A Domain Specific Language for Security Model in Database-centric applications

Final Project of Formal Model Driven Engineering

Student: Hoàng Nguyễn Phước Bảo

Universidad Autónoma de Madrid, Spain

Abstract. In this report, I propose a low-code approach to realize a model-driven proposal to enforce fine-grained access control in database-centric applications. The works involve designing the related metamodels and their domain specific languages, then, applying some model transformation and model manipulation techniques to generate final executable artifacts on the database level.

1 Introduction

Model-Driven Engineering (MDE) is a software development methodology that focuses on creating *models* of different views of a system, and then automatically generating different system artifacts from these models, such as code and configuration data. *Model-Driven Security* (MDS) is a specialization of model-driven engineering for developing secure systems. In a nutshell, designers specify system models along with their security requirements and use tools to automatically generate security-related system artifacts, such as access control infrastructures.

SQL Security Injector (SQLSI) is my first and latest work in the MDS discipline. The project attempts to enforce *fine-grained access control* (FGAC) in database using model-driven methodology. So far, this involves (i) to prospose a formal model to express such access control policy and a rigorous design to enforce it in database-centric application[], then (ii) to realize the aforementioned work in a notably case study[], finally, (iii) to prove its correctness. In regards to the second goal, since the inputs in [] are indeed models of different kinds, I believe it makes sense to apply a more *low-code* approach, taking advantage of the knowledge I have learnt from the course, to replace the current *ad-hoc* realization represented in [].

Organization The rest of the report is organized as follows. In Section 2, I provide some main remarks about the SQLSI idea and design, including some related works and their drawbacks, from which uplift the approach in this report. Next, in Section 3, I describe the input and output of this new approach. Then, in Section 4, provides the in more detail the realization and technology used. Finally, in Section 5, I provide conclusion and possible future works.

2 Background and Motivation

2.1 Traditional approach vs SQLSI approach

Figure 1 shows the main difference between traditional and model-driven approach in enforcing FGAC in database-centric application. In particular:

Traditional approach On the left-hand side is the traditional approach to enforce FGAC in database-centric applications. Firstly, the policy will be manually implemented and enforced on the application level. Next, given a SQL statement, the application layer sends it to the data layer from which returns the result. Then, the application layer must enforce the FGAC policy to the result and finally return the answer to the client.

SQLSI approach On the right-hand side is the SQLSI model-based approach. This approach requires an application *data-model* that corresponds to the underlying database in the data layer and a *security-model* that defines the FGAC policy and respects the given data-model. The SQLSI component have two main functionalities:

(i) Given those data- and security-models, SQLSI component generates a set of SQL *authorization functions* which are executable in the database.
(ii) Then, instead of issuing directly the SQL statements from the application layer, SQLSI will *rewrite* these statements into semantically-equivalent SQL *secure stored-procedures* in which *injects* the given access control policy via means of aforementioned SQL *authorization functions* in (i).

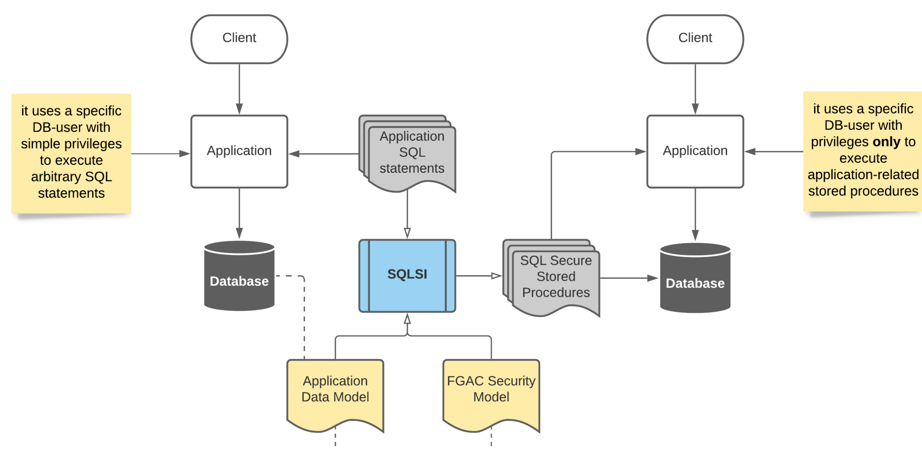

Fig. 1: Traditional approach vs. SQLSI approach

2.2 SQLSI realization vs low-code realization

In this paper, we focus on realizing the first functionality of SQLSI, in a more low-code fashion. Particularly, the following work will be primary:

(i) Define the metamodel for data-model, security-model, and their domain-specific language.
(ii) Define customized-model of relational database schema and the transformation from data-model to this relational database schema model.
(iii) Define the code-generation from my relational database schema model to actual SQL database script.
(iv) Define the transformation from the *user-input* version to the *normalized* version of the security-model.
(v) Define the code-generation from security-model to a set of SQL authentication functions.

