A Pattern Based Approach for Secure Database Design

Jenny Abramov[2], Arnon Sturm[1], and Peretz Shoval[1]

[1] Department of Information Systems Engineering
[2] Deutsche Telekom Laboratories (T-Labs)
Ben-Gurion University of the Negev
Beer Sheva 84105, Israel
{jennyab,sturm,shoval}@bgu.ac.il

Abstract. Security in general and database protection from unauthorized access in particular, are crucial for organizations. Although it has long been accepted that system requirements should be considered from the early stages of the development, non-functional requirements, such security, tend to be neglected or dealt-with only at the end of the development process. Various methods have been proposed, however, none of them provide a complete framework to guide, enforce and verify the correct implementation of security policies within a system design, and generate source code from it. In this paper, we present a novel approach that guides database designers, to design a database schema that complies with the organizational security policies related to authorization. First, organizational policies are defined in the form of security patterns. Then, during the application development, the patterns guide the implementation of the security requirements and the correct application of the patterns is verified. Finally, the secure database schema is automatically generated.

Keywords: Secure software engineering, database design, authorization.

1 Introduction

Data is the most valuable asset for an organization as its survival depends on the correct management, security, and confidentiality of the data [5]. In order to protect the data, organizations must secure data processing, transmission, and storage. Nowadays, organizational systems are developed with minor emphasis on security aspects; system developers tend to neglect security requirements or deal with them only at the end of the development process.

Various efforts to promote the security aspect at the early stages of the software development life cycle have been made. These include the development of modeling methods such as UMLsec [11] and SecureUML [2,13] and methodologies for designing secure databases, such as [7]. Nevertheless, such methods primarily provide guidelines of how to incorporate security within a certain stage of the software development process, or address a specific aspect in the development of secure applications. In addition, these methods do not refer to the organizational security policies. We are not aware of existing methods that provide a complete framework to both guide and enforce particular organizational security policies on a system design, and finally transform the design to code. Since most organizational data are stored and managed using databases, this research focuses on protecting databases against unauthorized access and use.

In this paper, we propose a comprehensive approach that addresses the aforementioned gaps. The approach deals with both the organizational level and the application level. At the organizational level, organizational security policies are defined by security officers and domain experts in the form of security patterns. These patterns capture expert knowledge about a recurring problem and provide a well-proven solution to it [8]. As the patterns are independent of specific application, they can be used in the development of many applications. At the application level, the designers create a conceptual data model. This model is refined to include the security requirements, and then it is verified compared to the organizational security policies (the patterns). Finally, the secure database schema is automatically generated.

The rest of this paper is structured as follows: section 2 provides a review of related studies. Section 3 elaborates on the approach, and section 4 concludes and set plans for future research.

2 Related Work

Since it has been recognized that security must be treated throughout all stages of the software development life cycle, it is the task of the designer to ensure that all required security requirements are included in the specifications, and that adequate mechanisms are implemented to address those specifications. Over the years, many techniques and methods have been suggested in order to incorporate security within the development process. In the following we review major approaches for secure software development, with an emphasis on Model Driven Development (MDD) [22].

Several specification techniques for representing different security policies in a MDD process have been proposed. For example, UMLsec [11] is a UML extension that enables specifying security concerns in the functional model. It uses the standard UML extension mechanisms, stereotypes with tagged values to formulate the security requirements, and constraints to check whether the security requirements hold in the presence of particular types of attacks. In the context of access control, UMLsec provides a notation to represent only Role-Based Access Control (RBAC) policies; nonetheless it does not support specification of fine grained constraints (i.e., specification at the property or the instance levels). Furthermore, UMLsec does not provide generation or transformation from the application model to code.

Another example is the SecureUML [2, 13], which is a modeling language based on RBAC that is used to formalize role based access control requirements and to integrate them into application models. It is basically a RBAC language (where the concrete syntax is based on UML) with authorization constraints that are expressed in OCL [16]. SecureUML supports automatic generation of access control specification for particular technologies. Basin et al. [1] extended SecureUML and showed how OCL expressions can be used to formalize and check security properties of application models. Although it provides rich facilities for specifying role based authorization, SecureUML does not support enforcements of security constraints, and it is suitable only for role based authorization models.

Koch and Parisi-Presicce [12] present another approach to integrate specifications of access control policies into UML. The models are specified in three levels: the access control metamodel, the application information model and the instance of the access control model in a specific application. In addition, they present a verification mechanism of the model with respect to consistency properties. However, this method does not support the transformation of the models into database schemata.

