Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A methodology for integrating access control policies within database development

Jenny Abramov [a,b], Omer Anson [b], Michal Dahan [a], Peretz Shoval [a], Arnon Sturm [a,*]

[a] Department of Information Systems Engineering, Ben-Gurion University, Markus Campus, Beer-Sheva 84105, Israel
[b] Deutsche Telekom Laboratories (T-Labs), Ben-Gurion University, Markus Campus, Beer-Sheva 84105, Israel

ARTICLE INFO

Article history:
Received 30 January 2011
Received in revised form
8 January 2012
Accepted 17 January 2012

Keywords:
Authorization
Access control
Database design
Security patterns
Domain analysis
FOOM
ADOM
UML
Security

ABSTRACT

Security in general and database protection from unauthorized access in particular, are crucial for organizations. While functional requirements are defined in the early stages of the development process, non-functional requirements such as security tend to be neglected or dealt with only at the end of the development process. Various efforts have been made to address this problem; however, none of them provide a complete framework to guide, enforce and verify the correct design of security policies, and eventually generate code from that design.

We present a novel methodology that assists developers, in particular database designers, to design secure databases that comply with the organizational security policies that are related to access control. The methodology is applied in two main levels: organizational level and application development level. At the organizational level, which takes place before the development of a specific application, organizational policies are defined in the form of security patterns. These patterns encapsulate accumulated knowledge and best practices on security related problems. At the application development level, the data-related security requirements are defined as part of the data model. The security patterns, which have been defined at the organizational level, guide the definition and implementation of the security requirements. The correct implementation of the security patterns is verified during the design stage of the development process, before the automatic generation of the database code. The methodology is supported by a CASE tool that assists its implementation in the various stages.

© 2012 Elsevier Ltd. All rights reserved.

1. Introduction

Data is a most valuable asset for an organization, as its survival depends on the correct management, security, and confidentiality of the data (Dhillon, 2001). In order to protect the data, organizations must secure data processing, transmission and storage. In spite of that, even nowadays, organizational systems are being developed with minor treatment of security aspects; system developers tend to neglect dealing with security requirements, or deal with them only at the end of the development process.

It has been recognized that security must be treated from the early stages of the software development lifecycle, and various efforts have been made to address this need. Examples for studies that deal with incorporation of security aspects within the development process vary from UML extensions such as UMLsec (Jürjens, 2005) and SecureUML (Lodderstedt et al., 2002; Basin et al., 2006), to methods for designing

* Corresponding author.
E-mail addresses: jennyab@bgu.ac.il (J. Abramov), oaanson@gmail.com (O. Anson), dahanmic@bgu.ac.il (M. Dahan), shoval@bgu.ac.il (P. Shoval), sturm@bgu.ac.il (A. Sturm).
0167-4048/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2012.01.004

secure databases such as Fernández-Medina and Piattini (2005). Yet, such studies mainly provide guidelines about the way security should be handled within certain stages of the software development process, or address specific aspects of security. To the best of our knowledge, no existing methodology provides a complete framework that both guides and enforces organizational security policies on a system design, and then generates executable code from that design.

We propose a comprehensive methodology that provides such a framework. The proposed methodology deals with the organizational level and with the application development level. At the organizational level, the organizational security policies are defined by security officers and domain experts, in the form of security patterns. At the application development level, the system analysts/designers create a conceptual data model and a functional model of the application, which include security requirements. These models are further refined and adjusted following the security patterns, and then are verified compared to the organizational level security policies. Finally, the code of a secure database schema is generated. The whole process, at both levels, is supported by a software tool that has been developed as part of this study.

Although many different security aspects should be dealt with when developing systems, in this study we deal mainly with the database aspect, while in follow-up work we also deal with other aspects (e.g., system behavior/functionality, and users' interfaces). In this study we focus on secure database development for the following reasons: 1) Data are the most important resource, that have to be protected from undesired access and manipulations; 2) Usually, when designing the security within the application layer, the connection to the database is done using a single user, thus eliminate the option of auditing and identification of malicious activities within the database; and 3) Databases can be accessed directly and through various applications, thus it is important to secure the database itself. Nevertheless, there are other important security aspects that should be addressed; the presented methodology can be extended to deal with such aspects as well.

The contribution of this work is the following: We provide a mechanism to precisely specify security patterns using well-known and widely-used modeling techniques, and add transformation rules to the security patterns. The precise definition of the patterns enables the enforcement of their content over application designs, whereas the transformation rules allow the automatic generation of the desired artifacts based on the specified patterns. Furthermore, we deal with conformance checking of application designs vis-à-vis the organizational security patterns.

The rest of this paper is structured as follows: Section 2 reviews related studies; Section 3 presents the background related to the proposed methodology; Section 4 describes the methodology; and Section 5 concludes and set plans for future research.

2. Related studies

Over the years, many methods and techniques have been proposed to incorporate security aspects within the development process of information systems. The studies in this survey are organized in two categories: a) general security specification techniques; and b) access control specification techniques. Finally, we discuss the limitations of the existing methods.

2.1. *General security specification techniques*

UML use cases (UC) are a common method to model functional requirements; thus, it is natural to expect that this method will be extended to deal with security requirements. One such extension is "misuse cases" (Sindre and Opdahl, 2005). Misuse cases describe how different kinds of actors might attack or misuse the system and their relationships with the desired functionality. Gomaa and Shin (2009) suggested another way to define security requirements using UCs. They proposed to separate the functional UCs from the security UCs, as opposed to the above use and misuse cases. The security UCs handle only the security requirements, and are later on combined with the functional UCs.

Several specification techniques for presenting security policies in a model-driven software development process have been proposed. UMLsec (Jürjens, 2005) was one of the first to integrate security with UML. UMLsec extends UML to enable specifying security concerns in the functional model. It uses 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 RBAC (Role-Based Access Control) policies and to specify guarded access.

Secure Tropos (Mouratidis and Giorgini, 2007) is a security-oriented extension to the Goal-Driven Requirements Engineering methodology, which enables to model security concerns of agent-based systems by allowing functional and non-functional requirements to be defined together, yet being clearly distinguished. Secure Tropos methodology is applied in four main phases: (1) During the early requirements analysis phase, functional as well as security requirements are analyzed in terms of constraints imposed by the stakeholders of the system. Then, secure goals and entities are identified, which guarantee the satisfaction of the constraints. (2) During the late requirements analysis phase, further analysis of the system within its operational environment, together with relevant functions and security specification, is performed. (3) During the architectural design phase, the architectural style of the system is defined. In this stage, the designer describes in detail all of the actors with respect to their goals and tasks, identifies and assigns agent capabilities while considering the identified security requirements, and transforms the requirements to a design with the aid of security patterns. (4) During the detailed design phase, the designer specifies each architectural component in further detail. A major limitation of Secure Tropos is that it mainly focused on the phases of early and late requirements analysis, but it does not provide adequate support to the design of security policies.

To overcome this limitation, Mouratidis and Jürjens (2010) combined Secure Tropos and UMLsec to create a structured methodology for secure software development that supports all software development phases. Since Secure Tropos' main focus is on requirements analysis, and UMLsec's main focus is on security analysis and design, the two approaches

complement each other. The combined approach consists of four stages: the first two stages, security analysis of the system environment, and security analysis of the system itself, aim at elicitation the security requirements based on a goal-driven security requirements analysis. Next, during the secure system design stage, the design of the system is developed with the use of Tropos and UMLsec models. Finally, during the secure components definition stage, UMLsec is used to specify in detail the components of the system while validating the security properties of the design according to the security requirements of the system. Note, however, that, UMLsec does not enforce security policies.