Figure 2 shows the workflow in a nutshell, including the technology used, whose description will be explained in Section 3 and Section 4.

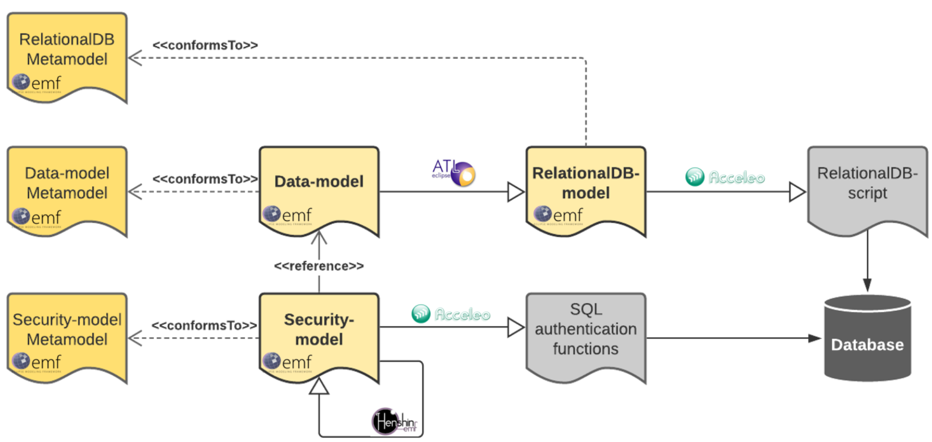

Fig. 2: Proposed approach - in a nutshell

2.3 Related work

[] presents a Java implementation of the whole SQLSI component, including the SQL authorization functions generation. In paritcular, the datamodel and securitymodel are represented as Java object hierarchies and consequently, every model transformation and manipulation was implemented manually through the help of the visitor pattern. Indeed, this is a very ad-hoc approach and less modern, compared to the "one-click-of-a-button" implementation in this report.

[?,?] are the latest attempts to implement intelligent web-based editors for datamodel and securitymodel. In particular, these implementations are the final products of bachelor theses. On the one hand, these approaches are, again, very ad-hoc, since any change in the model requires (a handful of) manual changes in the source code. And on the other hand, due to the time allotted for the final thesis, the students do not have enough time to thoroughly understand the idea behind these models and therefore implemented in a way that is very hard to maintain or extend. In my approach, we avoid both drawbacks by using the available modelling technology, for instances, Xtext and ATL.

3 The SQLSI Metamodels

3.1 Input Metamodels

Metamodel for data-models For SQLSI, a data-model contains entities and associations between them. An entity may have properties which are attributes or associations-ends.

The data-model metamodel for SQLSI is shown in Figure 3. The `DataModel` is the root element: it has a `name` and contains a set of `Entity`s. Every `Entity` represents an entity in the data model: it has a `name` and contains a set of `Property`(-ies). Every `Property` can be either an `Attribute` or an `AssociationEnd`.

– Each `Attribute` represents an attribute of the entity: it unique-ability depends on a boolean value `isUnique` and its `type` is either `String` or `Integer`.
– Each `AssociationEnd` represents an association between its container `Entity` with another `Entity`[1]: its `Multiplicity` is either `MANY` or `ONE`. Finally, each `AssociationEnd` is linked to its opposite `AssociationEnd`, and also with its `targetEntity`.

Metamodel for security-models For SQLSI, a security-model of a data-model contains a list of "fine-grained" rules which represent the authorization constraints for a data-model. Each rule formally states that under which *roles*, under which *actions*, under which *conditions*, the user can access which *resources*.

The security-model for SQLSI is shown in Figure 4. The `SecurityModel` is the root element, it has a `name` and contains a set of `ProtectedResource`s, a set of `Rule`s and a set of `Role`s. Naturally, every instance of `SecurityModel` is associated with an instance of `DataModel`.

