Integration of access control in Information Systems: from role engineering to implementation

Thion Romuald and Coulondre Stéphane
LIRIS / INSA University of Lyon
20 Av. Albert Einstein
69621 Villeurbanne Cedex, France
romuald.thion@insa-lyon.fr and stephane.coulondre@insa-lyon.fr

Keywords: role–based access control, object–oriented models, role engineering, security

Received: November 12, 2005

Pervasive computing and proliferation of smart gadgets make organizations open their information systems, especially by extensive use of mobile technology: information system must be available any-time, any-where. This cannot be performed reasonably without thorough access control policies. Such an access control must be able to deal with user’s profile, time and eventually with other complex contexts like geographical position. This paper shows that it is possible to take into account confidentiality constraints straight into the logical data model in a homogeneous way, for various aspects treated independently (user profile, time, geographical position, etc.). We propose a language called LORAAM which includes a way to express authorizations at the class level. We first present the syntactical aspects, then the semantics of such a language, based on the object-oriented paradigm.

Povzetek:

1 Introduction

Companies and public interest for new technologies keeps growing, either for mobile use (laptops, Wi-fi, pocket-PC, GPS, UMTS, Java technology in GSM, etc..) or for “traditional” use. Information Systems (IS) now become open and online. As the use of IS arises, the demand to secure those systems also arises, particularly since systems contains confidential data. The security of IS must be guaranteed.

New mobile and pervasive technologies lead to the concept of context: a new constraint to consider in access control to the information system services. From now on, access control mechanisms tend towards integration of user profile, time, state of the computing environment and even geographical position [8].

1.1 Motivations

Security is considered as a non-functional requirement in software engineering. Contrary to other non–functional requirements, such as efficiency or modularity, confidentiality have been unconsidered for long. Thus, access control management is often postponed until the end of the design cycle and is implemented at the very end of the design process. The software is therefore developed without taking confidentiality constraints into account. This usual approach often leads to serious design challenges (e.g. integration of roles) and problems (e.g. software vulnerabilities, information leakage) [16].

In this paper, we show how to take into account general context data (user roles, spatio-temporal environment, etc.) in a homogeneous way, straight in the object data model (and more generally in Information Systems, Objects, Web Services, etc) to provide an object–oriented language including authorizations policies at the class level. We do think that security must be present throughout the whole development cycle. Our proposal describes a logical data model in which contextual role-based access control is integrated: we thus provide a support to upstream
design methods [9, 13] which rely on it.

1.2 Our approach

We do think that security concerns must be considered all the development cycle long. As user interface development recommends to work on GUI and core application in a parallel direction to avoid non user–friendly interfaces, we argue that access control, and in a broader sense security, must be considered as soon as possible in the software development cycle, and not neglected until the very end of this process.

To fill this gap between security will and its implementation, our approach is to provide a consistent logical data model including role–based access control policies. Since Role–Based Access Control (RBAC) is actually one of the most seducing security models and since extending object–oriented (OO) systems with roles has been abundantly studied in the literature, we chose to integrate role–based authorization policies at the class level. With such embedded authorizations in an OO language, developers can now integrate their security policies in their code in a declarative manner. As inheritance is used by programmers without worrying about how polymorphism or dynamic linking are implemented, authorization policies can be used without worrying about the mechanisms involved in the authorization decision.

The rest of the paper is organized thus. Section 2 presents the original Role-Based Access Control that our proposal uses for the organization of privileges within an information system. Section 3 details syntactical aspects of the LORAAM (for role–based authorizations policies within object–oriented language) language we propose. Section 4 details functional aspects with an illustrative example in the medical area, this section also describes how to implement our language. Section 5 surveys previous work in attempting to integrate the role concept in logical object data models and compare our approach to related works about integration of security in the software design process. Section 6 finally concludes the paper and discusses some perspectives on integration of security in software design process and.

2 Modelling authorizations with roles

Role modelling has been introduced into many computer sciences areas: databases, programming languages, ontologies or agent oriented modelling. For security purpose, roles have been introduced to make access control policies administration easier: this is the main idea of the “Role–Based Access Control” (RBAC) model.

2.1 An Access Control model

Access control denotes the fact of determining whether a subject (process, computer, human user, etc.) is able to perform an operation (read, write, execute, delete, search, etc.) on an object (a tuple in a database, a table, an object, a file, etc.). An operation right on an object is called permission. An access control model define how to organize the permissions of subjects.

