U N I V E R S I T Y OF T A R T U

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Software Engineering

Kaarel Tark

Role Based Access Model in XML based Documents

Master's thesis (30 ECTS)

Supervisor: Raimundas Matulevičius

Author: ..... "....." May 2013

Supervisor: ..... "....." May 2013

Allow to defence

Professor: ..... "....." May 2013

TARTU 2013

Abstract

Nowadays most of documents are held in digital form. Often document repositories or databases are used to store the info. The info is getting bigger in size and needs to be transferred to partner parties faster. The documents can contain delicate info that not all the actors on the partner side should see and edit. This leads to the need of restricting the actions that a user can do with the document and see in the document. In this work we research possibility to integrate existing technologies to dynamically define forms and their security. For the solution we will introduce a dynamic way to define security on XML documents.

As the info transferred might be high in integrity and confidentiality, we need to keep in mind that despite restricting and giving permissions to user we must also always keep the integrity of the document. In order to fill these needs we will introduce a document structure based Role Based Access Control on each of the document's info element together with a merging strategy to keep document's integrity.

We will validate the approach with a case study following a set of business scenarios: check whether our solution can capture all permissions, roles and subjects from the model and whether the document content stays complete when the document is manipulated.

Table of contents

List of figures

List of tables

Chapter 1: Introduction

Nowadays more and more documents are held in digital form, either in central document repositories or local databases [34]. The amount of data is increasing therefore it is equally important to store and transfer the data between parties. The data content can be in different confidentiality level. Defining security policies to restrict unauthorized access and manipulation of the documents has become necessary.

Extensible Mark-up Language (XML) [17] is a mark-up language that describes the content and the structure of the document. Documents often contain a vast amount of information and their structure is dynamic, therefore it is difficult to apply access restriction rules based on document content. Calculation of the documents confidentiality level associated access permissions requires extensive computational resources. Security policies can be defined for full documents or document categories (*i.e.* contracts, sales orders, public documents). In addition, supplementary permission rules can be defined based on the structure of the documents [13, 20, 32].

Role-based Access Control (RBAC) [25] is a method to generate security in information systems depending on the role definition of the user. It provides a methodology to establish authorization policies and permissions on resources. However the implementation of RBAC is typically done after the information system itself is developed, often leading to additional time spent on development. As the documents are stored in web-based repositories, and their content is requested via a web service, the security solution needs to be applied before the document is sent to the content displaying component. Only authorized information should be transferred to avoid security risks.

As document access permission rules are often defined after the structure and the document have already been developed, setting these permission rules is often complex and requires developer skills. The underlying thesis evolves around the problem outlined above and specifically addresses the following research questions:

> *Can existing technology be integrated to dynamically define forms and security without losing context based information?*

More specifically,

1. *Can we dynamically define forms and permissions of the document?*
2. *Can we keep the document context complete when applying permissions on documents?*

The first sub-question investigates how to define forms and security permissions using existing tools. As outcome an architecture solution will be obtained. The second sub-question addresses how to ensure the document's completeness throughout its lifecycle.

As a solution we apply security modelling language – SecureUML [25] to dynamically define RBAC policy (forms and permissions) on XML documents and XML Schema based form building (DynaForm [29]). Combining XML Schema based form building and dynamic permissions on XML structure yields an automatic solution for defining

document access permissions, eliminating the need for additional implementations for newly created documents.

Developing the security rules and forms simultaneously simplifies the creation of new documents and application of the RBAC solution on them. Here we introduce an architectural model where only data permitted by RBAC rules are displayed to the external partners to limit the possibility of data access and manipulation by unauthorized parties. To validate our results we performed a case study on documents with permissions being defined by test subjects using SecureUML. Based on these SecureUML definitions we generated code and checked the information and completeness of the test-documents when processed.

The thesis is organised in three parts and eight chapters (Figure 1):

Part I, Background consists of two chapters dedicated to the theoretical overview of the used technologies. *Chapter 2: Role-based Access Control (RBAC) and SecureUML* defines security architecture and security modelling tools used in this thesis to define security policies. *Chapter 3: Background* introduces technologies that are supporting the architecture developed in this thesis.

Part II, Contribution consists of three chapters which define how the system is structured, forms are built, permissions are defined and how these three aspects are integrated into one solution. *Chapter 4: Architecture* defines how the technologies mentioned above are integrated. *Chapter 5: RBAC and XML-based Forms* presents how we apply security on documents and how the documents are transformed based on permissions applied.. *Chapter 6: Modelling and Templates to do RBAC* introduces how security is modelled using SecureUML and how code is generated from the model.

Part III, Contribution consists of two chapters dedicated to the validation and conclusion. *Chapter 7: Case Study* reports how we evaluated our research. *Chapter 8: Conclusions and Future Work* presents the conclusion of our research and defines possible future work.

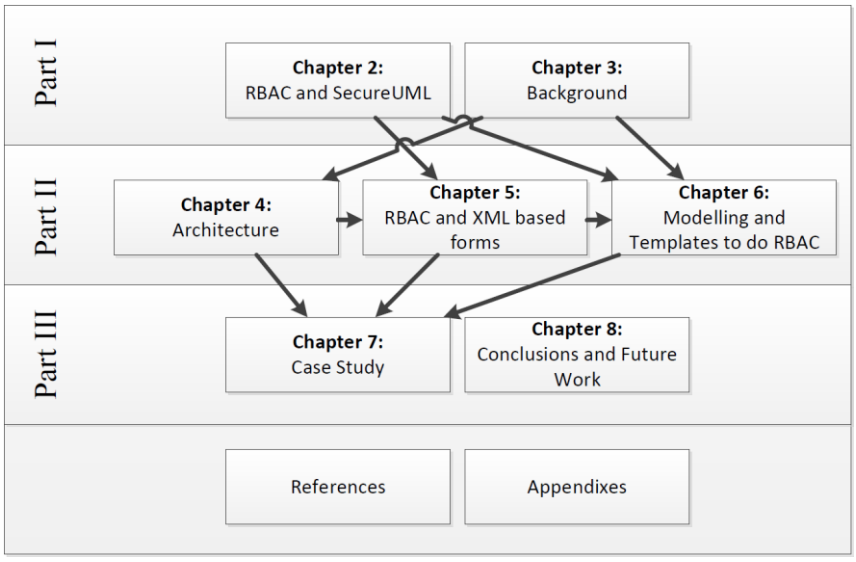


Figure 1 - Structure of the current thesis

References include the list of related work used in our research.
Appendix A describes Velocity template used to generate permissions and XML Schema from SecureUML models.
Appendix B presents Velocity template that is used to generate RBAC permissions from SecureUML models.
Appendix C includes rational database model that is used in our prototype.
Appendix D presents the source code of our application and models created during validation.
Appendix E introduces case study that was given to our experiment group during validation.
Appendix F shows process diagrams (UMLSec [19]) that were used during our validation.

PART I

Background

Chapter 2: Role-based Access Control and SecureUML

In this chapter we give an overview of two technologies that mainly used to define security policies in this thesis: Role-based Access Control (RBAC) and security modelling language SecureUML.

2.1 Role-based Access Control (RBAC)

RBAC is a role and permission based architecture to assign rights to subjects (users, agents). In our project we will be using flat RBAC [31], which contains of roles, permissions, resources and users (subjects). The basic concept of RBAC is that users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of roles. One user can have many roles and single user can be assigned to multiple users. The rights to a user are defined by a session where the user is authenticated and then authorized based on roles to permissions. The permissions define what actions are allowed to be done with the resource [25].

The resource can be a document or piece of information that is allowed to be manipulated. The process of assigning policies to an authenticated subject is described in Figure 2. The subject is authenticated and is related through an active session to one-to-many roles which are related to permissions. The subject is assigned the permissions in the session and is authorized thorough permissions to do actions on resources.

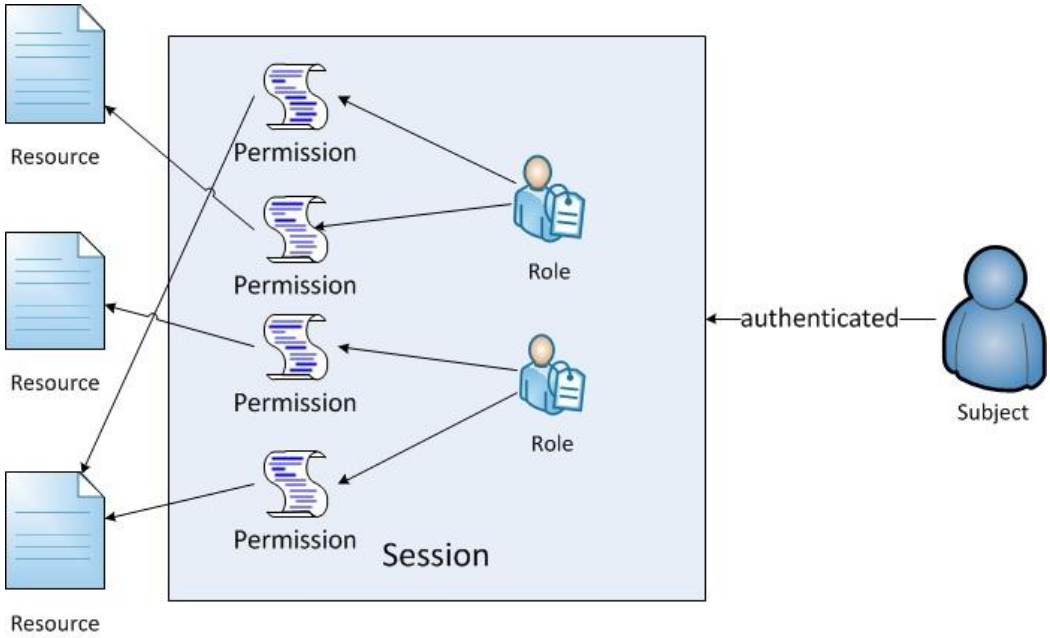


Figure 2 - Flat RBAC (adapted from [31])

2.2 SecureUML

In this section we will introduce SecureUML modelling language that we used to model RBAC on our resources.

SecureUML is a modelling language that defines a vocabulary for annotating UML-based models with information relevant to access control. It is based on the model for RBAC with additional support for specifying authorization constraints. SecureUML defines a vocabulary for expressing different aspects of access control, like roles, role permissions and user-role assignments. Due to its general access-control model and extensibility, SecureUML is well suited for business analysis as well as design models for different technologies [21].

The language contains of three main parts [25]: abstract syntax, concrete syntax and semantics. We will describe the main parts individually.

An **abstract syntax** of SecureUML is organized as a UML class diagram and displayed in Figure 3. It adapts the principles of the RBAC model, and introduces concepts like *User*, *Role*, and *Permission* as well as relationships *RoleAssignment* and *PermissionAssignment*. Here secured objects and operations are expressed through protected objects, which are modelled using the standard UML constructs (e.g., see concept of *ModelElement*). In addition, *ResourceSet* represents a user defined set of model elements used to in permissions and authorisation constraints [25].

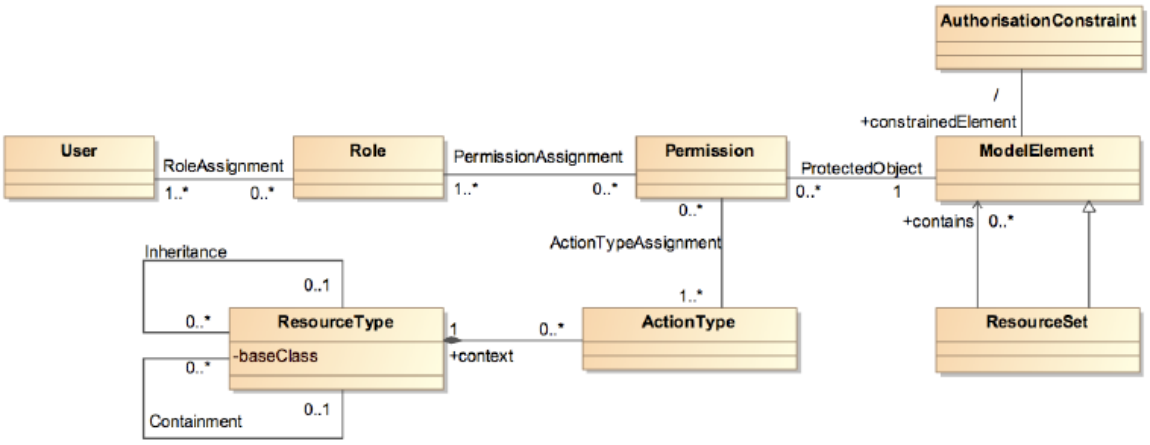


Figure 3 - SecureUML meta-model (adapted from [7,21,24])

The semantics of *Permission* is defined through *ActionType* elements used to classify permissions. Here every *ActionType* represents a class of security-relevant operations on a particular type of protected resource [25]. In Figure 4 we introduce four specific security actions: *Read*, *Write*, *Insert*, and *Delete*, which we will define later in Chapter 5. *ResourceType* define all possible actions for this type of resource. An *AuthorisationConstraint* expresses a precondition imposed to every call to an operation of a particular resource. This precondition usually depends on the dynamic state of the resource, the current call, or the environment. The authorization constraint is attached either directly or indirectly to a particular model element that represents a protected resource [25].

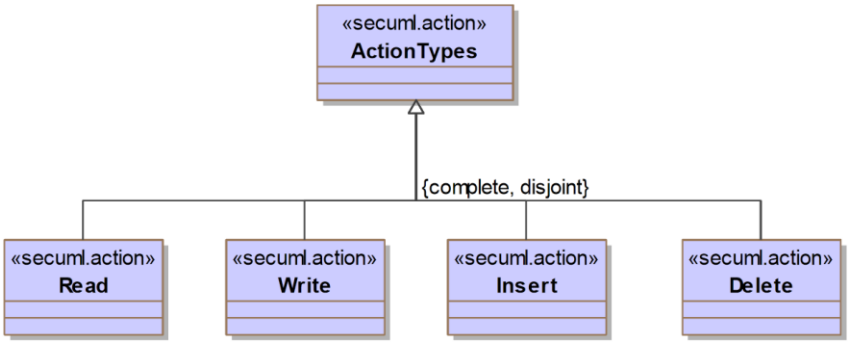


Figure 4 - Action types

At the **concrete syntax** level SecureUML is a "lightweight extension" of UML, namely through stereotypes, tagged values and constraints. The stereotypes are defined for the classes and relationships in the class diagrams and are specifically oriented to the RBAC terminology [25]. In Figure 5 we illustrate the SecureUML concrete syntax through the *Medical Records* permissions example.

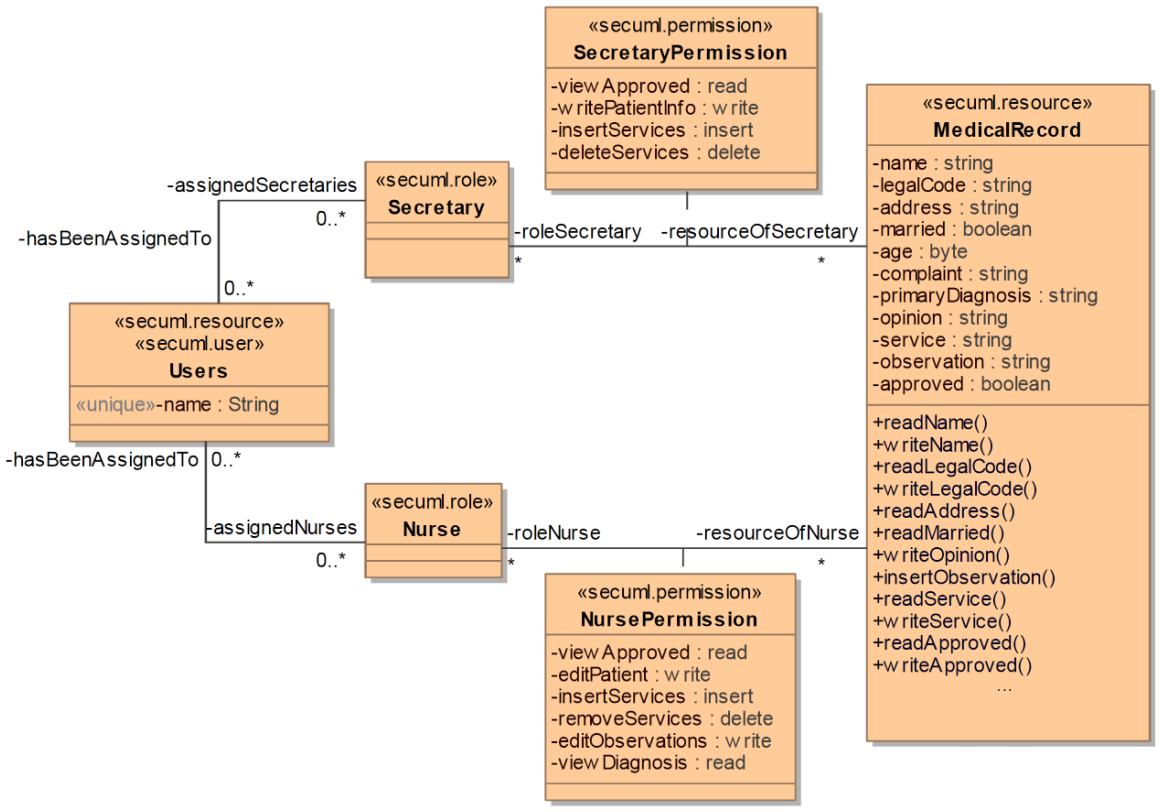


Figure 5 – *MedicalRecord* permissions with SecureUML

Figure 5 defines security for resource *MedicalRecord* which has eleven attibutes. Five of them describe the patient: *name, legalCode, aadress, married, age*. All these information units must be secured from unauthorized accesses.

