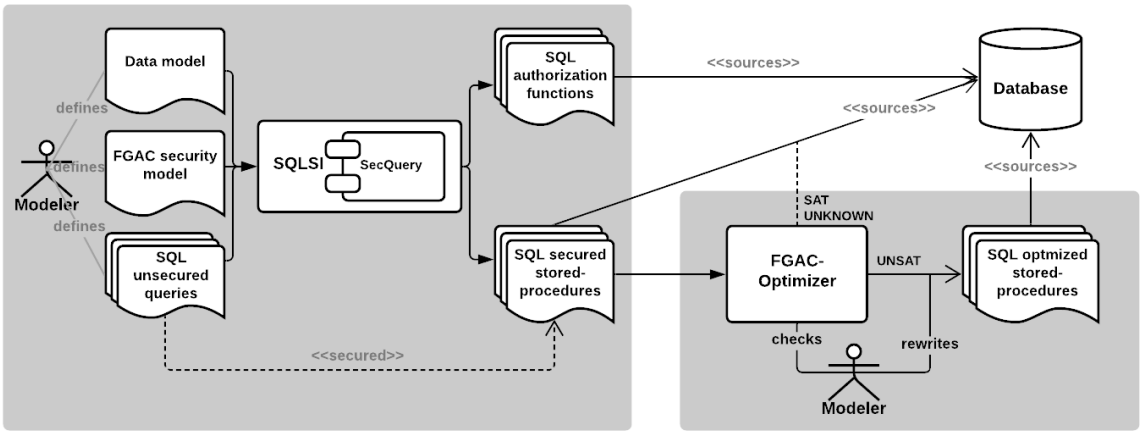


Figure 6.2: The SQLSI use-case (extended)

In Figure 6.2 , given an application with the underlying database modelled by a data model $\mathcal{D}$, given the FGAC security model $\mathcal{S}$, and given collection of SQL queries Q, the typical workflow, to enforce fine-grained access control for Q is the following:

- For each query $q \in Q$, the modeller inputs the data model $\mathcal{D}$, security model $\mathcal{S}$ and the query q into the `SQLSI` tool. Then, the `SQLSI` tool *automatically* generates the corresponding secure stored-procedure $\text{SecQuery}(\mathcal{S}, q)$.
- Next, the modeller analyzes the stored-procedure $\text{SecQuery}(\mathcal{S}, q)$ and identifies potential unnecessary authorization checks. Then, for each identification, the modeller creates a different input configuration, and feeds it into the `FGAC-Optimizer` tool.
  - if the result is `SAT` or `UNKNOWN`, then the check cannot be removed with the given configuration.
  - otherwise, if the result is `UNSAT`, then the check can be removed with the given configuration. In this case, the modeller can rewrite the stored-procedure in a way that makes use of this new information.

Chapter 7

Evaluation

In this chapter, we evaluate different criteria of our proposed model-based methodology for optimizing secure stored-procedure. Firstly, we revisit the experiments in Chapter 5 once more, this time applying the tool and the use-case proposed in the previous chapter. Then, we compare the execution-time performance of these *optimized* stored-procedure with the original.

7.1 Generating and Solving MSFOL theories

To evaluate the correctness of our generated MSFOL theories, with respect to the examples in Chapter 5, we rely on the two state-of-the-art SMT solvers, namely the Cooperating Validity Checker 4 (CVC4) [7], version 1.8., and the Microsoft Research Z3 [21], version 4.8.12. In this evaluation, for each example in Chapter 5, we generate the MSFOL theories using (i) the exact configuration introduced at the beginning and (ii) the configuration mentioned in the remarks at the end of each example. The interested readers can find in Appendix F the satisfiability problem that corresponds to these experiments.

Table 7.1 shows the output results as well as the solving time of each SMT solver. [1] For each of the example in Chapter 5, we generate the corresponding MSFOL theory using the `FGAC-Optimizer` tool, then we feed these generated MSFOL theory to the SMT-solvers. In Table 7.1, ∘ and — denote that the solver returns `SAT` and `UNKNOWN`, respectively; whereas • denotes the solver returns `UNSAT`. The solving time here is

[1] The transformation time of the MSFOL theory is another metric that can be included in this evaluation. However, since the transformation was implemented without using any transformation tool but ad-hoc and the recorded time is not significant, we decided not to report it.

	Ex. 5.1	Ex. 5.2			Ex. 5.3			Ex. 5.4a			Ex. 5.4b		
	(1)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
CVC4	●	●	—	—	●	—	—	●	—	—	●	—	—
	0.74	0.18	0.17	0.31	0.11	0.07	0.11	0.06	0.1	0.07	0.15	0.1	0.08
CVC4†	●	●	○	○	●	○	○	●	○	○	●	○	○
	0.03	0.12	0.08	0.11	0.09	0.08	0.06	0.06	0.07	0.07	0.1	0.06	0.07
Z3	●	●	○	○	●	○	○	●	○	○	●	○	○
	0.47	0.09	0.15	0.1	0.07	0.06	0.06	0.07	0.06	0.06	0.15	0.07	0.06

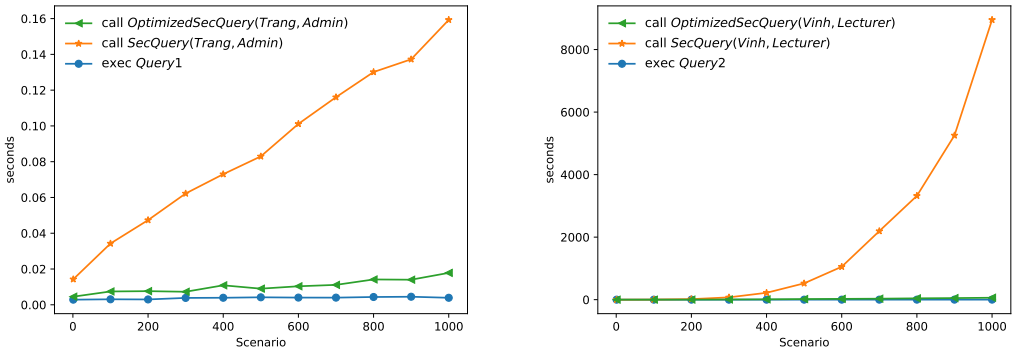
Table 7.1: The experiment results for examples in Chapter 5, under different input configurations, solved by different SMT-solvers, namely the CVC4, the CVC4 with `-finite-model-find` mode (denoted by CVC4†), and Z3.

measured in seconds, and by the arithmetic average of 10 executions. Overall, for all input theories, both CVC4 and Z3 solver response in less than 1 second. This solving time is acceptable since, as mentioned in our use-case, this proving process only happens at compile-time. More importantly, the result returned is as *expected*, i.e. with the configurations in category (i), the solvers always return `UNSAT`—which is as expected, since we have formally proved in Chapter 5 that the authorization checks in these examples are indeed unnecessary; with category (ii), the solvers return either `UNKNOWN` or `SAT`—which is as expected, since the authorization checks in these cases cannot be removed. [2]

7.2 Calling the *optimized* stored-procedures

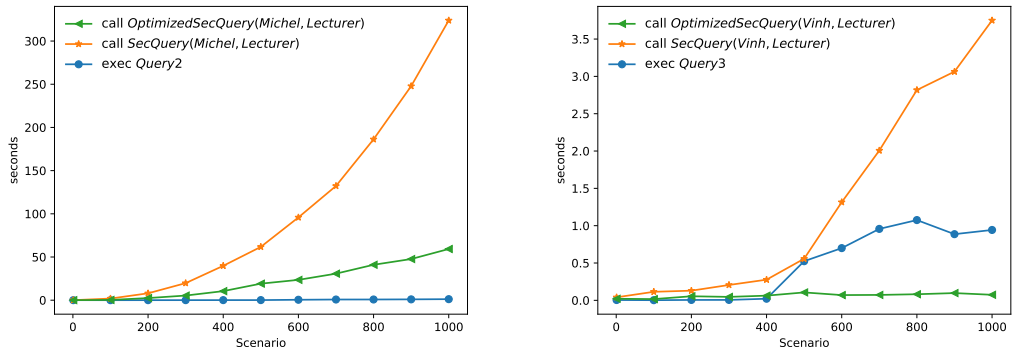
Figure 7.1 shows the execution-time of the *optimized* stored-procedure, calculated by the average of 10 executions. The interested readers can find in Appendix E the source code of the optimized stored-procedure for these experiments. As expected, the execution-time of the secured stored-procedure after being rewritten has reduced significantly. In Example 5.2 and 5.4, the execution of the optimized stored-procedures are even on par with the "unsecured" query. In particular, as depicted in Table 7.2, given the scenario `Uni`(10^3), Example 5.1, 5.2, 5.3 and 5.4, the optimized stored-procedures execute approximately 9, 144, 5 and 50 times faster than the generated stored-procedures from SecQuery().

[2] In the case where the solver returns `SAT`, the end-user can also obtain a counter-example, i.e. the scenario where the authorization check returns false, as a proof that the check cannot be removed.



(a) Example 5.1

(b) Example 5.2



(c) Example 5.3

(d) Example 5.4

Figure 7.1: The execution-time overall comparison of the optimized stored-procedures in Chapter 5. In each experiment, the line marked with •, ⋆ and ◁ indicate the execution-time of the original query, the generated stored-procedure and the optimized stored-procedure, respectively.

Ex.		$\{n \mid \texttt{Uni}(n)\}$									
		100	200	300	400	500	600	700	800	900	1000
5.1	execution-time	0.007	0.008	0.007	0.011	0.009	0.010	0.011	0.014	0.014	0.018
	speed-up	4.598	6.214	8.539	6.708	9.204	9.771	10.42	9.190	9.789	8.931
5.2	execution-time	0.345	1.989	5.739	10.57	19.39	25.64	31.45	42.57	50.6	61.96
	speed-up	8.914	8.444	12.91	20.81	26.93	41.06	69.72	78.01	103.8	144.4
5.3	execution-time	0.320	2.467	5.368	10.49	19.14	23.53	30.71	41.00	47.64	59.26
	speed-up	5.941	3.232	3.680	3.796	3.224	4.071	4.305	4.543	5.201	5.465
5.4	execution-time	0.017	0.054	0.046	0.062	0.105	0.070	0.072	0.081	0.097	0.074
	speed-up	6.875	2.377	4.445	4.409	5.259	18.67	27.66	34.47	31.55	50.53

Table 7.2: The detail execution-time of the optimized secured stored-procedures and the speed-up obtained with respect to the execution-time of the generated stored-procedures.