Secure Tropos [14] is a security extension to the Goal-Driven Requirements Engineering methodology, which enables to model security concerns of agent-based systems. The most notable limitation of Secure Tropos is that it mainly focuses on the requirements analysis phase, and therefore does not provide adequate support to the design of security policies. To overcome this, Mouratidis and Jurjens [15] combined Secure Tropos and UMLsec to create a structured methodology for software development, which supports most of the software development phases. In the context of our work, the limitations of the combined approach derived from those of UMLsec.

Another approach for security specification is security patterns, which are based on the classic idea of design patterns introduced by Gamma et al. [8]. Security patterns were proposed to assist developers to handle security concerns and provide guidelines to be used from the early stages of the development lifecycle [21]. To successfully utilize a security pattern, there must be systematic guidelines supporting its application throughout the entire software development lifecycle. Such a methodology to build secure systems using patterns is presented by Fernandez et al. [6]. The methodology integrates security patterns into each one of the software development stages, where each stage can be tested for compliance with the principles presented by the patterns. A catalog of security patterns can help in defining the security mechanisms at each architectural level and at each development stage. This method mainly provides guidelines about which security related procedural patterns could be useful in the various development stages, but no verification algorithm is proposed in order to verify the implementation of the design patterns in the application, and no concrete way to transform the application into code is provided.

Other methodologies present the use of the aspect-oriented software design to model security as separate aspects which would later become weaved into the functional model. For example, Ray et al. [18] propose to deal with access control requirements while utilizing UML diagrams. Access control patterns are modeled as aspects in template forms of UML diagrams, while other functional design concerns are addressed in the primary model. Then, the aspect models are woven into the primary functional model, creating an application model that addresses access control concerns. Yet, this method does not address verification of the application, nor transformations in any way. Another example is the framework proposed by Pavlich-Mariscal et al. [17], which enable to model access control policies with the use of built-in UML extensions and transform them into code. Additionally, a set of composable access control features, i.e., components that realize specific capabilities of RBAC, Mandatory Access Control (MAC), and Discretionary Access Control (DAC), are proposed in a set of access control diagrams. Although the presented extensions allow specifying a fairly broad range of requirements, there are still access control capabilities that are not included in the framework, such as the ability to specify time constraints indicating when subjects can access to the protected resources.

Several studies deal with secure database design, such Fernández-Medina and Piattini [7], who propose a methodology to design multilevel databases based on MAC policies. The methodology allows creating conceptual and logical models of multilevel databases, and implements the models by using Oracle Label Security [4]. The resultant database imposes that access of a user to a particular row is allowed only if (1) that user is authorized to do so by the DBMS; (2) that user has the necessary privileges; and (3) the label of the user dominates the label of the row. Following that methodology, the authors provide a way for transforming specification artifacts into code. However, they do not provide tools for verifying the application against security policies.

Information security is crucial to many organizations, and it is necessary to assure that the security policies of a database are not neglected during the development process. However, existing methods do not provide means to enforce particular organizational security policies on a database design. The aforementioned approaches mainly provide guidelines regarding how security can be handled within certain stages of the software development process, or address specific aspects in developing secure applications. Although some of the methods provide means for checking models, they do not support the ability to verify the correct application of the organizationl security policies. Our study specifically addresses these gaps by introducing a comprehensive approach for developing secure database schemata, and enables the organization to enforce its security policies within the developed systems.

3 The Pattern-Based Approach

The proposed approach aims at guiding, enforcing and verifying the correct usage of security patterns and utilizing the knowledge encapsulated in these patterns for generating secure database schemata. It is a part of a comprehensive methodology that supports the core phases of the development lifecycle of secured (authorization wise) database schemata. The approach was developed to enable its integration into an existing development process. Thus, the techniques and tools that are used are kept as similar as possible to the standard ones. For example, we adopt the modeling language UML along with OCL.

We use the Application-based Domain Modeling (ADOM) [19] framework as an infrastructure to the proposed approach. ADOM is rooted in the domain engineering discipline, which is concerned with building reusable assets on the one hand, and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and verification of the enterprise-specific models against the relevant reference models. Before application developed can be started, the security patterns to be enforced during the database development are defined within ADOM's domain layer (i.e., organizational level), along with their transformation rules which depict on how to transform the logical model (that is based on the pattern) into a database schema. These security patterns meant to reflect general access control policies within the organization. They are reusable and may be applied to numerous applications. During the development process of an application, the patterns are employed in the application layer (of ADOM). The designer is guided and compelled to apply the patterns that are defined in the organizational layer. After the application is verified against the patterns, the transformation rules are used to generate a secure database schema. As this work refers to database specifications, i.e., structural models, class diagrams are used for in both the organizational and the application layers.