We also have two roles: *Secretary* and *Nurse*. Their permissions to *MedicalRecord* resource are described through association classes *SecretaryPermission* and *NursePermission.*

To associate the permission attribute with the resource attributes we use authorization constraints, which are defined as abstract relations between them:

- *viewApproved -> approved*
- *writePatientInfo -> name*
- *writePatientInfo -> legalCode*
- *writePatientInfo -> address*
- *writePatientInfo -> married*
- *writePatientInfo -> age*
- *insertServices -> service*
- *deleteServices -> service*

Based on *SecretaryPermission* class the role *Secretary* permissions:

- action *viewApproved* (of type *Read*) defines that *Secretary* can read resource field *approved* (see class *MedicalRecord*);
- action writePatientInfo (of type *Write*) allows changing name, legalCode, address, married and age (see class MedicalRecord);
- action *insertServices* (of type *Insert*) allows adding new elements with tag name *service* (see class *MedicalRecord*);
- action *deleteServices* (of type *Delete*) allows to remove elements with tag name *service* (see class *MedicalRecord*).

In [8] **semantics** of Secure UML is formalised to satisfy two purposes: first to define a declarative access control decisions that depend on static information, namely the assignments of users and permissions to roles, and secondly to support implementation-based access control decisions that depend on dynamic information, namely, the satisfaction of authorisation constraints in the current system state [25]. Similarly to [2, 15] we discuss SecureUML semantics in respect to system modelling. We make use of the RBAC model to define semantics of the SecureUML constructs [23, 24] as illustrated in Table 1. Some mappings between RBAC and SecureUML are understood as a lexical correspondence. For example, the SecureUML classes with the stereotype <<*securml.user*>> correspond to the RBAC *users*, <<*securml.role*>> to the RBAC *roles*, and <*securml.permission*>> to the RBAC *permissions*. These SecureUML constructs and RBAC concepts are similar according to their textual expression, and also their semantic application (see *MedicalRecord* in Figure 5).

Subjects to be protected are defined as classes with stereotype<<*securml.resource*>>. In RBAC notation they are called as *objects.* Operations that can be applied on the objects that are defined as <<*securml.resource*>> class operations in SecureUML and as operations in RBAC. For RBAC *role assignment* we use association class between <<securml.role>> (*i.e. Secretary)* and <<securml.resource>> (*i.e. MedicalRecord* class). The permission assignment in SecureUML is described with association classes which defines actions that can be used on the associated resource elements.

Table 1 - Correspondence between RBAC concepts and secureUML constucts (based on [25])

RBAC concepts	**SecureUML contruct**	**MedicalRecord example**
Users (concept)	Class stereotype <<*secuml.user*>>	Class *Users*
Users assignment (relationship)	Association between classes with stereotypes <<*secuml.user*>> and <<*secuml.role*>>	Association relatsionship [hasBeenAssignedTo-assignedSecretary] and [hasBeenAssignedTo-assignedNurse]
Roles (concept)	Class stereotype <<*secuml.role*>>	Classes *Secretary* and *Nurse*
Permission assignment (relationship)	Associations class stereotype <<*secuml.permission*>>	Operations of associated classes *InitiatorPermissions* and *ParticipantPermissions*
Objects (concept)	Class stereotype <<*secuml.resource*>>	Class *MedicalRecord*
Operations (concept)	Operations of a class with stereotype <<*secuml.resource*>>	Operations *readName(), writeName(), readLegalCode(), writeLegalCode(), readAddress(), readMarried*, etc.
Permission (concept)	Authorization constraint	Abstract relations: *viewApproved -> approved* *writePatientInfo -> name* *writePatientInfo -> legalCode* *writePatientInfo -> address* *writePatientInfo -> married* *writePatientInfo -> age* *insertServices -> service* *deleteServices -> service*

2.3 Summary

In this chapter we introduced Role-based Access Control model and SecureUML modelling language. We discussed the associations between RBAC and SecureUML and defined the semantics to model RBAC. In the next chapter we will describe the existing tools we used to support our work.

Chapter 3: Background

In this chapter we define the main concepts and technologies that are used in the thesis. We will first cover the main technologies and then in the next chapter define how we use the technologies to support our work.

3.1 Extensible Mark-up Language

More and more data is stored today which leads to the need to store and transfer the data structurally. One possible format to use when transporting and storing data and documents is Extensible Mark-up Language (XML) [17]. XML is often used to transfer data to visible layer or between parties. In our work we use XML to store and transfer our data. We choose XML as it is understandable for humans and machines, and is platform independent.

XML was developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996 [17]. A sample XML document is given as Figure 6. The main line of the XML defines the XML version to be used and the encoding the proceeding document is given in. We have main node *file* which contains two main child elements *header* and *content*, which both also have sub-elements. We can see that the data type in the fields is different: field *created* is containing *date* format content, *createdBy* contains data of type *string* and *isAccepted* fields contains a boolean value. To define the rules which kind of data can be inserted to a field we use XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<files xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="XSD_example.xsd">
        <document>
                <header>
                        <created>2012-02-10</created>
                        <createdBy>John, Smith</createdBy>
                        <documentVersion>2</documentVersion>
                </header>
                <content>
                        <title>XML document's structure</title>
                        <paragraph>XML document's structure contains of tag's which can be
easily defined based on need</paragraph>
                        <isAccepted value="true"/>
                </content>
        </document>
</files>
```

Figure 6 – XML document

3.2 XML Schema

The structure of an XML document can be defined as a Document Type Definition (DTD) or XML Schema [35]. DTD contains info about the structure of the XML, but the XML Schema can contain info about the elements multiplicity, availability and type. The main limitation of DTD is that it is not itself a well-form and valid XML document. An additional special parser is needed when processing DTD and XML [32]. XML Schema

itself is a document that has needs to have a certain structure to be valid. For parsing XML Schema regular XML parsers can be used. For example XPath [36] or XQuerty [37]. XML Schema was published as a W3C recommendation in May 2001 [35]. It defines the structure and the rules which the respected XML document has to fill in order to be considered to be valid. It helps to validate the document before processing to ensure that it contains all needed elements.

An example XML Schema that validates our previously presented XML is given in Figure 7. The structure of the file is presented with the help of Altova XMLSpy [1] to define the schema. We can see that the document must contain of one element named *files* and then one to many elements of *documents*. The documents must contain of *header* and *content*.

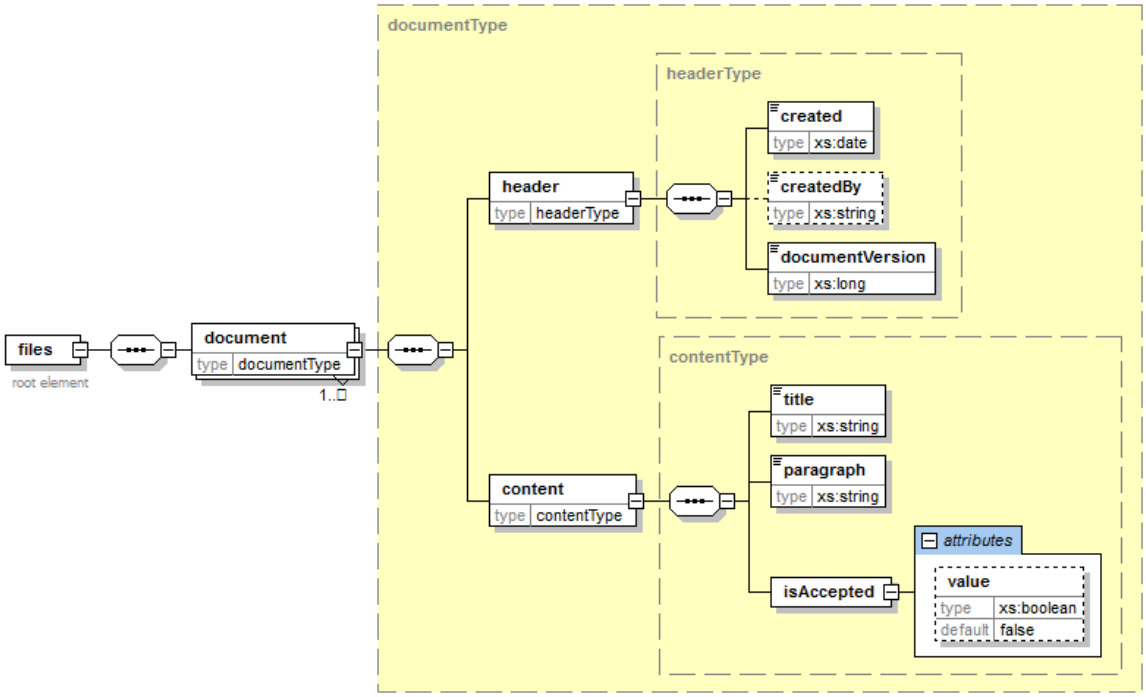


Figure 7 - XML Schema structure

If we look at the text definition (Figure 8) we can see the data types of each value and that *content*, *header* and *document* are defined as separate types. *HeaderType* contains elements *created*, *createdBy* and *documentVersion*. Element *created* is of type *xs:date* (where *xs* is the namespace) and is mandatory (*minOccurs* attribute not defined); *createdBy* is of type xs:string and is not mandatory; *documentVersion* is of type *xs:long* and is mandatory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2012 rel. 2 sp1 (x64) (http://www.altova.com) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
	<xs:element name="files">
		<xs:annotation>
			<xs:documentation>root element</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:sequence>
				<xs:element name="document" type="documentType"
maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:complexType name="contentType">
		<xs:sequence>
			<xs:element name="title" type="xs:string"/>
			<xs:element name="paragraph" type="xs:string"/>
			<xs:element name="isAccepted">
				<xs:complexType>
					<xs:attribute name="value" type="xs:boolean" default="false"/>
				</xs:complexType>
			</xs:element>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="headerType">
		<xs:sequence>
			<xs:element name="created" type="xs:date"/>
			<xs:element name="createdBy" type="xs:string" minOccurs="0"/>
			<xs:element name="documentVersion" type="xs:long"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="documentType">
		<xs:sequence>
			<xs:element name="header" type="headerType"/>
			<xs:element name="content" type="contentType"/>
		</xs:sequence>
	</xs:complexType>
</xs:schema>
```

Content type

Header type

Document type

Figure 8 - XML Schema text definition

3.3 DynaForm

Dynaform is a Dynamic Schema-Based Web Forms Generation in Java developed by Raudjärv [29]. It uses XML Schema to dynamically define forms and also uses Domain-Specific Language (DSL) [14] to specify the restrictions for the form, *i.e.* the length of a value. The form works by XML Schema, DynaData (DSL) and the form data. DynaForm Web form building is based on XForms [33] and presented with the help of Aranea framework and Java Servlets. We use the XML Schema based form builder as out central tool to display the form.

Figure 9 illustrates the inputs and outputs of DynaForm. XML schema is set as input based on what the web structure will be built. A presentation specific definition DynaData can be used to customize rendering the form. XML Instance can be used to pre-populate values to the form.

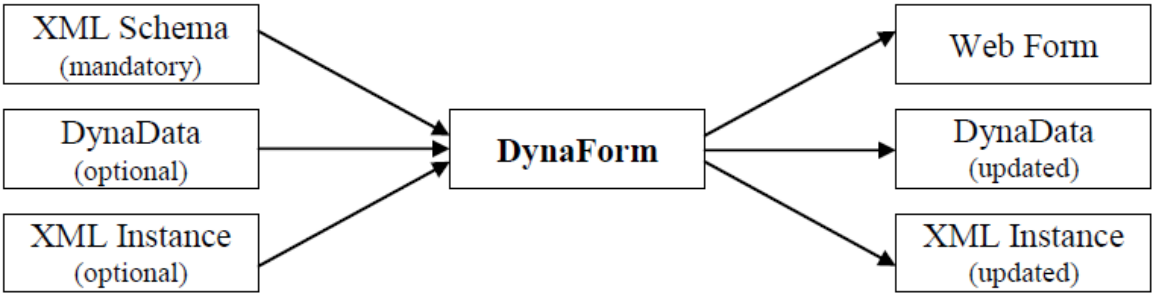


Figure 9 - DynaForm input and output (adapted from [29])

The DynaForm supports a limited number of XML Schema elements. They have chosen to focus on representative subset of XML Schema definition elements [29].

The supported subset of XML Schema should fulfil the following requirements [29]:

- The set of XML Schema elements allowed in the input of DynaForm are: <*all*>, <*attribute*>, <*choice*>, <*element*>, <*sequence*>.
- Reoccurring XML Schema elements (*minOccurs* and *maxOccurs* attributes) are also allowed.
- The supported XML Schema simple data types are:

    1. String data types: *string*.
    2. Date data types: *date, time, datetime.*
    3. Numeric data types: *integer, long, int, short, byte, decimal*.
    4. Miscellaneous data types: *boolean*.

DynaForms is the central presentation technology that we use to display forms.

3.4 Other tools

Velocity [5] is a Java-based template engine. It permits anyone to use a simple yet powerful template language to reference objects defined in Java code. The template defines the structure and the layout of the document with placeholders for Java objects. Placeholders will be defined from Java object and when processing the template Velocity will replace the placeholders with values from Java objects.

Velocity's capabilities reach well beyond the realm of the web; for example, it can be used to generate SQL and XML from templates. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems [5]. We use Velocity to generate SQL and XML Schemas from class diagrams.

MagicDraw [22] is business process, architecture, software and system modelling tool with teamwork support. Designed for Business Analysts, Software Analysts, Programmers, QA Engineers, and Documentation Writers, this dynamic and versatile development tool facilitates analysis and design of Object Oriented systems and databases. We use MagicDraw as our modelling tool.

ECLA XmlMerge [16] module is implemented to merge two or more XML documents together. We use the tool to merge two document versions into one without losing the data that was not displayed to the user.

Apache Tomcat is an open source software implementation of the Java Servlet and Java Server Pages technologies [4]. The servers are widely used on small and enterprise solutions. For out project we use Tomcat on both: client and server side. Version 6.0 is used on client side and latest stable version 7.0 on server side.

The **Apache Axis2** [3] project is a Java-based implementation of both the client and server sides of the Web services equation. The main reason for using the tool is that it supports sending Simple Object Access Protocol (SOAP) messages, receiving and processing SOAP messages, creating Web services out a Java class, creating client and server side Web Service Definition Language (WSDL) implementations dynamically. It is used to transfer the documents and document requests between client and server.

MySQL [27] is an open source relational database management system. It is available under the General Public License and is supported by a huge and active community of open source developers.

3.5 Summary

In this chapter we introduced main existing technologies and concepts used in our work: XML, XML Schema, DynaForm, Velocity, MagicDraw, ELCA XmlMerge, Apache Tomcat,Apache Axis2 and MySQL We describe the technology usage in the next chapter.

PART II

Contribution

Chapter 4: Architecture

In this chapter we introduce the architecture of our solution. First we discuss the high level architecture and then specify the solution in server and client side and integration separately. Additionally we go through the XML documents transformation and merging strategies.

4.1 Technologies integration

Figure 10 introduces how the used technologies are related to each other. The server machine contains info about the forms and forms data. The client displays the form using DynaForm and sends the saved data to the server. In the server machine the XML document and XML Schema RBAC transformations are performed based on the role and requested form from the client side. The client and the server are treated as trusted parties. The connection between these two parties is not encrypted and is supported by Axis2. The client machine does not store any form-related info or form responses.