Chapter 8

Related Work

The work presented in this thesis *optimizes* a recently proposed, model-driven approach for enforcing FGAC policies when executing SQL queries [2, 3, 4]. In this chapter, we discuss the works that are related to the aforementioned model-driven approach for enforcing FGAC policies as well as our model-driven methodology to optimize it.

A key feature of the approach proposed in [3, 4] is that it *does not modify* the underlying database, except for adding the stored-procedures that configure our FGAC enforcement mechanism. This is in clear contrast with the solutions currently offered by the major commercial RDBMS and some theoretical research, which recommend to manually create appropriate *views* —like in the case of MySQL or MariaDB [34]— or to automatically generates additional *policy* columns and tables —like in the case of [5]—, and then to modify the queries as to referencing these views/tables/-columns, or request — like Oracle [15], PostgreSQL [38], and IBM [24]— to use other non-standard, proprietary enforcement mechanisms. As argued in [2], the solutions currently offered by the major RDBMS are far from ideal: in fact, they are time-consuming, error-prone, and scale poorly.

The second key feature of the model-driven approach proposed in [3, 4] is that FGAC policies and SQL queries are kept *independent* of each other, except for the fact that they refer to the same underlying data model. This means, in particular, that FGAC policies can be specified without knowing which SQL queries will be executed, and vice versa. This is in clear contrast with the solution recently proposed in [33] where the FGAC policies must be (re-)written depending on the SQL queries that are executed. Nevertheless, the approach proposed in [3, 4] certainly shares

with [33], as well as with other previous approaches like [31], the idea of enforcing FGAC policies by *rewriting* the SQL queries, instead of by modifying the underlying databases or by using non-standard, proprietary RDBMS features.

The third key-feature of approach proposed in [3, 4] is that the enforcement mechanism can be *automatically generated* from the FGAC policies, by using available mappings from OCL to SQL —for example [23, 22, 35]— in order to implement the authorization constraints appearing in the FGAC policies. In practice, however, our experiments show that, for the sake of execution-time performance, manually implementing in SQL the authorization constraints appearing in the FGAC policies is to be preferred over using the implementations generated by the available mappings from OCL to SQL [17].

Notice that, in our approach, whenever a user is unauthorized to access a part of information which is used to answer the query, we immediately rollback the execution and return to the user an unauthorization error. This is, in fact, not the only approach to enforce fine-grained access control. The Truman Model, the terminology introduced in [45], favored in [16, 37], describes the mechanism where FGAC enforcement does not return an error but display as many information as the user is authorized to see. One major drawback, as described in [45], is that since a user is not aware of the enforcement underneath, he/she does not know whether the result obtained is complete or not. In clear contrast, our approach ensures the above consistency, in the sense that the user will either get the *expected* result or the unauthorized error. Consequently, we share the same remark with [45], that: "*one major concern about using this approach [the Non-Truman model] is the overhead of validity checking, especially for queries with a small execution time*". Nevertheless, to optimize the validity checking, the approach proposed in [45] differs from the solution in this thesis.

Finally, it is worthwhile to include in this chapter the work related with the mapping from OCL to first-order logic (FOL). To begin with, there have been many proposed mapping from OCL to different formalisms, [29, 1, 40] to name a few, but OCL2MSFOL [19], to the best of our knowledge, is the current state of the art. Furthermore, in references, this mapping has used in many lines of research that have similar context, e.g. formal reasoning about the validity of the data models [18], as well as the policy consistency of SecureUML models [20].

Chapter 9

Limitations, Conclusions and Future Work

Recently, [4] has proposed a *model-driven* approach for *enforcing* fine-grained access control (FGAC) policies when executing SQL queries. In a nutshell, to enforce FGAC policies when executing SQL queries, a function SecQuery() is defined that, given a policy $\mathcal{S}$ and a select-statement q, generates an SQL stored-procedure, such that: if a user is authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure will return the same result that executing q; otherwise, if a user is not authorized, according to $\mathcal{S}$, to execute q, then calling the stored-procedure will signal an error.

Not surprisingly, since enforcing FGAC policies for SQL queries implies performing authorization checks at execution-time, when following the approach proposed [4] there is a loss in performance. Clearly, however, there are situations in which the required authorization checks are in fact unnecessary, because they will always return true.

In this thesis we have developed a formal, model-based methodology for *optimizing* the stored-procedures generated by the function SecQuery(). In particular, whenever "secure", subqueries are favored over temporary tables, in order to allow the SQL optimizer to do its job. The decision of whether it is "secure" or not to use sub-queries instead of temporary tables ultimately depends on the underlying security model, and more particularly on the authorization constraints responsible in each case of the case-statements generated by SecQuery(). If these authorization constraints (i) can be proved to be trivial, or if they (ii) can be proved to be always satisfied given the invariants of the underlying data model, and/or (iii) can be proved to be satisfied

given the known properties of the objects involved in the authorization request, then the case-statements do not need to be generated, and the corresponding temporary tables can be safely replaced by sub-queries. To illustrate our approach we have provided a number of examples, involving different FGAC policies, queries, and scenarios, and we have evaluated the performance overhead incurred when executing the stored-procedures generated by SecQuery(). Finally, we have also implemented our approach as a prototype, which is currently an on-going project.

As far as the limitation concerned, our approach currently has several limitations. The following items describe these limitations and our future work in their regards.

Firstly, the data model does not support generalizations as well as m-to-n associations, where m and n are different from 1 or *many*. In addition, the FGAC security model does not consider role-hierarchies and we only consider the read-actions. It is part of our future work to extend these considerations.

Secondly, we define our own mapping from data model to SQL schemata. However, other mappings from data models to SQL databases are also possible. Of course, in this case, the implementation of enforcing FGAC policies must be changed accordingly.

As far as the function SecQuery() is concerned, its current implementation, which is described in Appendix B, only works for the MySQL Server. However, since all of the major SQL database systems follow the common standard [49], it is feasible to extend (syntactically) the implementation to support as well other relational database management systems. For non-relational databases, the general approach underlying is applicable. These are also parts of future work.

The definition of the function SecQuery(), which takes an SQL query as input, only covers the query patterns in [2]. As part of the future work, we plan to extend this definition to cover as much as possible of the SQL language, including, in particular, left/right-joins, group-by clauses and user-defined functions.

As mentioned before, ideally, the implementations of the OCL authorization constraints used by the function AuthFunc() could be automatically generated from the FGAC security models, by using available mappings from OCL to SQL —for example [35]. In practice, however, for the sake of execution-time performance, manually

implementing in SQL the authorisation constraints is to be preferred over using the implementations generated by the available mappings from OCL to SQL.

And finally, our methodology is not fully automated and requires human intuition. Firstly, in some cases, in order to prove a case-expression is unnecessary, a database invariant or a property of the user need to be introduced. Since this methodology operates at the compile-time, these properties cannot be derived automatically from the given resources but rather to be manually inserted by the modeller. Secondly, whenever a case-expression is proven to be unnecessary, the modeller is responsible to rewrite the stored-procedure in a way that makes use of this information. It is our future work to replace some of these ad-hoc steps into automation.

References

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On Challenges of Model Transformation from UML to Alloy. *Journal of Software and Systems Modeling*, 9(1):69–86, 2010.

[2] Hoang Nguyen Phuoc Bao and Manuel Clavel. Model-based Characterization of Fine-Grained Access Control authorization for SQL Queries. *Journal of Object Technology*, 19(3):3:1–13, 2020.

[3] Hoang Nguyen Phuoc Bao and Manuel Clavel. A Model-Driven Approach for Enforcing Fine-Grained Access Control for SQL Queries. In Tran Khanh Dang, Josef Küng, Makoto Takizawa, and Tai M. Chung, editors, *Future Data and Security Engineering - 7th International Conference, FDSE 2020, Quy Nhon, Vietnam, November 25-27, 2020, Proceedings*, volume 12466 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2020.

[4] Hoang Nguyen Phuoc Bao and Manuel Clavel. A Model-Driven Approach for Enforcing Fine-Grained Access Control for SQL Queries. *Springer Nature Computer Science*, 2(5):370, 2021.

[5] Steve Barker. Dynamic Meta-level Access Control in SQL. In Vijay Atluri, editor, *Data and Applications Security XXII, 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security, London, UK, July 13-16, 2008, Proceedings*, volume 5094 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[7] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In

Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[8] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[9] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. A Metamodel-Based Approach for Analyzing Security-Design Models. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems (MODELS)*, volume 4735 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2007.

[10] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated Analysis of Security-design Models. *Information and Software Technology*, 51(5):815–831, 2009.

[11] David Basin, Manuel Clavel, and Marina Egea. A decade of Model-Driven Security. In Ruth Breu, Jason Crampton, and Jorge Lobo, editors, *16th ACM Symposium on Access Control Models and Technologies, SACMAT 2011, Innsbruck, Austria, June 15-17, 2011, Proceedings*, pages 1–10. ACM, 2011.

[12] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model-Driven Security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[13] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[15] Kristy Browder and Mary Ann Davidson. The Virtual Private Database in Oracle9iR2. Technical report, Oracle Corporation, 2002. `https://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf`.

[16] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. Fine-Grained Authorization through Predicated Grants. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1174–1183. IEEE Computer Society, 2007.

[17] Manuel Clavel and Hoang Nguyen Phuoc Bao. Mapping OCL into SQL: Challenges and Opportunities Ahead. In A. D. Brucker, G. Daniel, and F. Jouault, editors, *19th International Workshop in OCL and Textual Modeling (OCL 2019) co-located with MODELS 2019*, volume 2513 of *CEUR Workshop Proceedings*, pages 3–16. CEUR-WS.org, 2019.

[18] Carolina Dania and Manuel Clavel. Model-Based Formal Reasoning about Data-Management Applications. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 9033 of *LNCS*, pages 218–232. Springer, 2015.

[19] Carolina Dania and Manuel Clavel. OCL2MSFOL: A Mapping to Many-Sorted First-Order Logic for Efficiently Checking the Satisfiability of OCL Constraints. In Benoit Baudry and Benoît Combemale, editors, *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 65–75. ACM, 2016.

[20] Miguel Angel García de Dios, Carolina Dania, and Manuel Clavel. Formal Reasoning about Fine-Grained Access Control Policies. In Motoshi Saeki and Henning Köhler, editors, *Asia-Pacific Conference on Conceptual Modelling (APCCM)*, volume 165 of *CRPIT*, pages 91–100. Australian Computer Society, 2015.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] Marina Egea and Carolina Dania. SQL-PL4OCL: An Automatic Code Generator from OCL to SQL Procedural Language. In *20th ACM/IEEE International*

Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017, page 54. IEEE Computer Society, 2017.

