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.

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 attempt in the MDS discipline. The project involves (i) prosposing a model-based characterization for fine-grained access control (FGAC) authorization for Structural Query Language (SQL) statements[] and (ii) to realize the aforementioned FGAC enforcement mechanism in database-centric applications[]. In regards to the latter goal, since the approach is indeed model-based, 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[1] the current ad-hoc realization.

Organization The rest of the report is organized as follows. In Section 2, I provide some background about the SQLSI idea and design, including some related works and their drawbacks, from which uplift the novel 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.

[1] TODO: A better word?

2 Background and Motivation

2.1 Traditional approach vs SQLSI approach

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. 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.
- 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 formally defines the FGAC policy that conforms to the data-model. From those models, SQLSI component generates a set of SQL authorization functions which are executable to the database. Then, instead of issuing directly the SQL statements from the application layer, these statements will be rewritten into semantically equivalent SQL secure stored procedures which "injects" the access control policy and be called afterwards.

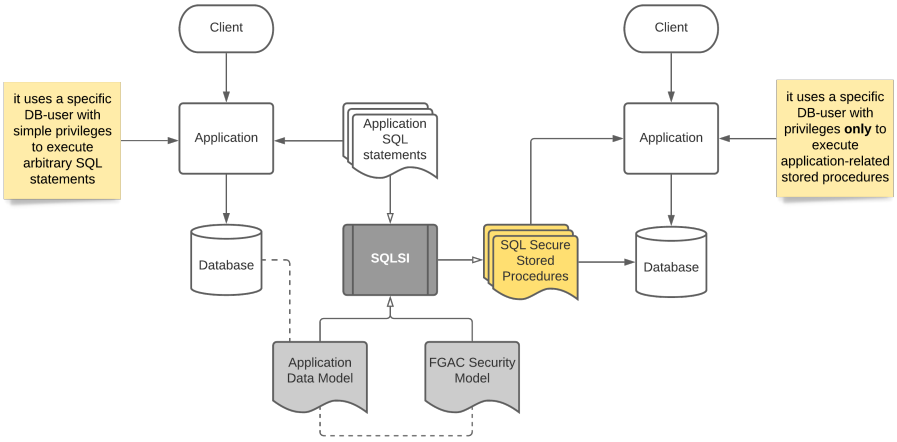

Fig. 1: Traditional approach vs. SQLSI approach

2.2 SQLSI realization vs low-code realization

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 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 extend[2]. 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 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)[3]. A `Property` can be either an `Attribute` or an `AssociationEnd`.

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

Metamodel for security-models [4] For SQLSI, a security-model of a data-model contains a list of rules which represent the authorization constraints for the given data-model. Each rule formally states that under which roles, under which actions, under which conditions, the user can access what resources.

The security-model for SQLSI is shown in Figure 3. The `SecurityModel` is the root element, it has a `name` and contains a set of `Rule`s and `Role`s. Naturally, every instance of `SecurityModel` is associated with an instance of `DataModel`.

[2] TODO: find a better word...
[3] TODO: Check this
[4] TODO: Mentioned all metamodel invariants will be specified in Xtext, for the sake of consistency.

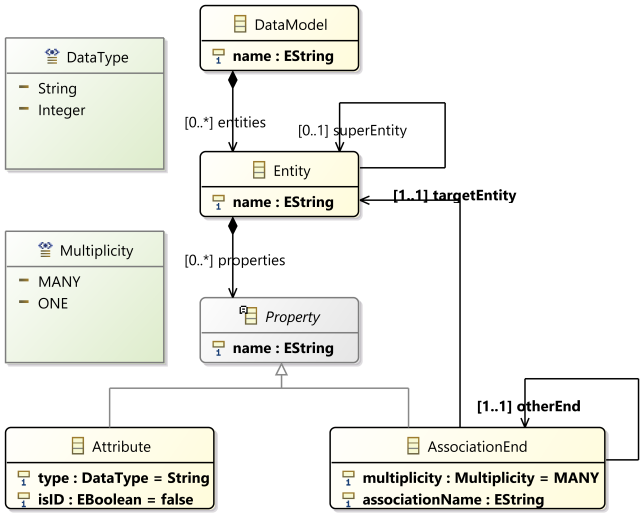

Fig. 2: SQLSI metamodel for data models.

- Each `Rule` represents a set of authorization rules in the policy.[5]
  - 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 cetain `Entity` from the `DataModel`, possibly the *user*-`Entity`.

3.2 Output Metamodels

Metamodel for Relational Schema The relational schema metamodel is shown in Figure 4. It is self-explanatory, therefore its descrption shall be omitted for the sake of space limit. [6]

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.

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

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

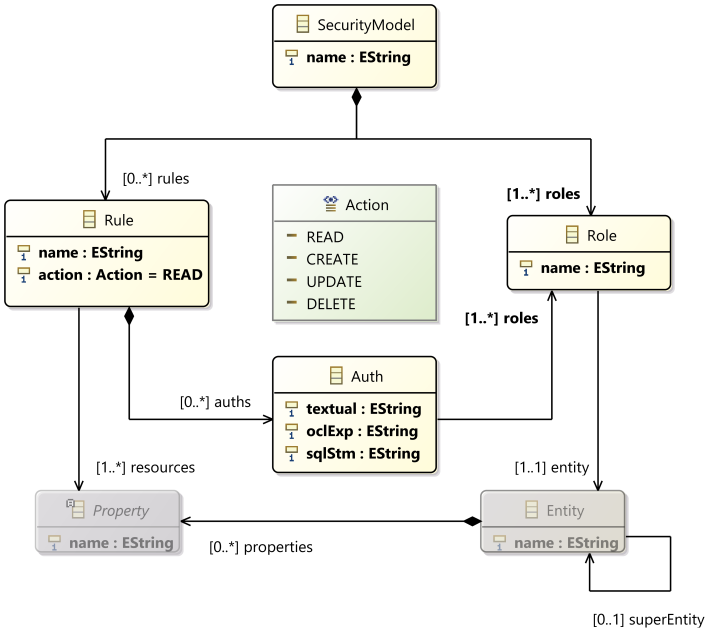

Fig. 3: SQLSI metamodel for security models.