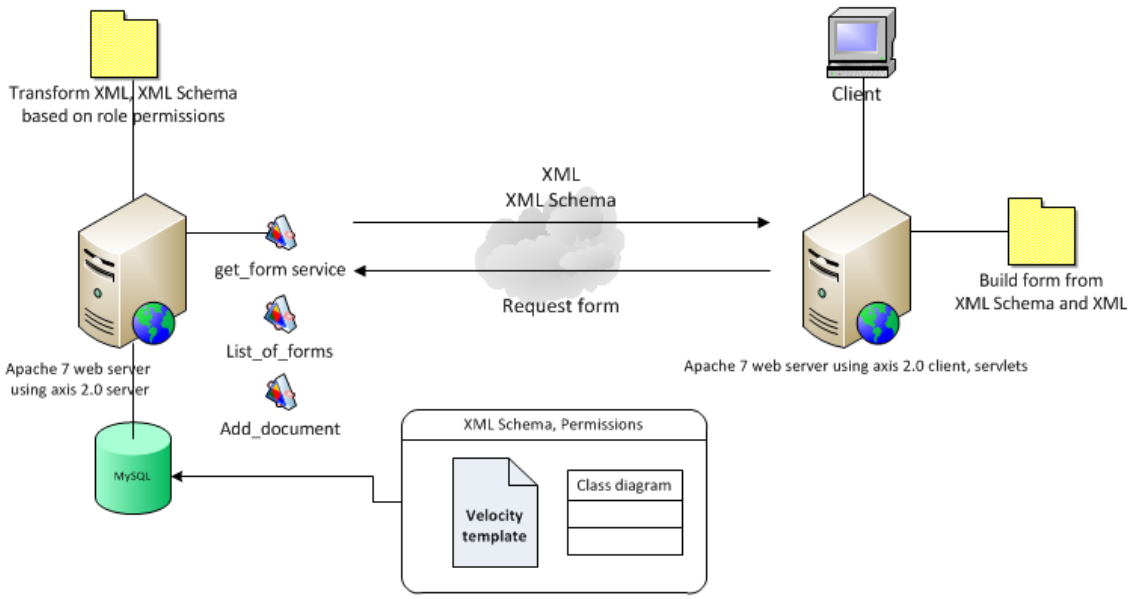


Figure 10 – Architecture solution

The **client side** contains of three parts: a Web Server (Apache Tomcat 6.0), DynaForm and web services clients (Axis2 clients). The client side is responsible for displaying the data got from server and sending the changed data back to the server.

DynaForm is used to display the form to the client, to collect values from the forms and send them to the server. The client server displays the requested form and validates it against XML Schema (with DynaForm) rules and if client side rules are accepted, the result is sent to the server. We have implemented the DynaForm not to save any data locally, but send all the data to integration services which sends the data to the server.

The **server side** is responsible for storing the data, permission, form definitions, transforming the documents and XML Schemas based on the role permissions and sending

it to the client. When modifying the document the server is also responsible for merging the existing document with the modified one. As suggested in [12] the server side is responsible for processing the permissions. This avoids any security risks in the client side.

The server has four main services that the client side uses:

getForms(roles) – returns the list of documents available for the roles with the documents permissions. The data is used to display an existing XML document to the user.
getXsdForms(roles) – returns the list of possible new documents to the user with XML Schema level permissions. The returned data is used to create a new document for the user.
updateForm(XML, user, XML Schema) – is a service that is used to update an existing document in the system.
addNewForm(XML, user, XML Schema) – a service that is used to add a new document to the system.
Server runs on Apache Tomcat 7 server which contains Axis2 integration library.

The **integration between server and client** is done through integration messages and with the help of Axis2 integration engine. The messages are sent synchronously, meaning each message will be waiting for a response [6].

The security between parties is not in the scope and the integration partners are treated as being trusted.

4.2 Document transformation and transportation

In this paragraph we describe the process to securely provide XML document to the end-user. After introducing the main process we describe each process step in detail.

We implement RBAC on XML documents based on document's structure. The XML documents can be big in size, so we will not implement RBAC on XML level as this might cause poor performance due to the vast amount of data to be processed. We define the rules on XML Schema level and calculate the XML for each role group beforehand, as the number or different rights combinations can be large. We calculate it before the document is requested from the server.

Figure 11 displays the document and its associated resources. Every XML document in the server side has a XML Schema assigned to it, which refer to the structure of the document and will be used to display the form in the client side. Every XML Schema can also have assigned a list of permissions which limits the actions that can be done with the document by subjects with different roles.

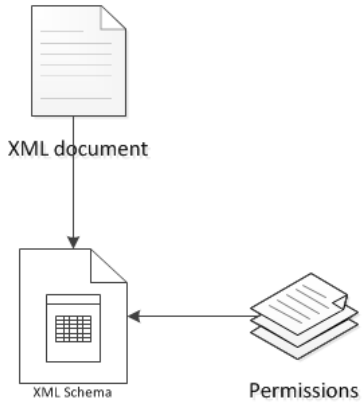


Figure 11 - XML with XML schema and Permissions

If a request from client side is initiated, then the role of active user is sent to the server. The process is described in Figure 12. After the server receives the request for a document, the server checks if there are any permission restrictions for the associated document. If exists then it will transform based on the permissions the XML and XML Schema and return them to the client. The client will display the form based on XML Schema and fill it with data from XML.

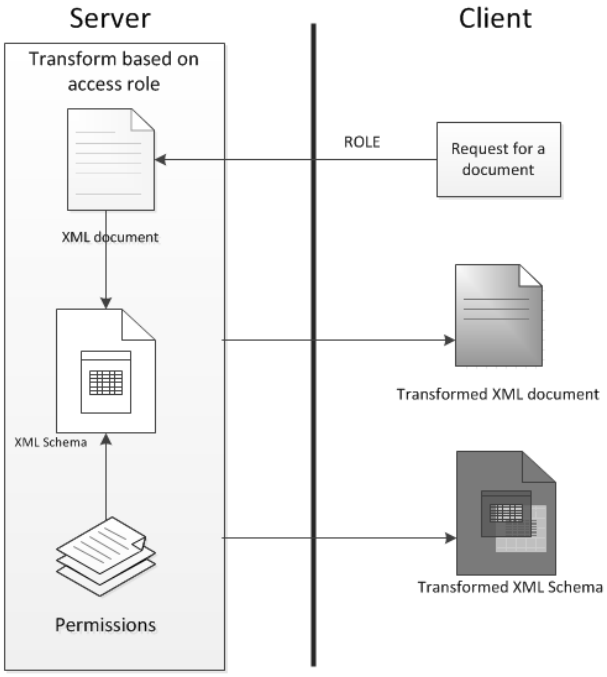


Figure 12 - Request a Document

When the client saves the document it will be sent back to the server and be merged together with the original XML document (Figure 13). The merging strategies are discussed in more detail in Chapter 5.

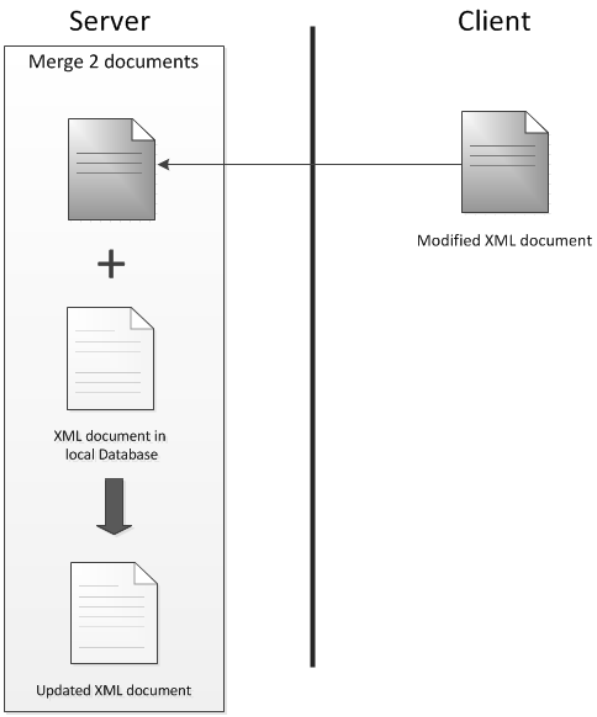


Figure 13 - Save a document and merge in server

4.3 Discussion

There exist architecture solutions in securing documents. In [32] Sandhu suggests having a separate security administration part which contains of an authorization server that is connected to role server and policy server. A server would be responsible for authorization and the requested XML transportation. In our case the authorization and session handling is in the client side and for simplicity we transfer only role level information to the XML Schema transformation server. There also exists work related to XML security. Organization for the Advancement of Structured Information Standards (OASIS) has developed Security Assertion Markup Language (SAML) [30] in XML to transfer security information: permissions, authorizations and authentications, but does not restrict directly XML data. Extensible Access Control Markup Language [28] (XACML) provides a method to define access controls based on sets of policies. XACML is similar to our work to define rules on XML elements level. The difference to XACML is that we use a specific control mode – RBAC to secure our resources as in [32]. XACML has no specific access control method model supported. Work in [12] is similar to ours by architecture: Samarati suggest having all permission calculations in server side and giving out only information for the server that the users are authorized to, but they define rules on DTD, not on XML Schema.

4.4 Summary

In this chapter we described the architecture of our solution from client side, server side and from integration. We introduced the documents transformation and merging strategy from the architecture perspective. Also, we pointed out architecture related work in XML security. In the next chapter we will describe how RBAC is applied on XML Schema.

Chapter 5: RBAC on XML Schema based forms

In this chapter we will describe how RBAC is applied on XML Schema. Firstly, we will define permissions using defined Domain Specific Language [14], secondly introduce possible actions, and thirdly discuss XML and XML Schema transformation rules together with document merging strategies and finally we will illustrate it with an example.

5.1 Defining RBAC on elements

To secure the XML documents RBAC is implemented in the XML Schema level. Permissions in an XML Schema document can be defined in the level of elements which do not have any child elements themselves. Figure 14 displays XML Schema for a document with 11 elements which do not have any child elements: *name, legalCode, address, married, age, complaint, primaryDiagnosis, opinion, service, observation* and *approved*. These are also elements which are displayed on the input form.

To define the role based access control for each of these elements we define a new Domain Specific Language (DSL) [14].

{Role}<>{element}>>{permissions}<break>

Each user can be assigned to one role and the role is provided by the client to the server. The *{Role}* field is case sensitive and has to match in both parties. Server checks for the access rights requested to the document. The *{element}* field name has to match the local name of the field: in the XML Schema the element attribute property *name* without the namespace. For example in Figure 14 *name*, *legalCode*, *etc*. should be used. For *{permissions}* we can define four actions: *Read*, *Write*, *Insert* and *Delete*. In the notation we use them as *"R,W,I,D"*. This kind of notation means that all these permissions are granted. Notation *"R,W,-,D"* means that it is allowed to read, write and delete. <*break*> is used to distinguish permission granting rights from each other.

One permission grant to read and write *address* field can be defined as: *Admin<>address>>R,W,-,-<break>*.

5.2 Action definitions

Role can have four different permissions on an element:

- *Read* (*R*) – element value is visible.
- *Write* (*W*) – If the element is visible then it is allowed to change the value.
- *Insert* (*I*) – If multiple sections allowed, then allows adding another section.
- *Delete* (*D*) – Allowed to remove elements more frequent in data XML, than in the XML Schema definition.

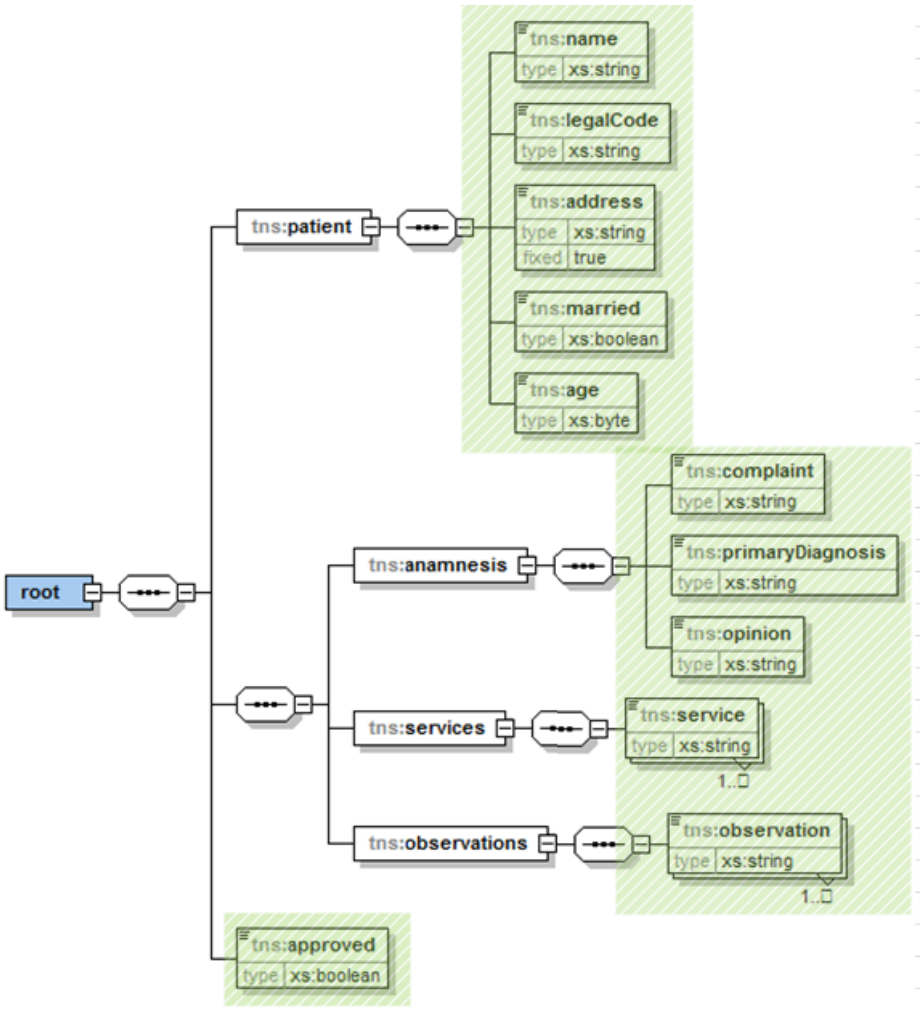


Figure 14 - Permissions on element level

Write permission is a prerequisite order to be able to add a new section. *Read* permission is a prerequisite in order to be able to write, insert and delete the element. Table 2 describes the permissions that can be assigned to manipulate an element. It is also illustrated with examples for XML Schema and XML. The transformations are done based on four actions.

Read permission is defined by attribute *fixed*. It specifies that the form element is in read-only mode. The attribute with value *true* is added only if the actor does not have additionally writing permission and the field is of type *boolean* or *string.* Value *0* is added if the field is of numeric type.

When *Write* permission is granted, then element is displayed as editable.

Insert permission controls the maximum number of occurrences that element can be in the document. Inserting is possible if the *maxOccurs* attribute's value in XML Schema is bigger than the number of elements occurrences in the document. Inserting right gives the permission to add the elements as many times as it is in the XML Schema definition. If the user does not have inserting permission then the value of attribute *maxOccurs* is calculated based on the number of times the element is used in the XML document. *I.e If the address element exists 3 times in the XML and in the form schema it is allowed to define 5 times. Then in the transformed XML Schema the maxOccurs attribute value is set to 3.*

Delete permission controls if it is allowed to delete the existing element. Deleting is possible if the attribute *minOccurs* value in XML Schema is smaller than occurrence of existing elements in the XML. Deleting right gives the permission to remove elements from the XML which are more frequent in the XML than permitted in the XML Schema. If the user does not have deleting right then the value of *minOccurs* is calculated based on the occurrences of the element in the XML. *I.e If the address element exists 3 times in the XML and in the form schema it is required to define at least once, then XML Schema minOccurs attribute value is set to 3.*

Table 2 - Permission descriptions

Permission	XML Schema	XML
READ	<xs:element name="address" fixed="true" type="xs:string"/>	<address>String</address>
WRITE	<xs:element name="address" type="xs:string"/>	<address>String</address>
INSERT	<xs:element name="address" type="xs:string" maxOccurs="2"/>	<address>String</address> <address>String</address>
DELETE	<xs:element name="address" type="xs:string" maxOccurs="3"/>	<address>String</address> <address>String</address> <address>String</address>

5.3 XML and XML Schema transformations

In this paragraph we will describe the process in order to transform the XML document and XML Schema respect to the permissions. The process is described as a Flowchart (legend defined in [11]) in Figure 15. The process can be separated into three main steps: transforming *minOccurs* and *maxOccurs* attributes, fixing read-only fields, removing fields from XML and XML Schema that the role does not have access *Read*. The first two steps are manipulating only XML Schema. The third one is manipulating also XML data.