[23] Marina Egea, Carolina Dania, and Manuel Clavel. MySQL4OCL: A Stored Procedure-based MySQL Code Generator for OCL. *Electronic Communication of the European Association of Software Science and Technology*, 36, 2010.

[24] Jim Bainbridge et. al. Row and Column Access Control Support in IBM DB2 for i. Technical report, International Business Machines Corporation, 2014. `https://www.redbooks.ibm.com/redpapers/pdfs/redp5110.pdf`.

[25] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[26] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, Inc., USA, 2nd edition, 2007.

[27] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[28] David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[29] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based Specification Environment for Validating UML and OCL. *Journal of Science of Computer Programming*, 69(1-3):27–34, 2007.

[30] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. Redundancy and Information Leakage in Fine-Grained Access Control. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 133–144, New York, NY, USA, 2006. Association for Computing Machinery.

[31] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting Disclosure in Hippocratic Databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller,

José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 108–119. Morgan Kaufmann, 2004.

[32] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[33] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for Database-backed Systems. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1463–1479. USENIX Association, 2017.

[34] Geoff Montee. Row-Level Security in MariaDB 10: Protect Your Data, 2015. `https://mariadb.com/resources/blog/`.

[35] Hoang Phuoc Bao Nguyen and Manuel Clavel. OCL2PSQL: An OCL-to-SQL Code-Generator for Model-Driven Engineering. In Tran Khanh Dang, Josef Küng, Makoto Takizawa, and Son Ha Bui, editors, *Future Data and Security Engineering - 6th International Conference, FDSE 2019, Proceedings*, volume 11814 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2019.

[36] Object Constraint Language Specification, version 2.4. Technical report, Object Management Group, February 2014. `https://www.omg.org/spec/OCL/`.

[37] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A Formal Framework for Reflective Database Access Control Policies. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 289–298. ACM, 2008.

[38] PostgreSQL 12.2, 2017. Part II. SQL The Language. Chapter 5. Data Definition. 5.8. Row Security Policies. `https://www.postgresql.org/docs/10/ddl.html`.

[39] Roger Pressman and Bruce Maxim. *Software Engineering: A Practitioner's Approach, Ninth Edition.* McGraw Hill Publishers, 9th edition, 2019.

[40] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas. *Data Knowl. Eng.*, 73:1–22, 2012.

[41] MariaDB Server Documentation - User & Server Security - Roles. Technical report, 2021. `https://mariadb.com/kb/en/roles/`.

[42] MySQL 8.0 Refenrence Manual - 6.2.10 Using Roles. Technical report, Oracle Corporation, 2021. `https://dev.mysql.com/doc/refman/8.0/en/roles.html`.

[43] PostgreSQL 13 Documentation - Chapter 21. Database Roles. Technical report, 2021. `https://www.postgresql.org/docs/13/user-manag.html`.

[44] Security Center for SQL Server Database Engine and Azure SQL Database. Technical report, Microsoft Corporation, September 2017. `https://docs.microsoft.com/en-us/sql/relational-databases/security`.

[45] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 551–562, New York, NY, USA, 2004. Association for Computing Machinery.

[46] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[47] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, 1996.

[48] Ravi S. Sandhu, David F. Ferraiolo, and D. Richard Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In Klaus Rebensburg, Charles E. Youman, and Vijay Atluri, editors, *Fifth ACM Workshop on Role-Based Access Control, RBAC 2000, Berlin, Germany, July 26-27, 2000*, pages 47–63. ACM, 2000.

[49] ISO/IEC 9075-(1–10) Information technology – Database languages – SQL. Technical report, International Organization for Standardization,

2011. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=63555.

[50] Unified Modeling Language Specification Version 1.1. Technical report, Object Management Group, December 1997. https://www.omg.org/spec/UML/1.1/About-UML/.

[51] Unified Modeling Language Specification Version 2.0 Infrastructure. Technical report, Object Management Group, July 2005. https://www.omg.org/spec/UML/2.0/About-UML/.

[52] Data Security Guide: Using Oracle Virtual Private Database to Control Data Access. https://docs.oracle.com/database/121/DBSEG.

Appendices

Appendix A

Mapping data and object models to databases

In this appendix, we recall the specific mappings from data models and object models to SQL that are used in this thesis for enforcing FGAC policies when executing SQL queries.

The mapping of data models

In characterizing access control authorization for SQL queries [2], we assume that SQL queries are executed on databases according to the mappings defined below.

Definition 8. *Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Our mapping of data model $\mathcal{D}$ to SQL, denoted by $\overline{\mathcal{D}}$, is defined as follows:*

- *For every $c \in C$, a corresponding table c, with a primary key column $c\_\texttt{id}$, is created:*

  `CREATE TABLE` c `(`c`_id VARCHAR PRIMARY KEY);`

- *For every attribute $at \in AT$, $at = \langle atn, c, t \rangle$, a column atn, with the corresponding SQL type, is added into table c:*

  `ALTER TABLE` c `ADD COLUMN` atn $\mathrm{SqlType}(t)$`;`

where:

- *if* $t =$ `Integer`, *then* $\mathrm{SqlType}(t) =$ `INT`.
- *if* $t =$ `String`, *then* $\mathrm{SqlType}(t) =$ `VARCHAR`.
- *if* $t \in C$, *then* $\mathrm{SqlType}(t) =$ `VARCHAR`.

Moreover, if $t \in C$, *then a constraint stating that the value of this column refers to the primary key column of class* t *is included:*

```
ALTER TABLE c
  ADD FOREIGN KEY fk_c_atn(atn) REFERENCES t(t_id);
```

- *For every association* $as \in AS$, $as = \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}} \rangle$, *a corresponding table* asn, *with two columns* ase_{l} *and* ase_{r} *refers to the primary key column of class* c_l *and* c_r, *is created:*

```
CREATE TABLE asn (
  ase_l varchar NOT NULL,
  ase_r varchar NOT NULL,
  FOREIGN KEY fk_c_l_ase_l(ase_l) REFERENCES  c_l(c_l_id),
  FOREIGN KEY fk_c_r_ase_r(ase_r) REFERENCES  c_r(c_r_id)
);
```

Moreover, a constraint stating that the tuple in this table is unique, is included:

```
ALTER TABLE asn
  ADD UNIQUE unique_link(ase_l, ase_r);
```

The mapping of objects models

Definition 9. *Let* $\mathcal{D} = \langle C, AT, AS \rangle$ *be a data model. Let* $\mathcal{O} = \langle OC,\ OAT,\ OAS \rangle$ *be an object model of* $\mathcal{D}$. *Our mapping of object model* $\mathcal{O}$ *to SQL, denoted by* $\overline{\mathcal{O}}$, *is defined as follows:*

- *For every object $o \in OC$, $o = \langle oi, c \rangle$, a tuple contains only the unique object identifier oi is inserted into the primary column $c\_\texttt{id}$ of table c:*

```
INSERT INTO c(c_id) VALUES (oi);
```

- *For every attribute value $atv \in OAT$, $atv = \langle \langle atn, c, t \rangle, \langle oi, c \rangle, vl \rangle$, the value vl is updated at the attribute atn of the corresponding tuple of object $\langle oi, c \rangle$:*

```
UPDATE c SET atn = vl WHERE c_id = oi;
```

- *For every association link $asl \in OAS$, $asl = \langle \langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c_{\mathrm{r}} \rangle, \langle oi_{\mathrm{l}}, c_{\mathrm{l}} \rangle, \langle oi_{\mathrm{r}}, c_{\mathrm{r}} \rangle \rangle$, a tuple contains the object identifications of the left object $\langle oi_{\mathrm{l}}, c_{\mathrm{l}} \rangle$ and the right object $\langle oi_{\mathrm{r}}, c_{\mathrm{r}} \rangle$ is inserted into the table asn:*

```
INSERT INTO asn(ase_l, ase_r) VALUES (oi_l, oi_r);
```

Appendix B

Defining secure SQL queries

In this appendix, we recall from [4] the key components introduced in this thesis for defining the enforcement of FGAC policies when executing SQL queries.

The function SecQuery()

Informally, given an FGAC policy $\mathcal{S}$ and an SQL select-statement q, the function SecQuery() generates an SQL stored-procedure satisfying the following: if a user is authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure returns the same result as executing q; otherwise, if a user is not authorized, according to $\mathcal{S}$, to execute q, then calling this stored-procedure signals an error.

In the definition below, $\ulcorner \text{SecQuery}(\mathcal{S}, q) \urcorner$ denotes the name of the stored-procedure generated by SecQuery(), for an FGAC policy $\mathcal{S}$ and a query q. SecQuery() uses the auxiliary function SecQueryAux() that is defined in the next section.

Definition 10. *Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth} \rangle$ be a security model for $\mathcal{D}$. Let q be an SQL query in $\overline{\mathcal{D}}$. Then,* $\text{SecQuery}(\mathcal{S}, q)$ *generates the following stored-procedure:*

```
CREATE PROCEDURE ⌜SecQuery(S, q)⌝ (
    caller varchar(250), role varchar(250))
BEGIN
DECLARE _rollback int DEFAULT 0;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
  % If an error is signalled, then set _rollback to 1 and
  % return the error message.
  SET _rollback = 1;
  GET STACKED DIAGNOSTICS CONDITION 1
    @p1 = RETURNED_SQLSTATE, @p2 = MESSAGE_TEXT;
  SELECT @p1, @p2;
  ROLLBACK;
END;
START TRANSACTION;
  % For each authorization condition applicable to the original query,
  % create the corresponding temporary table.

  SecQueryAux(S, q)

  % If after creating all the temporary tables, no error has
  % been signalled yet, i.e., _rollback still has value 0,
  % then execute the original query.
IF _rollback = 0
  THEN q;
END IF;
END
```

The function SecQueryAux

The definition of SecQueryAux() proceeds recursively. In the definition below, ⌜TempTable(q, *exp*)⌝ denotes the name of the temporary table generated by SecQuery, for a query q and a (sub-)expression *exp* in q.

