

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 work involve designing the metamodels and domain specific language for the models related in the proposal, then, applying some model transformation and model manipulation techniques to generate final products.

1 Introduction

Model-Driven Engineering 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 attempt in the MDS discipline. The project involves (i) proposing a model-based characterization for fine-grained access control (FGAC) authorization for SQL queries[] and (ii) to realize the aforementioned FGAC enforcement mechanism in database-centric applications[]. Since the approach is indeed model-based, I believe it makes sense to apply a more “low-code” approach, using the technology I have learnt from the course, to replace the current ad-hoc realization.

Organization The rest of the report is organized as follows. In Section 2, I provide some background about our previous realization from which uplift the new approach written 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 implementation and technology used. Finally, in Section 5, I provide some conclusion and possible future work.

2 Background and Motivation

2.1 SQLSI in a nutshell

Figure 1 depicts the main key design of SQLSI. In particular:

- On the left-hand side is the traditional approach to enforce fine-grained access control in database-centric applications. Namely, the policy will be manually implemented and enforced on the application level. Given a SQL statement, the application layer sends it to the database 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.
- On the right-hand side is the SQLSI model-based approach. Firstly, given the application data model that corresponds to the database schema and the security model that formally defines the FGAC policy that belongs to the data model, the SQLSI component generates a set of SQL authorization functions which are executed to the database. Secondly, instead of issuing directly the SQL statements from the application layer, these statements were replaced into SQL secure stored procedures that are semantically equivalent and called.

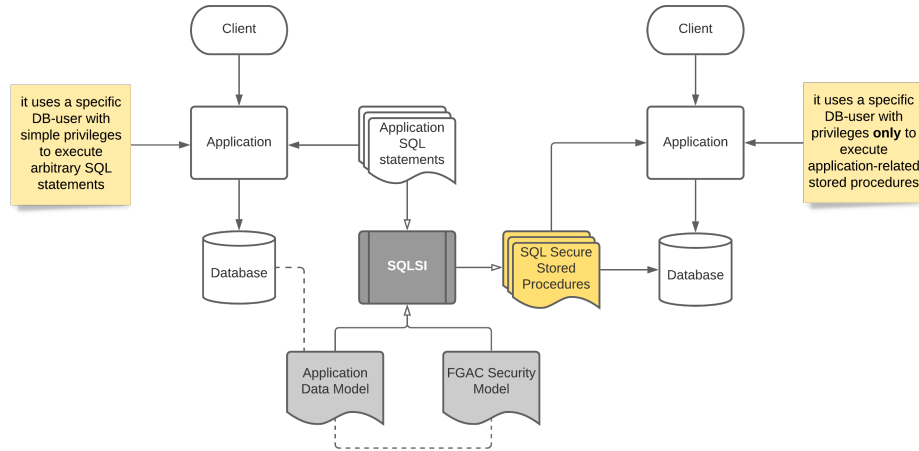


Fig. 1: Traditional approach vs. SQLSI approach

2.2 Related work

□ presents a Java implementation of the whole SQLSI component, including the SQL authorization functions generation. In particular, 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 implementation in this report.

[?,?] are the latest attempts to implement a web-based editor with intelligent validator for datamodel and securitymodel. In particular, these implementations are the final products of student bachelor theses. On the one hand, theses approach is, again, very ad-hoc, since any change in the model requires (a handful of) changes in their source code. And on the other hand, due to the time allotted for the final thesis, the students do not have enough time to understand the idea behind these models and therefore implemented in a way that is very hard to extend. In my approach, we avoid both drawbacks by using the available modelling technology, namely 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 2. The **DataModel** is the root element and contains a set of **Entity**s. Every **Entity** represents an entity in the data model: it has a unique name and is related to a set of **Property**(-ies)¹. A **Property** can be either an **Attribute** or an **AssociationEnd**.

- Each **Attribute** represents an attribute of an entity: its **type** is either **String** or **Integer**.
- Each **AssociationEnd** represents an association between this **Entity** with another **Entity**: its **Multiplicity** is either **MANY** or **ONE**. Each **AssociationEnd** is also linked to its opposite **AssociationEnd**, and with its **targetEntity**.

Metamodel for data models ² The securitymodel for SQLSI is shown in Figure 3. The **SecurityModel** is the root element, it has a **name** and contains a set of **Rules** and **Roles**. Naturally, every instance of **SecurityModel** is associated with an instance of **DataModel**.