First, we find all the elements in the XML Schema that have *minOccurs* or *maxOccurs* attributes. For each this element we also find how many times the element occurs in the XML data. After finding all the elements we check if the role has *Delete* permission. If it does we set the element attribute *minOccurs* value equal XML Schema attribute value (value in the original XML Schema). If the role does not have the *Delete* permission, we will set the attribute value to the number of times the element existed in the original XML document. Next we check if the role has *Insert* permission to the element. If it does, then we set the attribute *maxOccurs* to the XML Schema value (value in the original XML Schema). If the role does not have the permission, we will set the attribute value to the number of times the element existed in the original XML document. After finishing the process for one element we will check if there exist more multi-occurrence elements. If it did then we check the values again for the new element. If not, we will proceed.

After multi-occurrence elements have been all checked we will continue to find elements which the *role* has *Read* permission, but does not have *Write* permission. For all these

elements we will add an attribute *fixed = "true"* to the XML Schema. It states that the value is fixed and cannot be changed – it is read-only.

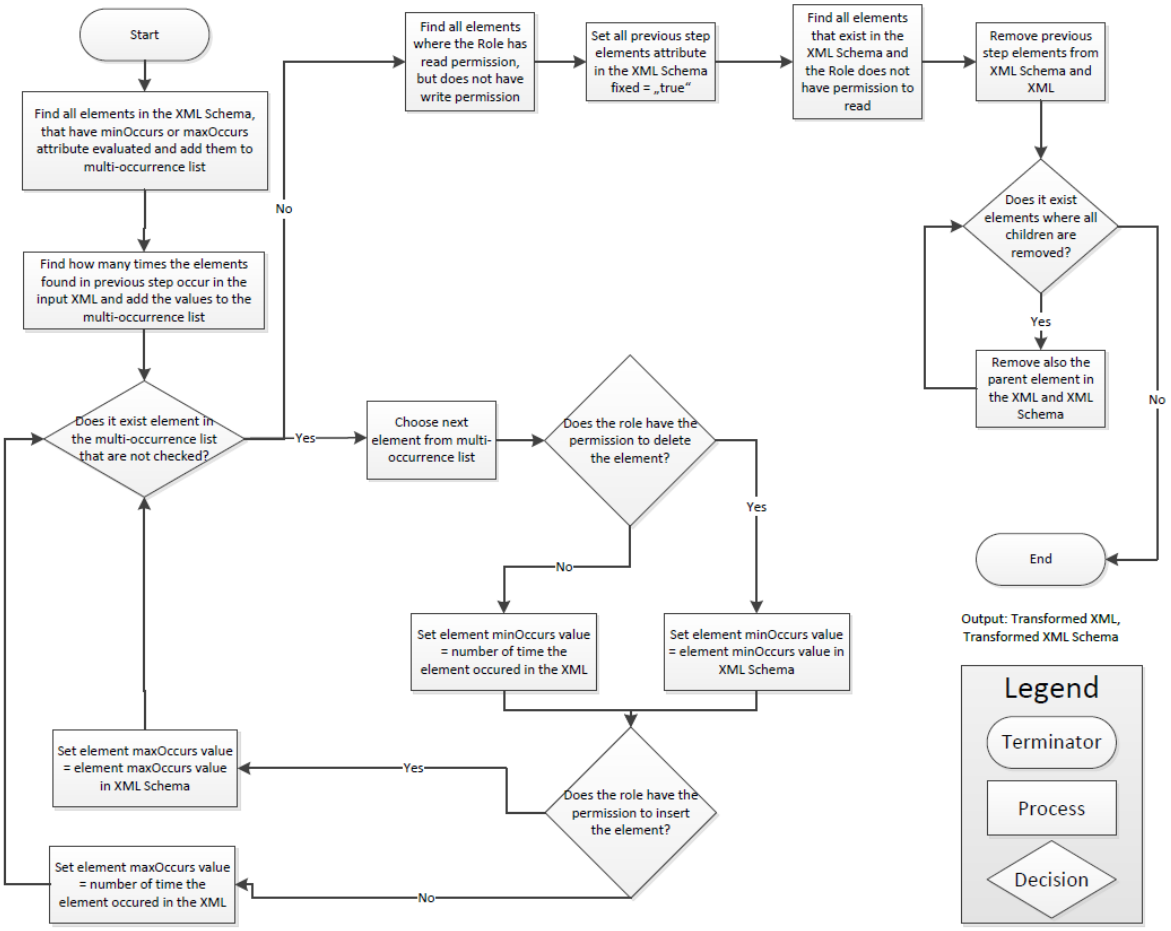


Figure 15 - XML and XML Schema transformation flowchart

After fixing read-only fields we will continue with finding elements that the *role* does not have *Read* permission. These elements should not be visible to the *role* and will be removed from the XML Schema and XML data. After removing the elements we might come to a point where the parent element does not have any more child elements (all child elements are removed), then we also remove the parent element in order to keep the XML Schema valid. We also remove the elements from XML data to keep it valid respect to XML Schema.

In this paragraph we described the process of permissions structure, possible actions and transformations to the XML Schema and XML data. After all the transformation process is finished the transformed XML Schema and transformed XML are sent to the client.

5.4 XML merging rules

After the transformed XML Schema and XML data is received in the client side the user can modify the existing data and send it back to the server for storage.

When a XML document is saved, then the document can be limited based on permissions (all element that the role does not have permission to read are removed). In order to keep the document complete we merge the documents in the server side. We use the original XML and the new saved document received from the client and merges them with ELCA module XmlMerge [16] with merging strategy *replace.*

Replace strategy means that it will replace the original with the saved XML element or create a new element if it does not exist in the original XML. This solution has also one limitation: when the element is not under main element (*root* element in the examples) and is not in the saved XML, the value will be removed during merging the XML (Table 3 row 4). Each time we save a document we take the original XML and merge it with saved XML from client. Examples are given in Table 3.

Table 3 - XML merging examples

Original XML	Saved XML	Merged XML	Comment
<root> <a>Value </root>	<root> <a/> Value </root>	<root> <a/> Value </root>	Element values are overridden.
<root> <a/> Value </root>	<root> <a/> <a>Value </root>	<root> <a/> <a>Value </root>	If an element exists in saved XML, then value is replaced or deleted if the element is not present.
<root> <a/> Value </root>	<root> <a/> <a>Value </root>	<root> <a/> <a>Value Value </root>	If an element does not exist in saved XML, then it is not deleted from the original XML document if it is under main element.
<root> <a> Value <c>Value</c> </root>	<root> <a> <c>Value</c> </root>	<root> <a> <c>Value</c> </root>	If an element does not exist in saved XML, then it is deleted from the original XML document if it not under main element.

5.5 An example of XML and XML Schema transformation

In the client side we always display all the info we get from the server, all the transformation of XML document and XML Schema are done in the server side. As defined previously we have four main actions *Read*, *Write*, *Insert* and *Delete*. We will describe the action transformations of XML and XML Schema based on an example.

Lets' assume we have a simple XML Schema described in Figure 17; a role *Secretary* who can read and write fields: *name, legalCode, address, married, age*. Read, write, insert and delete field *service* and only read fields *observation* and *approved.* The definition of role *Secretary* permissions in our MySQL [27] database are given in Table 4. There also exists a XML document for that schema. Example document XML is described in Figure 16.

Table 4 - Permissions for role Secretary

Secretary<>name>>R,W,-,-<break> Secretary<>legalCode>>R,W,-,-<break> Secretary<>address>>R,W,-,-<break> Secretary<>married>>R,W,-,-<break>	Secretary<>age>>R,W,-,-<break> Secretary<>service>>R,W,I,D<break> Secretary<>observation>>R,-,-,-<break> Secretary<>approved>>R,-,-,-<break>

We will first find all the elements that in XML Schema have an attribute *minOccurs* or *maxOccurs* describe the multiplicity of an element. From Figure 17 we can see that these elements are *service* and *observation.* Element *service* occurs in the XML document three times and *observation* two times. The occurrences are taken into account when *Insert* and *Delete* rights are calculated. For field *Service* the actor can *Insert* and *Delete*, therefore we do not change the XML Schema definition. For field *observation* the user does not have *Insert* nor *Delete* permissions. Therefore the maximum number of allowed observations must equal the number of elements existing in the XML and the minimum number of allowed observations must also equal the number of elements existing in the XML - two.

<xs:element name="service" type="xs:string" maxOccurs="unbounded"/>
<xs:element name="observation" type="xs:string" minOccurs="2" maxOccurs="2"/>

Next we will look for elements where the role has *Read* permission, but does not have *Write* permission. These elements are displayed as read only. These elements are *observation* and *approved*. For these elements we will add to the XML Schema attribute *fixed* with value *"true"* if it is of type string or boolean and *"0"* if it is numeric.

<xs:element name="observation" type="xs:string" minOccurs="2" maxOccurs="2" fixed="true"/>
<xs:element name="approved" type="xs:boolean" fixed="true"/>

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <root>
3      <patient>
4          <name>Smith, Johan</name>
5          <legalCode>322120102</legalCode>
6          <address>234c Baker St.</address>
7          <married>true</married>
8          <age>55</age>
9      </patient>
10     <anamnesis>
11         <complaint> Head ache</complaint>
12         <primaryDiagnosis>K22.1 Consistent head-ache</primaryDiagnosis>
13         <optionion/>
14     </anamnesis>
15     <services>
16         <service>Specialist appointment</service>
17         <service>Observation: Cholesterol, Lipid </service>
18         <service>Observation: Blood glucose </service>
19     </services>
20     <observations>
21         <observation>Cholesterol and Lipid Levels: OK</observation>
22         <observation>Diabetes (blood glucose): too low </observation>
23     </observations>
24     <approved>true</approved>
25 </root>
```

Figure 16 – Main XML Document

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.example.org/simple1"
   targetNamespace="http://www.example.org/simple1" elementFormDefault="qualified">
3    <xs:element name="root">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="patient">
7            <xs:complexType>
8              <xs:sequence>
9                <xs:element name="name" type="xs:string"/>
10               <xs:element name="legalCode" type="xs:string"/>
11               <xs:element name="address" type="xs:string" fixed="true"/>
12               <xs:element name="married" type="xs:boolean"/>
13               <xs:element name="age" type="xs:byte"/>
14             </xs:sequence>
15           </xs:complexType>
16         </xs:element>
17         <xs:sequence>
18           <xs:element name="anamnesis">
19             <xs:complexType>
20               <xs:sequence>
21                 <xs:element name="complaint" type="xs:string"/>
22                 <xs:element name="primaryDiagnosis" type="xs:string"/>
23                 <xs:element name="opinion" type="xs:string"/>
24               </xs:sequence>
25             </xs:complexType>
26           </xs:element>
27           <xs:element name="services">
28             <xs:complexType>
29               <xs:sequence>
30                 <xs:element name="service" type="xs:string" maxOccurs="unbounded"/>
31               </xs:sequence>
32             </xs:complexType>
33           </xs:element>
34           <xs:element name="observations">
35             <xs:complexType>
36               <xs:sequence>
37                 <xs:element name="observation" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
38               </xs:sequence>
39             </xs:complexType>
40           </xs:element>
41         </xs:sequence>
42         <xs:element name="approved" type="xs:boolean"/>
43       </xs:sequence>
44     </xs:complexType>
45   </xs:element>
46 </xs:schema>
```

Figure 17 – Main XML Document Schema

As the last transformation we find all elements where the role does not have *Read* permission. These elements will be removed from the XML Schema and also from the XML document. These elements are *complaint*, *primaryDiagnosis* and *opinion*. As by removing all these fields the element *anamnesis* would be empty in the XML Schema, we also remove the parent element (Lines 18-26 in Figure 17 are removed). After removing the elements that the role cannot read, we have XML Schema as defined in Figure 18. We also remove all XML document elements where the user does not have *Read* permission. As the XML Schema the parent element was empty after removing the elements we also removed the parent element *anamnesis* from the XML data. The final XML document is displayed as Figure 19.

```
5   ...
6       <xs:element name="patient">
7         <xs:complexType>
8           <xs:sequence>
9             <xs:element name="name" type="xs:string"/>
10            <xs:element name="legalCode" type="xs:string"/>
11            <xs:element name="address" type="xs:string"/>
12            <xs:element name="married" type="xs:boolean"/>
13            <xs:element name="age" type="xs:byte"/>
14          </xs:sequence>
15        </xs:complexType>
16      </xs:element>
17      <xs:sequence>
18        <xs:element name="services">
19          <xs:complexType>
20            <xs:sequence>
21              <xs:element name="service" type="xs:string" maxOccurs="unbounded"/>
22            </xs:sequence>
23          </xs:complexType>
24        </xs:element>
25        <xs:element name="observations">
26          <xs:complexType>
27            <xs:sequence>
28              <xs:element name="observation" type="xs:string" maxOccurs="unbounded"/>
29            </xs:sequence>
30          </xs:complexType>
31        </xs:element>
32      </xs:sequence>
33      <xs:element name="approved" type="xs:boolean"/>
34    </xs:sequence>
35  ...
```