The RBAC Model [20] was defined in the 90’s and has been extended in many ways since (temporal, geographical extensions, etc). It was introduced in order to tackle the weaknesses of DAC (Discretionary Access Control) and MAC (Mandatory Access Control) models: the former is difficult to implement with a large number of users, and the latter is too rigid for modern applications. We focused on RBAC rather than other recent access control models because it is actually the most seducing access control paradigm, as shown by its use in major databases management systems such as Oracle Enterprise Server v.8 or Sybase Adaptive Server v.11.5. Even for legacy systems which are not role–based, the use of RBAC may simplify management [17].

The basic RBAC philosophy is based on the observation that most of the access permissions are determined by a person authority or function, inside an organization. This defines the central concept of role. The introduction of role concept in access control policies as an intermediate layer between subjects and permissions, really facilitates and simplifies the system administration task. The RBAC definition of a role is “a job function within the organization with some semantics regarding the authority and responsibility conferred on the member of the role” ([20]).

The RBAC model family is based on the identification of a certain number of roles [19], each
of them representing a set of actions and responsibilities within the system (roles can be seen as a collection of permissions). Thus in the RBAC model (figure 1):

- no permission is granted directly to the subject (ex: user, process, object...), permissions are only granted to roles,
- the subjects endorse the roles which are given by the administrator (it is only possible to specify positive authorizations, no prohibitions),
- roles are defined and organized in a hierarchy: a child role has the permissions granted to his/her parents.

In OO systems, the concept of permission is related to objects’ methods. A permission is an access privilege on an resource. In OO systems, resources are objects (or attributes of objects) and access privileges are methods of objects (e.g. getAttribute() is a read-access, setAttribute() is a write-access). Thus, to grant an access privilege to a resource is to authorize the call of an object’s method. The rules defining permission assignment to role are “access control policies”.

2.2 Access control policies

Access control policies define the subjects’ rights on objects, in order to enforce security of an organization. In the RBAC model policies define which permissions are granted to roles (permission–role assignment in figure 1). Thus subjects are granted permissions through role–assignment (user–role assignment in figure 1).

An example of role–oriented access control policies in health sector would be:

- a nurse can only read the patient prescriptions. But she can write the last care date and time, provided it takes place during her working time,
- a doctor can only prescribe if he/she is geographically located in the hospital. He has access to the whole medical record, but he/she cannot write the last care date and time,
- a head nurse has read access to prescriptions and cares history without conditions of time.

Permissions associated to roles allow expressing access authorization in a generic way. Therefore we do not specify that “Dr. Johnson” has access to “Mr. Rabot” record. Instead we only specify that doctors have write access to patient records. According to RBAC’s principle “permissions are only granted to roles”, our proposal do not include policies related to individuals. Thus is is not possible to specify that only “Dr. Johnson” has access to “Mr. Rabot” record. The RBAC roles, their hierarchical organization and the associated permissions constitute the organization’s confidentiality policy.

The language we chose to express access control policies has been heavily influenced by [8] which formalize authorization policies, including temporal aspects, in first–order logic (FOL). Thus
we use a tractable fragment of FOL (no function symbol, no negation, no implication) suitable to express role–based authorization policies at the class level.

2.3 A suitable subset of RBAC

Our approach intends to include role–based authorizations into class models. Thus, authorizations policies are common to every objects instantiated from a class. As classes are static in OO systems, we are not able to express dynamic aspects of RBAC. The session concept for example, cannot be included into the class model: each session is related to exactly one user, and it represents the roles its user is actually endorsing.

Moreover, our proposal is an OO language designed for secured softwares. Its goal is not to include the whole RBAC model into class models, but only a subset describing authorizations policies of the application developed in such a language. Thus user–role assignments, sessions or delegations are outside our scope: we only model the authorization policies related to the application. For example, an hospital is developing an intranet web–portal. The user–role assignment are stored in a dedicated directory (which is used by other applications), not in web–portal itself. Only the policies describing “which role can access this method of this class of the web–portal” are included directly into the code.