A subtle, but important point in the definition of SecQueryAux() has to do with the way of handling read-access authorization for tables representing associations. The definition of SecQueryAux() assumes that the policies' underlying data models, as well as its object models, are implemented in SQL following the *mapping* introduced in Appendix A. According to this mapping, the rows in the association-tables only represent the links of the given association that exist between objects. In other words, if a link does not exist, this information is not stored anywhere. Thus, when checking if a user is authorized to know the links of a given association, it should be performed not only the appropriate checks on the rows contained in the corresponding association-table, but also on the rows contained in its (virtual) *complement*, i.e., on those rows represent the links that *do not exist* between objects. For this reason, in the definition of SecQueryAux() below, when handling read-access authorization for tables representing associations, it is considered the Cartesian product of the two end-tables involved in the given association, checking read-access authorization for *every* row in the Cartesian product.

Next, the different cases in the recursive definition of the function SecQueryAux() are introduced. For each case, the authorization conditions that need to be satisfied are informally introduced as well. As mentioned before, these conditions have been formally defined in [2]. According to these conditions, not only the data that appears in the final result, but any data that is *used* when executing a query (in particular, data used by sub-queries, where-clauses, and on-clauses) must be checked for policy-compliance. To this end, the function SecQueryAux() uses the function SecAtt() to add the corresponding authorization checks to any expression accessing specific attribute values, and the function SecAs() to add the corresponding authorization checks to access association links. These functions will be introduced in the next section. The function SecAttList(), also used by SecQueryAux(), simply iteratively applies SecAtt() to each of the expressions in an expression list. Finally, in the definitions below, RepExp() denotes the result of replacing, within an expression, each occurrence of the association's association-ends by the corresponding association-ends' class-identifier.

Case q = `SELECT` *selitems* `FROM` c `WHERE` *exp*.
To execute q, the following conditions must be satisfied:

- The user is authorized to access the information required to evaluate the where-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

```
CREATE TEMPORARY TABLE ⌜TempTable(q, exp)⌝ AS (
  SELECT * FROM c WHERE SecAtt(S, exp)
);
CREATE TEMPORARY TABLE ⌜TempTable(q, selitems)⌝ AS (
  SELECT SecAttList(S, selitems) FROM ⌜TempTable(q, exp)⌝
);
```

Case q = `SELECT` *selitems* `FROM` *as* `WHERE` *exp*.
To execute q, the following conditions must be satisfied:

- The user is authorized to access the information referred to by both association-ends, but only for the rows contained in the Cartesian product between the classes involved in the association that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

```
CREATE TEMPORARY TABLE ⌜TempTable(q, exp)⌝ AS (
  SELECT c_l_id as ase_l, c_r_id as ase_r FROM c_l, c_r
    WHERE RepExp(exp, as)
);
CREATE TEMPORARY TABLE ⌜TempTable(q, selitems)⌝ AS (
  SELECT selitems FROM ⌜TempTable(q, exp)⌝ WHERE SecAs(S, as)
);
```

Case $q =$ `SELECT` *selitems* `FROM` *subselect* `WHERE` *exp*.
To execute q, the following conditions must be satisfied:

- The user is authorized to execute the sub-query *subselect*.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)

Case $q =$ `SELECT` *selitems* `FROM` c `JOIN` *as* `ON` *exp* `WHERE` *exp*$'$.
To execute q, the following conditions must be satisfied:

- The user is authorized to access the information referred to by both association-ends in *as*.
- The user is authorized to access the information required to evaluate the on-clause *exp*.
- The user is authorized to access the information required to evaluate the where-clause *exp*$'$, but only for the objects/rows and links/rows that satisfy the on-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows and links/rows that satisfy the on-clause *exp* and the where-clause *exp*$'$.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, `SELECT` $*$ `FROM` *as*)
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*)⌝ `AS` (
  `SELECT` $*$ `FROM` c `JOIN` *as* `ON` SecAtt($\mathcal{S}$, *exp*)
);
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*$'$)⌝ `AS` (
  `SELECT` $*$ `FROM` ⌜TempTable(q, *exp*⌝) `WHERE` SecAtt($\mathcal{S}$, *exp*$'$)
);
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *selitems*)⌝ `AS` (
  `SELECT` SecAttList($\mathcal{S}$, *selitems*) `FROM` ⌜TempTable(q, *exp*$'$)⌝
);

Case $q =$ `SELECT` *selitems* `FROM` c `JOIN` *subselect* `ON` *exp* `WHERE` exp'.
To execute q, the following conditions must be satisfied:

- The user is authorized to execute the sub-query *subselect*.
- The user is authorized to access the information required to evaluate the on-clause *exp*.
- The user is authorized to access the information required to evaluate the where-clause exp'; but only for the objects/rows and links/rows that satisfy the on-clause *exp*.
- The user is authorized to access the information referred to by *selitems*, but only for the objects/rows and links/rows that satisfy the on-clause *exp* and the where-clause exp'.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*)⌝ `AS` (
  `SELECT * FROM` c `JOIN` *subselect* `ON` SecAtt($\mathcal{S}$, *exp*)
);
`CREATE TEMPORARY TABLE` ⌜TempTable(q, exp')⌝ `AS` (
  `SELECT * FROM` ⌜TempTable(q, *exp*)⌝ `WHERE` SecAtt($\mathcal{S}$, exp')
);
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *selitems*)⌝ `AS` (
  `SELECT` SecAttList($\mathcal{S}$, *selitems*) `FROM` ⌜TempTable(q, exp')⌝
);

Case $q =$ `SELECT` *selitems* `FROM` *as* `JOIN` *subselect* `ON` *exp* `WHERE` exp'.
Three cases must be considered:

(i) The case when ase_{l} appears in *exp*, but ase_{r} does not appear in *exp*. Let *col* be the column in *subselect* that ase_{l} is related to in *exp*. To execute q, the following conditions must be satisfied:

- The user is authorized to execute the sub-query *subselect*.

- The user is authorized to access the information referred to by both association-ends, but only for the rows contained in the Cartesian product between the classes involved in the association that satisfy the where-clause *exp*.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *exp*)⌝ `AS (`
  `SELECT` c_{l}`_id as` ase_{l}, *col* `as` ase_{r} `FROM` c_{l}, *subselect*
    `ON` RepExp(*exp*, *as*) `WHERE` RepExp(*exp*′, *as*)
`);`
`CREATE TEMPORARY TABLE` ⌜TempTable(q, *as*)⌝ `AS (`
  `SELECT * FROM` ⌜TempTable(q, *exp*)⌝ `WHERE` SecAs($\mathcal{S}$, *as*)
`);`

(ii) The case when ase_{r} appears in *exp*, but ase_{l} does not appear in *exp*. This case is resolved analogously to the previous case.

(iii) The case when both ase_{r} and ase_{l} appear in *exp*. To execute q, the following conditions must be satisfied:

- The user is authorized to execute the sub-query *subselect*.
- The user is authorized to access the information referred to by both association-ends.

For this case, SecQueryAux() returns the following create-statements:

SecQueryAux($\mathcal{S}$, *subselect*)
SecQueryAux($\mathcal{S}$, `SELECT * FROM` *as*)

Case q = `SELECT` *selitems* `FROM` $subselect_1$ `JOIN` $subselect_2$ `ON` *exp* `WHERE` *exp*′**.** To execute q, the following conditions must be satisfied:

- The user is authorized to execute the sub-queries $subselect_1$ and $subselect_2$.

For this case, SecQueryAux() returns the following create-statements:

$$\text{SecQueryAux}(\mathcal{S}, \mathit{subselect}_1)$$
$$\text{SecQueryAux}(\mathcal{S}, \mathit{subselect}_2)$$

The function SecAtt()

The function SecQueryAux() uses SecAtt() to *wrap* any access to a protected attribute *at* into a case-expression. The value of this case expression is a call to a function AuthFunc() that implements the authorization checks required for accessing the corresponding attribute. If the result of this function-call is `TRUE`, then the case-expression will return the requested resource; otherwise, it will signal an error. The function AuthFunc() is defined in the following section. In what follows, $\ulcorner\text{AuthFunc}(\mathcal{S}, at)\urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an attribute *at*; when the argument $\mathcal{S}$ is clear from the context, it may be omitted.

Definition 11. *Let $\mathcal{D} = \langle C, AT, AS\rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth}\rangle$ be a security model for $\mathcal{D}$. Let exp be an SQL expression in $\overline{\mathcal{D}}$.* $\text{SecAtt}(\mathcal{S}, exp)$ *denotes the SQL expression in $\overline{\mathcal{D}}$ that results from replacing each attribute $at = \langle atn, c, t\rangle$ in exp by the following case-expression:*

```
CASE ⌜AuthFunc(at)⌝ (c_id, caller, role)
  WHEN 1 THEN at
  ELSE throw_error() END as at.
```

where the function `throw_error()` *is defined as followed:*

```
CREATE FUNCTION throw_error()
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Unauthorized access';
  RETURN (0);
END
```

The function SecAs()

The function SecQueryAux() uses SecAs() to *wrap* any access to a protected association *as* into a where case-expression. The value of this case expression is a call to the function AuthFunc() that, in this case, implements the authorization checks required for accessing the corresponding association-ends. If the result of this function-call is `TRUE`, then the case-expression will also return `TRUE`; otherwise, it will signal an error. The function AuthFunc() is defined in the following section. In what follows, $\ulcorner \text{AuthFunc}(\mathcal{S}, as) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$ an association *as*; when the argument $\mathcal{S}$ is clear from the context, it may be omitted.

Definition 12. *Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth} \rangle$ be a security model for $\mathcal{D}$. Let as be an association class in $\mathcal{D}$. Let ase_l and ase_r be the association-ends of as.* $\text{SecAs}(\mathcal{S}, as)$ *denotes the SQL expression in $\overline{\mathcal{D}}$ that results by the following case-expression:*

```
CASE ⌜AuthFunc(as)⌝ (asel, aser, caller, role)
  WHEN 1 THEN TRUE
  ELSE throw_error() END
```

where the function `throw_error()` *is defined as before.*

The function AuthFunc()

The functions SecAtt() and SecAs() use this function to check that the access to a specific protected resource is authorized. For each protected resource, the required authorization checks depend on the role of the user attempting to access this resource. Accordingly, for each role, the function AuthFunc() calls a function AuthFuncRole() that implements the authorization checks required for a user with that role to access a specific protected resource. The function AuthFuncRole() will be introduced in the next section. In what follows, $\ulcorner \text{AuthFuncRole}(\mathcal{S}, rs, r) \urcorner$ denotes the name of the function generated by SecQuery() for a policy $\mathcal{S}$, a resource rs, and a role r; when the argument $\mathcal{S}$ is clear from the context, we may omit it.