– Each `ProtectedResource` represents a resource that the security-model intend to enforce fine-grained access control checks. Every protected resource has a `name` and is linked to a `Property` of the referenced `DataModel`.
– Each `Role` represents a role in the application: it has a `name`. Naturally, all the roles must referred to the *user*-`Entity` instance from the referenced `DataModel`.

[1] not necessary be a different one

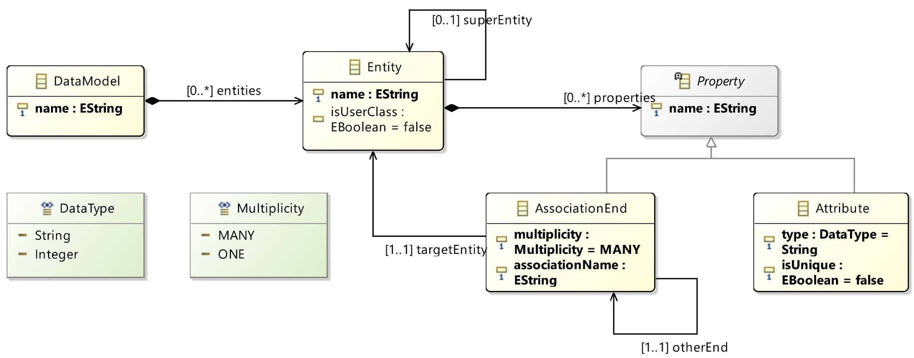

Fig. 3: SQLSI metamodel for data models.

– Each `Rule` represents a FGAC-rule in the application: it has a `name` and accomodates with an `action` (chosen from CRUD). Furthermore, every rule also refers to the resources that it intends to protect along a set of `Auth`orization conditions, whose meaning is to state the scenarios when the *resource* is allowed to perform the *action*.[2].

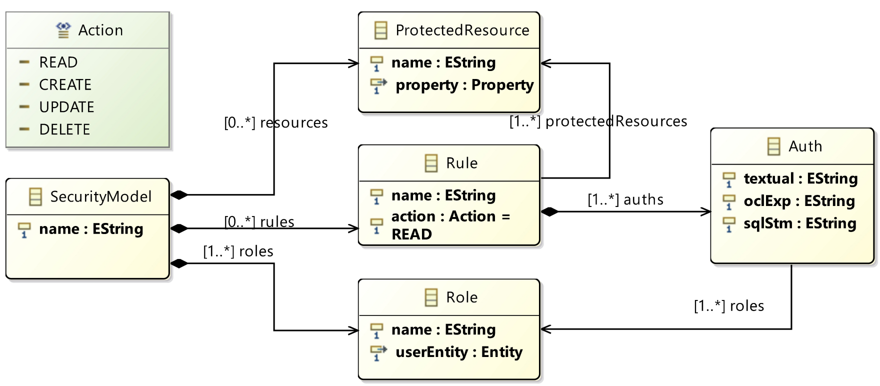

Fig. 4: SQLSI metamodel for security models.

[2] In this model, we allow user to write authorization constraints under three different means: either textual, OCL constraint or SQL boolean statement. Furthermore, each authorization constraint can be enforced for a set of `Roles`

3.2 Output Metamodels

Metamodel for Relational Schema The relational schema metamodel is shown in Figure 5. It is self-explanatory, therefore its descrption shall be omitted for the sake of space limit. [3]

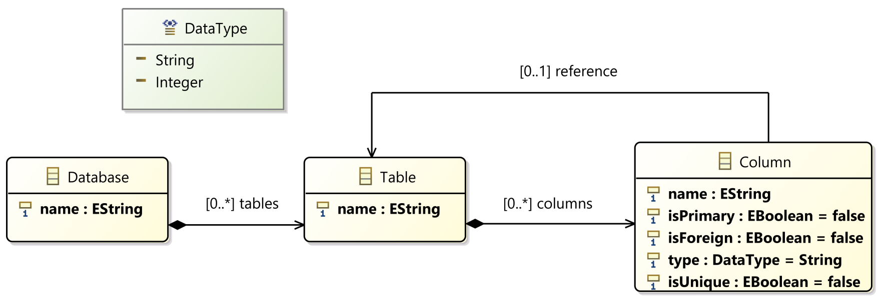

Fig. 5: SQLSI metamodel for relational database.