Another approach for security specification is security patterns, which is based on the classic idea of design patterns introduced by the Gang of Four (Gamma et al., 1995). 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 (Schumacher, 2003). 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 Schumacher et al. (2006) and Fernandez et al. (2006). This methodology integrates security patterns into each one of the software development stages, and each stage can be tested for compliance with the principles presented by the patterns. A catalog of security patterns helps to define the security mechanisms at each architectural level and at each development stage. In the requirements stage, each action within a use case (UC) is analyzed, inferring possible attacks, and creating secure UCs. Then, the developer determines which policies would stop these attacks. In the analysis stage, patterns are used to build the conceptual model, and authorization rules are generated to apply the conceptual model. In the design stage, patterns are used to enforce rules through the architecture. Finally, the implementation stage requires that the security rules defined in the design stage will be reflected in the code while incorporating COTS (commercial off the shelf) security applications. This method mainly provides guidelines about which security related procedural patterns[1] could be useful in the various development stages, and some design patterns such as role base access control (RBAC), but no verification algorithm is proposed in order to verify the implementation of the design patterns in the application, and no concrete ways to transform the application into code are provided.

Hafner and Breu (2009) proposed a Model Driven Security (MDS) methodology for service-oriented architectures. The conceptual framework is based on ProSecO for requirements engineering and a MDS framework called SECTET for the model-driven configuration and management of security infrastructures. SECTET weaves three software engineering paradigms: Model Driven Architecture as methodical concept, Service-Oriented Architecture as architectural paradigm, and Web Services as technical standard. It supports transformation of model information into configuration code for the components of the target architecture. In the context of access control, the authors show how to model access control policies with SECTET-PL, an OCL-based language. By comparison, this framework supports the realization of security-critical inter-organizational workflows, but does not address the development process of object-oriented applications and databases.

2.2. Access control specification techniques

Fernandez and Hawkins (1997) proposed a method to determine privileges and rights of roles with the least privilege principle taken in mind, i.e., each role gets only the necessary privileges in order to do his job. The method is applied during the analysis phase using an extension of UCs that describes the functional requirements along with other, non-functional specifications. This method enables the specification of access control constraints as pre-conditions, description, exceptions, and post-conditions of the UCs.

AuthUML (Alghathbar and Wijesekera, 2003) is a framework to specify and analyze access control at the UC level. The goal of AuthUML is to analyze (not necessarily to model) access control policies during the early stages of the development lifecycle in order to ensure consistent, conflict-free and complete requirements. AuthUML is further extended by Alghathbar (2007) to incorporate RBAC into UML, and validate the enforcement of Separation of Duty during the requirement engineering phase. In a later study, Alghathbar (2009) proposed a way to express access control and authorization of actors based on roles within UCs. This method added several notations to UC diagrams via stereotypes, which indicate the authorization specifications of actors with respect to specific UCs, i.e., it specifies which roles have privileges on which UCs.

Popp et al. (2003) suggested an approach based on UMLsec for conceptual modeling of security requirements. There are three main activities within this UC-oriented requirements engineering method: (1) The static aspects of the application domain along with their access policies and other security properties are modeled to create the security data model. In order to build the static security data model, stereotypes, tags and constraints are used – all of which are components available in UMLsec. Critical objects are defined by the stereotype <<critical>>, and the level is mentioned by a tag (i.e., secrecy, high, integrity). (2) UCs are specified with an extension of textual description, unfolding threats and the vulnerability of the inputs and the outputs. (3) The class model is integrated with the security UC model by creating message flows between the objects. Although this method enables to specify security related aspects in class diagrams and in UCs, it does not refer directly to privileges granted to users on the objects; the method mainly provides the level of importance for each object and UC in terms of security.

SecureUML (Lodderstedt et al., 2002; Basin et al., 2006) 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 Object Constraint Language (OCL) (OMG, 2010). The OCL extension is very useful since

[1] Procedural patterns are used to improve the process of secure software development, influencing software lifecycle phases other than design. On the other hand, structural (or design) patterns have an impact on the architecture and the design of the software (Kienzle et al., 2002).

RBAC does not support specifying policies that depend on dynamic properties of the system state (e.g., allowing an operation to take place only during weekdays). SecureUML also supports the automatic generation of access control specification for particular access control technologies based on the application models. Although it provides rich facilities for specifying RBAC based authorization, SecureUML does not support enforcements of security constraints, and is suitable only for RBAC based authorization models. Basin et al. (2009) extend SecureUML and show how OCL expressions can be used to formalize and check security properties of application models. In their method, RBAC policies are expressed as OCL queries that are evaluated on the class model or model instances (object diagram) under consideration.

Koch and Parisi-Presicce (2006) present an approach to integrate the specification of access control policies into UML. The access control models are specified in three levels: a) The access control meta-model level defines the language for specifying access control models such as RBAC, MAC (Mandatory Access Control), or DAC (Discretionary Access Control); these models show only the policy rules and constraints, and do not consist of any application specific information. b) The model level is an instance of the access control meta-model that introduces the application information. For example, if the meta-model specifies a RBAC model, then doctor and patient may be roles in a medical application. c) The instance level is an instance of the access control model in a specific application. UML class diagrams are used to specify the access control model entities, and UML object diagrams and OCL are used to specify rules and constraints. In addition, Koch and Parisi-Presicce (2006) present a transformation from UML diagrams to graphs that allow verifying the model with respect to consistency properties. However, this method does not support the transformation of the models into code.

Other methodologies present the use of aspect-oriented software design to model security as separate aspects that would later on be weaved within the functional model. For example, Ray et al. (2004) propose to deal with access control requirements while utilizing UML diagrams. Here, 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 automated verification of the application, nor transformations in any way.

Another example is Pavlich-Mariscal et al. (2010) who present a framework to model access control policies and transform them into code. This framework includes a set of access control diagrams – UML extensions – to model access control policies. In addition, they propose a set of composable access control features, i.e., components that realize specific capabilities of RBAC, MAC, and DAC, in a set of access control diagrams. Designers can select and compose features to achieve the desired behavior in an access control policy. The final design maps into code, which constrains access to methods based on information from the policy code. Although the presented extensions allow specifying a fairly broad range of requirements, there are still access control capabilities that are not included in these models or in the composable access control features, such as lack of ability to specify time constraints, indicating when subjects can access to the protected resources.

Fernández-Medina and Piattini (2005) deal with designing secure databases. They propose a methodology to design multilevel databases based on MAC policies. The methodology enables to create conceptual and logical models of multilevel databases, and to implement the models by using Oracle Label Security (Czuprynski, 2003). 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 of transforming specification artifacts into implementation. However, they do not provide tools for verifying the application vis-à-vis security policies.

2.3. *Limitations of existing methods*

It is crucial for organizations to assure that their security policies are not neglected or ignored during the application development processes. However, existing methods do not provide means to enforce organizational security policies on the development process. The methods that we have surveyed mainly provide guidelines regarding how security can be handled within certain stages of the development process, or address specific aspects of developing secure applications. Although some of the surveyed methods provide means for checking models, they do not support the ability to verify the correct application of security policies. Our study addresses these gaps by introducing a comprehensive methodology that enables: a) at the organizational level, to define the organizational security policies; b) then, at the application development level, to define security requirements while enforcing the above security policies, to verify them, and then to implement them in the database application.