3.1 Pattern Specification

The patterns are specified according to the ADOM approach within the domain layer. These patterns serve as a guideline for application developers as well as a verification template, and they provide infrastructure for the transformation process. Similarly to the classical pattern approach, security patterns are specified in a structured form. The standard template aids designers who are not security experts to identify and understand security problems and solve them efficiently. In order to specify the patterns, we use a common template that was introduced by Schumacher [20]. The template consists of five main sections: name, context, problem, solution, and consequence (due to space limitations, we discuss only the solution section as other sections are similar to those found in the literature). The solution section describes a generic solution to the problem. In the proposed approach, it consists of a UML class diagram that specifies the static structure of the solution and OCL constraints that are used to specify additional requirements that ought to be verified in the application model. In case that finer grained solution is required than the one provided by the diagrams, OCL rules in the form of general templates can be defined. These general OCL templates are specified using the specific elements and language that were already defined by the class diagrams specifying the structure of the pattern. The designer should use these templates to specify fine grained access control policies.

As RBAC patterns are commonly used for database authorization, in the following example we show a simple RBAC pattern that is adjusted to some specific organizational policies. Other access control patterns can be specified in a similar way. Figure 1 presents our RBAC pattern based on the proposed approach.

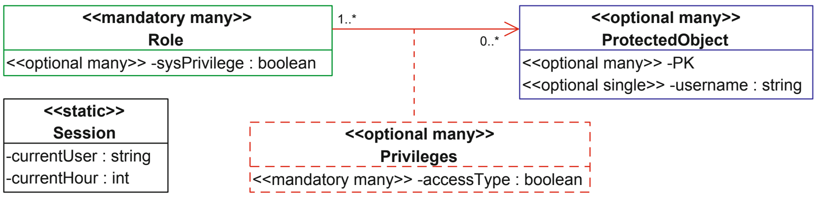

Fig. 1. The RBAC security pattern residing within the domain layer

In the described pattern, *Role* is akin to an external group of entities or users playing a specific role that needs to access the database. While applying or implementing this RBAC pattern, it is obligatory to define at least one *Role* as it is defined as a <<*mandatory*>> element. In addition, one can specify the system privileges assigned to some *Role* by using the *sysPrivilege* classification. *ProtectedObject* is akin to a database table, where the *PK* classification is used to indicate the primary keys of the table. Additionally, when a *ProtectedObject* is also classified as a *Role*, one can specify the user related to the object (or row in a table) by using the *username* classification. *Privileges* association class determines the schema object privileges of a *Role* with respect to a specific *ProtectedObject*. A class that is classified as *Privileges* must include at least one object privilege - *accessType*.

In addition to the pattern structure, OCL rules that further constraint and clarify the pattern can be specified. These OCL rules are evaluated in the application layer during the verification stage, rather than in the domain layer where they are defined. For example, the following constraint limits object privileges to `SELECT`, `INSERT`, `UPDATE` and `DELETE`:

```
context Privileges inv:
 Set{'SELECT','INSERT','UPDATE','DELETE'}->
  includesAll(self.accessType->collect(name))
```

Another example is the constraint that limits system privileges to `CREATE SESSION`, `CREATE VIEW` and `SYSDBA`:

```
context Role inv:
 Set{'CREATE SESSION', 'CREATE VIEW', 'SYSDBA'}->
  includesAll(self.sysPrivilege -> collect(name))
```

Additionally, the following constraint ensures that roles that have the system privilege `SYSDBA` do not have access to any object:

```
context Role inv:
 if self.protectedObject->size() > 0 then
   self.sysPrivilege->collect(name)->excludes('SYSDBA')
 endif
```

In general RBAC models define privileges for objects as one unit; however sometimes finer-grained access control is required. Access control on the objects' properties and instances (i.e., columns and rows in database tables, respectively) enable preservation of security principles such as the minimal privilege principle. In order to support fine-grained access control specifications, OCL templates are defined in the domain model so that the designer could use them to create property and instance access control rules in the application model. These templates are defined using the various elements that specify the structure of the pattern, such as the *Role*, and the *accessType* in our example. The OCL templates are used for two purposes: guiding the developer on fine-grained constraints; and defining transformation rules.