Figure 18 – Transformed XML Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <root>
3    <patient>
4      <name>Smith, Johan</name>
5      <legalCode>322120102</legalCode>
6      <address>234c Baker St.</address>
7      <married>true</married>
8      <age>55</age>
9    </patient>
10   <services>
11     <service>Specialist appointment</service>
12     <service>Observation: Cholesterol, Lipid </service>
13     <service>Observation: Blood glucose </service>
14   </services>
15   <observations>
16     <observation>Cholesterol and Lipid Levels: OK</observation>
17     <observation>Diabetes (blood glucose): too low </observation>
18   </observations>
19   <approved>true</approved>
20 </root>
```

Figure 19 - Transformed XML document

After all the transformations are finished the transformed XML Schema and transformed XML are sent to the client.

The existing form based on the transformed XML Schema and XML document is displayed in Figure 20. Elements *observation* and *approved* are in read-only mode, and elements *complaint*, *primaryDiagnosis* and *opinion* are not displayed.

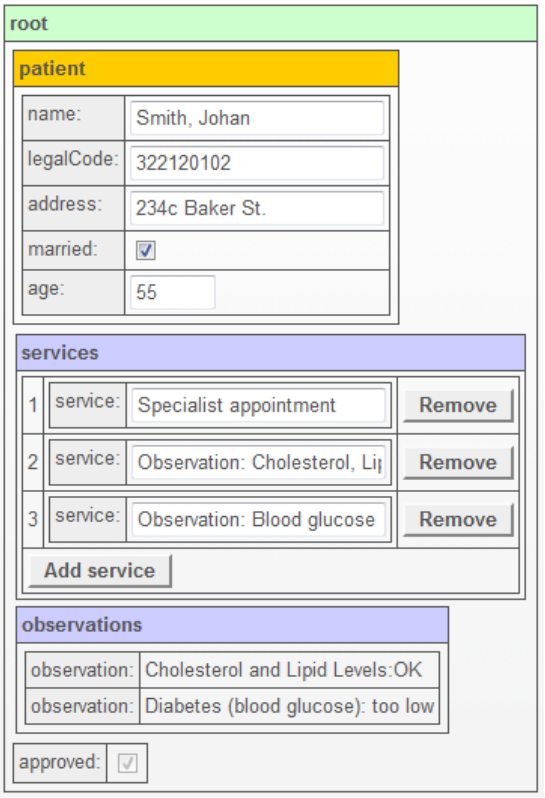


Figure 20 - Transformed XSD and XML based form

The user can add and change *name*, *legalCode*, *address*, *married*, age and can add or delete *service* elements. For example user changes field *legalCode* value to "*112233"* and adds a new *service* "20x25mg paracetamol". After saving we merge the two documents in order to keep the info that was not visible to the end user due to role permissions in the document and add the changes and additional elements that were changed by the user. The documents will be merged in the server to a new document version.

In Figure 21 the existing document in the server and the document sent by the client are displayed. The sent document is missing the *anamnesis* element with its sub-elements and has changed value in *legalCode* field and has one additional *service* element.

In the merging process field *legalCode* will be replaced by the value in the client side XML. As the anamnesis block is missing in the client side document, the values are taken from the original XML in the server. The added *service* field will also be added to the document in the server. Finally, in the database we will have merged XML document as described in Figure 21.

Document in the server	Client side saved and sent document to server
<root> <patient> <name>Smith, Johan</name> <legalCode>322120102</legalCode> <address>234c Baker St.</address> <married>true</married> <age>55</age> </patient> <anamnesis> <complaint> Head ache</complaint> <primaryDiagnosis>K22.1 Consistent head-ache</primaryDiagnosis> <opinion/> </anamnesis> <services> <service>Specialist appointment</service> <service>Observation: Cholesterol, Lipid</service> <service>Observation: Blood glucose </service> </services> <observations> <observation>Cholesterol and Lipid Levels:OK</observation> <observation>Diabetes (blood glucose): too low</observation> </observations> <approved>true</approved> </root>	<root> <patient> <name>Smith, Johan</name> <legalCode>112233</legalCode> <address>234c Baker St.</address> <married>true</married> <age>55</age> </patient> <services> <service>Specialist appointment</service> <service>Observation: Cholesterol, Lipid</service> <service>Observation: Blood glucose </service> <service>20x25mg paracetamol</service> </services> <observations> <observation>Cholesterol and Lipid Levels:OK</observation> <observation>Diabetes (blood glucose): too low</observation> </observations> <approved>true</approved> </root>

Merged XML document
<root> <patient> <name>Smith, Johan</name> <legalCode>112233</legalCode> <address>234c Baker St.</address> <married>true</married> <age>55</age> </patient> <anamnesis> <complaint>Head ache</complaint> <primaryDiagnosis>K22.1 Consistent head-ache</primaryDiagnosis> <opinion /> </anamnesis> <services> <service>Specialist appointment</service> <service>Observation: Cholesterol, Lipid</service> <service>Observation: Blood glucose</service> <service>20x25mg paracetamol</service> </services> <observations> <observation>Cholesterol and Lipid Levels:OK</observation> <observation>Diabetes (blood glucose): too low</observation> </observations> <approved>true</approved> </root>

Figure 21 - Merge documents

5.6 Discussion

There has been some related work to secure XML documents based on content or structure of the document. Sandhu *et al.* in [32] define RBAC on Document Type Definition and XML Schema. The work is similar to ours: they also have four main actions and calculate permissions on elements level, but they do not transform the XML Schemas together with XML document, assuming that the output schema is already known. Samarati *et al.* in [13] and Diamani *et al.* in [12] define access control on DTD level. Samarati uses XML Access Control List to specify the permissions, Diamani X-path expressions. We use our own defined domain specific language. Bertino *et al.* in [9] defines XML based RBAC policy specification framework – X-RBAC. It is not used to secure XML documents but to control access in dynamic XML-based web services. X-RBAC has also a Java based GUI-enabled application to define the rules. Bertino *et al.* in [8] define a Java based system for access control to XML sources – Author X. It can be used to secure XML documents directly and documents based on their Document Type Definition. Crampton in [10] creates separate security views with X-path and XQuery to secure documents. Murata *et al.* in [26] define static methods to secure documents so that it would not be needed to calculate visible fields in a schema every time, as there might exists policies that are common for all roles. Using static methods could also be a future work to improve our solution.

5.7 Summary

In this paragraph we described the four possible actions to assign role permissions to elements in XML, introduced XML Schema and XML transformation rules and proposed a method how to merge original XML and edited XML into one new document without losing the document's completeness. After describing the process steps we illustrated it with an example on simplified healthcare diary record. In the next chapter we will describe the modelling of RBAC permissions using SecureUML and how to generate needed output from the models.

Chapter 6: Modelling & Templates to do RBAC

In this chapter we will define rules to model RBAC for dynamic forms using SecureUML. For defining the rules we assume that there already exists a definition for a form in the system and it needs additional security to be applied or is simple to be modelled. If the XML Schema has only complex types of elements of simple types – in this case we will also be able to generate the XML Schema directly from the SecureUML model.

At first we will introduce used stereotypes, rules to define each stereotype and then how the objects are associated to each other. We will illustrate the usage with MagicDraw usage for SecureUML.

6.1 Stereotypes

As SecureUML [21] uses stereotypes that are not defined UML, we will introduce new stereotypes (discussed in Chapter 2) to cover the needed notation:

- <<*securml.resource*>> - This stereotype is applied to a class that defines the unit to be secured.
- <<*securml.role*>> - This stereotype is applied to a class that defines the actor's role in the process.
- <<*securml.permission*>> - This stereotype is applied to an association class that defines the relationship between the role and the resource.

For XML Schema we will introduce an additional stereotype:

- <<*xml.schema*>> - This stereotype is applied to a class that defines the main form class.

The stereotypes will cover RBAC resources, roles and permissions.

6.2 SecureUML Resources

In SecureUML resources are classes with stereotype <<*securml.resource*>>. We will define a form that needs to be secured as a separate class. The name of the class can be chosen freely. The attributes of the class are *ID* and form elements. *ID* is a special attribute which default value has to match the forms primary key identifier in the database (MySQL database structure given in Appendix C). Form element attributes must contain all XML Schema elements which do not have any child elements. The name of the attribute must match the name of the XML Schema element name tag. We also define the operations to the form class object to describe the possible actions for each element. The operations are illustrative and will not be used in the transformation from UML to SQL. In Figure 22 we have defined a form with *ID= 201* and ten different elements (with types *string*, *int*, *boolean*) on the form, also some possible operations are pointed out.

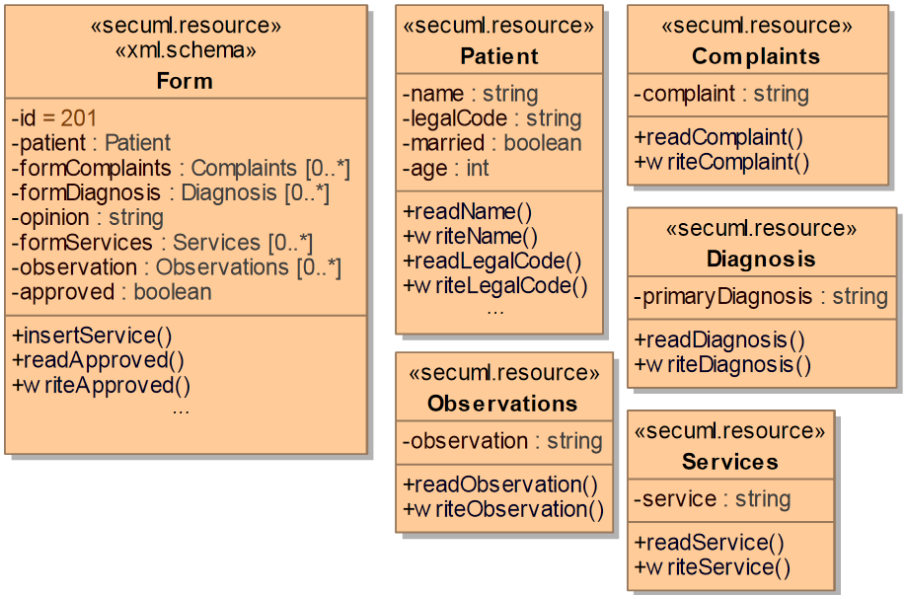


Figure 22 - Resource class

6.3 XML Schema Resources Limitations

The XML Schema has some limitations based on DynaForm and modelling. To generate an XML Schema the class of type *<<xml.schema>>* (class *Form*) can contain elements of types defines in Chapter 3 and complex elements defines in the same diagram. It is important that when modelling the complex element (class *Patient*) should not contain different element types than in Chapter 3.

6.4 Roles

In SecureUML roles are classes with stereotype *<<secuml.role>>*. The class name must match the role name used when requesting the form. Additional attributes and operations can be defined but they will be ignored when transforming the model. An example model for role *ADMIN* is displayed in Figure 23.

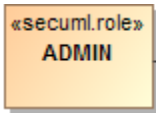


Figure 23 - Role class

6.5 Permissions

In SecureUML permissions are association classes with stereotype *<<secuml.permission>>*. Permissions are associations between role and resource object classes. The permission groups are defined as attributes. The class type for the attribute has to be one of four actions: *Read*, *Write*, *Insert* or *Delete*. The name of the attribute is chosen freely. The association between the permission and form field (stereotype *secuml.resource* class) is defined via abstract association between a certain permission attribute and form attribute. To define all four action permission there must be defined at least four attributes with everyone with a different action type class. Figure 24 displayed an

example permission class with four action types. Also we can see that attribute *viewName* gives actually *Read* permission for fields: *address*, *married* and *name*.

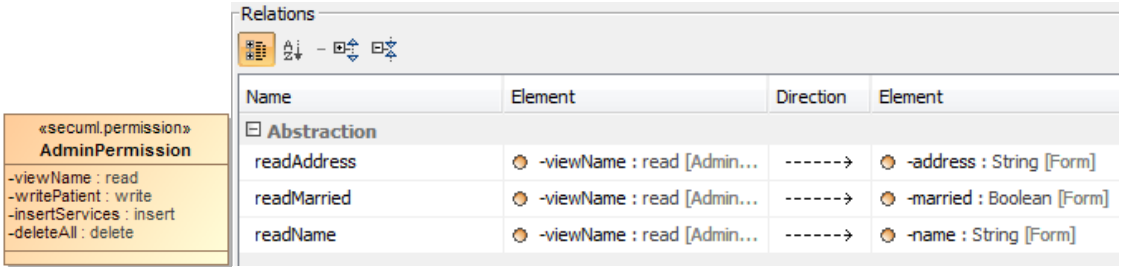


Figure 24 - Permission class

6.6 Template

To generate permissions from XML Schema we use templates in Velocity [5] language (interpretable by MagicDraw modelling tool) to transform the class diagrams into SQL update sentences the can be executed in the SQL console (database pre-configuration is given in Appendix C). Appendix A describes the transformer to generate the XML Schema and the permissions from class diagrams. The algorithm used to capture permissions from class diagram is given in the end of the paragraph. Using the transformer the XML Schema limitations apply. If the form should be more complex (have more than 2 level structures), then the XML Schema should be generated with a different tool *i.e.* Altova XML Spy. In this case only the XML Schema limitations by DynaForm will apply and template in Appendix B should be used for updating permissions. Both cases limitations are described in Chapter 3. The main benefit from modelling XML Schema together with permissions is that it is possible to use same tool to change the rules and document's structure, which will give a better overview how security is applied. Also there is a straightforward solution through Velocity templates to initiate propagating the changes to the application. The algorithm used to capture permissions from the model is given as pseudo code:

```
1. Find all Roles (RO), Resources (RC), Association classes (AC)
2. Take next RO
3. Find all R associated AC attributes (RACA)
4. Take next RC
5. Find all RC attributes (RCA)
6. Search all association RACA -> RCA
      a. Evaluate R:=true if exists RACA -> RCA where RACA type = read
      b. Evaluate W:=true if exists RACA -> RCA where RACA type =
         write
      c. Evaluate I:=true if exists RACA -> RCA where RACA type =
         insert
      d. Evaluate D:=true if exists RACA -> RCA where RACA type =
         delete
7. Write out RO>>RCA>>R,W,I,D<break>
8. If exists more RC elements then go to 4, otherwise continue
9. If exists more RO elements then go to 2, otherwise end.
```

6.7 Discussion

Jin in [18] is close to our work. She uses UML to define RBAC rules to generate Extensible Access Control Mark-up Language system security rules out of the model. Also Matulevičius and Lakk in [25] have used SecureUML to generate PL/SQL database views and access control based on roles. The main difference of our work from all the others is that we model the permissions and resources using SecureUML and generate SQL sentences that can be directly executed in the application. We have also analysed the solution from modifying and merging the documents in the business process perspective in order to keep the information's integrity.

6.8 Summary

In this chapter we defined new stereotypes to define RBAC model in SecureUML, discussed each of the stereotype in respect to modelling, and introduced how permissions are assigned to users. Also we introduced the method how to model simple XML Schemas using class diagrams and how to generate the class diagrams into SQL sentences using MagicDraw and Velocity templates. In the next part we will introduce our validation.

PART III

Validation

Chapter 7: Case study

In this chapter we will introduce experiment that was conducted to test our contribution's validity. It includes modelling, generating code and executing the code in our prototype.

Experiment was conducted from 16th to 19th of April. The resources and permissions were modelled by an experiment group, pre-graduate students of University of Tartu on 18th of April 2013, associated to course Secure Software Design. The research point for our case study was: is it possible to use SecureUML together with RBAC on XML documents structure to fulfil a predefined business process in our prototype?

The validation process contained of six main sub-processes:

1. Defining business process as a UMLSec [19] diagram;
2. Defining scenarios;
3. Modelling resources (documents);
4. Modelling RBAC in SecureUML;
5. Generating SQL sentences to be inserted to the database;
6. Testing the document workflow.

First, we defined two business processes: purchase order and course subscription using UMLSec [19] notation (Appendix F).

We will discuss the second scenario based on Figure 25. The process starts with student filling in an application: writing his contact info, previous studying and course he would like to participate. Then evaluation team views the application: they can only see the previous study info and applied course. They will mark the subscription as accepted or rejected. If the application was rejected, then the process stops. Otherwise, the application is sent to study committee who will see the document's content and can add a scholarship decision. Next, the document is sent to an accountant who can see all fields in document and marks the payment as rejected or accepted. For each protected method we use association tags to relate them to roles. For example to give role *Evaluation team* permission to *addDecision* we describe it:

AT6#:
{protected = addDecision}
{role = (<username>, Evaluation team)}
{right = (Evaluation team, addDecision)}

The full list of association tags is given in Appendix F. The pre-defined process was used in the end to validate the document manipulation and merging. The operations used to manipulate the document are differently defined by experiment group but the document still go through the same lifecycle.

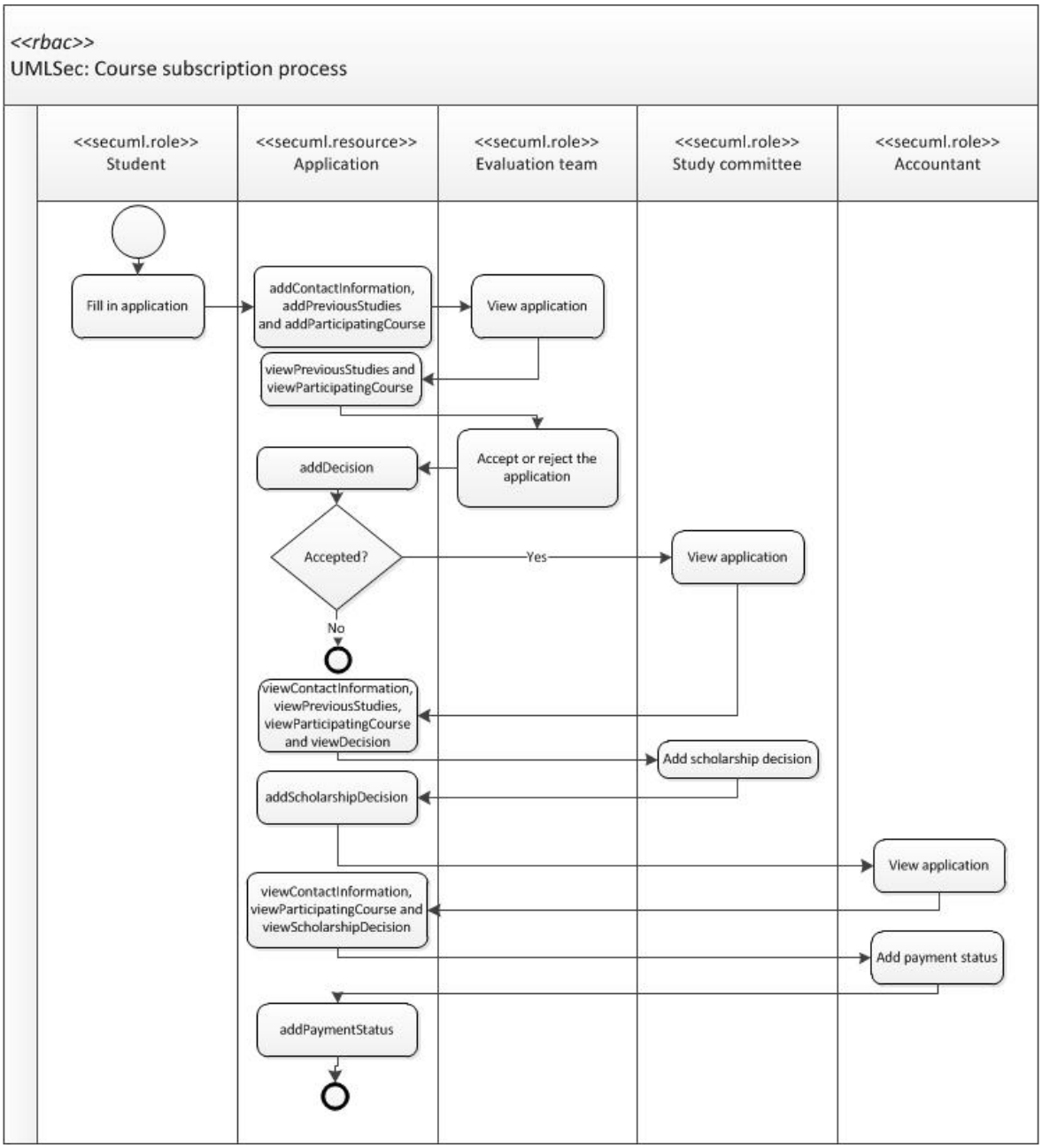


Figure 25 - UMLSec: Course subscription process

Secondly textual scenarios were defined based on the process description (part of Appendix E). These descriptions of two processes were given to our experiment group to model resources and RBAC. The scenarios also included an example document which was used to model the resource (Appendix E).

Modelling resources and RBAC in SecureUML was done by our experiment group. They were split into 4-6 student groups, nine groups altogether. The groups were generated by students themselves and the scenario was chosen randomly to the teams. They had 50 minutes to describe the solution based on instructions.

The pre-knowledge about modelling in SecureUML for the majority of the students was the same. They had had one 1.5h lecture about modelling in SecureUML which included

one practical exercise. The students did not know about participating in a validation process and were dealing the exercise as being a Secure Software Design course task. After gathering the SecureUML models we analysed them and concluded that the models need additional reviewing in order to be possible to use them in our prototype. An example solution is in Figure 26. It contains of four role classes; one resource class and four association classes. Each role is assigned to the resource class which attributes are through abstract association (not visible from the figure) to the resource attributes.

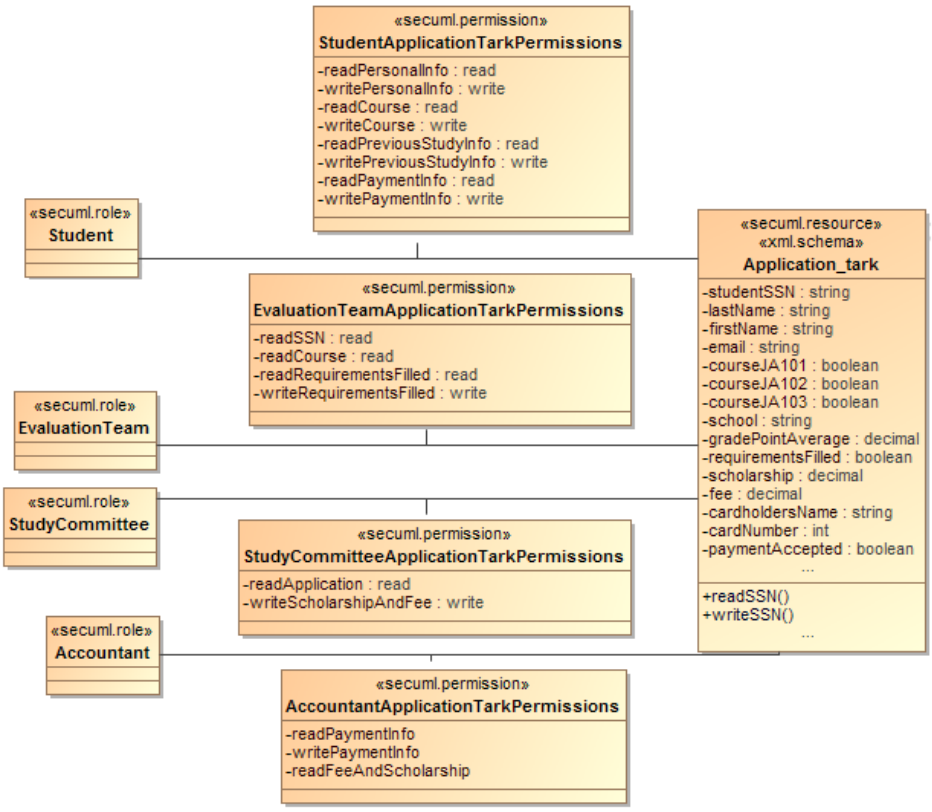


Figure 26 - Example solution - Course application

After reviewing the models we used our Velocity templates to generate XML Schema and permissions as SQL sentences. The data update queries were excecuted in our application database.

As a final step we tested the workflow of the process with the experiment group outcomes based on UMLSec diagrams given in Figure 25 and Appendix F.

7.1 Results

All the experiment groups were able to define SecureUML diagrams. From nine of the workgroups all generated role-based access control so that the models could be used with minimal add-ons to apply security. The adding of different attributes was based on the thesis team opinion. This is also one of threats to validity.

Modelled class diagrams are added to the work as Appendix D (an example solution in Figure 26). The count of manipulations done to the model is given in Table 5, where each column describes one working group.

Attributes definitions in Table 5:

Roles defined – Number of RBAC roles defined by the group.
Association classes defined – Number of correct association classes between role and resource.
Permission attributes defined – Number of attributes defined in permission association classes.
Permission attributes added – Number of additional attributes definitions added by us to the permission association classes.
Resource classes defined – Number of resource classes defined.
Resource classes added – Number of resource classes added to the diagram by us.
Resource attributes defined – Number of attributes defined in resource classes.
Resource attributes added – Number of attributes definition added to resource classes by us.
PRR defined – Number of association combinations permission attribute, resource operation and resource attribute defined.
PRR added – Number of association combinations permission attribute, resource operation and resource attribute added by us.
Operations defined – Number of operations defined in the resource class.
Process complete – Was it possible to go through the predefined process? (yes/no)
Document complete - Was all the data manipulations reflected to the document? Was it reflecting the actual manipulations and did it contain all info added? (yes/no)

Table 5 - Validation results

	Course subscription					**Purchase order**				
Attribute	**Group1**	**Group2**	**Group5**	**Group8**	**Group9**	**Group3**	**Group4**	**Group6**	**Group7**	**Average**
Roles defined	4	4	4	4	4	4	4	4	4	4,0
Association classes defined	4	4	4	4	4	4	4	4	4	4,0
Permission attributes defined	9	15	9	10	18	17	8	7	15	12,0
Permission attributes added	1	0	1	0	0	3	12	6	2	3,4
Resource classes defined	1	1	1	2	1	4	4	2	5	2,7
Resource classes added	0	0	1	0	1	0	0	0	0	0,3
Resource attributes defined	10	8	13	8	14	17	16	12	16	13,7
Resource attributes added	0	0	1	2	1	0	0	0	0	0,3
PRR defined	21	0	0	0	0	0	0	0	0	0,0
PRR added	14	27	47	32	52	88	55	44	54	53,1
Operations defined	14	14	3	6	8	0	16	9	22	9,1
Process complete	yes	yes	yes	yes	yes	yes	no	yes	yes	
Document complete	yes	yes	yes	yes	yes	yes	yes	yes	yes	

All the groups were able to define correct roles and association classes. We added missing attributes and elements in order to make the models complete respect to our implementation. We tried to minimize the changes needed in changing the resource class, but defined as many permission attributes and PRR definitions as needed to support the flow. By manipulating the permission attributes and PRR we specified how the permissions were applied, but with minimal changes to resource class we ensured that the secured document structure varied in order to test our solution. By average we had 12 permission attributes defined and added 3.4 attributes to the permission association class for one diagram. For Group5, Group8 and Group9 we also had to change the resource document as they did not define multiple occurrences element as a separate class. This limitation was found during the case study. It was also concluded that the most complex tasks, for experiment group, was to define permission attribute, resource operation and resource attribute associations. Only one team out of nine manage to define any of them.

After adding missing concepts we followed predefined process (Appendix F) in the developed prototype, checked the possibility to fill the business process and that all content added or manipulated were reflected to the XML document. Eight out of nine teams solutions were able to finish the business flow. Group4 case failed as they had not specified a field to collect delivery date (in Purchase Order scenario), therefore we could not move through step "Specify delivery date". Going through the process all teams solutions did not lose any data entered or modifications made.

7.2 Threats to validity

During the case study we also found couple of threats to the validity of our case study:

1. None of the teams could create a fully working process. This could be due to the 50 minutes timeframe to do the exercise, but can also be due to the complexity.
2. All the additional manipulations were done by us in order to make the teams solutions work in our implementation. We are more focused on the working solution than in the groups' vision of the security.
3. The experiment group students were related to the performers – students from the same faculty. As the students did not know that they were part of a validation process this should be minimal.

7.3 Summary

The results show that our transformation rules can be used for the defined document and business process. We have shown that based on nine different document definitions and permission descriptions the associated business processes can be completed without losing any data in the process. The task to model SecureUML diagrams was harder than expected and the subjects could not finish totally the modelling in given 50 minutes. With reviewing the solutions and making minor updates the solutions were usable in respect to our prototype. In the next chapter we will conclude our work and introduce future work.

Chapter 8: Conclusions and Future Work

In this chapter we summarize the main conclusions of our thesis. We present the limitations to the solution provided, summarize our studies and also introduce possible future work to improve our solution.

In this thesis we provided a solution to integrate existing technology to dynamically define forms and security permissions without losing context information.

8.1 Limitations

We limited our solution to secure elements by their attribute name value. This means if two fields with the same name exist, they cannot be distinguished and it will be not clear which security policy to apply on them. To prevent that the field names in XML Schema has to be unique. When a multi-occurrence element is needed it has to be defined as a complex type for the merging to keep the XML document valid respect to the schema. This limitation was noted in the case study. Although this limits possibility to model XML Schema, it does not limit permission calculations. If a complex form is needed, then the form can be defined with some other tool, modelled as a flat class in SecureUML (all fields as attributes in one class) and permissions assigned to the resource class.

The connection between the server and the client is presumed to be secure, so the traffic of the data is not encrypted. The server and the client are treated as trusted parties.

8.2 Conclusions

> The main research problem addressed in this work as stated in Chapter 1 is "*Can existing technology be integrated to dynamically define forms and security without losing context based information?".*

This research problem was divided into two research questions for investigation. We discuss the answers to the questions separately and then summarize the work.

Can we dynamically define forms and permissions of the document?

> To answer this question we analysed an architecture solution with the support of modelling forms and RBAC permissions using SecureUML [25] language. We also developed Velocity [5] templates to generate codes from the models.

Can we keep the document context complete when applying permissions on documents?

> We investigated how manipulations carried out on the document (adding/ substituting/ deleting data) affects its completeness. To secure documents we developed a transformer which gives to the displaying component only authorized information. To avoid losses upon modification of the document (as not all info

was visible) we introduced ELCA XmlMerge module [16] to ensure the context info completeness. By integrating the technology and implementing RBAC and document transformation we have a solution where forms and permissions are dynamically defined.

As a result we created: SecureUML modelling strategy to dynamically control permissions and forms structure; Velocity templates to generate code from SecureUML models; application where the code is executed and RBAC and DynaForm [29] are implemented; solution in the server to keep the document content complete. This shows that it is possible to integrate together modelling tools and applications to create dynamic solutions. To validate the solution correctness we conducted a case study in respect to capturing all information from the models and documents context completeness.

In the case study we analysed whether our proposed architecture and policy applying strategy worked. Eight out of nine permission combinations, defined by subjects, were supporting the process in all steps. One solution missed a field definition so the process was not complete. In all of the cases all permission were captured from the model and the context data was intact after every data manipulation from the application. The results from the case study verify that we are able to generate solution from existing technologies that dynamically defines form and its permissions.

8.3 Future work

In the current study we use our own Domain-Specific Language [14] to define permissions on resource files. As one of the improvements, we could use XML based structure (for example XACML [28]) for defining permissions to make the solution more uniform. Also more business processes and documents could be defined to be able to find the additional needs to the documents and the process. In order to improve the solution we could combine structure based security together with file level security. Additionally the XML Schema [35] could be improved to be able to hold info on the visual output of the fields: *i.e.* field length, where to get data and field titles in multiple languages.

Dünaamiline rollipõhine ligipääsu kontroll XML dokumentidele

Magistritöö
Kaarel Tark

Kokkuvõte

Tänapäeval hoitakse enamus dokumente elektroonsetel andmekandjatel. Samaaegselt kasvab andmete maht ja vajadus nende kiireks transportimiseks. Dokumendid võivad sisaldada salajast või infot, mis valede osapoolte käes võib olla ohtlik või tähendada äririske, mis tekitab vajadust dokumentide sisu kaitsta. Peamiselt käsitletakse dokumenti kui tervikut – kuulutatakse terve dokument näiteks salastatuks. Selline lähenemisviis ei ole alati põhjendatud ja osaliselt võib dokumendi sisu siiski olla avalik: näiteks anonüümsete uuringute tegemiseks.

Rollipõhine ligipääsu kontroll (inglise keeles – Role Based Access Control) on meetod defineerimaks kasutajate õiguseid infosüsteemis vastavalt neile omistatud rollidele. Turvameetmete defineerimine ja realiseerimine on aeganõudev töö, mida tihti tehakse paralleelselt või pärast rakenduse loogika realiseerimist.

Lahenduseks probleemile pakume rollipõhist ligipääsukontrolli dünaamiliselt defineeritavatele dokumentidel. Kuna dokumendid võivad olla oma mahult suured, siis defineerime õigused dokumendi struktuuri tasandil (XML Schema). Kontrolli ja dokumendi vormi modelleerimiseks kasutame SecureUML notatsiooni, mis on UML'i laiendus. Antud mudelist genereerime SQL koodi, mida andmebaasis jooksutades saame muudatused viia mudelist serveri rakendusse. Server vastutab antud privileegide ja dokumendi struktuuri kaitstud osade eest ja annab välja ainult antud rollile lubatud infot.

Antud töö raames teostatud uuring näitas, et on võimalik modelleerida nii vorm kui õigused nii, et ligipääsupiiranguid omavad andmed on kaitstud. Kuvatavate andmete muutmisel ja lisamisel ei rikuta olemasoleva dokumendi terviklikkust ja andmed säilivad.

References

[1] Altova XML Editor http://www.altova.com/xml-editor/ (Last checked: 13.04.2013)

[2] Anaya V., Berio G., Harzallah M., Heymans P., Matulevičius R., Opdahl A. L., Panetto H.,Verdech M. J. (2010) The Unified Enterprise Modelling Language – Overview and Further Work, Computers in Industry, Elsevier Science Publication, Vol 61, No 2, 99-111

[3] Apache Axis http://axis.apache.org/ (last checked: 12.05.2013)

[4] Apache Tomcat http://tomcat.apache.org/ (Last checked: 13.04.2013)

[5] Apache Velocity http://velocity.apache.org/ (last checked: 13.04.2013)

[6] Asynchronous vs. Synchronous http://www.inetdaemon.com/tutorials/basic_concepts/communication/asynchronous_vs_synchronous.shtml (last checked: 31.03.2013)

[7] Basin, D., Doser, J., Lodderstedt, T. (2006) Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15 Issue 1, January 2006, 39 - 91

[8] Bertino, E., Braun, M, Castano S., Ferrari E., Mesiti, M. (2000) Author-X: a Java-Based System for XML Data Protection, Data and Application Security, Springer US, pp. 15 - 26

[9] Bertino, E., Joshi, J., Bhatti, R., Ghafoor, A. (2003) Access Control in Dynamic XML-based Web-services with X-RBAC, ICWS, 2003, pp.243-249

[10] Crampton, J. (2004) Applying hierarchical and role-based access control to XML documents, SWS '04 Proceedings of the 2004 workshop on Secure web service

[11] Create a basic flowchart – Visio – Office.com http://office.microsoft.com/en-us/visio-help/create-a-basic-flowchart-HP001207727.aspx#BM1 (last checked: 31.03.2013)

[12] Damiani, E., Vimercati, S. C., Paraboschi, S., Samarati, P. (2002) A Fine-Grained Access Control System for XML Documents, ACM Transactions on Information and System Security (TISSEC), Volume 5 Issue 2, May 2002, pp. 169 - 202

[13] Damiani, E., Vimercati, S. C., Paraboschi, S., Samarati, P. (2000) Securing XML Documents, Advances in Database Technology — EDBT 2000, Springer Berlin Heidelberg, pp. 121-135

[14] Deursen, A., Klint, P., Visser, J. (2000) Domain-Specific Languages: An Annotated Bibliography, ACM SIGPLAN Notices, Volume 35 Issue 6, June 2000, pp. 26 - 36

[15] Dubois, E., Heymans, P., Mayer, N., Matulevičius, R. (2010) A Systematic Approach to Define

the Domain of Information System Security Risk Management, Intentional Perspectives on Information Systems Engineering, Springer
Heidelberg, pp. 289 - 306

[16] ELCA, Document for module XmlMerge
http://fisheye.collectionspace.org/browse/~raw,r=4964/collectionspace/src/sandbox/XMLMergeSample/XMLMerge-ReferenceDoc.pdf (Last checked: 13.04.2013)

[17] Extensible Markup Language http://www.w3.org/XML/ (last checked: 13.04.2013)

[18] Jin, X. (2006) Applying Model Driven Architecture approach to
Model Role Based Access Control System

[19] Jürjens, J. (2005) Secure Systems Development with UML, Springer-Verlag Berlin Heidelberg

[20] Kudo, M., Hada, S. (2000) XML Document Security based on Provisional Authorization, CCS '00 Proceedings of the 7th ACM conference on Computer and communications security, pp. 87 – 96

[21] Lodderstedt T., Basin, D., Doser, J. (2002) SecureUML: A UML-Based Modeling Language for Model-Driven Security, ≪UML≫ 2002 — The Unified Modeling Language, Springer Berlin Heidelberg, pp. 426 - 441

[22] Magic Draw http://www.nomagic.com/products/magicdraw.html (last checked: 13.04.2013)

[23] Matulevičius, R., Dumas M. (2010) A comparison of SecureUML
and UMLsec for role-based access control, The 9th Conference on Databases and Information Systems; Riga, Latvia; July 5-7, 2010, Databases and
Information Systems. University of Latvia Press, pp. 171 - 185

[24] Matulevičius, R., Dumas, M. (2011) Towards Model Transformation between SecureUML and
UMLsec for Role-based Access Control, Databases and Information
Systems VI, IOS Press, Inc., pp. 339 – 352

[25] Matulevičius, R., Lakk, H. (2011) A model-driven Role-based Access Control for SQL Databases

[26] Murata, M., Tozawa, A., Kudo, M., Hada, S. (2003) XML Access Control Using Static Analysis, CCS '03 Proceedings of the 10th ACM conference on Computer and communications security, pp. 73 - 84

[27] MySQL :: The world's most popular open source database www.mysql.com (Last checked: 12.05.2013)

[28] Oasis eXtensible Access Control Markup Language (XACML) TC OASIS
https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml (last checked: 31.03.2013)

[29] Raudjärv, R. (2010) Dynamic Schema-Based Web Forms Generation in Java

[30] SAML XML.org Online community for the Security Assertion Markup Language (SAML) OASIS Standard http://saml.xml.org/ (last checked: 31.03.2013)

[31]Sandhu, R., Ferraiolo, D., Kuhn, R. (2001) The NIST Model for Role-Based Access Control: Towards A Unified Standard, ACM Transactions on Information and System Security (TISSEC), Volume 4 Issue 3, August 2001, pp. 224 - 274

[32]Sandhu, R., Zhang, X., Park, J. (2004) Schema Based XML Security: RBAC Approach, Data and Applications Security XVII, Springer US, pp. 330-343

[33]The Forms Working Group: XForms http://www.w3.org/MarkUp/Forms/ (last checked: 05.04.2013)

[34] Trace the growth of XML over 10 years http://www.ibm.com/developerworks/library/x-xml10years/ (last checked: 02.05.2013)

[35] W3C XML Schema http://www.w3.org/XML/Schema (last checked: 13.04.2013)

[36] XPath http://www.w3schools.com/xpath/ (last checked: 13.04.2013)

[37] XQuery http://www.w3schools.com/xquery/ (last checked: 31.03.2013)

Appendix A – RBAC and XML Schema Velocity template

Velocity template to generate XML Schema and permissions update SQL sentence. Used in MagicDraw after modelling the document and RBAC rules.