- Each **Rule** represents a set of authorization rules in the policy. In particular, it states that under which **action**, which **resources** can be accessed by whom.³

¹ TODO: Check this

² TODO: Mentioned all metamodel invariants will be specified in Xtext, for the sake of consistency.

³ TODO: Clarify this with SecureUML, maybe a paragraph at the introduction would clarify the decision

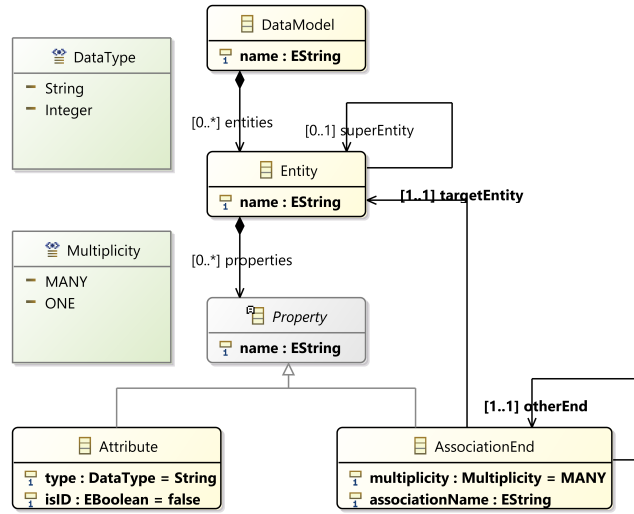


Fig. 2: SQLSI metamodel for data models.

- Each *protected*-resource will associate with a **Property** from the **DataModel**.
- In this model, we allow user to write authorization constraints under three different means: either textual, OCL boolean expression or SQL query. Furthermore, each authorization constraint can be enforced for a set of users with specific **Roles**.
- Each **Role** is associated with a certain **Entity** from the **DataModel**, possibly the *user-Entity*.

3.2 Output Metamodels

Relational Schema Metamodel The relational schema metamodel is shown in Figure 4. It is self-explanatory, therefore its description shall be omitted for the sake of space limit.⁴

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 5 containing four entities: **RegisterUser**, for representing register users; **Lecturer**, for

⁴ All metamodel invariants will be specified in Xtext, for the sake of consistency.

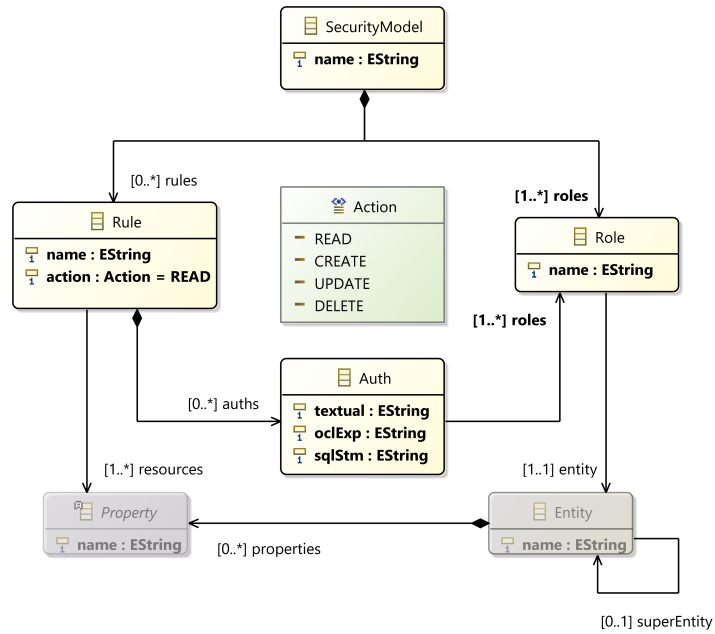


Fig. 3: SQLSI metamodel for security models.