Text templates [24] are essentially exemplars of the desired output code with "blanks" that should be filled in with a value of an attribute or with an output code from a nested template. These "blanks" contain meta-code and are delimited between "< >". For example, the following OCL expression defines a simple OCL template:

```
context <class name> inv <constraint name>:
  <simple OCL expression>
```

For every OCL template, we define a corresponding SQL template, with the same set of "blank" elements. The corresponding template in SQL is defined as:

```
ALTER TABLE <class name> ADD CONSTRAINT <constraint name>
  CHECK (<simple OCL expression>)
```

After the missing values are inserted, a template engine is used to create the output code. The processing of the inserted elements can be a simple replacement of a variable with its value, or a more complex transformation. Following the template example above, let's assume we would like to constraint an *Enrollment* class so that the property *grade* is limited to values between 0 and 100. The missing values for `<class name>`, `<constraint name>`, and `<simple OCL expression>` are `Enrollment`, `grade_in_range`, and `grade>=0 and grade<=100`, respectively. After processing, the OCL output code is as follows:

```
context Enrollment inv grade_in_range:
  grade>=0 and grade<=100
```

The corresponding SQL output code is as follows:

```
ALTER TABLE Enrollment ADD CONSTRAINT grade_in_range
  CHECK (grade>=0 and grade<=100)
```

As in the target language (SQL and Oracle VPD) the approach for handling object privileges is to first grant the privilege to the table and then to limit the access by some condition; in this RBAC pattern we apply the same approach. First, the privileges are given to the role (in the model) and then the access can be limited to some of the instances or properties by OCL constraints.

In order to incorporate **property (column or cell) level** access control constraints, an extension of OCL is required. The Boolean operation `isAuthorized` is used to restrict a role from performing an action on an element, depending on the result of the condition: `role.isAuthorized(action, element, guard): Boolean`, where `role` is the activating role, `action` is the operation (e.g., select, insert, update, or delete) to be performed on the `element`, `element` is an object or a property of an object that needs to be protected, and `guard` is the condition that has to be fulfilled in order to allow the role to perform the action on the `element`. For each property within a class that requires different restrictions on its access than its class, the following **property (column) level** OCL template can be used:

```
context <Privileges> inv:
  self.<Role>.isAuthorized(self.<accessType>,
    self.<ProtectedObject>.<property>,
    <guard>)
```

This OCL template means that in the context of a class that is classified as <<*Privileges*>>, a privilege of type <<*accessType*>> of a <<*Role*>> to a *property* of the <<*ProtectedObject*>> is limited by the *guard*.

To provide support for **instance (row) level** access control, the following OCL template could be applied on application models:

```
context <Privileges> inv:
  [self.<constrained-type :accessType> implies]
  Session.currentUser = <ProtectedObject>.<username>
  and <ProtectedObject>.[<navigate-between-the-
  ProtectedObject-and-the-Role>].<Role>->includes(<Role>)
```

This OCL means that in the context of some <<*Privileges*>>, the user of the application (indicated by *Session.currentUser*) may have the access only to objects related to this user. In addition, the user has to have a <<*Role*>> that is part of the collection related to the <<*ProtectedObject*>>. This template is meaningful only if the <<*ProtectedObject*>> is additionally classifies as a <<*Role*>>. Note that this template enables the designer to specify constraints on a specific <<*accessType*>> (line 2) or all the <<*accessTypes*>> within a <<*Privileges*>>.

Additional templates can be specified to include other requirements, such as policies that restrict access based on time.

3.2 Transformation Rules Specification

Model transformation plays an important role in MDD approaches. As models become first class artifacts that drive the whole development process, software engineers should be supported during the development process by mature MDD tools and techniques in the same way that IDEs, compilers, and debuggers support them in traditional programming paradigms. Thus, in order to complete the process, the transformation rules of a pattern need to define how application elements should be transformed into a database schema. To transform the UML class diagram into SQL code, a “model to code” transformation tools have to be provided. This type of transformation can use technologies such as the ATLAS Transformation Language (ATL) [10] to first transform the application model into an SQL application model, and then to translate it to SQL code. The following example specifies the rule to transform application object privileges - *accessType*:

```
rule Permission {
  from element :
    ADOM!"Model::RBAC::Role::Privilege::accessType"
  to permission : SQL!Permission (
    roles <- element.getParent().getSource(),
    object <- element.getParent().getTarget(),
    operation <- element.getName()
 )}
```

This rule iterates over all elements stereotyped as *accessType* and assigns the specified permission (specified by the attribute `name`) to the connected roles (specified by the association class *Privilege* end - `source`) on the objects (specified by the association class *Privilege* end - `target`).