```
#foreach ($diagram in $Diagram)
#set($eList = $report.getDiagramElements($diagram))
update document_template set permissions = '#foreach ($role
in $eList)
#if($report.containsStereotype($role, "securml.role"))
#foreach ($form in $eList)
#if($report.containsStereotype($form, "securml.resource"))
#foreach($attributes in $form.ownedAttribute)
#if($attributes.name == "ID")
#set($formId = $attributes.defaultValue)
#end##if
#set($attributeClass="")
#foreach ($formClass in $eList)
#if($report.containsStereotype($formClass,
"securml.resource"))
#if($attributes.type.name == $formClass.name)
#set($attributeClass = $attributes.type.name)
#end##if
#end##if
#end##foreach
#if($attributeClass!="")
#set($fieldName = $attributeClass)
#end##if
#if($attributeClass=="")
#set($fieldName = $attributes.name)
#end##if
#set($hasRead = "-")
#set($hasWrite = "-")
#set($hasInsert = "-")
#set($hasDelete= "-")
#foreach($roleAssociationClass in $AssociationClass)
#if($report.containsStereotype($roleAssociationClass,
"securml.permission"))
#if($roleAssociationClass.relatedElement.get(1).name ==
$role.name ||
$roleAssociationClass.relatedElement.get(0).name ==
$role.name)
#foreach($associationProp in
$roleAssociationClass.ownedAttribute)
#foreach($roleRight in
$report.getRelationship($associationProp))
#if($attributes.name==$roleRight.target.get(0).name)
```

```
#if($associationProp.type.name == "read")
#set($hasRead="R")
#end##if
#if($associationProp.type.name == "write")
#set($hasWrite="W")
#end##if
#if($associationProp.type.name == "insert")
#set($hasInsert="I")
#end##if
#if($associationProp.type.name == "delete")
#set($hasDelete="D")
#end##if
#end##if
#end##foreach
#end##foreach
#end##if
#end##if
#end##foreach
#if($fieldName != "" && $fieldName !=
"ID")$role.name<>$fieldName>>$hasRead,$hasWrite,$hasInsert,$h
asDelete<break>#end##if
#end##foreach
#end##if
#end##forearch
#end##if
#end##foreach
' ,
TEMPLATE_XSD='<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
#foreach ($form in $eList)
#if($report.containsStereotype($form, "xml.schema"))
<xs:element name="$form.name">
<xs:complexType>
<xs:sequence>
#set($i = 0)
#set($j = 0)
#foreach($attributes in $form.ownedAttribute)
#set($isSimple = 1)
#foreach($types in $eList)
#if($attributes.type.name==$types.name &&
$attributes.name!="" && $attributes.name!="ID")
#set($string = $attributes.multiplicity)
#set($mult = $string.split("\.."))
#set($minoccurs="1")
#set($maxoccurs="1")
#set($Integer=0)
#if($Integer.parseInt($mult.get(0))>0)
#set($minoccurs="$mult.get(0)")
#end##if
```

```
#if($mult.get(1)=="*")
#set($maxoccurs="unbounded")
#end##if
#if($Integer.parseInt($mult.get(1))>0)
#set($maxoccurs=$mult.get(1))
#end##if
<xs:element name="$attributes.name">
<xs:complexType>
<xs:sequence>
<xs:element name="$attributes.type.name"
minOccurs="$minoccurs" maxOccurs="$maxoccurs">
<xs:complexType>
<xs:sequence>
#foreach($attrib in $types.ownedAttribute)
#set($string = $attrib.multiplicity)
#set($mult = $string.split("\.."))
#set($minoccurs="1")
#set($maxoccurs="1")
#set($Integer=0)
#if($Integer.parseInt($mult.get(0))>0)
#set($minoccurs="$mult.get(0)")
#end##if
#if($mult.get(1)=="*")
#set($maxoccurs="unbounded")
#end##if
#if($Integer.parseInt($mult.get(1))>0)
#set($maxoccurs=$mult.get(1))
#end##if
<xs:element name="$attrib.name"
type="xs:$attrib.type.name.toLowerCase()"
minOccurs="$minoccurs" maxOccurs="$maxoccurs"/>
#end##foreach
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
#set ($isSimple = 0)
#end##if
#end##foreach
#if($isSimple == 1  && $attributes.name!="" &&
$attributes.name!="ID")
#set($string = $attributes.multiplicity)
#set($mult = $string.split("\.."))
#set($minoccurs="1")
#set($maxoccurs="1")
#set($Integer=0)
#if($Integer.parseInt($mult.get(0))>0)
#set($minoccurs="$mult.get(0)")
#end##if
```

```
#if($mult.get(1)=="*")
#set($maxoccurs="unbounded")
#end##if
#if($Integer.parseInt($mult.get(1))>0)
#set($maxoccurs=$mult.get(1))
#end##if
#set($simpleComplex=0)
#if($maxoccurs!="1")
<xs:element name="$attributes.name" >
<xs:complexType>
<xs:sequence>
<xs:element name="$attributes.name"
type="xs:$attributes.type.name.toLowerCase()"
minOccurs="$minoccurs" maxOccurs="$maxoccurs"/>
</xs:sequence>
</xs:complexType>
</xs:element>
#end##if
#if($maxoccurs=="1")
<xs:element name="$attributes.name"
type="xs:$attributes.type.name.toLowerCase()"
minOccurs="$minoccurs" maxOccurs="$maxoccurs"/>
#end##if
#end##if
#end##foreach
</xs:sequence>
</xs:complexType>
</xs:element>
#end##if
#end##foreach
</xs:schema>'
where id=$formId.value;
#end##foreach
```

Appendix B – RBAC Permissions Velocity template

Velocity template to generate permissions update SQL sentence. Used in MagicDraw if only RBAC rules are modelled. Document is modelled with some other tool.

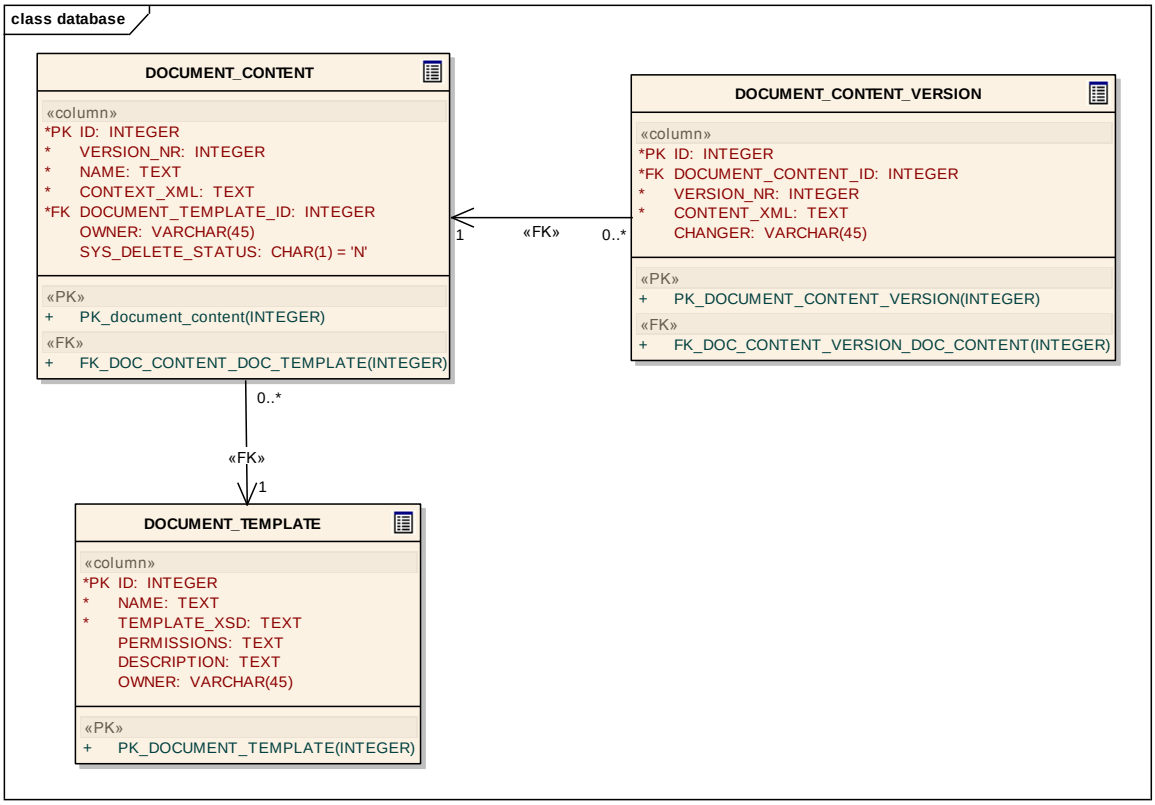
```
#foreach ($diagram in $Diagram)
#set($eList = $report.getDiagramElements($diagram))
update document_template set permissions = '#foreach ($role
in $eList)
#if($report.containsStereotype($role, "securml.role"))
#foreach ($form in $eList)
#if($report.containsStereotype($form, "securml.resource"))
#foreach($attributes in $form.ownedAttribute)
#if($attributes.name == "ID")
#set($formId = $attributes.defaultValue)
#end##if
#set($attributeClass="")
#foreach ($formClass in $eList)
#if($report.containsStereotype($formClass,
"securml.resource"))
#if($attributes.type.name == $formClass.name)
#set($attributeClass = $attributes.type.name)
#end##if
#end##if
#end##foreach
#if($attributeClass!="")
#set($fieldName = $attributeClass)
#end##if
#if($attributeClass=="")
#set($fieldName = $attributes.name)
#end##if
#set($hasRead = "-")
#set($hasWrite = "-")
#set($hasInsert = "-")
#set($hasDelete= "-")
#foreach($roleAssociationClass in $AssociationClass)
#if($report.containsStereotype($roleAssociationClass,
"securml.permission"))
#if($roleAssociationClass.relatedElement.get(1).name ==
$role.name ||
$roleAssociationClass.relatedElement.get(0).name ==
$role.name)
#foreach($associationProp in
$roleAssociationClass.ownedAttribute)
#foreach($roleRight in
$report.getRelationship($associationProp))
#if($attributes.name==$roleRight.target.get(0).name)
```