Definition 13. *Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data model. Let $\mathcal{S} = \langle R, \text{auth} \rangle$ be a security model for $\mathcal{D}$, with $R = \{r_1, r_2, \ldots, r_n\}$. Let at be an attribute in AT. Then,* $\text{AuthFunc}(at)$ *generates the following SQL function:*

```
CREATE FUNCTION ⌜AuthFunc(at)⌝ ( self varchar(250),
  caller varchar(250), role varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (role = r_1)
    THEN RETURN ⌜AuthFuncRole(at, r_1)⌝(self, caller)
  ⋮
  ELSE IF (role = r_n)
    THEN RETURN ⌜AuthFuncRole(at, r_n)⌝(self, caller)
  ELSE RETURN 0
  END IF;
  ⋮
  END IF;
END
```

Similarly, let as be an association in AS. Then AuthFunc(as) *generates the following SQL function:*

```
CREATE FUNCTION ⌜AuthFunc(as)⌝ ( left varchar(250),
  right varchar(250), caller varchar(250), role varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (role = r_1)
    THEN RETURN ⌜AuthFuncRole⌝(as, r_1) (left, right, caller)
  ⋮
  ELSE IF (role = r_n)
    THEN RETURN ⌜AuthFuncRole(as, r_n)⌝ (left, right, caller)
  ELSE RETURN 0
  END IF;
  ⋮
  END IF;
END
```

The function AuthFuncRole()

The function AuthFuncRole() implements the authorization constraints associated with the permission for users of a given role for executing a given read-action on a specific resource. There are many different ways of implementing in SQL an OCL authorization constraint. The definition of the function AuthFuncRole() only assumes that there exists a function map() that, for each authorization constraint of interest, it returns its preferred SQL implementation. Without loss of generality, it also assumes that this implementation, when executed, will return an SQL Boolean.[1]

Definition 14. *Let* $\mathcal{D} = \langle C, AT, AS \rangle$ *be a data model. Let* $\mathcal{S} = \langle R, \text{auth} \rangle$ *be a security model for* $\mathcal{D}$*. Let* r *be a role in* R*. Let* $at = \langle atn, c, t \rangle$ *be an attribute in* AT*. Then,* AuthFuncRole(at, r) *generates the following SQL function:*

```
CREATE FUNCTION ⌜AuthFuncRole(at, r)⌝ ( self varchar(250),
  caller varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT * INTO result FROM map(auth(r, read(at))) AS TEMP;
  RETURN result;
END
```

Similarly, let $as = \langle asn, ase_{\text{l}}, c_{\text{l}}, ase_{\text{r}}, c_{\text{r}} \rangle \in AS$*, be an association in* AS*. Then,* AuthFuncRole(as, r) *generates the following SQL function:*

```
CREATE FUNCTION ⌜AuthFuncRole(as, r)⌝ ( left varchar(250),
  right varchar(250), caller varchar(250))
RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT * INTO result FROM map(auth(r, read(as))) AS TEMP;
  RETURN result;
END
```

[1] Recently, OCL2PSQL [35] was introduced as a mapping which only uses standard SQL subselects and joins for translating OCL iterators. This mapping can certainly be used as map()-function. However, current experiments in [17] suggest that, for non-trivial authorization constraints, manually-written implementations significantly outperforms those automatically generated by OCL2PSQL, when checking FGAC authorization in large databases.

Appendix C

SQLSI: representing data models using JSON

In this appendix, we recall the JSON representation of data model. Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data models.

Let $c \in C$ be a class in $\mathcal{D}$. We denote by $\text{Atts}(c, AT)$ the attributes of the class c in $\mathcal{D}$. Let $at = \langle atn, c, t \rangle, at \in \text{Atts}(c, AT)$ be an attribute in $\mathcal{D}$. Then, the corresponding JSON-object $\text{json}(at)$ is defined as follows:

{
    `name` : atn,
    `type` : t
}

Also, we denote by $\text{json}(\text{Atts}(c, AT))$ the JSON-array containing the JSON-objects corresponding to the attributes in $\text{Atts}(c, AT)$.

Moreover, let $c \in C$ be a class in $\mathcal{D}$. We denote by $\text{Ends}(c, AS)$ the associations in $\mathcal{D}$ that have the class c at one of their ends. Let $as = \langle asn, ase_{\text{l}}, c, ase_{\text{r}}, c_{\text{r}} \rangle, as \in \text{Ends}(c, AS)$ be an association in $\mathcal{D}$. Then, the corresponding JSON-object $\text{json}(as)$ is defined as follows:

{
    `association` : asn,
    `name` : ase_{r},
    `target` : c_{r},

`opp` : ase_{l},
`mult` : $*$
}

Analogously, let $\langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c\rangle \in \mathrm{Ends}(c, AS)$ be an association in $\mathcal{D}$. Then, the corresponding JSON-object json(as) is defined as follows:

{
`association` : asn,
`name` : ase_{l},
`target` : c_{l},
`opp` : ase_{r},
`mult` : $*$
}

Also, we denote by json(Ends(c, AS)) the JSON-array containing the JSON-objects corresponding to the associations in Ends(c, AS).

Next, let $c \in C$ be a class in $\mathcal{D}$. Then, the corresponding JSON-object json(c) is defined as follows:

{
`class` : c,
`attributes` : json(Atts(c, AT)),
`ends` : json(Ends(c, AS))
}

Finally, we denote by json($\mathcal{D}$) the JSON-array containing the JSON-objects corresponding to the classes in $\mathcal{D}$.

Appendix D

SQLSI: representing security models using JSON

In this appendix, we recall the JSON representation of our FGAC security model. Let $\mathcal{D} = \langle C, AT, AS \rangle$ be a data models. Let $c \in C$ be a class in $\mathcal{D}$. We denote by Res(c) the *resources* of the class c, i.e., the union of the sets Atts(c, AT) and Ends(c, AT).

Let $\mathcal{S} = \langle R, auth \rangle$ be a security model of $\mathcal{D}$. Let $r \in R$ a role in $\mathcal{S}$. Notice that the function auth() together with the role r define an equivalence relationship Res(c), as follows: let rsc, rsc' in Res(c), then $[r, c, rsc] \equiv [r, c, rsc']$ if and only if auth($r, read(rsc)$) = auth($r, read(rsc')$).

We denote by $Auths(r, c)$ the authorization constraints corresponding to the different equivalence classes defined by the function auth(), together with the role r, in Res(c).

Let $rsc \in$ Res(c) be a resource of the class c. Then, the corresponding JSON-object json(rsc) is defined as follows:

- if $rsc = \langle atn, c, t \rangle$, then json($rsc$) is the following object:

```
{
    entity : c,
    attribute : atn
}
```

- if *rsc* is either $\langle asn, ase_{\mathrm{l}}, c_{\mathrm{l}}, ase_{\mathrm{r}}, c\rangle$ or $\langle asn, ase_{\mathrm{l}}, c, ase_{\mathrm{r}}, c_{\mathrm{r}}\rangle$, then json(*rsc*) is the following object:

  {
  `association` : *asn*
  }

Let *auth* be an authorization constraint in $Auths(r, c)$. Then, we denote by json(*auth*) the following JSON-object:

{
`role` : r,
`action`: `read`,
`resources` : json($[r, c, auth]$),
`auth` : *auth*
}

Appendix E

SQLSI: generated artifacts

In this appendix, we display the SQL statements, functions and stored-procedures related to the examples of this thesis.

SQLSI implementation of the `Uni` data model

Listing E.1: `Uni` data model: The SQLSI implementation

```
1  /* create Lecturer table */
2  CREATE TABLE Lecturer (Lecturer_id VARCHAR (100) PRIMARY KEY);
3  ALTER TABLE Lecturer ADD COLUMN email VARCHAR (100);
4  ALTER TABLE Lecturer ADD COLUMN age INT (11);
5  ALTER TABLE Lecturer ADD COLUMN name VARCHAR (100);
6
7  /* create Student table */
8  CREATE TABLE Student (Student_id VARCHAR (100) PRIMARY KEY);
9  ALTER TABLE Student ADD COLUMN email VARCHAR (100);
10 ALTER TABLE Student ADD COLUMN age INT (11);
11 ALTER TABLE Student ADD COLUMN name VARCHAR (100);
12
13 /* create Enrollment association */
14 CREATE TABLE Enrollment (
15     lecturers VARCHAR (100), students VARCHAR (100),
16     FOREIGN KEY (lecturers) REFERENCES Lecturer (Lecturer_id),
17     FOREIGN KEY (students) REFERENCES Student (Student_id)
18 );
19 ALTER TABLE Enrollment
20     ADD UNIQUE unique_link(lecturers,students);
```

SQLSI implementation of the security model

Listing E.2: `Sec#1` security model: The SQLSI implementation

```
DROP FUNCTION IF EXISTS auth_READ_Lecturer_age;
/* FUNC: auth_READ_Lecturer_age */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_email;
/* FUNC: auth_READ_Lecturer_email */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_email(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_name;
/* FUNC: auth_READ_Lecturer_name */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_name(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_age;
/* FUNC: auth_READ_Student_age */
DELIMITER //
CREATE FUNCTION auth_READ_Student_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
```

```
44    DECLARE result INT DEFAULT 0;
45    IF (krole = `Admin`)
46      THEN IF (auth_READ_Student_age_Admin(kself, kcaller))
47        THEN RETURN (1);
48        ELSE RETURN (0);
49      END IF;
50    ELSE RETURN 0;
51    END IF;
52  END //
53  DELIMITER ;
54
55  DROP FUNCTION IF EXISTS auth_READ_Student_age_Admin;
56  /* FUNC: auth_READ_Student_age_Admin */
57  DELIMITER //
58  CREATE FUNCTION auth_READ_Student_age_Admin(
59    kself varchar(100), kcaller varchar(100)
60  ) RETURNS INT DETERMINISTIC
61  BEGIN
62    DECLARE result INT DEFAULT 0;
63    SELECT res INTO result FROM
64    (SELECT (TRUE) AS res) AS TEMP;
65    RETURN (result);
66  END //
67  DELIMITER ;
68
69  DROP FUNCTION IF EXISTS auth_READ_Student_email;
70  /* FUNC: auth_READ_Student_email */
71  DELIMITER //
72  CREATE FUNCTION auth_READ_Student_email(
73    kcaller varchar(100), krole varchar(100), kself varchar(100)
74  ) RETURNS INT DETERMINISTIC
75  BEGIN
76    DECLARE result INT DEFAULT 0;
77    RETURN 0;
78  END //
79  DELIMITER ;
80
81  DROP FUNCTION IF EXISTS auth_READ_Student_name;
82  /* FUNC: auth_READ_Student_name */
83  DELIMITER //
84  CREATE FUNCTION auth_READ_Student_name(
85    kcaller varchar(100), krole varchar(100), kself varchar(100)
86  ) RETURNS INT DETERMINISTIC
87  BEGIN
88    DECLARE result INT DEFAULT 0;
89    RETURN 0;
```

```
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Enrollment;
/* FUNC: auth_READ_Enrollment */
DELIMITER //
CREATE FUNCTION auth_READ_Enrollment(
  kcaller varchar(100), krole varchar(100),
  klecturers varchar(100), kstudents varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
  IF (krole = 'Admin')
    THEN IF (auth_READ_Enrollment_Admin(klecturers,
      kstudents, kcaller))
      THEN RETURN (1);
      ELSE RETURN (0);
    END IF;
  ELSE RETURN 0;
  END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Enrollment_Admin;
/* FUNC: auth_READ_Enrollment_Admin */
DELIMITER //
CREATE FUNCTION auth_READ_Enrollment_Admin(
  klecturers varchar(100), kstudents varchar(100), kcaller
     varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT res INTO result FROM
  (SELECT (TRUE) AS res) AS TEMP;
  RETURN (result);
END //
DELIMITER ;
```