The transformation of OCL templates to SQL or PL/SQL code can combine text templates such as StringTemplate [24], with more SQL oriented transformation techniques such as the Dresden OCL Toolkit [9]. The following template example is for the specific case where the privileges refer to a class that is classified both as *Role* and as *ProtectedObject*; this template is a short version of the *instance level template*:

```
-- From: OCL
context <Privileges> inv:
  [self.<constrained-type:accessType> implies]
  Session.currentUser = <ProtectedObject>.<username>
```

```
-- To: Oracle VPD function and policy
CREATE FUNCTION <Privileges>[_<accessType>]
   (SCHEMAV VARCHAR2, OBJ VARCHAR2) RETURN VARCHAR2 AS
BEGIN
  IF (NOT DBMS_SESSION.IS_ROLE_ENABLED('<Role>')) THEN
    RETURN NULL;
  END IF;
  RETURN '<username> = ' ||
               SYS_CONTEXT('USERENV', 'SESSION_USER');
END;
BEGIN DBMS_RLS.add_policy(
  object_schema => '<application-class-diagram-name>',
  object_name => '<ProtectedObject>',
  policy_name => '<Privileges>[_<accessType>]',
  policy_function => '<Privileges>[_<accessType>]',
  statement_type => '<constrained-type: accessType |
                              {Privileges.accessType*}>'
  <if {INSERT, UPDATE, DELETE} •
        <constrained-type:accessType |
         {Privileges.accessType*}> != ∅>
   , update_check => TRUE
  <endif>
);
END;
```

This VPD function first checks if the activating *Role* is relevant to the constraint, if not it returns Null (i.e., no constraint) otherwise, it returns a filtering string with the current user's username. The VPD policy registered to the stated object, specifies when the function should be activated by stating the relevant *accessType*s. This template contains both simple values substitution and nested template (lines 22-26).

To summarize, the outcome of the two tasks at the organizational level are: a) the specification of security patterns that will guide developers and compel them to specify security requirements as determined by the security policies, and b) the definition of transformation rules from application models into database schemata. The security patterns are defined using UML and OCL, and the transformation rules are expressed in ATL and templates.

3.3 Application Development

Before elaborating on application development phases, we present a simple example that will be used to demonstrate the various tasks and the resulting artifacts. The example deals with a university application that manages lecturers, course, students, and their enrollments and grades. Each department offers some courses, and each course offering is given by a lecturer. Most of the information about departments, lecturers, courses and course offerings is public and therefore all users may view their details. However, the information about students enrolled to certain course offerings may be visible only to the lecturers of those courses. A lecturer's phone number may be visible only to students who are enrolled to one of the lecturer's courses; other lecturers and secretaries may view this data unconditionally. Secretaries may add new courses to the system and update course's level or credits. Secretaries may also add or delete course offerings and assign lecturers to them; however the secretaries may not read or modify the students enrolled to course offerings. Lecturers may update all of the information related to their assigned course offerings, except the students enrolled. In addition, lecturers may see the students enrolled in each of their course offerings (i.e., access their IDs and names), but they may not see to which other courses their students are enrolled. Lecturers are only allowed to give (add) grades to their students. Lecturers may update their personal information, excluding the department they belong to and their assignments to course offerings. Students may read and modify their own personal information and enrolled courses, and they may see their grades. Secretaries may add or delete students and lecturers from the system.

During the design stage at the application level, the designer formalizes the data model and the privilege specifications into a unified class diagram. Meaning that the authorization rules are weaved into the class diagram as specified by the patterns; the relevant elements are classified, via stereotypes, according to the predefined patterns. Figure 2 presents a class diagram of the university system along with the security specification as determined by the RBAC pattern, specified at the organizational level (see section 3.1). The various elements that are relevant for the RBAC pattern are classified (by stereotypes) with the pattern elements: *Role*, *ProtectedObject*, and *Privileges*. Note that *accessType* and *sysPrivilege* elements are presented only if their assigned value is true. For example, the *Student* role has a system privilege of {`CREATE SESSION`} and the following schema object privileges: {`SELECT`} to *Department*, *Lecturer*, *Course* and *CourseOffering*, {`SELECT`, `INSERT`, and `DELETE`} to *Enrollment*, and {`SELECT` and `UPDATE`} to *Student*.