```
#if($associationProp.type.name == "read")
#set($hasRead="R")
#end##if
#if($associationProp.type.name == "write")
#set($hasWrite="W")
#end##if
#if($associationProp.type.name == "insert")
#set($hasInsert="I")
#end##if
#if($associationProp.type.name == "delete")
#set($hasDelete="D")
#end##if
#end##if
#end##foreach
#end##foreach
#end##if
#end##if
#end##foreach
#if($fieldName != "" && $fieldName !=
"ID")$role.name<>$fieldName>>$hasRead,$hasWrite,$hasInsert,$h
asDelete<break>#end##if
#end##foreach
#end##if
#end##forearch
#end##if
#end##foreach
'
where id=$formId.value;
#end##foreach
```

Appendix C – Relational Database Model

Server side relational database model. Document content holds the latest content of the XML. Document version hold all previous versions of a document. Document template contains info about the form and permissions. To define a new document we need to define a document template entry:

- ID – unique identifier for the form (has to match model resource attribute *ID*) NAME – name for the document;
- TEMPLATE_XSD – XML Schema definition for the form;
- PERMISSIONS – permissions for the document;
- DESCRIPTION – free text describing the document;
- OWNER – Owner name for the document (not used in our prototype).



Appendix D – Source code and models

Structure of CD:

/application/ - prototype application associated filed.

/application/compiled/ - compiled war files of AddServiceServer.war (server) and dynaform.war (client).

/application/source/ - zipped source of projects.

/application/database/ - database related files: create_database.txt (schema and tables creation scripts); database_server_conf.txt (database configuration); insert_* (database example data insert scripts).

/modelling/ - modelling related files.

/modelling/case study/ - case study diagrams as MagicDraw file.

/modelling/velocity templates/ - Velocity templates used to generate code in MagicDraw from models: permissions.txt (for only permissions generation), permissions_and_xml_schema.txt (for permissions and schema generation).

Appendix E – Case study

Principles of Secure Software Design

RBAC Application to Secure Documents

Workshop 18.04.2013

Goal: For the given scenario create the RBAC model in SecureUML (class diagram).

Task 1. Elicit the relevant RBAC information. Answer the following questions:

1.1. What are the **objects** (e.g., in your scenario *document*) and their **attributes**?

Note: *it is expected that you will characterise the document regarding its contents, i.e., field, entries. For instance, in the case of the Phonebook, its field entries are its name, issue year, list of persons; A person could be characterise by its name and phone number.*

1.2. What are **operations** that changes values of the attributes?

1.3. What are the **roles**?

1.4. What are the **security actions**?

Note: *the following security actions and their meanings could be used:*

- Read – *element value is visible.*
- Write – *if the element is visible (allowed to read) then it is allowed to change the value.*
- Insert – *if multiple sections (form field multiplicity) allowed, then it is possible to add another section.*
- Delete – *allowed removing elements.*

Task 2. Model the RBAC solution. Apply the template given in Figure 1.

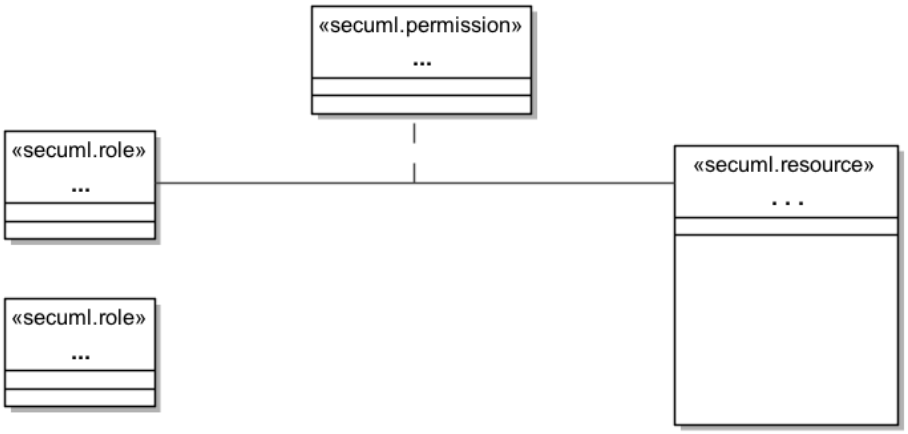


Figure 1: RBAC template

2.1. Model the resource (i.e. document) that needs to be protected.

- Define classes of secure resources;
- Define field entries as class attributes;
- Define attribute types. *Simple* (i.e., the which needs to be secured) attribute types for an element:
    - String data types: string.
    - Date data types: date, time, datetime.
    - Numeric data types: integer, long, int, short, byte, decimal.
    - Miscellaneous data types: boolean.
- Define complex types for logical groups. For example: contact information of a person can be a complex type containing of personName and phoneNumber attributes.

2.2. Define the attribute multiplicities

Note: *Define attributes multiplicity: 1, 1..*, 0..*, 0..5, etc. By default 1. For example (see Figure 2) a person have 0..* friends, but he can make a party where he can invite 0 to 20 (0..20) persons as the pub does not have more places.*

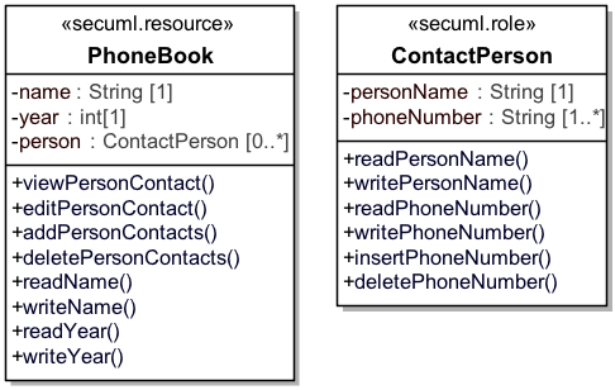


Figure 2: Secure resource definition

2.3. Model **the roles** and their **permissions**.
Security action model is given in Figure 3.

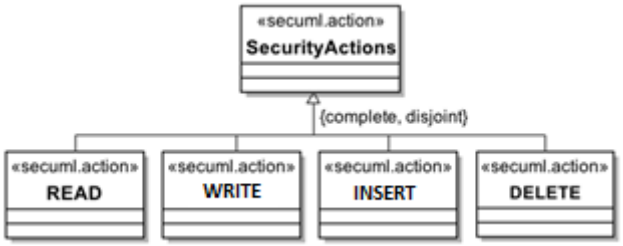


Figure 3: Security actions

2.4. Define the security authorisation constraints (textually). This means, describe the link between the security actions of the roles and the resource operations (over the attributes).

For example:
<permission attribute> – <resource opertation> – <resource attribute>
obtainPersonsPhoneNumber:READ – readPhoneNumber() – phoneNumber

Scenario 1: Order document / Client placing a purchase order

The process starts with client filling in an order form. Client specifies his contact info and requested items and sends the request to the seller. Seller views the purchase order and assigns unit codes and prices to the offering and sends the document to delivery team. Delivery department will specify when they can ship the package to the client. After the offering is complete it will be sent back to the client for acceptance. If the client refuses the offering, then the process stops and the document is deleted. If the offering is accepted by the client the document is sent to the accounting department who will transmission a credit card payment. If the payment is accepted, then accounting department will mark the order as being paid and send it to delivery department. The delivery department will pack and ship the ordered items and mark the order as shipped.

Roles: client, seller, accounting, delivery department

Purchase order – form

Seller Seller Company LLC
380 Franscisco St
San Francisco
CA 94133
US

Client Company LLC
1272 Rockafeller Str
Silverthorne, CO 80498
US

Shipping date: 21-May-2012

No.	Quantity	Item Code	Description	Unit Price	Amount	Delivered
1	100.00 PCS	INV00005	1900mAh Slim Battery Charging Case for iPhone4/4S	15.00	1,500.00	X
2	200.00 PCS	INV00006	2400mAh Solar Powered Rechargable Battery Pack for iPhone or iPod	12.00	2,400.00	X
3	200.00 PCS	INV00002	Apple iPad casing - black	20.00	4,000.00	X
4	200.00 PCS	INV00003	Samsung Galaxy Tab 10.1" casing - black	10.00	2,000.00	X
5	200.00 PCS	INV00001	Apple iPad casing - white	20.00	4,000.00	X

Payment info
Cardholder's name: Aleksandr Skafandr
Card Number: 213213123122312 Exp. Date. 01-May-2015

Payment: Accepted

Application (form 2): Application for Japanese course

The process starts with student filling in the subscription. Student specifies his contact information, school and current grade point average and registered course number. The subscription requests are sent to an evaluation team in the university who will not see the actual contact information of the subscriber, but only the previous study information, student SSN and course the student would like to register to. The evaluation team will mark requirements filled or not filled. If the if the requirements are not filled then the document is rejected, process stops and the student is notified. If the requirements are filled, then it is sent to study committee, who will decide in what amount or if any the course cost will be funded by a scholarship. The committee can only view the application with an exception to add scholarship amount to the document. After the committee has made its decision the document is sent to accounting department to approve payment. If the payment is accepted, then the accounting department marks it as paid.

Roles: student, study committee, evaluation team, accounting

Application for Japanese Courses – form

Student SSN: A62112
Last name: Skafandr
First name: Aleksandr
Email: skanfandr.aleksandr@ut.ee
Full name of School: University X

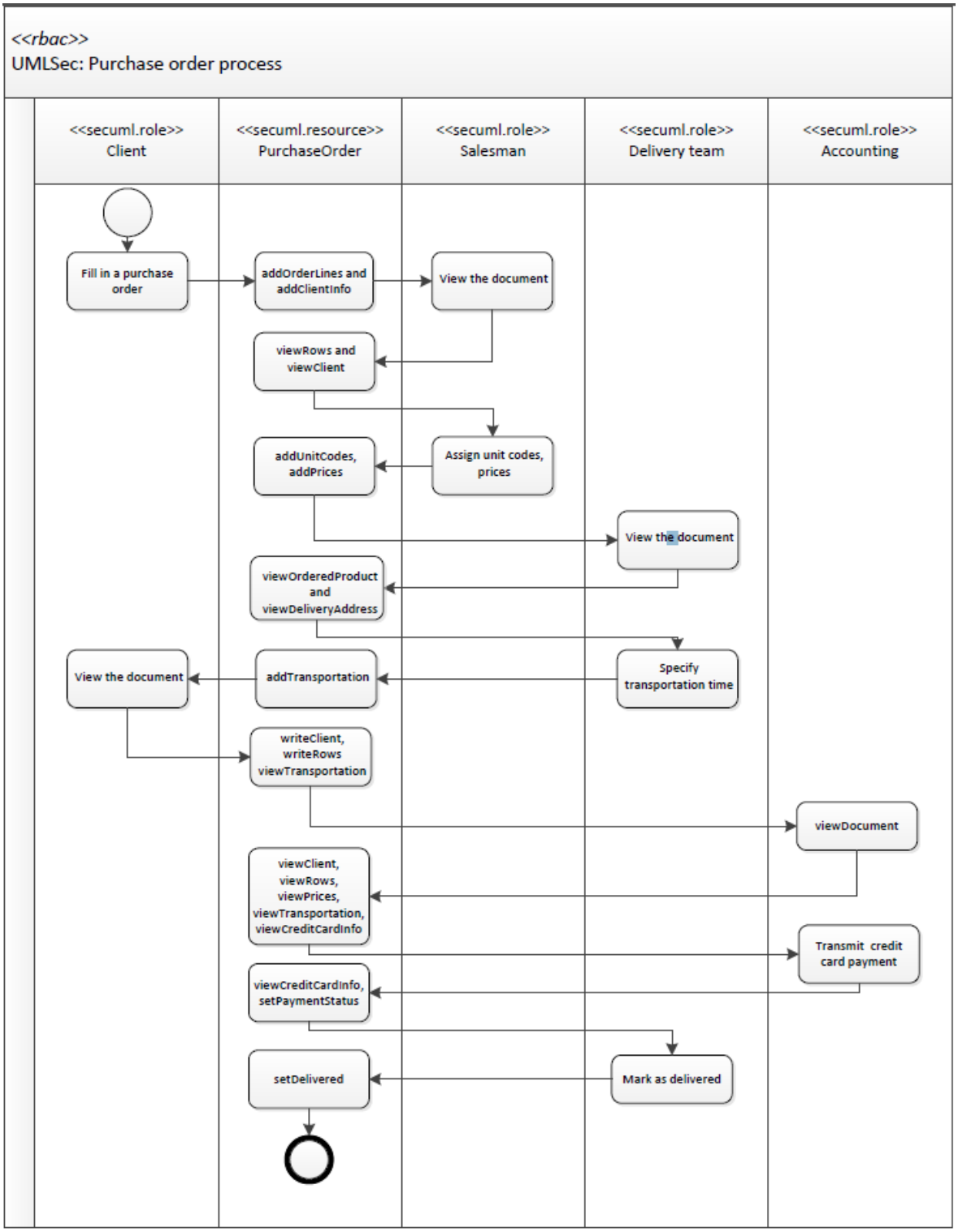
Registered course number
[ ] JA 101
[x] JA 102
[ ] JA 103

Fee: 400$
Scholarship: 100$
Grade point average (previous year): 4.2
Requirements filled: Yes

Payment info
Cardholder's name: Aleksandr Skafandr
Card Number: 213213123122312
Payment: Accepted

Appendix F – UMLSec process diagrams

Describes business processes used in validation process. First process describes purchase order document, then second one course subscription.



AT1#:
{protected = addOrderLines and addClientInfo}
{role = (<username>, Client)}
{right = (Client, addOrderLine and addClientInfo)}

AT2#:
{protected = viewRows and viewClient }
{role = (<username>, Salesman)}
{right = (Salesman, viewRows and viewClient)}

AT3#:
{protected = addUnitCodes and addPrices}
{role = (<username>, Salesman)}
{right = (Salesman, addUnitCodes and addPrices)}

AT4#:
{protected = viewOrderedProduct and viewDeliveryAddress}
{role = (<username>, Delivery Team)}
{right = (Delivery Team, viewOrderedProduct and viewDeliveryAddress)}

AT5#:
{protected = addTransportation}
{role = (<username>, Delivery Team)}
{right = (Delivery Team, addTransportation)}

AT6#:

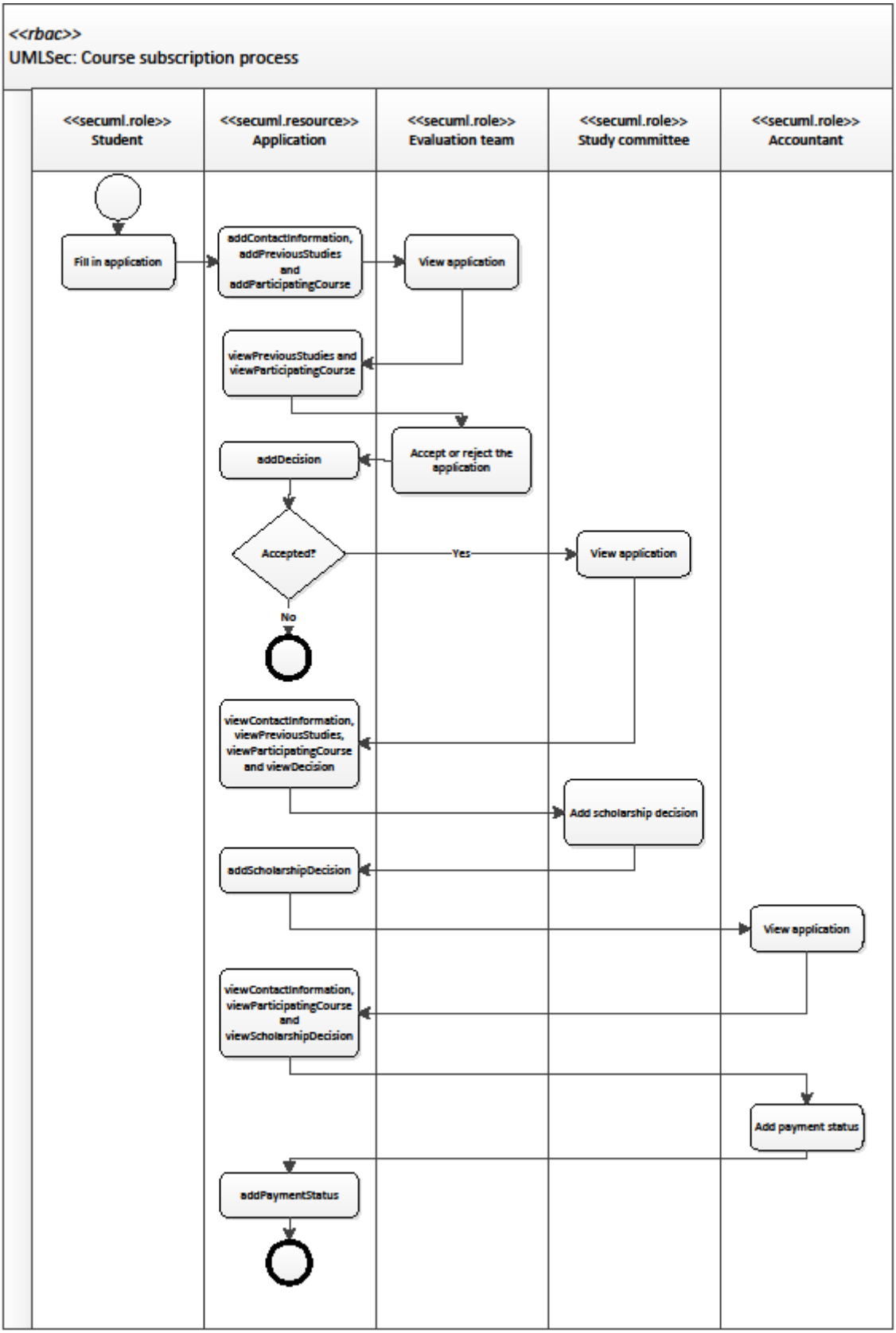
{protected = writeClient, writeRows and viewTransportation}

{role = (<username>, Client)}
{right = (Client, viewClient,viewRows and viewTransportaion)}

AT7#:
{protected = viewClient, viewRows, viewPrices, viewTransportation and viewCreditCardInfo}
{role = (<username>, Accounting)}
{right = (Accounting, viewClient, viewRows, viewPrices, viewTransportation and viewCreditCardInfo)}

AT8#:
{protected = viewCreditCardInfo and setPaymentStatus}
{role = (<username>, Accounting)}
{right = (Accounting, viewCreditCardInfo and setPaymentStatus)}

AT9#:
{protected = setDelivered}
{role = (<username>, Delivery Team)}
{right = (Delivery Team, setDelivered)}


<<rbac>>
UMLSec: Course subscription process
<<secuml.role>>
Student
<<secuml.resource>>
Application
<<secuml.role>>
Evaluation team
<<secuml.role>>
Study committee
<<secuml.role>>
Accountant
Fill in application
addContactInformation, addPreviousStudies and addParticipatingCourse
View application
viewPreviousStudies and viewParticipatingCourse
Accept or reject the application
addDecision
Accepted?
Yes
No
View application
viewContactInformation, viewPreviousStudies, viewParticipatingCourse and viewDecision
Add scholarship decision
addScholarshipDecision
View application
viewContactInformation, viewParticipatingCourse and viewScholarshipDecision
Add payment status
addPaymentStatus

AT1#:
{protected = addContractInformation, addPreviousStudies and addParticipatingCourse}
{role = (<username>, Student)}
{right = (Student, addContractInformation, addPreviousStudies and addParticipatingCourse)}

AT2#:
{protected = viewPreviousStudies and viewParticipatingCourse}
{role = (<username>, Evaluation team)}
{right = (Evaluation team, viewPreviousStudies and viewParticipatingCourse)}

AT3#:
{protected = addDecision}
{role = (<username>, Evaluation team)}
{right = (Evaluation team, addDecision)}

AT4#:
{protected = viewContactInformation, viewPreviousStudies, viewParticipatingCourse and viewDecision}
{role = (<username>, Study committee)}
{right = (Study committee, viewContactInformation, viewPreviousStudies, viewParticipatingCourse and viewDecision)}

AT5#:
{protected = addScholarshipDecision}
{role = (<username>, Study committee)}
{right = (Study committee, addScholarshipDecision)}

AT6#:
{protected = viewContactInformation, viewParticipatingCourse and viewScholarshipDecision}
{role = (<username>, Accountant)}
{right = (Accountant, viewContactInformation, viewParticipatingCourse and viewScholarshipDecision)}

AT7#:
{protected = addPaymentStatus}
{role = (<username>, Accountant)}
{right = {(Accountant, addPaymentStatus)}

Non-exclusive licence to reproduce thesis and make thesis public

I, Kaarel Tark

(date of birth:09.10.1987),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2.make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, Role Based Access Model in XML based Documents supervised by Raimundas Matulevičius.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, ...**05.2013**