3. Background for the proposed methodology

As a background for the proposed methodology, we provide a brief survey of two underlying approaches: one is the Application-based Domain Modeling (ADOM) approach, which is used for enforcing and validating the security constraints on application models. The other is the Functional Object-Oriented Methodology (FOOM), which we use as the base methodology for the analysis and design of applications.

3.1. *The ADOM approach*

The Application-based Domain Modeling (ADOM) (Reinhartz-Berger and Sturm, 2009) 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 conformance checking of the enterprise-specific models vis-à-vis the relevant reference models. The architecture of ADOM is based on three layers:

1. **The language layer** comprises meta-models and specifications of the modeling languages. In this work, UML 2.0 is used as the modeling language and in particular the class diagram.
2. **The domain layer** holds the building elements of the domain and the relations among them. It consists of specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The structure and the behavior of the domain layer are modeled using the modeling language defined in the language layer. In this work, the structure of each pattern is introduced in the domain model.
3. **The application layer** consists of domain-specific applications, including their structure and behavior. The application layer is modeled using the knowledge and constraints presented in the domain layer and the modeling constructs specified in the language layer. An application model uses a domain model as a conformance template. All the static and dynamic constraints enforced by the domain model should be applied in any application model of that domain. In order to achieve this goal, any element in the application model is classified according to the elements declared in the domain model using UML built-in stereotype. In this work the application model elements are classified by the patterns (domain) model elements.

For describing variability and commonality, ADOM uses multiplicity stereotypes that can be associated to all UML elements, including classes, attributes, methods, and associations. The multiplicity stereotypes in the domain model aim to define the number of times a model element of this type may appear in an application model. This stereotype has two associated tagged values – min and max – which define the lowest and the upper most multiplicity boundaries. For the purpose of clarity, four commonly used multiplicity groups were defined: <<optional many>> (0:n), <<optional single>> (0:1), <<mandatory many>> (1:n), and <<mandatory single>> (1:1).

The relations between reusable (domain) elements and their logical instantiations (application) are maintained by the UML stereotypes mechanism: each domain element can serve as a stereotype to an application element of the same type (e.g., a class that appears in a domain model may serve as a classifier of classes in an application model). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain model. Some reusable elements are optional, and may be omitted from the application model. In this case, there will be no elements in the application model classified as the omitted reusable elements. On the other hand, elements that are application specific (and therefore do not have a corresponding reusable domain element) may be added to the application model. These elements do not have a classifying stereotype.

To prevent application developers from violating domain constraints while (re)using domain artifacts in the context of a particular application, ADOM provides a conformance checking mechanism. The algorithm is briefly describes in Section 4.3.2.

3.2. FOOM

FOOM is a methodology for analysis and design of information systems that combines the functional- (process-) oriented approach and the object-oriented (OO) approach (Shoval, 2007). According to FOOM, two main models are created in the analysis phase, based on the users' requirements:

1. **Initial class diagram**: this diagram consists of data classes, attributes and various relationship types between the classes (but no functionally is included yet).
2. **OO-DFDs**: these data flow diagrams are similar to traditional DFDs but with a major difference: instead of data-stores they included data classes, which are taken from the initial class diagram.

The two models, i.e., the initial class diagram and the OO-DFDs, are synchronized in the sense that every class in the class diagram must appear (i.e., be used) in at least one OO-DFD, and each class appearing (used) in any OO-DFD must be taken from that class diagram.

In the design phase, the OO-DFDs are decomposed into "transactions". A transaction, originally defined in ADISSA methodology (Shoval, 1988), is an independent process/activity that performs some functionality of the system for a user. Conceptually, a transaction is very similar to a UML UC; however, it includes more components: besides the functions, a transaction diagram may include external entities from which input data arrives or where the output information goes, and data classes from which data (objects and attributes) are retrieved or updated (added, deleted or changed). Moreover, the links between the components of the transaction are directed data flows that carry certain data elements from the source component to the target component.

In further steps of FOOM, the process logic of each transaction is described in detail using pseudo-code or some flowchart, and the input and output screens and reports involved in each transaction are designed. Eventually, each transaction description is further decomposed in methods, each being attached to a proper class. Another step performed at the design phase of FOOM is mapping of the class diagram into a relational database schema. This step too is utilized as part of the methodology proposed in this study.

Note that transaction diagrams may be created directly from the users' requirements, without creating the OO-DFDs and then decomposing them into transactions. This is quite similar to the process of creating UCs in use-case-driven methodologies. But still, a major difference between transactions and UCs is that the transaction diagrams include more components, notably the data classes. Later on we will show how the transactions descriptions can be extended to enable definition of user roles and access privileges, definitions that will eventually be implemented in the database.

4. The methodology

The proposed methodology supports the entire development lifecycle of a secured (authorization wise) database. The

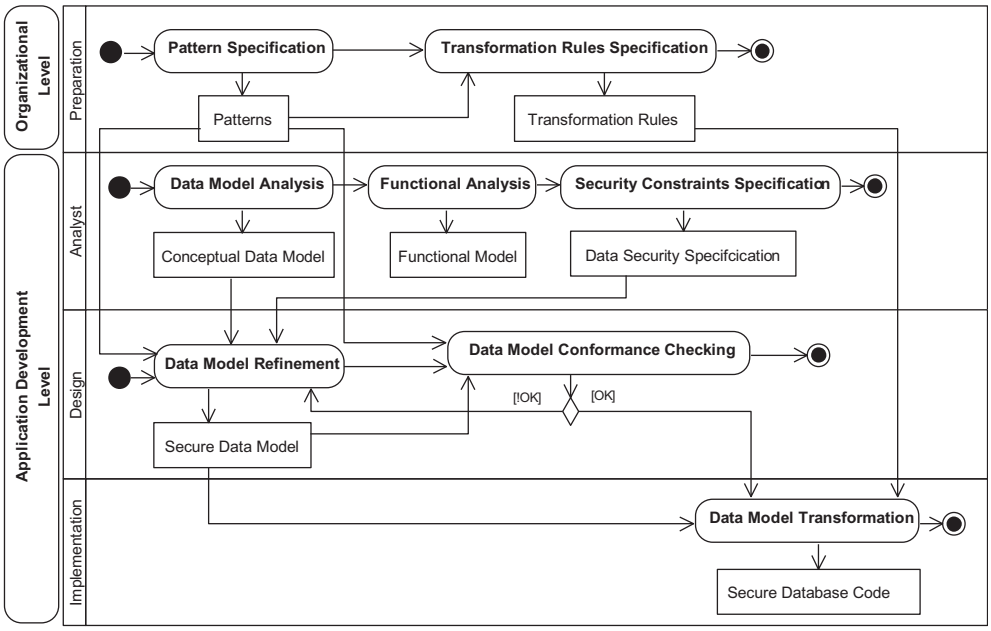

Fig. 1 – The methodology overview.

methodology was developed to enable its weaving into existing development processes. Thus, the techniques we utilize are kept as similar as possible to the standard ones; we adopt the modeling language UML along with OCL, and extend them when required.

Fig. 1 presents the methodology scope in terms of the tasks to be performed in each phase and the various artifacts generated during the development process. We roughly divide the development process into four phases: preparation, analysis, design and implementation. The preparation phase occurs at the organizational level, whereas the other three phases occur at the application development level. Note that the methodology provides the actions and artifacts to be introduced, yet it does not enforce the order of these activities as long as the outcomes are consistent.