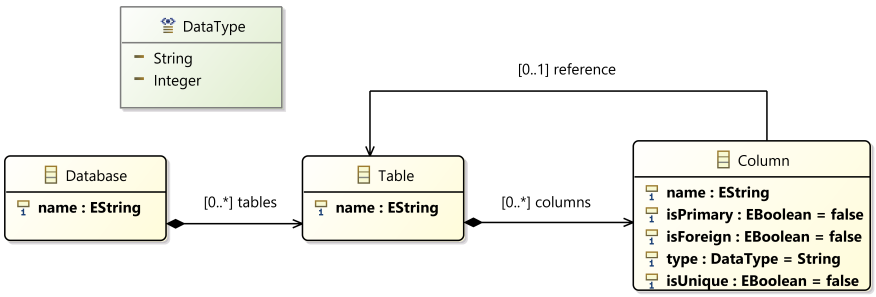

Fig. 4: SQLSI metamodel for relational database.

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 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 `Student`s. Additionally, every `Lecturer` can teach none or many `Course`s as well as every `Student` can enroll in none or many `Course`s.

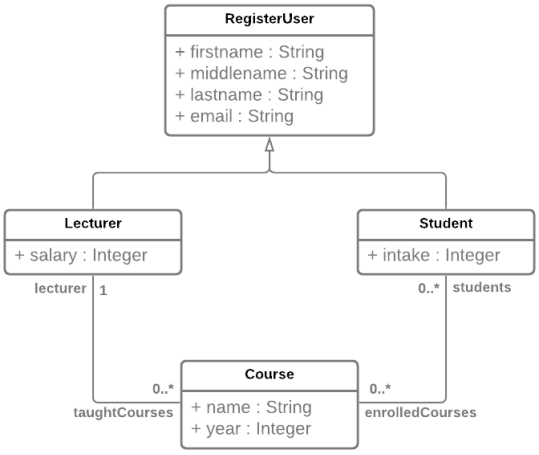

Fig. 5: SQLSI metamodel for security models.

4.2 Domain-Specific Language for the Input Models

DSL for data-model Figure 6 shows `UniversityDM` written in our domain specific language.[7] This language syntax is inspired by the ActionGUI[] datamodel language specification.

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`.

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

```
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

– 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 7 shows a fragment of the above security-model written in our domain specific language.[8] This language syntax is inspired by the SecureUML[] language specification, which is [9].

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

[9] TODO: captures the work of SecureUML

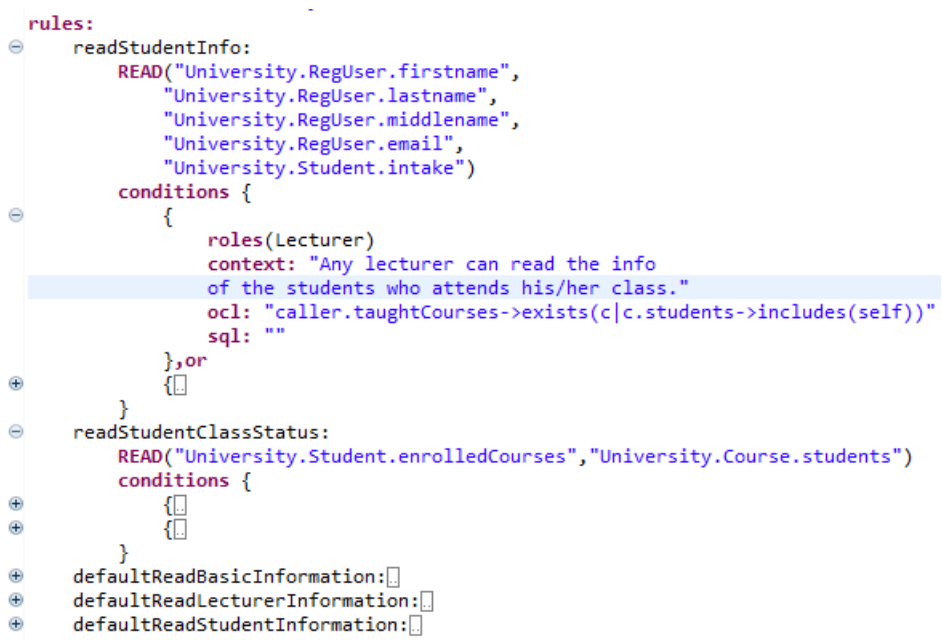

Fig. 7: A snipper of `UniversitySM`, written in our DSL

4.3 Transformation from Data Model to Relational Schema

Definition [10]

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 generation with Acceleo In addtion to the above transformation, we also provide a database schema generation. In this current version, we only support database schema generation that is executable on MySQL[11].

[10] TODO: Write something here

[11] 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.)

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.
- 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. 8: A MySQL database script of `UniversitySM`