Listing E.3: Sec#2 security model: The SQLSI implementation

```
DROP FUNCTION IF EXISTS auth_READ_Lecturer_age;
/* FUNC: auth_READ_Lecturer_age */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_email;
/* FUNC: auth_READ_Lecturer_email */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_email(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_name;
/* FUNC: auth_READ_Lecturer_name */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_name(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_age;
/* FUNC: auth_READ_Student_age */
DELIMITER //
CREATE FUNCTION auth_READ_Student_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (krole = 'Lecturer')
```

```
46      THEN IF (auth_READ_Student_age_Lecturer(kself, kcaller))
47        THEN RETURN (1);
48        ELSE RETURN (0);
49      END IF;
50    ELSE RETURN 0;
51    END IF;
52  END //
53  DELIMITER ;
54
55  DROP FUNCTION IF EXISTS auth_READ_Student_age_Lecturer;
56  /* FUNC: auth_READ_Student_age_Lecturer */
57  DELIMITER //
58  CREATE FUNCTION auth_READ_Student_age_Lecturer(
59    kself varchar(100), kcaller varchar(100)
60  ) RETURNS INT DETERMINISTIC
61  BEGIN
62    DECLARE result INT DEFAULT 0;
63    SELECT res INTO result
64    FROM (SELECT ((SELECT MAX(age) FROM Lecturer)
65      = (SELECT age FROM Lecturer
66        WHERE Lecturer_id = kcaller)) AS res) AS TEMP;
67    RETURN (result);
68  END //
69  DELIMITER ;
70
71  DROP FUNCTION IF EXISTS auth_READ_Student_email;
72  /* FUNC: auth_READ_Student_email */
73  DELIMITER //
74  CREATE FUNCTION auth_READ_Student_email(
75    kcaller varchar(100), krole varchar(100), kself varchar(100)
76  ) RETURNS INT DETERMINISTIC
77  BEGIN
78    DECLARE result INT DEFAULT 0;
79    RETURN 0;
80  END //
81  DELIMITER ;
82
83  DROP FUNCTION IF EXISTS auth_READ_Student_name;
84  /* FUNC: auth_READ_Student_name */
85  DELIMITER //
86  CREATE FUNCTION auth_READ_Student_name(
87    kcaller varchar(100), krole varchar(100), kself varchar(100)
88  )
89  RETURNS INT DETERMINISTIC
90  BEGIN
91    DECLARE result INT DEFAULT 0;
```

```
92    RETURN 0;
93  END //
94  DELIMITER ;
95
96  DROP FUNCTION IF EXISTS auth_READ_Enrollment;
97  /* FUNC: auth_READ_Enrollment */
98  DELIMITER //
99  CREATE FUNCTION auth_READ_Enrollment(
100   kcaller varchar(100), krole varchar(100),
101   klecturers varchar(100), kstudents varchar(100)
102 ) RETURNS INT DETERMINISTIC
103 BEGIN
104   DECLARE result INT DEFAULT 0;
105   IF (krole = 'Lecturer')
106     THEN IF (auth_READ_Enrollment_Lecturer(klecturers,
107       kstudents, kcaller))
108       THEN RETURN (1);
109       ELSE RETURN (0);
110     END IF;
111   ELSE RETURN 0;
112   END IF;
113 END //
114 DELIMITER ;
115
116 DROP FUNCTION IF EXISTS auth_READ_Enrollment_Lecturer;
117 /* FUNC: auth_READ_Enrollment_Lecturer */
118 DELIMITER //
119 CREATE FUNCTION auth_READ_Enrollment_Lecturer(
120   klecturers varchar(100), kstudents varchar(100),
121   kcaller varchar(100)
122   ) RETURNS INT DETERMINISTIC
123 BEGIN
124   DECLARE result INT DEFAULT 0;
125   SELECT res INTO result
126   FROM (SELECT ((SELECT MAX(age) FROM Lecturer)
127     = (SELECT age FROM Lecturer
128       WHERE Lecturer_id = kcaller)) AS res) AS TEMP;
129   RETURN (result);
130 END //
131 DELIMITER ;
```

Listing E.4: `Sec#3` security model: The SQLSI implementation

```
DROP FUNCTION IF EXISTS auth_READ_Lecturer_age;
/* FUNC: auth_READ_Lecturer_age */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_email;
/* FUNC: auth_READ_Lecturer_email */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_email(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Lecturer_name;
/* FUNC: auth_READ_Lecturer_name */
DELIMITER //
CREATE FUNCTION auth_READ_Lecturer_name(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_age;
/* FUNC: auth_READ_Student_age */
DELIMITER //
CREATE FUNCTION auth_READ_Student_age(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  IF (krole = 'Lecturer')
```

```
    THEN IF (auth_READ_Student_age_Lecturer(kself, kcaller))
      THEN RETURN (1);
      ELSE RETURN (0);
    END IF;
  ELSE RETURN 0;
  END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_age_Lecturer;
/* FUNC: auth_READ_Student_age_Lecturer */
DELIMITER //
CREATE FUNCTION auth_READ_Student_age_Lecturer(
  kself varchar(100), kcaller varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  SELECT res INTO result
  FROM (SELECT (EXISTS (
    SELECT 1 FROM Enrollment
    WHERE lecturers = kcaller AND kself = students)
    )as res
  ) AS TEMP;
  RETURN (result);
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_email;
/* FUNC: auth_READ_Student_email */
DELIMITER //
CREATE FUNCTION auth_READ_Student_email(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
  DECLARE result INT DEFAULT 0;
  RETURN 0;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_Student_name;
/* FUNC: auth_READ_Student_name */
DELIMITER //
CREATE FUNCTION auth_READ_Student_name(
  kcaller varchar(100), krole varchar(100), kself varchar(100)
) RETURNS INT DETERMINISTIC
BEGIN
```

```
92    DECLARE result INT DEFAULT 0;
93    RETURN 0;
94  END //
95  DELIMITER ;
96
97  DROP FUNCTION IF EXISTS auth_READ_Enrollment;
98  /* FUNC: auth_READ_Enrollment */
99  DELIMITER //
100 CREATE FUNCTION auth_READ_Enrollment(
101   kcaller varchar(100), krole varchar(100),
102   klecturers varchar(100), kstudents varchar(100)
103 ) RETURNS INT DETERMINISTIC
104 BEGIN
105   DECLARE result INT DEFAULT 0;
106   IF (krole = 'Lecturer')
107     THEN IF (auth_READ_Enrollment_Lecturer(klecturers,
108       kstudents, kcaller))
109       THEN RETURN (1);
110       ELSE RETURN (0);
111     END IF;
112   ELSE RETURN 0;
113   END IF;
114 END //
115 DELIMITER ;
116
117 DROP FUNCTION IF EXISTS auth_READ_Enrollment_Lecturer;
118 /* FUNC: auth_READ_Enrollment_Lecturer */
119 DELIMITER //
120 CREATE FUNCTION auth_READ_Enrollment_Lecturer(
121   klecturers varchar(100), kstudents varchar(100), kcaller
        varchar(100)
122 ) RETURNS INT DETERMINISTIC
123 BEGIN
124   DECLARE result INT DEFAULT 0;
125   SELECT res INTO result FROM
126   (SELECT (EXISTS (SELECT 1 FROM Enrollment
127       WHERE lecturers = kcaller AND kstudents = students)
128     )as res
129   ) AS TEMP;
130   RETURN (result);
131 END //
132 DELIMITER ;
```

SQLSI implementation of the secure stored-procedure of Query#1

Listing E.5: Query#1: The generated SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query1;
DELIMITER //
CREATE PROCEDURE Query1(
  in kcaller varchar(250),
  in krole varchar(250)
)
BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
  SELECT @p1, @p2;
  ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT * FROM Student
    WHERE CASE auth_READ_Student_age(kcaller,
      krole, Student_id) WHEN 1 THEN age
      ELSE throw_error() END > 18
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP2;
  CREATE TEMPORARY TABLE TEMP2 AS (
    SELECT Student_id AS Student_id FROM TEMP1
  );
  IF _rollback = 0
  THEN SELECT COUNT(*) FROM TEMP2;
  END IF;
END //
DELIMITER ;
```

SQLSI implementation of the optimized stored-procedure of Query#1

Listing E.6: Query#1: The optimized SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query1Opt;
DELIMITER //
CREATE PROCEDURE Query1Opt(
  in kcaller varchar(250),
  in krole varchar(250)
)
BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
    SELECT @p1, @p2;
    ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT * FROM Student WHERE age > 18
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP2;
  CREATE TEMPORARY TABLE TEMP2 AS (
    SELECT Student_id AS Student_id FROM TEMP1
  );
  IF _rollback = 0
  THEN SELECT COUNT(*) FROM TEMP2;
  END IF;
END //
DELIMITER ;
```