4.1. *Organizational level – the preparation phase*

In this phase, the security officers and the domain experts specify the organizational security patterns, which will later on be enforced during the development of any application and its database schema. In addition, these experts define transformation rules that depict on how to transform the application model that is based on the patterns into database code. These artifacts are reusable and may be applied to numerous applications. The security officers are expected to be familiar with the available technologies, as they determine the security patterns to reflect general security policies within the organization. The domain experts model the security policies as security patterns.

4.1.1. *Specification of security patterns*

Security patterns have been introduced by Schumacher (2003) and Fernandez et al. (2006). Such patterns serve as informal guidelines of how to look for and address security threats. In the proposed methodology, the patterns are specified according to the ADOM approach. These patterns serve as guidelines for application developers as well as conformance checking templates, and 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 necessarily 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 (2003). The template consists of five main sections: name, context, problem, solution, and consequence. Here we discuss only the solution in bold, as other sections are similar to those found in the literature. The solution section describes a generic solution to the problem. In this study, 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 a 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 elements that were already defined by the class diagram specifying the structure of the pattern. During the application design phase, the designer will use these templates in order to specify fine grained access control policies.

In the following we show an example of the Role-Based Access Control (RBAC) pattern[2] that is commonly used in the context of database authorization. Fig. 2 demonstrates the way according to which it is modeled in the proposed method. ***Role***

[2] Our specification of the RBAC pattern is comparable to the one appears in Fernandez et al. (2006), yet its actual usage is different.

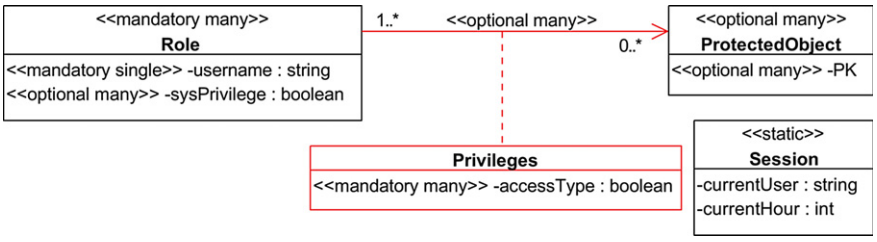

Fig. 2 – The RBAC security pattern residing within the domain layer.

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. ***ProtectedObject*** is akin to a database table. ***Privileges*** association class determines the schema object privileges of a ***Role*** with respect to a specific ***ProtectedObject***. A class of that type within an application must include at least one object privilege – ***accessType***. In addition, one can specify system privileges[3] assigned to some ***Role*** by using the ***sysPrivilege*** classification. The ***Session*** class appears in the pattern aims at specifying information that is related to the session (in this case the connected user and time); this information is used within the transformation rules and templates. Yet, as it is not classified using the multiplicity indicator, it means that the application developers are not required to explicitly refer to this when (visually) specifying their applications.

In addition to the pattern class diagram, OCL rules that further constraint and clarify the pattern can be specified. These OCL rules are evaluated in the application development layer during the conformance checking stage. 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(e|e.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(p|p.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(e|e.name)->includes('SYSDBA')
     else
        self.sysPrivilege->collect(e|e.name)->excludes('SYSDBA')
     endif
```

In general, RBAC models define privileges for objects as one unit; however sometimes fine grained access control is required. Access control on the objects' properties and instances (i.e., columns and rows in relational databases tables, respectively) enable preservation of security principles such as the minimal privilege principle. In order to enable 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. 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 have to be given to a ***Role*** on a specific ***ProtectedObject***, and then the access can be limited to some of the instances or properties by OCL constraints.

In order to incorporate property level (column or cell level) access control constraints, an extension of the OCL standard library 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, condition):
Boolean
```

where `role` is the activating role, `action` is the operation (i.e., select, insert, update, delete) to be performed on the element,

[3] System privileges allow the user to perform system level activities.

element is an object or a property of an object that needs to be protected, and condition 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 control, the **property (column) level OCL template** can be used as follows[4]:

```
context ⟨Privileges⟩
inv: self.⟨Role⟩.isAuthorized(self.⟨accessType⟩,
                              self.⟨ProtectedObject⟩.⟨property⟩,
                              ⟨condition⟩)
```

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

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 = ⟨Role⟩.⟨username⟩ and
     ⟨ProtectedObject⟩.[⟨navigation-between-the-protected-object-and-
                        the-role⟩].⟨Role⟩ -> includes(⟨Role⟩)
```

This OCL means that in the context of some <<Privileges>>, a user of the application has to have a <<role>> that is part of the collection related to the <<ProtectedObject>>. This navigation is possible only if the <<role>> is additionally classified as a <<ProtectedObject>>. Note that this template enables the designer not only to specify constraints on all <<accessTypes>> within <<Privileges>> specified between the ***Role*** and the ***ProtectedObject***, but also to specify constraints on a specific <<accessType>> within a <<Privileges>>.

Note that in this paper we specified an example of only one pattern. During our study, we have also specified the DAC and MAC patterns. Yet, it is clear that many other patterns can be defined by an organization.

4.1.2. *Specification of transformation rules*

Model transformation plays an important role in the Model Driven Development (MDD) approach (Selic, 2003). 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 the relevant pattern need to define the way in which application elements should be transformed into database code. To transform the UML class diagram into SQL code, "model to code" transformation tools have to be provided. This type of transformation can use technologies such as ATLAS Transformation Language (ATL) (Jouault et al., 2008), which first transforms the application model into an SQL application model, and then translates this model to SQL code.

To enforce access control within a database, one can use the inherited capabilities of the database product like privileges grant and views, as well as other advanced mechanisms. In this study, we used the grant privilege options as well as the Virtual Private Database (VPD) of Oracle (2011), as this was our target database technology, and it also enables handling fine grained access control in a unified manner. Nevertheless, the transformation rules could also be applied to generate views or other access control mechanisms.

As already said, in order to incorporate fine grained access control, we use text templates in the form of OCL templates. Text templates are essentially exemplars of the desired output code with "blanks" that should be filled in with values of attributes or with output code from a nested template. These "blanks" contain meta-code and are usually delimited between "< >". For example, the following OCL constraint defines a simple OCL template:

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

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

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

After the missing values were inserted, a template engine is used to create the output code. The processing of the inserted variables can be a simple replacement of a variable with its value, or a more complex transformation.

[4] In the following, syntax parts enclosed in between angular brackets ("<>") indicate mandatory elements, while those enclosed in square brackets ("[ ]") are optional.

Following the example above, using the class diagram in Fig. 3, a constraint will be set on the class *Enrollment*, where 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)
```

The transformation of OCL constraints to SQL or PL/SQL code can combine text templates such as StringTemplate (2010), with more SQL oriented transformation techniques such as the Dresden OCL Toolkit (Heidenreich et al., 2008).

To summarize, the outcome of the two tasks within the preparation phase are: a) specification of security patterns that will guide developers and compel them to specify security requirements as determined by the security policies and b) definition of transformation rules (and templates) from application models into database code.

4.2. *Application development level*

Before describing the methodology in the application development level, we present a simple example that will be used to demonstrate the tasks and outcomes in the various phases of the application development process. The example is part of a university system, dealing with students, enrollment to courses, and reporting of course grades.