As explained before, the expressiveness of class diagrams is limited in representing fine grained access control privileges. For example, in the case of the university system as presented in figure 2, a student can enroll other students to courses and update their personal information. Thus, working under the RBAC pattern defined in section 3.1, the designer should constrain this by using the predefined OCL templates. The following OCL example illustrates the use of the *instance* level template that limits students to read, insert, or delete an enrollment only for them:

```
context Student-Enrollment inv:
 Session.currentUser = self.student.username
 and self.enrollment.student -> includes(self.student)
```

Another example is of a *property* level constraint, in which students may see a lecturer's phone number only if they are enrolled to one of the lecturers' courses:

```
context Student-Lecturer inv:
 self.student.isAuthorized(self.SELECT,
   self.lecturer.phone,
   Session.currentUser = self.student.usermane
   and self.lecturer.courseOffering.enrolled ->
         includes(self.student))
```

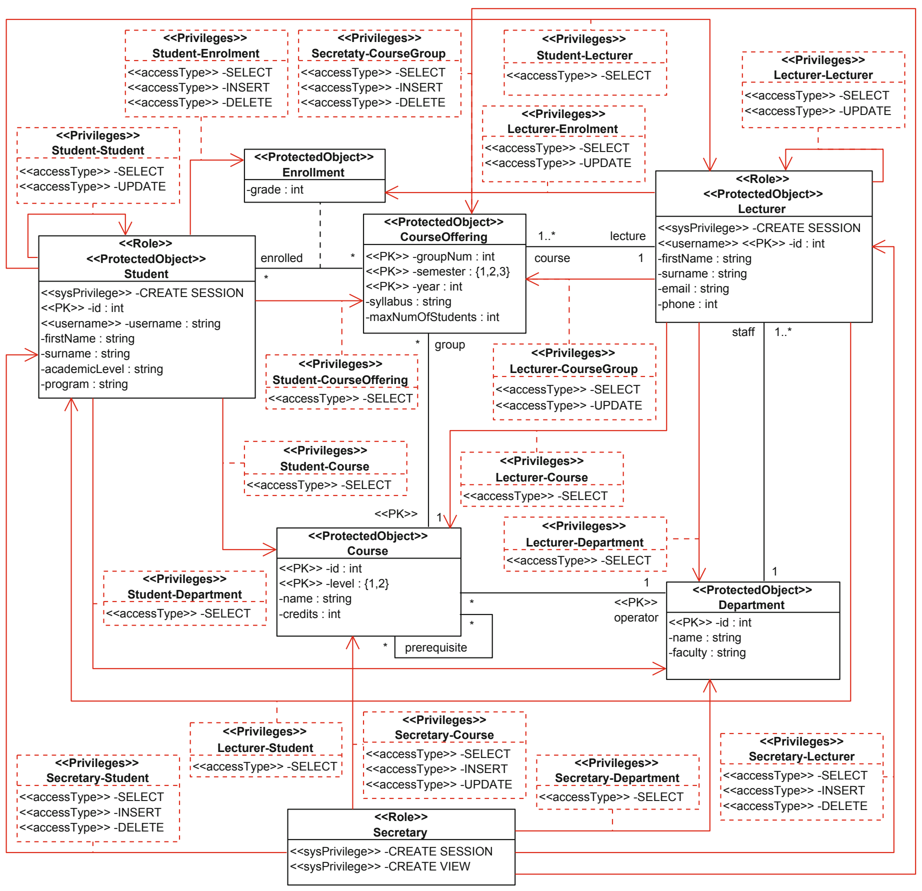

Fig. 2. The university system application model

3.4 Data Model Verification

Having created a refined data model, we need to check if it adheres to the organizational security policies as defined by the patterns. In the following we define a pattern-based verification which is based on the ADOM validation algorithm [19] for checking the adherence of application models with the domain model. The application is verified in several facets: (a) multiplicity, (b) OCL, (c) language, and (d) cardinality.

Multiplicity verification checks that the application adheres with the patterns (which reside within the domain layer) with respect to elements multiplicity. It is performed in three phases as defined in ADOM's validation algorithm: element reduction, element unification, and model matching. In the *element reduction step*, elements that are not stereotyped by elements of the security pattern are disregarded. During the *element unification step*, elements having the same domain stereotype are unified, leaving only one element instantiating the same type in the resultant model. The multiplicity of that element denotes the number of distinct elements in the application model having the same stereotype. In the *model matching step*, the resultant model of the previous step is matched against the pattern (the domain model) in order to verify the multiplicity of the elements, and the application model structure with respect to the pattern (the domain model). In the university example there are no unclassified classes, thus the reduction step of the multiplicity verification is redundant. The resultant model after performing the unification step consists of three classes: *Role* with multiplicity 3, *sysPrivilege* with multiplicity <<1..2>, *ProtectedObject* with multiplicity 6, *PK* with multiplicity <<0..3>>, *username* with multiplicity <<0..2>>, *Privileges* with multiplicity <<2..5>>, and *accessType* with multiplicity <<1..3>>. Thus, the matching step finds no violation. An example to a possible violation would have been in case that there were association classes that are classified as *Privileges* with no relevant *accessType*s specified.

