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

Student: Hoàng Nguyễn Phước Bảo

Universidad Autónoma de Madrid, Spain

Abstract. To be decided.

1 Introduction

Model-Driven Engineering (MDE) turns the attention on models, instead of code.

Model-Driven Security (MDS) is a specialization on the domain of security.

One of my recent effort in MDS is to propose a model-driven approach in defining security policies for accessing data in database-centric application.

In this report, I am going to realize this proposal into a prototype by using the technology I learnt from the course.

Organization

2 Background and Motivation

Relational Databases and SQL

Access Control in Relational Databases

OCL and Authorization Constraints

Related work on realizing SQLSI

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 1. 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`.

[1] TODO: Check this

– 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`.

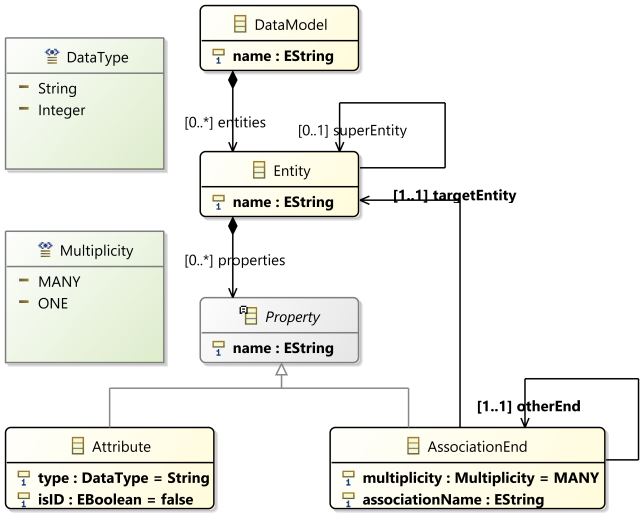

Fig. 1: SQLSI metamodel for data models.

Metamodel for data models [2] The securitymodel for SQLSI is shown in Figure 2. 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`.

– 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]
  • 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 `Role`s.

[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

– Each `Role` is associated with a cetain `Entity` from the `DataModel`, possibly the *user*-`Entity`.

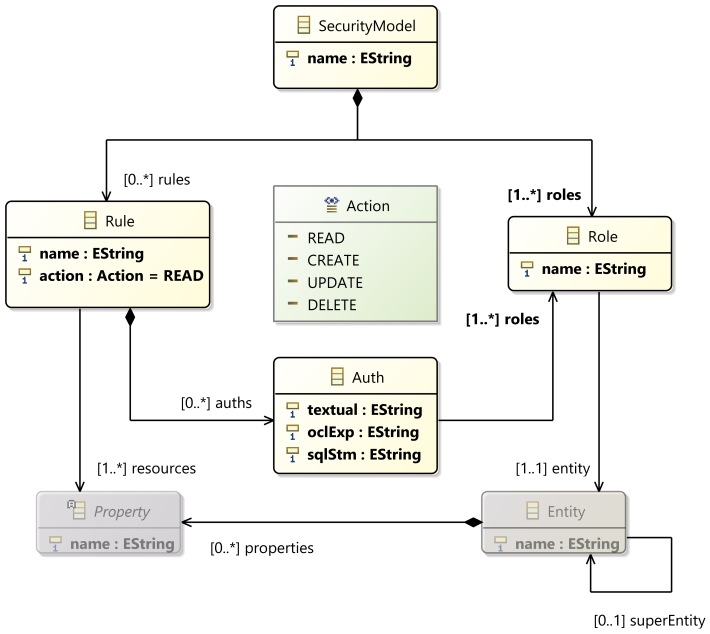

Fig. 2: SQLSI metamodel for security models.

3.2 Output Metamodels

Relational Schema Metamodel The relational schema metamodel is shown in Figure 3. It is self-explanatory, therefore its descrption 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] All metamodel invariants will be specified in Xtext, for the sake of consistency.

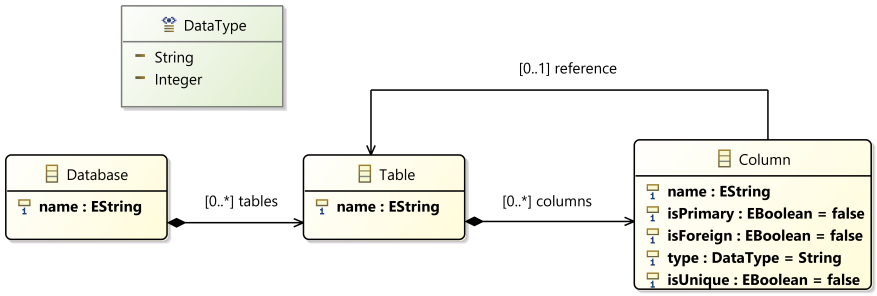

Fig. 3: SQLSI metamodel for relational database.

4.1 A Simple University Management System

Considering a simple UML class diagram `UniversityDM` in Figure 4 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 `Couse` 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.

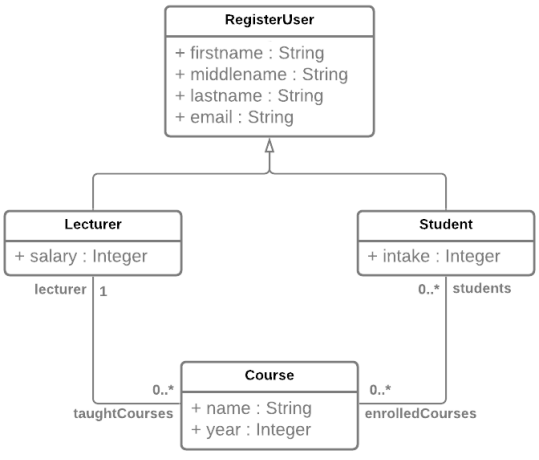

Fig. 4: SQLSI metamodel for security models.

4.2 Domain-Specific Language for the Input Models

A Domain-Specific Language for DataModel Figure 6 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).

```
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. 5: `UniversityDM` written in our DSL

A Domain-Specific Language for SecurityModel Considering the following security model for `UniversityDM`, denoted as $\mathcal{S}\rceil\lfloor$`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:
  - 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).

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

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.
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`.

[5] TODO: Write something here

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

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