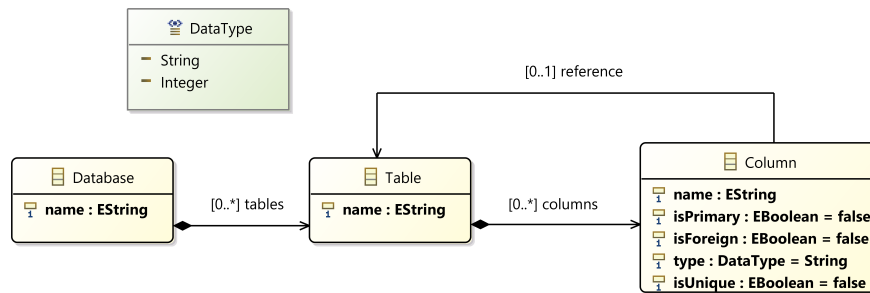


Fig. 4: SQLSI metamodel for relational database.

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 **Couse** has a **name** and a **year**. Every **Course** 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**.

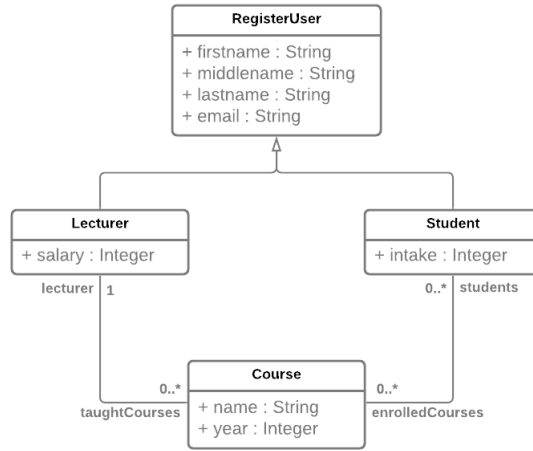


Fig. 5: SQLSI metamodel for security models.

4.2 Domain-Specific Language for the Input Models

A Domain-Specific Language for DataModel Figure 7 shows **UniversityDM** written in our domain specific language. This language syntax is inspired by the **ActionGUI** datamodel language specification. For the interested readers, the specification of this language can be seen at [here](#) (for the syntax) and [here](#) (for the additional features).

A Domain-Specific Language for SecurityModel Considering the following security model for **UniversityDM**, denoted as $S \upharpoonright \text{UniversityDM}$:

- Role: There are two roles, namely, the role **Lecturer** (associate with the entity **Lecturer**) and the role **Student** (associate with the entity **Student**).
- Permissions:

```

DataModel University:
  entity RegUser {
    attribute (firstname) String,
    attribute (middlename) String,
    attribute (lastname) String,
    attribute (email) unique String
  },
  entity Lecturer extends RegUser {
    attribute (salary) Integer,
    association Set(Course) taughtCourses
    oppositeTo "Course.lecturer" in Teaching
  },
  entity Student extends RegUser {
    attribute (intake) Integer,
    association Set(Course) enrolledCourses
    oppositeTo "Course.students" in Enrollment
  },
  entity Course {
    attribute (name) String,
    attribute (year) Integer,
    association Single(Lecturer) lecturer
    oppositeTo "Lecturer.taughtCourses" in Teaching,
    association Set(Student) students
    oppositeTo "Student.enrolledCourses" in Enrollment
  }

```

Fig. 6: UniversityDM written in our DSL

- 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`

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

4.3 Transformation from Data Model to Relational Schema

Definition ⁵

1. Every datamodel can be transformed into a relational database schema with the same name.

⁵ TODO: Write something here

```

rules:
  readStudentInfo:
    READ("University.RegUser.firstname",
          "University.RegUser.lastname",
          "University.RegUser.middlename",
          "University.RegUser.email",
          "University.Student.intake")
    conditions {
      {
        roles(Lecturer)
        context: "Any lecturer can read the info
of the students who attends his/her class."
        ocl: "caller.taughtCourses->exists(c|c.students->includes(self))"
        sql: ""
      },or
      {}
    }
  readStudentClassStatus:
    READ("University.Student.enrolledCourses", "University.Course.students")
    conditions {
      {}
      {}
    }
  defaultReadBasicInformation: {}
  defaultReadLecturerInformation: {}
  defaultReadStudentInformation: {}

```

Fig. 7: A snippet of UniversitySM, written in our DSL

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 generation with Acceleo In addition to the above transformation, we also provide a database schema generation.⁶ Due to the space limitation, I attached the sample MySQL database schema script in Appendix C.

⁶ Please note that we have only provide a mapping for that conforms MySQL. 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. Oracle, PostgreSQL, etc.)

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.
- An ID attribute must be 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. 8: A MySQL database script of UniversitySM