4 The SQLSI Language and Design

For the sake of clarity, let us continue this section of SQLSI language and design by associating it with a running example.

4.1 A Simple University Management System

Considering a simple UML class diagram `UniversityDM` in Figure 6 containing four entities: `RegisterUser`, for representing register users; `Lecturer`, for representing lecturers; `Student`, for representing students; and `Course`, for representing the courses in the university. Every `RegisterUser` have a `firstname`, a `middlename`, a `lastname` and a unique `email`. A `Lecturer` is a `RegisterUser` with a `salary` attribute, for representing his/her monthly income. A `Student` is also a `RegisterUser` with an `intake`. A `Course` has a `name` and a `year`. Every `Cource` is taught by exactly one `Lecturer` and can have none or many `Students`. Additionally, every `Lecturer` can teach none or many `Courses` as well as every `Student` can enroll in none or many `Courses`.

4.2 Domain-Specific Language for the Input Models

DSL for data-model Figure 7 shows `UniversityDM` written in our domain specific language.[4] This language syntax is inspired by the ActionGUI[] datamodel language specification.

[3] All metamodel invariants will be specified in Xtext, for the sake of consistency.

[4] For the interested readers, the specification of this language can be seen at here (for the syntax) and here (for the additional features).

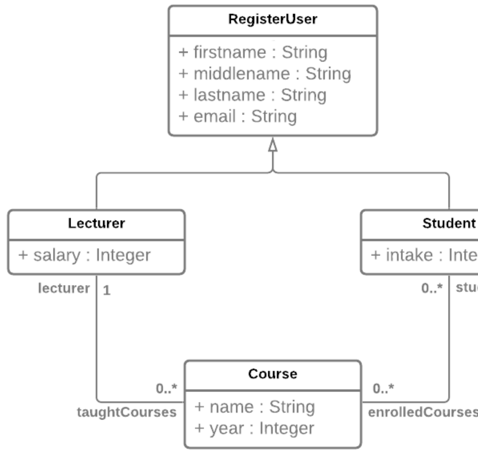

Fig. 6: SQLSI metamodel for security models.

```
DataModel University :
user entity RegUser {
    attribute name String,
    attribute email unique String
},
entity Student extends RegUser {
    attribute intake Integer,
    association Course[*] enrolled
        oppositeTo "Course.students" in Enrollment
},
entity Lecturer extends RegUser {
    attribute salary Integer,
    association Course[*] taught
        oppositeTo "Course.lecturer" in Teaching
},
entity Course {
    attribute name String,
    attribute year Integer,
    association Student[*] students
        oppositeTo "Student.enrolled" in Enrollment,
    association Lecturer[1] lecturer
        oppositeTo "Lecturer.taught" in Teaching
}
```

Fig. 7: `UniversityDM` written in our DSL

DSL for security-model Considering the following security-model for `UniversityDM`:

- Role: There are two roles, namely, the role `Lecturer` and the role `Student`, both refer to the user entity `Reg_User`.
- Permissions:
  - Anyone can read their own information. Formally,
    `caller = self`.
  - Any student can read the basic information of other students who are classmates. Formally,
    `caller.enrolledCourses→exists(c|c.students→includes(self))`
  - Any lecturer can read the basic information of the students who attends his/her class. Formally,
    `caller.taughtCourses→exists(c|c.students→includes(self))`
  - Any lecturer can read the teaching's scheme of other lecturers. Formally,
    `caller.taughtCourses→collect(c|c.lecturer)→includes(self)`
  - Any one can read the basic information about the courses. Formally,
    `true`

Figure 8 shows a fragment of the above security-model written in our domain specific language.[5] This language syntax is inspired by the SecureUML[] language specification, which is [6].

4.3 Transformation from Data Model to Relational Schema

Definition [7]

1. Every datamodel can be transformed into a relational database schema with the same name.
2. Every entity can be transformed into a relational table with the same name. In addition, every table needs to have a generated identifier `<table_name>ID` of type `Integer`, which acts as a primary key. Furthermore, in case the transforming entity has a super one, it requires to have an additional foreign key `<super_table_name>ID` of type `Integer`.
3. Every association can be transformed into a relational table with two foreign keys, one for each association end.
4. Finally, every attribute can be transformed into a column with the same name and corresponding type.