Thus our language includes the following concepts of RBAC models:

– object hierarchy: class hierarchy,
– access privileges: methods,
– permissions: methods of objects,
– permission-role assignments: access to methods of objects are conditionned by roles,
– roles: our proposal integrates the definition of roles involved in authorization policies,
– role hierarchy: roles can be organized in hierarchies in our language.

Thus, the concepts of user, user–role assignment, sessions and context retrieval are outside our proposal and will be referred as “the user’s profiles” (section 4).

3 The LORAAM Language

In order to tackle the problems of RBAC integration within object data models, we propose a generic language LORAAM allowing the expression of RBAC authorizations and integrating an access control mechanism. The declarative part of the language is composed of:

– the body, which relies on C++ syntax (on a purely illustrative basis, as any class–based language could have been used: Java, Python, etc.) while adding access authorizations formulae to methods,

– the header, which defines the roles which are to be used in the definition of access authorizations.

3.1 The Header

The header is used to specify (annex: 6):

– various categories of roles to be taken into account. In this example we included the categories of [3] which are adapted to organisations: functional, seniority and context. These categories, freely chosen by the developer, form groups of roles. These groups represent transverse role aspects, which are combined to form complex roles. It would be possible to add some other groups such as wards, (ex: cardiology, radiology, etc. which remains static), or “classification” (ex: white, grey, black information according to the sensitivity of data) which can be used for example to simulate a MAC access control,

– hierarchical relations between roles [14]. For example head << assistant means that the head has (at least) all the privileges of the assistant. Thus, the conjunction of these roles with a functional role “doctor” makes it possible to specify complex roles, for example “head doctor”, who would have more privileges than a “simple” doctor, but fewer privileges than the manager (who is also a doctor),

– the various contexts in which the access authorizations are defined. These contexts can be geographical (by using the predicate position) or temporal (with the predicate hour).
We suppose that the position of the user is obtained by reliable mechanisms which are not in the scope of this paper. We suppose we can get an absolute reference as a couple of (X, Y) co-ordinates, indicating the user position from where he/she invokes the service. In practice, space modelling by mean of linear constraints is sufficient for many cases [7]. Within the header, we can for example restrict access only if the user is located within the hospital or the building.

All simple roles defined in the header are combiable via conjunctions and disjunctions, in order to create complex roles, modelling access control constraints based on the transverse aspects of the profile, time and space at the same time.

Functional Roles {
Roles : nurse, doctor, day_nurse, night_nurse;
Hierarchy : day_nurse << nurse, night_nurse << nurse;
}

Seniority Roles {
Roles : manager, head, assistant;
Hierarchy : manager << head << assistant;
}

Contextual Roles {
Hospital_enclosure = (position(X,Y)
and X>10 and X<50 and Y<10 and Y>30);
First_shift = (hour(H) and H>=4 and H<12);
Second_shift = (hour (H) and H>=12 and H<20);
Third_shift = (hour (H) and H>=20)
or (hour(H) and H<4));
}

3.2 The Body

In LORAAM, the body part allows the expression of access authorizations at the method level. This is made possible using the auth keyword, followed by an appropriate logical formula. The authorization logical formulae condition access to each method, according to the roles defined in the header. These access authorizations can model access control rules defined in the confidentiality policy.

```cpp
Class CElectronicPatientRecord {
Public:
  contact getPatientContact()
  auth (doctor or nurse);
  string getLastPrescription() 
  auth (doctor or nurse);
  string getPrescriptionHistory()
  auth (doctor or (nurse and head));
  string getCareHistory()
  auth (doctor or (nurse and head));
  void setPrescription(string prescription)
  auth (doctor and Hospital_enclosure);
  void setLastCare(hour h, string care)
  auth ((day_nurse and first_shift)
  or (day_nurse and second_shift)
or (night_nurse and third_shift));
  /* This authorization prevent day nurse from 
  filling the LastCare field of the e-Patient 
  record during night and night nurse during the day */
```

4 Functional aspects

As the access control we propose is defined at the class level, the following statements hold:

- for confidentiality-critical applications, access control authorizations should be taken into account from the very start of an information system design cycle [16]. We do think that it does not have to be postponed until the end of the cycle,

- roles must be defined as soon as the requirement engineering stage [19, 11],

- roles and authorizations can only be static [3], as the class structure is modified, therefore recompiling is necessary. We consider that this is not necessarily a major problem, as the set of information defined in the header and authorizations are very static (ex: hierarchical levels, internal organisation, administrative responsibilities, etc.). However, no recompiling is necessary for dynamic user role assignment or revocation. Moreover, privilege delegation is possible between users.