OCL verification checks that the OCL rules defined in the pattern hold in the application. This verification iterates over all pattern elements. For each such element, it extracts its OCL rules, and retrieves its logical instantiations from the application. For each logical instantiation, the verification checks that the extracted OCL rules hold. In the university example, all of the OCL constraints defined in the pattern are evaluated to true. An example for a violation would have been if one or more of the roles would have some object privilege and the `SYSDBA` system privilege, or one of the *Role*s would have object privileges different from the ones defined, i.e., `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

Language verification confirms that logical instantiations in the application have the same types and modifiers as their classifying element in the pattern. Specifically, all attributes in the application model that instantiate domain attributes must have the same type, or a subtype thereof. For instance, in the RBAC pattern defined in figure 1, the *username* attribute in the *Role* class is defined as string, that means that all attributes that are classified as *username* in the application have to be of type string. However, for attributes such as the *PK* in the *ProtectedObject* class, that were defined as object (or do not define the type), classified attributes can be of any type.

Cardinality verification confirms that the cardinality defined on association ends in the pattern hold. The number of logical instantiation of associations must be within the cardinality bounds defined on the classifying association in the pattern (the domain). For instance, the cardinality defined on the *Privilege* association class states that at least one *Role* must be connected to every *ProtectedObject*, yet not vice versa.

3.5 Data Model Transformation

In this stage the developer has a verified data model enhanced with security constraints that need to be transformed into a database schema. The transformation of the class diagram to the relational tables can be done according to existing algorithms, such as Blaha et al. [3] and Shoval [23]. This and pattern related transformations are done by following the transformation rules and templates which were defined at the organizational level, and the application specification created in the design phase. The generated SQL commands for the *Student* role and a sample of the SQL fine-grained code for the university application are as follows:

```
-- Role creation
CREATE ROLE STUDENT;
-- Granting privileges to Student
GRANT CREATE SESSION TO STUDENT;
GRANT SELECT ON DEPARTMENT TO STUDENT;
GRANT SELECT ON LECTURER TO STUDENT;
GRANT SELECT ON COURSE_OFFERING TO STUDENT;
GRANT SELECT ON COURSE TO STUDENT;
GRANT SELECT ON PREREQUISITE TO STUDENT;
GRANT SELECT, INSERT, DELETE ON ENROLLMENT TO STUDENT;
GRANT SELECT, UPDATE ON STUDENT TO STUDENT;
-- Instance level template transformation
-- Students can update only their personal information:
CREATE FUNCTION STUDENT_STUDENT_UPDATE
   (SCHEMAV VARCHAR2, OBJ VARCHAR2) RETURN VARCHAR2 AS
BEGIN
  IF (NOT DBMS_SESSION.IS_ROLE_ENABLED('STUDENT')) THEN
    RETURN NULL;
  END IF;
  RETURN 'username = ' ||
           SYS_CONTEXT('USERENV', 'SESSION_USER');
END;
BEGIN DBMS_RLS.add_policy(
  object_schema   => 'UNIVERSITY',
  object_name     => 'STUDENT',
  policy_name     => 'STUDENT_STUDENT_UPDATE ',
  policy_function => 'STUDENT_STUDENT_UPDATE ',
  statement_types => 'UPDATE',
  update_check    => TRUE);
END;
```

4 Summary

In this paper we have presented a novel approach that utilizes organizational security patterns for enforcing security in database application design. The approach guides developers on how to incorporate security aspects, in particular authorization, within the development process. It handles the specification and implementation of the authorization aspect from the early stages of the development process to implementation, leading to a more secure system design. The proposed method addresses the limitations of existing studies by providing a coherent framework that facilitates the enforcement of security organizational patterns on applications development.

So far we have used the approach in a few small cases. Yet, it should be further tested and evaluated in terms of applicability and usage. We already have conducted an experiment to verify that the use of the proposed approach indeed ease the developer efforts and increase specifications correctness. In that experiment we examine whether the abstraction achieved by the patterns proposed within the approach helped undergraduate students to achieve better security specification than directly coding in SQL. In that experiment, the students received a requirement document that explicitly lists all of the security requirements. They were required to add the security constraints over an existing database schema using the patterns

(Group A) and using SQL (Group B). The experiment results showed that the group that used the patterns achieved statistically significant better results than the other in terms of security specification related to table/object, column/property, and row/instance levels. Yet, the results still requires further inspection and analysis.