> The university system manages students, lecturers, course enrollments and grades. Each department offers 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, information about students' enrollments to course offerings may be visible only to the lecturers of the offered courses. A lecturer's phone number may be visible only to students who are enrolled to one of the lecturer's courses; lecturers and secretaries may view this data unconditionally. Secretaries may add new courses to the system and update their level or credits. Secretaries may also add or delete course offerings and assign lecturers to them; however, secretaries may not see or modify the students who are enrolled to course offerings. Lecturers may update all of the information related to their course offerings, except the students enrolled. 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 allowed to grade only their students. Lecturers may update their personal information, excluding the department they belong to and their assignments to course offerings. Students may see and modify their own personal information and enrolled courses, and may see (but not update) their grades. Secretaries may add or delete students and lecturers from the system.

4.2.1. *The analysis phase*

In the analysis phase of the development process, we follow the FOOM methodology (Shoval, 2007) and create two

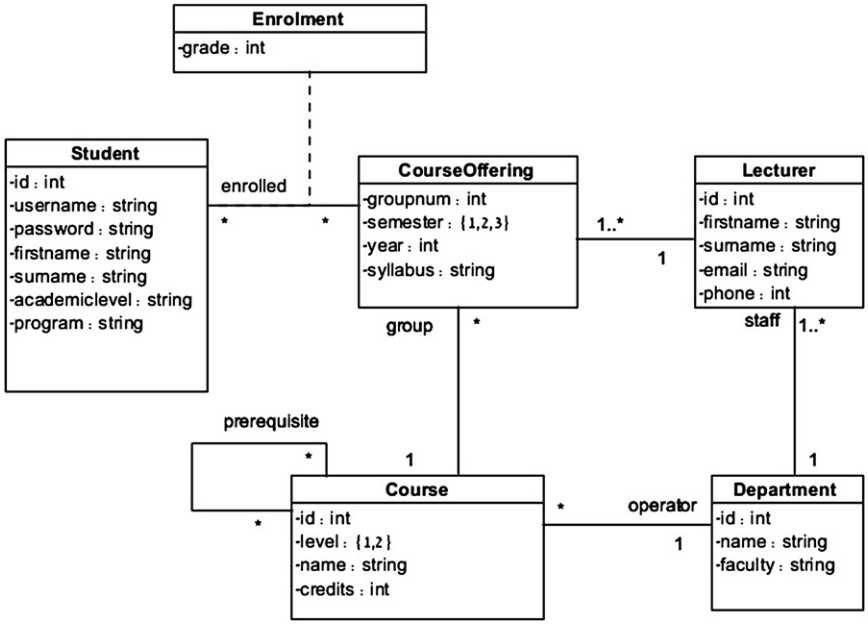

Fig. 3 – Initial class diagram of the university system.

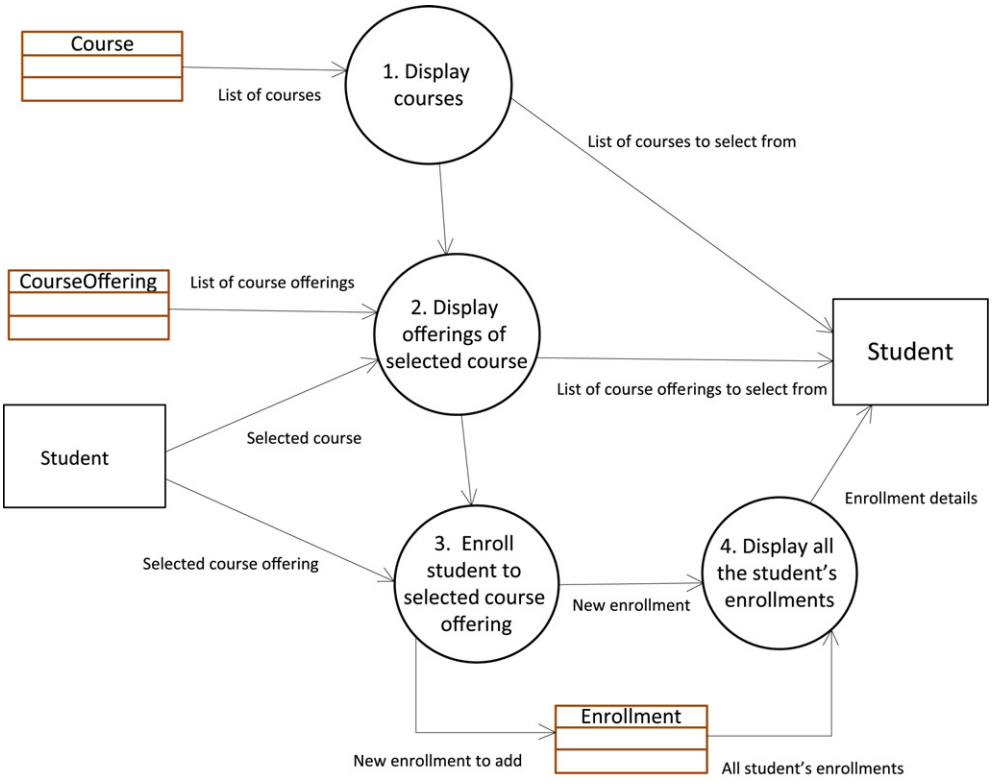

Fig. 4 – EUC diagram of a course enrollment.

models: a conceptual data model and a functional model, that are augmented with specifications related to authorization.

4.2.1.1. *Conceptual data model*. The first task in the analysis phase is to create a conceptual data model based on the users' requirements. The conceptual data model is an initial class diagram that consists of data classes, attributes and various types of relationships. Following the widely-used modeling language, we adopt the notation of the standard UML class diagram (that is somewhat different from the notation used in FOOM). Fig. 3 depicts the initial class diagram of the above university system example.

4.2.1.2. *Functional model*. The second task in the analysis phase is to create a functional model. In use case (UC) driven methodologies, the functional requirements are defined using UML UCs. However, such UCs do not include data classes and other elements that are needed in order to enable definition related to authorizations, i.e., inputs and outputs, and data classes. Therefore, we opt to use "extended use cases" (EUC). A EUC diagram, originally termed "transaction" in the FOOM methodology, may include one or more functions connected to one another with directed data flows, along with data classes that are used by those functions in order to retrieve or update data, and external/user entities from which input data comes or output/information goes to. (Hence, a EUC can be seen as a data flow diagram version of a UC.) Note that at this stage we have already created an initial class diagram; the data classes included in any EUC diagram are taken from this class diagram[5]. Fig. 4 shows an example of a EUC diagram for the enrollment of a student to a course offering.

As in ordinary UC, for every EUC diagram we also prepare a description. The template for a EUC description is somewhat different from an ordinary UC description, as it includes also definitions related to authorization. In the following we provide a description of the above EUC diagram. Note specifically that for each class included in the EUC we define: the *access type* (add, read, update or delete), the *attributes* involved in that operation, and the *authorized operators* of the EUC, i.e., who are authorized to run this EUC.

EUC description:
Authorized operators: Student
Pre-condition: none
Post-condition: the student is enrolled to a course offering
Main success scenario:

1. The system retrieves a list of courses from class "Course" and displays them to the student.
2. The student selects a course to enroll to.

[5] It should be noted that external/user entities in the EUC do not signify the operators but rather the sources and destinations of data/information. The roles of operators of each EUC are not signified in the diagram but will be defined separately in the EUC description.