SQLSI implementation of the secure stored-procedure of Query#2

Listing E.7: Query#2: The generated SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query2;
DELIMITER //
CREATE PROCEDURE Query2(
  in kcaller varchar(250),
  in krole varchar(250)
)
BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
    SELECT @p1, @p2;
    ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT Lecturer_id AS lecturers, Student_id AS students
    FROM Lecturer, Student
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP2;
  CREATE TEMPORARY TABLE TEMP2 AS (
    SELECT * FROM TEMP1
    WHERE CASE auth_READ_Enrollment(kcaller, krole,
      lecturers, students) WHEN TRUE THEN TRUE
      ELSE throw_error() END
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP3;
  CREATE TEMPORARY TABLE TEMP3 AS (
    SELECT students FROM Enrollment
  );
  IF _rollback = 0
  THEN SELECT COUNT(*) FROM TEMP3;
  END IF;
END //
DELIMITER ;
```

SQLSI implementation of the optimized stored-procedure of Query#2

Listing E.8: Query#2: The optimized SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query2Opt;
DELIMITER //
CREATE PROCEDURE Query2Opt(
  in kcaller varchar(250),
  in krole varchar(250)
)
BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
    SELECT @p1, @p2;
    ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT Student_id AS students, Lecturer_id AS lecturers
    FROM Student, Lecturer WHERE TRUE
  );
  IF (SELECT (SELECT COUNT(*) FROM Student)
    = (SELECT COUNT(*)
        FROM (SELECT COUNT(*) AS size FROM Enrollment
                  GROUP BY students) AS TEMP
        WHERE TEMP.size = (SELECT COUNT(*) FROM Lecturer)))
  THEN
    DROP TEMPORARY TABLE IF EXISTS TEMP2;
    CREATE TEMPORARY TABLE TEMP2 AS (
      SELECT * FROM TEMP1 WHERE TRUE
    );
  ELSE
    DROP TEMPORARY TABLE IF EXISTS TEMP2;
    CREATE TEMPORARY TABLE TEMP2 AS (
      SELECT * FROM TEMP1
      WHERE CASE auth_READ_Enrollment(kcaller, krole,
        lecturers, students) WHEN TRUE THEN TRUE
        ELSE throw_error() END
    );
  END IF;
  DROP TEMPORARY TABLE IF EXISTS TEMP3;
```

```
44    CREATE TEMPORARY TABLE TEMP3 AS (
45      SELECT students FROM Enrollment
46    );
47    IF _rollback = 0
48    THEN SELECT COUNT(*) FROM TEMP3;
49    END IF;
50  END //
51  DELIMITER ;
```

SQLSI implementation of the secure stored-procedure of Query#3

Listing E.9: Query#3: The generated SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query3;
DELIMITER //
CREATE PROCEDURE Query3(
  in kcaller varchar(250),
  in krole varchar(250)
)
BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
    SELECT @p1, @p2;
    ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT Student_id AS students, Lecturer_id AS lecturers
    FROM Student, Lecturer WHERE Lecturer_id = kcaller
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP2;
  CREATE TEMPORARY TABLE TEMP2 AS (
    SELECT * FROM TEMP1
    WHERE CASE auth_READ_Enrollment(kcaller, krole,
      lecturers, students) WHEN TRUE THEN TRUE
      ELSE throw_error() END
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP3;
  CREATE TEMPORARY TABLE TEMP3 AS (
    SELECT * FROM Student JOIN TEMP2
    ON Student_id = students
  );
  DROP TEMPORARY TABLE IF EXISTS TEMP4;
  CREATE TEMPORARY TABLE TEMP4 AS (
    SELECT CASE auth_READ_Student_age(kcaller,
      krole, Student_id) WHEN 1 THEN age
      ELSE throw_error() END as age
    FROM TEMP3
  );
  IF _rollback = 0
```

```
44    THEN SELECT AVG(age) FROM TEMP4;
45    END IF;
46  END //
47  DELIMITER ;
```

SQLSI implementation of the optimized stored-procedure of Query#3

Listing E.10: Query#3: The optimized SQL stored-procedure.

```
DROP PROCEDURE IF EXISTS Query3Opt;
DELIMITER //
CREATE PROCEDURE Query3Opt(
  in kcaller varchar(250),
  in krole varchar(250)
)
  BEGIN
  DECLARE _rollback int DEFAULT 0;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    SET _rollback = 1;
    GET STACKED DIAGNOSTICS CONDITION 1
      @p1 = RETURNED_SQLSTATE,
      @p2 = MESSAGE_TEXT;
    SELECT @p1, @p2;
    ROLLBACK;
  END;
  START TRANSACTION;
  DROP TEMPORARY TABLE IF EXISTS TEMP1;
  CREATE TEMPORARY TABLE TEMP1 AS (
    SELECT Student_id AS students, Lecturer_id AS lecturers
    FROM Student, Lecturer WHERE Lecturer_id = kcaller
  );
  IF (SELECT (SELECT COUNT(*) FROM Student)
    = (SELECT COUNT(*)
        FROM (SELECT COUNT(*) AS size FROM Enrollment
                  GROUP BY students) AS TEMP
        WHERE TEMP.size = (SELECT COUNT(*) FROM Lecturer)))
  THEN
    DROP TEMPORARY TABLE IF EXISTS TEMP2;
    CREATE TEMPORARY TABLE TEMP2 AS (
      SELECT * FROM TEMP1 WHERE TRUE
    );
  ELSE
    DROP TEMPORARY TABLE IF EXISTS TEMP2;
    CREATE TEMPORARY TABLE TEMP2 AS (
      SELECT * FROM TEMP1
      WHERE CASE auth_READ_Enrollment(kcaller, krole,
        lecturers, students) WHEN TRUE THEN TRUE
        ELSE throw_error() END
    );
  END IF;
  DROP TEMPORARY TABLE IF EXISTS TEMP3;
```

```
44    CREATE TEMPORARY TABLE TEMP3 AS (
45      SELECT * FROM Student JOIN TEMP2
46      ON Student_id = students
47    );
48    DROP TEMPORARY TABLE IF EXISTS TEMP4;
49    CREATE TEMPORARY TABLE TEMP4 AS (
50      SELECT age FROM TEMP3
51    );
52    IF _rollback = 0
53    THEN SELECT AVG(age) FROM TEMP4;
54    END IF;
55  END //
56  DELIMITER ;
```

Appendix F

MSFOL: generated theories

In this appendix, we display the generated MSFOL formulae, theories related to the case study in Chapter 5 and other examples in the thesis.

The MSFOL theory for the `Uni` data model

Listing F.1: `Uni` data model: The generated MSFOL theory in SMT-LIB

```
; sort declaration
(declare-sort Classifier 0)

; null and invalid object and its axiom
(declare-const nullClassifier Classifier)
(declare-const invalClassifier Classifier)
(assert (distinct nullClassifier invalClassifier))

; null and invalid integer and its axiom
(declare-const nullInt Int)
(declare-const invalInt Int)
(assert (distinct nullInt invalInt))

; null and invalid string and its axiom
(declare-const nullString String)
(declare-const invalString String)
(assert (distinct nullString invalString))

; unary predicate Lecturer(x) and its axiom
(declare-fun Lecturer (Classifier) Bool)
(assert (not (Lecturer nullClassifier)))
(assert (not (Lecturer invalClassifier)))
```

```
23
24 ; unary predicate Student(x) and its axiom
25 (declare-fun Student (Classifier) Bool)
26 (assert (not (Student nullClassifier)))
27 (assert (not (Student invalClassifier)))
28
29 ; axiom: disjoint set of objects of different classes
30 (assert (forall ((x Classifier))
31     (=> (Lecturer x) (not (Student x)))))
32 (assert (forall ((x Classifier))
33     (=> (Student x) (not (Lecturer x)))))
34
35 ; function get the age of lecturer and its axiom
36 (declare-fun age_Lecturer (Classifier) Int)
37 (assert (= (age_Lecturer nullClassifier) invalInt))
38 (assert (= (age_Lecturer invalClassifier) invalInt))
39 (assert (forall ((x Classifier))
40     (=> (Lecturer x)
41         (distinct (age_Lecturer x) invalInt))))
42
43 ; function get the email of lecturer and its axiom
44 (declare-fun email_Lecturer (Classifier) String)
45 (assert (= (email_Lecturer nullClassifier) invalString))
46 (assert (= (email_Lecturer invalClassifier) invalString))
47 (assert (forall ((x Classifier))
48     (=> (Lecturer x)
49         (distinct (email_Lecturer x) invalString))))
50
51 ; function get the name of lecturer and its axiom
52 (declare-fun name_Lecturer (Classifier) String)
53 (assert (= (name_Lecturer nullClassifier) invalString))
54 (assert (= (name_Lecturer invalClassifier) invalString))
55 (assert (forall ((x Classifier))
56     (=> (Lecturer x)
57         (distinct (name_Lecturer x) invalString))))
58
59 ; function get the age of student and its axiom
60 (declare-fun age_Student (Classifier) Int)
61 (assert (= (age_Student nullClassifier) invalInt))
62 (assert (= (age_Student invalClassifier) invalInt))
63 (assert (forall ((x Classifier))
64     (=> (Student x)
65         (distinct (age_Student x) invalInt))))
66
67 ; function get the name of student and its axiom
68 (declare-fun name_Student (Classifier) String)
```

```
(assert (= (name_Student nullClassifier) invalString))
(assert (= (name_Student invalClassifier) invalString))
(assert (forall ((x Classifier))
    (=> (Student x)
        (distinct (name_Student x) invalString))))

; function get the email of student and its axiom
(declare-fun email_Student (Classifier) String)
(assert (= (email_Student nullClassifier) invalString))
(assert (= (email_Student invalClassifier) invalString))
(assert (forall ((x Classifier))
    (=> (Student x)
        (distinct (email_Student x) invalString))))

; binary predicate of the Enrollment association
; and its axiom
(declare-fun Enrollment (Classifier Classifier) Bool)
(assert (forall ((x Classifier))
    (forall ((y Classifier))
        (=> (Enrollment x y)
            (and (Lecturer x) (Student y))))))
```

OCL expression: Sample generated MSFOL formula

Listing F.2: OCL expression: The generated MSFOL formulae
`Student.allInstances()` → `select(s|s.age` ≥ `19)` → `isEmpty()`

```
; function repr. of non-boolean expression
; exp' = Student.allInstances()->select(s|s.age > 19)
(declare-fun temp (Classifier) Bool)

; definition of predicate temp
(assert (forall ((s Classifier))
    (= (temp s)
    (and (Student s)
         (and (> (age_Student s) 19)
              (not (or (= (age_Student s) nullInt)
                       (or (= s nullClassifier)
                           (= s invalClassifier))
                       false false)))))))

; 19 cannot be interpreted by nullInt constant symbol
(assert (distinct nullInt 19))

; 19 cannot be interpreted by invalIn constant symbol
(assert (distinct invalInt 19))

; map from exp'->isEmpty
(assert (forall ((x Classifier))
    (and (not (temp x)) (not false))))
```

Example 5.1 generated theory

Listing F.3: Example 5.1: The generated MSFOL formulae.

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of self and its axiom
(declare-const kself Classifier)
(assert (Student kself))

; authorization constraint: Admin can read student age
(assert (not true))
```

Example 5.2 generated theories

Listing F.4: Example 5.2: The generated MSFOL formulae.

```
; the generated MSFOL theory for data model
; is removed due to its length

; invariant: Every lecturer is lecturer of every student
(assert (forall ((l Classifier))
  (and (=> (Lecturer l)
       (forall ((s Classifier))
          (and (=> (Student s)
                   (exists ((temp Classifier))
                       (and (Enrollment l temp)
                            (= temp s)
                              (not (or (= l nullClassifier)
                                       (= l invalidClassifier)))
                            (not (= s invalidClassifier)))))
               (not false))))
   (not false))))

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
                 (and (Enrollment temp kstudents)
                      (= temp kcaller)
                      (not (or (= kstudents nullClassifier)
                               (= kstudents invalidClassifier)))
                      (not (= kcaller invalidClassifier))))))
```

Listing F.5: Example 5.2: The generated MSFOL formulae, without the data invariant.