4.1 The authorization decision

The principle of access control decision is as follows: when a method call is detected, the LORAAM engine checks if the dynamic user’s profile fulfills the method’s authorization policy. As described in section 2.3, our proposal do not include the management of users’ profiles: we suppose that a system storing role assignments, running sessions and providing contextual informations (e.g. time) exists. Once these informations are retrieved (a cache mechanism can be used to improve retrieval efficiency), the LORAAM engine can check if the requested access is granted. An architecture for such a context repository is described in [15].
The basic idea of the access control decision is based on logical implication. The user’s profile and the authorization policy of the requested method need to be transformed into first-order logic formulae:

1. each role is replaced by itself and the conjunction of all its parents roles. If two roles are set to be mutually inherited, they are considered as a same role,
2. each category \( c \) act as a predicate symbol. Each role \( r \) defined within \( c \) is replaced by atom \( c(r) \),
3. if a role is equivalent to a formula, then it is replaced by this formula,

Once theses transformations applied to both user’s profile and requested authorization formula, we need to add contextual informations to the user’s profile. These informations are obtained by mean of software/hardware tools such as LDAP, GPS, time clock, etc. and also translated into atoms. E.g. \( \text{hour}(18) \) or \( \text{position}(10, 23) \). Then if the user’s profiles (plus context) implies the authorization formula of the requested method, the method is invoked, else an catchable exception is raised.

### 4.2 Example of authorization decision

Let us suppose that a user, John, wants to access the \( \text{setLastCare()} \) method from his mobile device. John, who has previously identified himself on the information system, has the following profile:

\[
\text{functional(nurse)} \land \text{functional(night_nurse)} \land \text{hour}(H) \land H < 4 \lor (\text{functional(nurse)} \land \text{functional(night_nurse)} \land \text{hour}(H) \land H \geq 20) \lor (\text{functional(nurse)} \land \text{functional(day_nurse)} \land \text{hour}(H) \land H \geq 4 \land H < 12) \lor (\text{functional(nurse)} \land \text{functional(day_nurse)} \land \text{hour}(H) \land H \geq 12 \land H < 20)
\]

The LORAAM engine checks if the dynamic user profile logical formula implies this formula. As the user profile is \( \text{functional(nurse)} \land \text{functional(night_nurse)} \land \text{position}(150, 45) \land \text{hour}(23) \). Implication holds, therefore access is granted.

### 4.3 Implementation in an object–oriented framework

The first approach to implement LORAAM is to add a layer on an existing object–oriented language. Such a layer have to retrieve user’s profile and contextual informations from role–assignment database. This layer needs to implement the profile and authorization policy transformation into logical formulae (section 4.1). This framework allows software designers to integrate access control in a declarative manner, without worrying about the mechanisms involved in authorization decision.

A proof–of–concept pre–processor LORAAM to C++ has been implemented. For a developer, the LORAAM to C++ pre–processor is a “black–box” transforming his code into C++. The basic steps of the pre–processor are:

- the input is a loraam source file, as written by developers according to LORAAM grammar,
- the pre–processor includes the C++ framework files to the source code,
- the pre–processor parse the header of LORAAM and suppress it from the source file,
– the pre–processor analyse the body of LORAAM and transform authorization policies into logical formulae (according to the previously parsed header),

– the pre–processor add a call to the authorization decision mechanism (included from C++ framework files) at the beginning of each method,

– the output is a C++ source file, obtained from LORAAM source file once header and authorization policies are tranformed into calls to the framework.

4.4 Implementation role–based object–oriented model

In many class–based object–oriented systems the association between an instance and a class is exclusive and permanent. Therefore these systems have serious difficulties in representing dynamic evolution of objects over time. The problem is the most severe for OO databases in which objects are stored over long periods during which the entities evolve. Object–role oriented models intend to fill this shortcomings of object–oriented models by adding an orthogonal concept to classes: roles.