Database Schema script generation In addtion to the above transformation, we also provide a database schema generation. In this current version, we only

[5] For the interested readers, the specification of this language can be seen at here (for the syntax) and here (for the additional features).
[6] TODO: captures the work of SecureUML
[7] TODO: Write something here

```
Rule readStudentSpecificInfo {
    action READ (studentIntake)
    auths {
        roles (Administrator)
        condition: {
            textual "Administrator can read
            any student intake year"
            oclExp "true"
            sqlStm "TRUE"
        },
        roles (Student)
        condition: {
            textual "Student can read its classmates' intake year"
            oclExp "caller.enrolled->exists(c|c.students->includes(self))"
            sqlStm "caller = self"
        },
        roles (Lecturer)
        condition: {
            textual "Lecturer can read its students' intake year"
            oclExp "caller.taught->exists(c|c.students->includes(self))"
            sqlStm "EXISTS (SELECT 1
                FROM (SELECT * FROM teaching WHERE lecturer = caller) as TEMP1
                JOIN (SELECT * FROM enrollment WHERE students = self) as TEMP2
                ON TEMP1.taught = TEMP2.enrolled)"
        }
    }
},
```

Fig. 8: A snipper of `UniversitySM`, written in our DSL

support database schema generation that is executable on MySQL[8]. For interested readers, Appendix C shows the generated `UniversityDM` MySQL database schema script.

4.4 Code generation from Security Model to a Secure Authorization SQL-function

5 Conclusions and Future Work

A Validator for the DSL of DataModel

For the sake of consistency, all metamodel invariants (including the ones that can be expressed by the metamodel) will be specified in Xtext. Considering these invariants:

- Within the same datamodel, the name of the entity must be unique.
- Within the same entity, the name of the property must be unique.
- Every entity either has no superclass and has exactly one ID attribute or has a superclass and has no ID attribute.

[8] Since our definition about data model conforms with the standard SQL in 1999, we should be able to provide generation for any given relational databases that follows this standard (i.e. Orable, PostgreSQL, etc.)

– An ID attribute must by of type Integer.
– The relation otherEnd is not reflexive.
– The relation superEntity (if exists) is not reflexive nor acyclic.
– For every association end, there exists exactly one another association end such that it be its otherEnd.

B Validator for the DSL of SecurityModel

C Sample MySQL database schema script

```
DROP DATABASE IF EXISTS UNIVERSITY;
CREATE DATABASE UNIVERSITY;
USE UNIVERSITY;

DROP TABLE IF EXISTS RegisterUser;
CREATE TABLE RegisterUser(
  RegisterUserID INT(11) PRIMARY KEY,
  firstname VARCHAR(200),
  middlename VARCHAR(200),
  lastname VARCHAR(200),
  email VARCHAR(200) UNIQUE
);
DROP TABLE IF EXISTS Lecturer;
CREATE TABLE Lecturer(
  LecturerID INT(11) PRIMARY KEY,
  RegisterUserID INT(11) UNIQUE,
  FOREIGN KEY (RegisterUserID) REFERENCES RegisterUser(RegisterUserID),
  salary INT(11)
);
DROP TABLE IF EXISTS Student;
CREATE TABLE Student(
  StudentID INT(11) PRIMARY KEY,
  RegisterUserID INT(11) UNIQUE,
  FOREIGN KEY (RegisterUserID) REFERENCES RegisterUser(RegisterUserID),
  intake INT(11)
);
DROP TABLE IF EXISTS Course;
CREATE TABLE Course(
CourseID INT(11) PRIMARY KEY,
name VARCHAR(200),
year INT(11)
);
DROP TABLE IF EXISTS Teaching;
CREATE TABLE Teaching(
  taught INT(11) UNIQUE,
  FOREIGN KEY (taught) REFERENCES Course(CourseID),
  lecturer INT(11) UNIQUE,
  FOREIGN KEY (lecturer) REFERENCES Lecturer(LecturerID)
);
DROP TABLE IF EXISTS Enrollment;
CREATE TABLE Enrollment(
  students INT(11) UNIQUE,
  FOREIGN KEY (students) REFERENCES Student(StudentID),
  enrolled INT(11) UNIQUE,
  FOREIGN KEY (enrolled) REFERENCES Course(CourseID)
);
```

Fig. 9: A MySQL database script of `UniversitySM`