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
                (and (Enrollment temp kstudents)
                     (= temp kcaller)
                     (not (or (= kstudents nullClassifier)
                              (= kstudents invalidClassifier)))
                     (not (= kcaller invalidClassifier))))))
```

Listing F.6: Example 5.2: The generated MSFOL formulae, under security model `Sec#2`

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; this TEMP0 function is the OCL expression
; Lecturer.allInstances()->select(l|l.age > caller.age)
(declare-fun TEMP0 (Classifier) Bool)
(assert (forall ((l Classifier))
    (= (TEMP0 l)
       (and (Lecturer l)
            (and (> (age_Lecturer l)
                    (age_Lecturer kcaller))
                 (not (or (= (age_Lecturer l) nullInt)
                          (or (= l nullClassifier)
                              (= l invalidClassifier))
                          (= (age_Lecturer kcaller) nullInt)
                          (or (= kcaller nullClassifier)
                              (= kcaller invalidClassifier)))))))
                                 ))

; authorization constraint: caller is the oldest lecturer
(assert (not (forall ((x Classifier))
                     (and (not (TEMP0 x))
                          (not false))))))
```

Example 5.3 theories

Listing F.7: Example 5.3: The generated MSFOL formulae.

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; caller property: caller is indeed the oldest lecturer
(assert (forall ((l Classifier))
    (and (=> (Lecturer l)
             (and (<= (age_Lecturer l) (age_Lecturer kcaller))
                  (not (or (= (age_Lecturer l) nullInt)
                           (or (= l nullClassifier)
                               (= l invalidClassifier))
                           (= (age_Lecturer kcaller) nullInt)
                           (or (= kcaller nullClassifier)
                               (= kcaller invalidClassifier))))))
         (not false))))

; this TEMP0 function is the OCL expression
; Lecturer.allInstances()->select(l|l.age > caller.age)
(declare-fun TEMP0 (Classifier) Bool)
(assert (forall ((l Classifier))
  (= (TEMP0 l)
     (and (Lecturer l)
          (and (> (age_Lecturer l) (age_Lecturer kcaller))
               (not (or (= (age_Lecturer l) nullInt)
                        (or (= l nullClassifier)
                            (= l invalidClassifier))
                        (= (age_Lecturer kcaller) nullInt)
                        (or (= kcaller nullClassifier)
                            (= kcaller invalidClassifier)))))))))

; authorization constraint: caller is the oldest lecturer
(assert (not (forall ((x Classifier))
                     (and (not (TEMP0 x)) (not false))))))
```

Listing F.8: Example 5.3: The generated MSFOL formulae, without `caller` properties

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; this TEMP0 function is the OCL expression
; Lecturer.allInstances()->select(l|l.age > caller.age)
(declare-fun TEMP0 (Classifier) Bool)
(assert (forall ((l Classifier))
  (= (TEMP0 l)
     (and (Lecturer l)
          (and (> (age_Lecturer l) (age_Lecturer kcaller))
               (not (or (= (age_Lecturer l) nullInt)
                        (or (= l nullClassifier)
                            (= l invalidClassifier))
                        (= (age_Lecturer kcaller) nullInt)
                        (or (= kcaller nullClassifier)
                            (= kcaller invalidClassifier)))))))))

; authorization constraint: caller is the oldest lecturer
(assert (not (forall ((x Classifier))
                     (and (not (TEMP0 x)) (not false))))))
```

Listing F.9: Example 5.3: The generated MSFOL formulae, under security model `Sec#3`

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; caller property: caller is indeed the oldest lecturer
(assert (forall ((l Classifier))
    (and (=> (Lecturer l)
             (and (<= (age_Lecturer l) (age_Lecturer kcaller))
                  (not (or (= (age_Lecturer l) nullInt)
                           (or (= l nullClassifier)
                               (= l invalidClassifier))
                           (= (age_Lecturer kcaller) nullInt)
                           (or (= kcaller nullClassifier)
                               (= kcaller invalidClassifier))))))
         (not false))))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
                (and (Enrollment temp kstudents)
                     (= temp kcaller)
                     (not (or (= kstudents nullClassifier)
                              (= kstudents invalidClassifier)))
                     (not (= kcaller invalidClassifier))))))
```

Example 5.4 theories

Listing F.10: Example 5.4: The generated MSFOL formulae for the first authorization checks

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; invariant: Every lecturer is lecturer of every student
(assert (forall ((l Classifier))
  (and (=> (Lecturer l)
       (forall ((s Classifier))
          (and (=> (Student s)
                   (exists ((temp Classifier))
                       (and (Enrollment l temp)
                            (= temp s)
                              (not (or (= l nullClassifier)
                                       (= l invalidClassifier)))
                            (not (= s invalidClassifier)))))
               (not false))))
   (not false))))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
                 (and (Enrollment temp kstudents)
                      (= temp kcaller)
                      (not (or (= kstudents nullClassifier)
                               (= kstudents invalidClassifier)))
                      (not (= kcaller invalidClassifier))))))
```

Listing F.11: Example 5.4: The generated MSFOL formulae for the first authorization checks, without the data invariant

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
                (and (Enrollment temp kstudents)
                     (= temp kcaller)
                     (not (or (= kstudents nullClassifier)
                              (= kstudents invalidClassifier)))
                     (not (= kcaller invalidClassifier))))))
```

Listing F.12: Example 5.4: The generated MSFOL formulae for the first authorization checks, under security model `Sec#2`

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of lecturers and its axiom
(declare-const klecturers Classifier)
(assert (Lecturer klecturers))

; constant symbol of students and its axiom
(declare-const kstudents Classifier)
(assert (Student kstudents))

; invariant: Every lecturer is lecturer of every student
(assert (forall ((l Classifier))
  (and (=> (Lecturer l)
       (forall ((s Classifier))
          (and (=> (Student s)
                   (exists ((temp Classifier))
                       (and (Enrollment l temp)
                            (= temp s)
                              (not (or (= l nullClassifier)
                                       (= l invalidClassifier)))
                            (not (= s invalidClassifier)))))
               (not false))))
   (not false))))

; this TEMP0 function is the OCL expression
; Lecturer.allInstances()->select(l|l.age > caller.age)
(declare-fun TEMP0 (Classifier) Bool)
(assert (forall ((l Classifier))
  (= (TEMP0 l)
     (and (Lecturer l)
          (and (> (age_Lecturer l) (age_Lecturer kcaller))
               (not (or (= (age_Lecturer l) nullInt)
                        (or (= l nullClassifier)
                            (= l invalidClassifier))
                        (= (age_Lecturer kcaller) nullInt)
                        (or (= kcaller nullClassifier)
                            (= kcaller invalidClassifier)))))))))

; authorization constraint: caller is the oldest lecturer
(assert (not (forall ((x Classifier))
                     (and (not (TEMP0 x)) (not false))))))
```

Listing F.13: Example 5.4: The generated MSFOL formulae for the second authorization checks

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of self and its axiom
(declare-const kself Classifier)
(assert (Student kself))

; self property: self is a student of lecturer
(assert (exists ((temp Classifier))
    (and (Enrollment kcaller temp)
         (= temp kself)
         (not (or (= kcaller nullClassifier)
                  (= kcaller invalidClassifier)))
         (not (= kself invalidClassifier)))))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
    (and (Enrollment kcaller temp)
         (= temp kself)
         (not (or (= kcaller nullClassifier)
                  (= kcaller invalidClassifier)))
         (not (= kself invalidClassifier))))))
```

Listing F.14: Example 5.4: The generated MSFOL formulae for the second authorization checks, without the `self` properties

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of self and its axiom
(declare-const kself Classifier)
(assert (Student kself))

; authorization constraint: a lecturer can know the
; students of any lecturer, if the student is his
; or her student
(assert (not (exists ((temp Classifier))
    (and (Enrollment kcaller temp)
         (= temp kself)
         (not (or (= kcaller nullClassifier)
                  (= kcaller invalidClassifier)))
         (not (= kself invalidClassifier))))))
```

Listing F.15: Example 5.4: The generated MSFOL formulae for the second authorization checks, under security model `Sec#2`

```
; the generated MSFOL theory for data model
; is removed due to its length

; constant symbol of caller and its axiom
(declare-const kcaller Classifier)
(assert (Lecturer kcaller))

; constant symbol of self and its axiom
(declare-const kself Classifier)
(assert (Student kself))

; self property: self is a student of lecturer
(assert (exists ((temp Classifier))
    (and (Enrollment kcaller temp)
         (= temp kself)
         (not (or (= kcaller nullClassifier)
                  (= kcaller invalidClassifier)))
         (not (= kself invalidClassifier)))))

; this TEMP0 function is the OCL expression
; Lecturer.allInstances()->select(l|l.age > caller.age)
(declare-fun TEMP0 (Classifier) Bool)
(assert (forall ((l Classifier))
  (= (TEMP0 l)
     (and (Lecturer l)
          (and (> (age_Lecturer l) (age_Lecturer kcaller))
               (not (or (= (age_Lecturer l) nullInt)
                        (or (= l nullClassifier)
                            (= l invalidClassifier))
                        (= (age_Lecturer kcaller) nullInt)
                        (or (= kcaller nullClassifier)
                            (= kcaller invalidClassifier)))))))))

; authorization constraint: caller is the oldest lecturer
(assert (not (forall ((x Classifier))
                     (and (not (TEMP0 x)) (not false))))))
```

(This is the end of the thesis)