The authors of [22] describe an RBAC framework organized into 7 layers, as the OSI (Open Systems Interconnect) network stack is (from physical layer #1 to abstract application layer #7). The least abstract layer is the object layer, used by the directly higher one: objects handles. This second layer is used to keep the association between objects and roles. An object–role oriented model integrate directly such a layer: handles are no more needed, associations between roles and objects are handled in declarative manner in the object–role oriented paradigm.

The implementation of LORAAM in a role–based object–oriented model is possible if:

– the model integrates role hierarchy,

– the role hierarchy is independent of the class hierarchy,

– the model respects the Object Data Management Group (ODMG) standard (e.g. encapsulation, inheritance, polymorphism, etc.) to be compliant with an existing object–oriented language.

Section 5.2 survey previous works on integration of role mechanisms into object–oriented models. According to [2, 10] the LORAAM language can be reduced into the Samovar model. This model respects the ODMG standard and integrate roles into object–oriented model. In this model, class can be seen as “abstract roles container”, no method is directly associated to a class. Attributes are only associated to roles or combinations of roles (including conjunction and disjouction). These combinations are expressed in first–order logic formulae (the same fragment of FOL used to express authorization policies in LORAAM).

Thus, to reduce LORAAM in a role–based object–oriented model, we must carry a prior transformation step on access control policies. In LORAAM, each authorization is associated to an object’s method, roughly said as “policies are organized by permissions”. In order to reduce our language, we must infer on these policies to organize them by roles, rather than by permissions.

For example, assuming we are working with the example from section 3.2. Policies of the CElectronicPatientRecord class are organized by permissions:

– permission getPatientContact() is granted to roles (doctor or nurse),

– permission getLastPrescription() is granted to roles (doctor or nurse),

– permission getPrescriptionHistory() is granted to roles (doctor or (nurse and head)).

In order to be reduced in a role–based object–oriented model, we must organize policies by roles. Thus, the above example will be organized as is:

– role (doctor) is granted access to getPatientContact(), getLastPrescription() and getPrescriptionHistory(),

– role (nurse) is granted access to getPatientContact() and getLastPrescription()

– role (nurse and head) is granted access to getPrescriptionHistory().
Everybody who is assigned to several roles is granted all permissions assigned to each role, as permitted by role hierarchy. Thus, (nurse and head) is also granted access which (nurse) is granted.

5 Related work

Very few work focused on integrating of access control models within logical data models. This section survey related work on introduction of security in object-oriented models and on the role-object oriented paradigm.

5.1 Integration of access control in object–oriented systems

Many papers have described how to implement security mechanisms involving roles and contexts (e.g. [15, 3]) in a role–oriented system. Our goal is not to describe how role–based access control can be implemented with classes, but is to describe how (and which subset of) role–based access control mechanisms can be implemented in classes.

Integration of access control into OO systems has already been studied. The authors of [23] describe how to implement mandatory access control (MAC) in OO database systems. Roughly stated, MAC is the “military like model”, in which subjects and resources are associated to labels. Access is granted iff the subject’s label as least as high as the requested resource’s label. Commonly used labels are unclassified, confidential, secret and top-secret. However, MAC has been shown to be too much rigid for actual applications, particularly when multiple users with different profiles are working on the system.

A more recent approach in [5] integrate MAC to UML diagrams. Their framework bridges the gap between software engineers and an organization’s security. A very interesting contribution which is not limited to class diagram and intends to integrate MAC in use cases and sequence diagrams. This paper describes a logic data model, but we are working on a conceptual model integrating role–based authorizations (section 6).

Figure 2: Empirical solutions for role implementation

5.2 Extending object–oriented systems with roles

The object paradigm is a very expressive framework, largely used. According to [21], implementing object roles is a difficult task. Indeed, the multiplicity of roles and their lifecycle (creation, deletion) is incompatible with the hard constraints of class–based models: object identity, strong typing, etc.

This problem could be partly solved with multiple inheritance (figure 2) in an object programming language. But each combination of role must lead to create a new class, which leads to an explosion of the number of necessary classes. Moreover, their existence is only motivated by technical reasons and not by a modelling need.

Another solution is to create a structure of “handles” (figure 2) which corresponds to the desired multiple-role instances. The handle references several OIDs, each of them corresponding to a role played by this instance. This leads to a referencing problem and involves the use of message delegation. Moreover, “Jacques” would be only a “handle”, losing its encapsulation, and therefore not an object anymore.