3. The system retrieves a list of the selected course's offerings from class "Course Offering" and displays them to the student.
4. The student selects one of the course's offerings to enroll to.
5. The system adds the selected course offering to class "Enrollment".
6. The system retrieves all the enrollments of the student from class "Enrollment" and displays them to the student.

Extensions: none
Access privileges:

Class name	Access type	Attributes	Authorized operators	Constraints
Course	Read	id, name, level, credits	Student	
Course Offering	Read	group num, year, semester, syllabus	Student	
Enrollment	Read, Add, Update		Student	Only his/her own

Note that for each class participating in a EUC, we define in the Access Privileges Specifications table, who is allowed to access it (i.e., the authorized role or operator), the details of the access (e.g., read or write of certain attributes of the respective class) and other possible constraints.

4.2.2. *Specification of security constraints*

This task actually involves the aggregation of all access privileges defined for all the EUCs of the application. The outcome of this step can be expressed in the form of a table, similar to the table shown above for one EUC. Table 1 presents a part of the aggregated table, only for the authorized operator *Student*. The complete table will include the privileges granted to all the operators of the application. Obviously, this table can be sorted by class names, as shown, or by the authorized operators, so that the developer could see easily the authorized accesses to the various classes for each operator.

authorization rules are added to the initial class diagram as specified by the predefined patterns, and the relevant elements are classified via stereotypes according to the patterns. During this task, additional changes to the authorization rules may be applied. Note that choosing patterns to be assigned for each element within the application can be guided by using existing approaches such as misuse cases and attack trees.

Fig. 5 presents a class diagram of the university system example along with the security specification as determined by the RBAC pattern specified in the Preparation phase. The various elements that are relevant for the RBAC pattern are classified by stereotypes with the pattern elements: *Role*, *ProtectedObject*, and *Privileges*. For example, the *Student* role has a system privilege of CREATE SESSION and the following schema object privileges: (1) SELECT privilege to Department, Lecturer, Course and Course Offering classes; (2) SELECT, INSERT, DELETE privileges to Enrollment class; and (3) SELECT and UPDATE privileges to Student class.

As explained in Section 4.1.1, the expressive power of a class diagram is limited when it comes to representing fine grained access control privileges. For example, in university system presented in Fig. 5, a student can enroll other students to courses and update their personal data. Thus, working under the RBAC pattern defined in Section 4.1.1, whenever fine grained access control is desired, the designer should specify these constraints using the predefined OCL templates. The following OCL example illustrates the use of the instance level constraint that limits students to read, insert, or delete only their own enrollments:

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

4.3. *The Design Phase*

In the design phase, the analysis phase models are transformed into a coherent model including security specifications that adhere with the organizational policy.

Another example is of a property level constraint in which a student may see a lecturer's phone number only if enrolled in 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))
```

4.3.1. *Refinement of the data model*

In this task, the designer formalizes the initial class diagram and the specification of the access privileges specifications and creates a unified class diagram. This means that the

Note that not all elements have to be classified, as some of them may not be related to the patterns; such elements will be ignored by the conformance checking mechanism. Note also that an element may be classified

Table 1 – The university application privileges specification for the Student role.

Class name	Access Type	Attributes	Authorized operators	Constraints
Student	Read	id, username, academic level, program	Student	Only his/her own
Student	Read, Update	firstname, surname	Student	Only his/her own
Department	Read	id, name, faculty	Student	
Course	Read	id, name, level, credits	Student	
Course Offering	Read	group num, year, semester, syllabus	Student	
Enrollment	Read, Update, Add		Student	Only his/her own
Enrollment	Read	grade	Student	Only his/her own

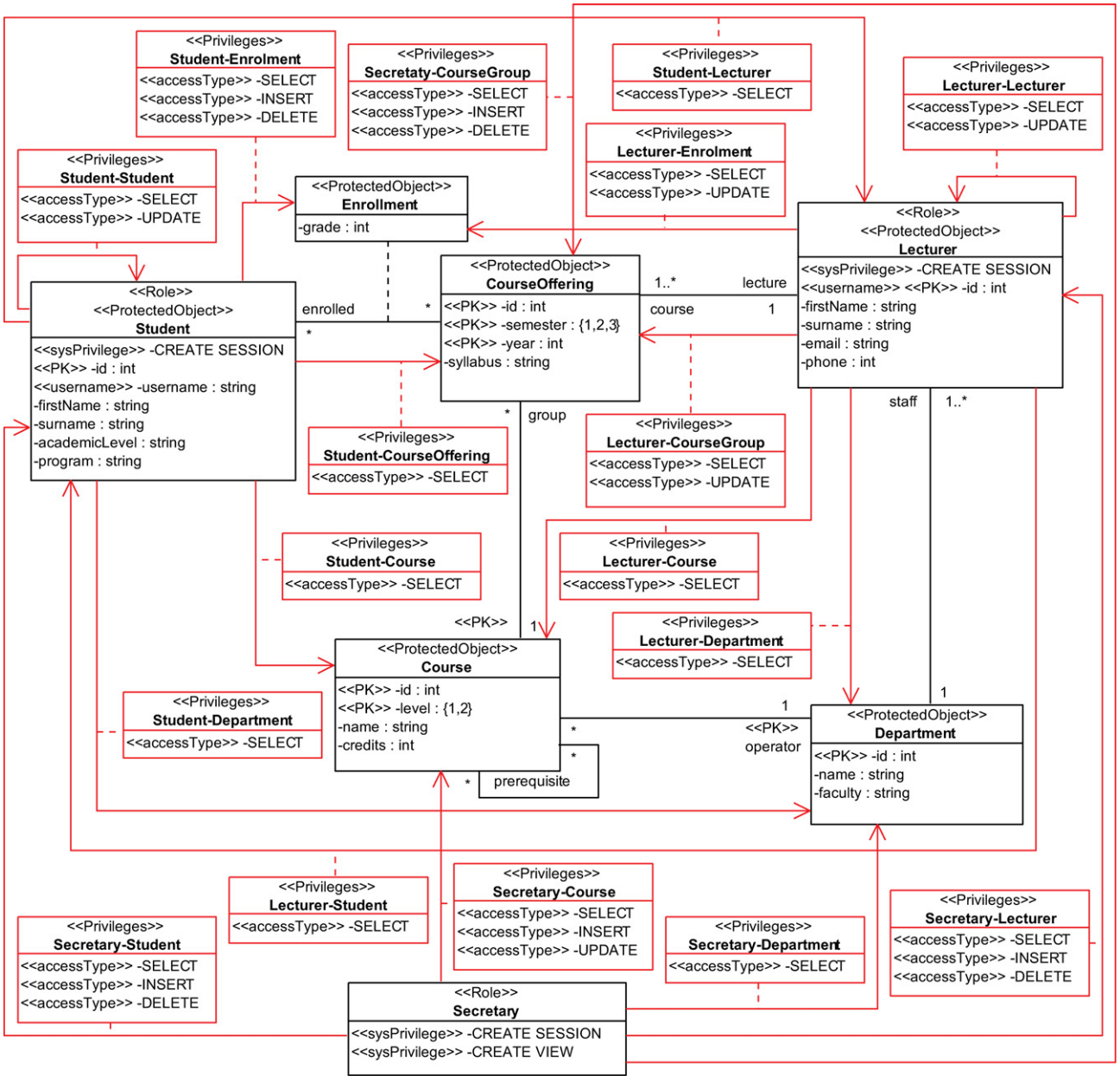

Fig. 5 – The RBAC-base refined data model of the university system.

by multiple patterns; such elements will be checked for conformance with respect to all assigned patterns, and the generated artifacts will weave all related patterns transformations.

4.3.2. *Data model conformance checking*

After having created a refined data model, we need to check if it adheres to the security policies as defined by the specified security patterns. In the following we define a pattern-based conformance checking for checking the adherence of application models with the domain model. When using these patterns at the application design phase, a conformance checking will be performed separately and independently for each of the patterns. The application is verified in several facets: a) multiplicity, b) OCL, c) language, and d) cardinality.

a) **Multiplicity** conformance, which is based on the ADOM validation algorithm (Reinhartz-Berger and Sturm, 2009), checks that the application adheres to the security patterns with respect to elements multiplicity. It is performed in three steps, 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 pattern are disregarded. In the **element unification** step, elements having the same pattern 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 in order to verify the multiplicity of the elements, and the application model structure with respect to the pattern.
b) **OCL** conformance checks that OCL rules defined in the pattern hold in the application. This conformance checking iterates over all pattern elements. For each of them it extracts its OCL rules and retrieves its logical instantiations from the application. For each logical instantiation, the conformance checks that the extracted OCL rules hold.
c) **Language** conformance 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 pattern attributes must have the same type, or a subtype thereof. For instance, in the RBAC pattern defined in Fig. 2, the *username* attribute in the *Role* class is defined as *string*; this means that all attributes that are classified as *username* in the application have to be of type *string*. However, for attributes that were defined as *object* (or their type was not defined), such as the *PK* in the *ProtectedObject* class, they can be classified as any type.
d) **Cardinality** conformance confirms that the cardinality defined on association 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.

In the university example, there are no unclassified classes, thus the reduction step of the multiplicity conformance checking is redundant. The resultant model after performing the unification step consists of three classes: *Role* with multiplicity <<3..3>>, *systemPrivilege* with multiplicity <<1..2>>, *ProtectedObject* with multiplicity <<6..6>>, *PK* with multiplicity <<0..3>>, *username* with multiplicity <<1..1>>, *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 access types specified. In addition, all of the OCL constraints defined in the pattern evaluate to true and there are no language or cardinality violations. An example for a violation in the context of OCL conformance checking would have been if one or more of the roles would not have the `CREATE SESSION` system privilege, or one of the roles would have object privileges different from the ones defined, i.e., `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

4.4. *The implementation phase*

In this phase, the developer has a compliant (with respect to the security patterns) conceptual data model enhanced with security constraints. The data model transformation task (see Fig. 1) aims at transforming this model into SQL statements defining the database schema, along with the authorization constraints.

The transformation of the class diagram into the relational schema can be done using existing algorithms, such as Blaha et al. (1994) and Shoval (2007). In the proposed methodology, the transformation of the security specification is done by following the transformation rules and templates that have been defined at the organizational level by the security officer and the domain expert. Following the rules and templates defined at that level, and the application specification created in the design phase, the SQL statements generated for the Student role and a sample of the SQL fine grained code for the university application are as follows:

```
-- Student role creation:
CREATE ROLE STUDENT;

-- system and object privileges for the student role:
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 (defined in section 4.1.1.).
-- A student may read, insert, or delete an enrollment to a course offering
-- only for herself:
CREATE FUNCTION STUDENT_ENROLMENT (SCHEMA_NAME VARCHAR2, OBJECT_NAME VARCHAR2)
  RETURN VARCHAR2 AS
    USERNAME VARCHAR(30);
    CNT NUMBER;
    ID NUMBER;
BEGIN
  IF (NOT DBMS_SESSION.IS_ROLE_ENABLED('STUDENT')) THEN
    RETURN NULL;
  END IF;
  USERNAME := SYS_CONTEXT('USERENV', 'SESSION_USER');
  SELECT COUNT(*) INTO CNT FROM STUDENT WHERE STUDENT.USERNAME = USERNAME;
  IF (CNT = 0) THEN
    RETURN '1=2';
  END IF;
  SELECT "ID" INTO ID FROM STUDENT WHERE STUDENT.USERNAME = USERNAME;
  RETURN 'STUDENT = ' || ID;
END;

BEGIN
  DBMS_RLS.add_policy(
    object_schema   => 'UNIVERSITY',
    object_name     => 'ENROLLMENT',
    policy_name     => 'STUDENT_ENROLMENT',
    policy_function => 'STUDENT_ENROLMENT',
    statement_types => 'SELECT, INSERT, DELETE',
    update_check    => TRUE);
END;
```

4.5. *Security modeling tool*

We have developed a software tool entitled Security Modeling Tool (SMT) to assist developers utilizing the methodology (Abramov et al., 2011). Since the methodology is to be incorporated within an ordinary development process, we utilized in SMT a set of existing technologies and frameworks. It is based on the Eclipse framework and its various modeling capabilities, such as Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF), UML, OCL (for the OCL solver we select the Dresden OCL; Heidenreich et al., 2008), and ATL (Jouault et al., 2008). On top of this infrastructure we adopt an open source tool named TOPCASED (TOPCASED, 2010), which provides a graphical UML editor. It was selected for its adherence to an open standard as well as its extendibility and its native implementation in Eclipse. We further adjust TOPCASED to support ADOM and other required models.

SMT supports the following activities: At the organizational level, it supports the specification of security pattern using UML class diagram, adopting the ADOM principles and OCL via specific editors; specification of transformation rules using ATL; and definition of templates. In the analysis phase, SMT supports the creation of an initial UML class diagram, specification of EUCs (including EUC diagrams and their descriptions), and the specification of the access privileges and security constraints. In the design phase, SMT supports the refinement of the initial class diagram along with the security pattern elements. It also provides conformance and adherence checking of the application model vis-à-vis the relevant patterns. Finally, in the implementation phase, the tool automatically generates the SQL statements that include specifications of all the database security constraints. Additionally, SMT provides documentation of the various artifacts created along the application development process.

5. Conclusion

We presented a methodology that guides organizations and developers on how to incorporate security aspects related to database access authorization within the development process. We aim at weaving the sought methodology into the regular development process. Therefore, we based it on well-known and used techniques such as UML and OCL. The

methodology is applied at two main levels: At the organization level it enables to define and formalize the security policy required for applications to be developed. This is done using class diagrams and OCL rules as well as form-based template specification and ATL transformations. At the application development level the methodology involves the three main phases of development: In the analysis phase, the requirements are defined using a class diagram, extended use cases, and security constraints specification. In the design phase, the requirements are formalized and verified using automatic verifiers (for the patterns and for OCL). Finally, in the implementation phase, the design products are transformed into a database code.

Along the development of this methodology, we have worked closely with experts who have extensive knowledge about problems and difficulties involved in dealing with security aspects. During mutual discussions, various usage forms of the methodology were proposed, including of using it for the development of new applications as well as for dealing with security problems in existing systems. We also discussed the potential of extending the methodology to deal with additional security aspects besides authorization, and the percolation of security aspects to other software layers, not only to the database. Another possibility to improve the methodology is to adopt ideas from other studies that addressed security aspects and weave these within the proposed methodology.

So far we have used the methodology in a few small cases; obviously, it should be further tested and evaluated. We already have conducted a controlled experiment aimed at verifying that the methodology indeed eases the developers' efforts and improves the quality of the design. In that experiment, we examined whether the abstraction achieved by the proposed security patterns helped designers to achieve better security specifications than when using direct coding in SQL. Software Engineering students played the roles of designers; they received a requirement document that included the security requirements of some application in narrative form, and were asked to add security constraints to the existing database schema of that application using either security patterns or just SQL. The experiment results showed that the designers who used the security patterns achieved better results than the designers who used just SQL, with respect to security specification related to table/object, column/property, and row/instance levels. The improvements were above 25% and were statistically significant. Yet, that experiment involved only a relatively small example, and we did not utilize all aspects of the proposed methodology; for example, at that time we did not have yet the SMT tool.

Despite providing many advantages, the methodology may suffer from several drawbacks. Pattern specification is not a straightforward task. In order to provide the necessary support to database designers, the security officer and the domain expert who specify the patterns need to have a lot of knowledge regarding general access control mechanisms, and to be aware of different technologies that can be utilized for the generation of secure databases. At the application level, analysts/designer may have difficulties related to exploiting the OCL templates, or specifying complex requirements in an object-oriented model.