Despite providing many advantages, the Pattern Based Approach may suffer from several drawbacks. Pattern specification is not a straightforward task. In order to provide the necessary support to database designers, the security expert specifying the pattern has to have a lot of knowledge regarding general access control mechanisms, and different technologies that can be utilized for the generation of the secure database schema. The security expert must also be able to express organizational policies in the general access control patterns. In addition, at the application level, the designer may have difficulties related to exploiting the OCL templates, or in specifying complex requirements in an object oriented model for relational database.

Future research may include the evaluation of the proposed approach using real-world case studies and actual development and manipulation of databases, so as to enable comparisons of the alternative approaches in close to real conditions. Additionally, as specifying a pattern and its transformation rules is not an easy task; in order to fully understand this process, further investigation is required. In addition, the Pattern Based Approach presented in this study used access control patterns; however, the approach can be extended to deal with broader areas of security concerns in databases and secure application design, using other types of security design patterns. This is important, as working with a variety of approaches and standards on different security aspects is not sufficient and requires a lot of efforts in combining the artifacts. Therefore, a common infrastructure that will allow the smooth integration of the various security aspects is a necessity.

Acknowledgment. This work was partially funded by Deutsche Telekom Laboratories at Ben-Gurion University of the Negev.

References

1. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated Analysis of Security-Design Models. Information and Software Technology 51(5), 815–831 (2009)
2. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology 15(1), 39–91 (2006)
3. Blaha, M., Premerlani, W., Shen, H.: Converting OO Models into RDBMS Schema. IEEE Software 11, 28–39 (1994)
4. Czuprynski, J.: Oracle Label Security. The Database Journal (2003), `http://www.databasejournal.com`
5. Dhillon, G.: Information Security Management: Global Challenges in the New Millennium. IGI Publishing, Hershey (2001)
6. Fernandez, E.B., Larrondo-Petrie, M.M., Sorgente, T., VanHilst, M.: A Methodology to Develop Secure Systems Using Patterns. In: Integrating Security and Software Engineering: Advances and Future Vision, ch. 5, pp. 107–126. IDEA Press (2006)
7. Fernández-Medina, E., Piattini, M.: Designing Secure Databases. Information and Software Technology 47(7), 463–477 (2005)

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1994)
9. Heidenreich, F., Wende, C., Demuth, B.: A Framework for Generating Query Language Code from OCL Invariants. In: Akehurst, D.H., Gogolla, M., Zschaler, S. (eds.) Proceedings of the 7th OCL Workshop at the UML/MoDELS Conference, Ocl4All - Modelling Systems with OCL. ECEASST, vol. 9 (2008)
10. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming 72(1-2), 31–39 (2008)
11. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2004)
12. Koch, M., Parisi-Presicce, F.: UML Specification of Access Control Policies and their Formal Verification. Software and System Modeling 5(4), 429–447 (2006)
13. Lodderstedt, T., Basin, D.A., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
14. Mouratidis, H., Giorgini, P.: Secure Tropos: A Security-Oriented Extension of the Tropos Methodology. International Journal of Software Engineering and Knowledge Engineering 27(2), 285–309 (2007)
15. Mouratidis, H., Jurjens, J.: From Goal Driven Security Requirements Engineering to Secure Design. International Journal of Intelligent Systems 25(8), 813–840 (2010)
16. Object Constraint Language. OMG Specification, Version 2.2, `http://www.omg.org/spec/OCL/2.2/`
17. Pavlich-Mariscal, J.A., Demurjian, S.A., Michel, L.D.: A Framework of Composable Access Control Features: Preserving Separation of Access Control Concerns from Models to Code. Computers & Security 29(3), 350–379 (2010)
18. Ray, I., France, R.B., Li, N., Georg, G.: An Aspect-Based Approach to Modeling Access Control Concerns. Information and Software Technology 46(9), 575–587 (2004)
19. Reinhartz-Berger, I., Sturm, A.: Utilizing Domain Models for Application Design and Validation. Information & Software Technology 51(8), 1275–1289 (2009)
20. Schumacher, M.: Security Engineering with Patterns: Origins, Theoretical Models, and New Applications. Springer-Verlag New York, Inc., Secaucus (2003)
21. Schumacher, M., Fernandez, E.B., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, Chichester (2006)
22. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (2003)
23. Shoval, P.: Functional and Object-Oriented Analysis and Design - An Integrated Methodology. IGI Publishing, Hershey (2007)
24. String Template (2010), `http://www.stringtemplate.org`