The implementation of RBAC models in OO systems clearly points out that maintaining association between roles and classes can be a tough design challenge, particularly when dealing with role hierarchies. For example, [3, 4] describes a UML class–diagram to implement RBAC. Their framework includes “role” and “role instance” classes. Thus, software designer have to implement a mechanism to ensure that an object instance of “role” is linked with another object instance of “role instance”.

A review of role–based object models in the
programming and database areas can be found in [10, 2, 6]. However, these models are intended mainly to take into account the evolutive part of the objects during their life, but either they do not propose in general any access control primitive or they do not totally respect the standard paradigms of object programming (e.g. [24]).

5.3 Integration of security in role-oriented systems

To the best of our knowledge, the closest paper to ours is [24]. This approach intends to integrate a subset of RBAC into a role-oriented system: DOOR. This model permits modelling an owner relationship between roles and objects, but this approach does not respect the standard paradigms of object programming, particularly strong typing. Section 5.2 explain how LORAAM can be reduced in role-oriented systems, nevertheless in such implementation, roles can still be used for non role-based authorization purposes. The Samovar model [2] is well suited to include a basic form of RBAC because it includes role definition in a logic manner. These role formulae can easily capture permission-role assignments of RBAC.

The aspect-oriented paradigm, can be seen as an alternative to role-oriented one. Both of them aim at a more flexible use of objects in OO systems. The authors of [18] describe their approach based on an aspect-oriented modeling (AOM) technique that allows system developers to isolate cross-cutting design structures in aspects to support controlled evolution of the structures. This approach is very close to our since they use an “addition” on OO models to reflect access control model’s mechanisms, and explain how to deal with. We are investigating if LORAAM can be as easily reduced in an aspect-based model (such as AspectC++) as it is in a role-based model.

6 Discussion

Our proposal makes it possible to take into account RBAC access control to information systems straight into the logical object data model. We presented the generic LORAAM language, which contains two parts. The header allows specification of roles categories and hierarchies. The body part allows specification of authorizations at the method level, by use of logical connectors in order to build more complex ones. We also presented the functional part of LORAAM, which relies on a first-order logic engine.

Security is often divided into confidentiality, reliability and integrity. Confidentiality is the least considered non-functional requirements of security. Access control models, and nowadays role-based ones, are designed to enhance confidentiality. Integration of mandatory access control [5] in UML diagrams and security extensions for UML [12, 13] are promising. They will bridge the gap between a security will and its implementation. These conceptual and logical propositions can be core models for methodologies considering security as in integral part of the whole software design process such as [16]. We are currently working on automatic translation into LORAAM of UML diagrams expressed in specific security models. LORAAM can indeed be used as a target language for a CASE supporting a RBAC-based design method, such as SecureUML. We currently plan to validate this approach using our prototype, a LORAAM to C++ pre-processor, with the Foundstone SecureUML Visio template [1].

References


Annex: BNF Grammar of LORAAM’s header

Note: grammar of non–terminal symbol logical_formula is not included.

```plaintext
loraam_header := groups_list
groups_list := groups_list group |
              group := group_identifier 'Roles' '{' definitions_list '}'
              group_identifier := IDENTIFIER
definitions_list := definitions_list definition |
definition := role_definition |
              hierarchy_definition |
equivalency_definition
role_definition := 'Roles' ':' roles_list ';'
roles_list := role_identifier roles_list_suite
roles_list_suite := ',' role_identifier roles_list_next |
role_identifier := IDENTIFIER
hierarchy_definition := 'Hierarchy' ':' hierarchical_relations_list ';'
hierarchical_relations_list := hierarchical_relation |
hierarchical_relations_list_suite := ',' hierarchical_relation hierarchical_relations_list_suite |
hierarchical_relation := role_identifier '<<' role_identifier
equivalency_definition := equivalency_identifier := equivalency_identifier ':=' logical_formula ';'
equivalency_identifier := IDENTIFIER
 logical_formula := logical_formula ',' logical_conjunction |
logical_conjunction := logical_conjunction ',' |
logical_conjunction := logical_conjunction ',' |
logical_atom := logical_predicate |
logical_predicate := logical_predicate comparison |
logical_atom := logical_predicate |
logical_atom := logical_predicate |
```

logical_formula := logical_formula '(', logical_formula ')';