In the future we plan to further evaluate the methodology in experimental settings, including the software tool, as well as incorporating it in real world application development. Another research direction is, as said, to extend the methodology to deal with other security aspects, notably privacy and encryption, authentication, etc.

REFERENCES

Abramov J, Anson O, Sturm A, Shoval P. Tool support for enforcing security policies on databases. CAiSE Forum; 2011:41–8. CEUR-WS.

Alghathbar K, Wijesekera D. authUML: a three-phased framework to analyze access control specifications in use cases. In: workshop on formal methods in security engineering. ACM; 2003. p. 77–86.

Alghathbar K. Enhancement of use case diagram to capture authorization requirements. In: Fourth international conference on software engineering advances. IEEE Computer Society; 2009. p. 394–400.

Alghathbar K. Validating the enforcement of access control policies and separation of duty principle in requirement engineering. Information & Software Technology 2007;49:142–57.

Basin D, Clavel M, Doser J, Egea M. Automated analysis of security-design models. Information & Software Technology 2009;51:815–31.

Basin D, Doser J, Lodderstedt T. Model driven security: from UML models to access control infrastructures. ACM Transaction on Software Engineering and Methodologies 2006;15(1):39–91.

Blaha M, Premerlani W, Shen H. Converting OO models into RDBMS schema. IEEE Software 1994;11:28–39.

Czuprynski J. Oracle label security. The Database Journal; 2003.

Dhillon GS. Information security management: global challenges in the new millennium. IGI Publishing; 2001.

Fernandez EB, Hawkins JC. Determining role rights from use cases. In: Workshop on role-based access control; 1997. p. 121–5.

Fernandez EB, Larrondo-Petrie MM, Sorgente T, VanHilst M. A methodology to develop secure systems using patterns. In: Mouratidis H, Giorgini P, editors. Integrating security and software engineering: advances and future vision. IDEA Press; 2006.

Fernández-Medina E, Piattini M. Designing secure databases. Information & Software Technology 2005;47(7):463–77.

Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional; 1995.

Gomaa H, Shin ME. Separating application and security concerns in use case models. In: 15th workshop on Earlt aspects. ACM; 2009. p. 1–6.

Hafner M, Breu R. Security engineering for service oriented architectures. Springer; 2009.

Heidenreich F, Wende C, Demuth B. A framework for generating query language code from OCL invariants. Electronic Communications of the EASST 2008;9.

Jouault F, Allilaire F, Bézivin J, Kurtev IATL. A model transformation tool. Science of Computer Programming 2008; 72(1-2):31–9.

Jürjens J. Secure systems development with UML. SpringerVerlag; 2005.

Kienzle DM, Elder MC, Tyree D, Edwards-Hewitt J. Security patterns repository version 1.0. Technical report, http://www.scrypt.net/~celer/securitypatterns; 2002. last accessed: December 2010.

Koch M, Parisi-Presicce F. UML specification of access control policies and their formal verification. Software and System Modeling 2006;5:429–47.

Lodderstedt T, Basin DA, Doser J. SecureUML: a UML-based modeling language for model-driven security. In: 5th international conference on the unified modeling language. Springer-Verlag; 2002. p. 426–41.
Mouratidis H, Giorgini P. Secure Tropos: a security-oriented extension of the Tropos methodology. International Journal of Software Engineering and Knowledge Engineering 2007;17: 285–309.
Mouratidis H, Jürjens J. From goal-driven security requirements engineering to secure design. International Journal on Intelligent Systems 2010;25(8):813–40.
OMG. Object constraint language, version 2.2, http://www.omg.org/spec/OCL/2.2/; [last accessed: December 2010].
Oracle: Oracle@database – security guide, http://download.oracle.com/docs/cd/B28359_01/network.111/b28531.pdf; [last accessed: December 2011].
Pavlich-Mariscal JA, Demurjian SA, Michel LD. A framework of composable access control features: preserving separation of access control concerns from models to code. Computers & Security 2010;29:350–79.
Popp G, Jürjens J, Wimmel G, Breu R. Security-critical system development with extended use cases. In: 10th Asia-Pacific software engineering conference. IEEE Computer Society; 2003. p. 478–87.
Ray I, France RB, Li N, Georg G. An aspect-based approach to modeling access control concerns. Information & Software Technology 2004;46:575–87.
Reinhartz-Berger I, Sturm A. Utilizing domain models for application design and validation. Information & Software Technology 2009;51:1275–89.
Schumacher M, Fernandez-Buglioni E, Hybertson D, Buschmann F, Sommerlad P. Security patterns: integrating security and systems engineering. John Wiley & Sons; 2006.
Schumacher M. Security engineering with patterns: origins, theoretical models, and new applications. Springer-Verlag; 2003.
Selic B. The pragmatics of model-driven development. Software. IEEE software 2003;20(5):19–25.
Shoval P. ADISSA: architectural design of information systems based on structured analysis. Information Systems 1988;13:193–210.
Shoval P. Functional and object-oriented analysis & design – an integrated methodology. IGI – Idea Group; 2007.
Sindre G, Opdahl AL. Eliciting security requirements with misuse cases. Requirements Engieering 2005;10(1):34–44.
StringTemplate, http://www.stringtemplate.org/; [last accessed: December 2010].
TOPCASED, www.topcased.org; [last accessed: December 2010].

Jenny Abramov is a researcher at T-Labs of Ben-Gurion University. She earned her B.Sc. and M.Sc. degrees in Information Systems Engineering from Ben-Gurion University. Her research interests include domain engineering and system analysis and design in the context of security.

Omer Anson is a scientific programmer at T-Labs of Ben-Gurion University. He earned his B.Sc. in Computer Science and Physics from Ben-Gurion University.

Michal Dahan is a graduate student at the department of Information Systems Engineering of Ben-Gurion University. Her research interests include system analysis and design.

Peretz Shoval is a Professor at the Department of Information Systems Engineering at Ben-Gurion University. He earned his Ph.D. in Information Systems from the University of Pittsburgh (1981) where he specialized in expert systems for information retrieval. In 1984 he joined Ben-Gurion University and founded the Information Systems Program; later on he founded and headed the Department of Information Systems Engineering. Shoval's research interests include information systems analysis and design methods, data modeling, and information retrieval and filtering. Shoval has been selected by ACM as Distinguished Scientist. (2009).

Arnon Sturm is a Senior Lecturer at the Department of Information Systems Engineering of Ben-Gurion University. He earned his M.Sc. and Ph.D. in Information Systems at the Technion. Sturm's research interests include domain engineering, system analysis and design, and business process management.