ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2014, vol. 2, no. 1., p. 1-11.
Available online at www.orb-academic.org

International Journal of Computer Science: Theory and Application
ISSN: 2336-0984

Distributed and Typed Role-based Access Control Mechanisms Driven by CRUD Expressions

Óscar Mortágua Pereira, Diogo Domingues Regateiro, Rui L. Aguiar

Instituto de Telecomunicações, DETI, University of Aveiro, Aveiro, Portugal.
Email: omp@ua.pt, diogoregateiro@ua.pt, ruilaa@ua.pt

ABSTRACT
Business logics of relational databases applications are an important source of security violations, namely in respect to access control. The situation is particularly critical when access control policies are many and complex. In these cases, programmers of business logics can hardly master the established access control policies. Now we consider situations where business logics are built with tools such as JDBC and ODBC. These tools convey two sources of security threats: 1) the use of unauthorized Create, Read, Update and Delete (CRUD) expressions and also 2) the modification of data previously retrieved by Select statements. To overcome this security gap when Role-based access control policies are used, we propose an extension to the basic model in order to control the two sources of security threats. Finally, we present a software architectural model from which distributed and typed RBAC mechanisms are automatically built, this way relieving programmers from mastering any security schema. We demonstrate empirical evidence of the effectiveness of our proposal from a use case based on Java and JDBC.

KEYWORDS
RBAC — access control — information security — software architecture — middleware — distributed systems — relational databases.

1. Introduction

Information systems are traditionally protected by several security measures, among them we emphasize: user authentication, secure connections and data encryption. Another relevant security measure is access control [1][2], which "*is concerned with limiting the activity of legitimate users*" [3]. In other words, access control regulates every users' requests to access sensitive resources, in our case data stored in relational database management systems (RDBMS). Most of these requests are from users running client applications that need to access data. When client applications, and mainly business logics, are built from tools such as ODBC [4], JDBC [5], ADO.NET [6], LINQ [7], JPA [8] and Hibernate [9], users' requests can be materialized through several techniques provided by those tools (herein known as access modes). Two of them are the most popular and, therefore, widely used: requests based on Create, Read, Update and Delete (CRUD) expressions encoded inside strings (this is the Direct Access Mode) and requests are trigged when content of local data sets (LDS) retrieved by Select expressions are modified and committed to the host database (this is the Indirect Access Mode). Figure 1, Figure 2 and Figure 3 present typical usages of JDBC, ADO.NET and LINQ, respectively. Similarly to the other tools, JDBC, ADO.NET and LINQ are agnostic regarding the schema of databases and also regarding the schema of access control mechanisms. Programmers can write any CRUD expression (line 40, 28, 17) and execute it (line 44, 33, 17). In these cases it is a Select expression and, therefore, a LDS is instantiated (line 44, 33, 17). Once again, programmers can read attributes (line 44, 35, 18), delete rows (line 51, -, -), update rows (line 53-54, 37-39, 20-21) and, finally, insert new rows (line 55-57, -, -). After being committed, these modifications are replicated in the host databases. There is no possibility to make programmers aware of any established schemas (database and access control policies). In situations where database schemas and/or security policies are complex, programmers can hardly write source in accordance with the established security policies. To overcome this situation we propose an extension to basic Role-Based Access Control (RBAC) policy [10], which has emerged as one of the dominant access control policies [11]. In our proposed model, a role comprises the required security information to supervise the direct and the indirect access modes. Through this security information and from a software architectural model, to be herein presented, distributed security components are automatically built to statically enforce the established RBAC policies. This way, programmers are relieved from mastering any schema.

This paper is organized as follows: section 2 presents the related work; section 3 presents our conceptual proposal; section 4 presents our implementation proposal; section 5 discusses some

aspects of the presented solution and, finally, section 6 presents the conclusion.

```
void useJDBC() throws SQLException {

    sql="Select * from Person p Where p.id=?";
    ps=conn.prepareStatement(sql,
            ResultSet.TYPE_FORWARD_ONLY,
            ResultSet.CONCUR_UPDATABLE);
    ps.setInt(1,id);
    rs=ps.executeQuery();
    if (rs.next() ) {
      grade=rs.getFloat("fName");
      // ... read more attributes
      rs.deleteRow();
      // ... code
      rs.updateFloat("grade",value);
      rs.updateRow();
      // ... code
      rs.moveToInsertRow();
      rs.updateString("fName",fName);
      // more attributes
      rs.insertRow();
    }
}
```

Figure 1. *Typical usage of JDBC.*

```
private void useADO()
{
 String sql="Select * from Products where productId=10";
 SqlDataAdapter da = new SqlDataAdapter();
 da.SelectCommand = new SqlCommand(sql,conn);
 SqlCommandBuilder cb = new SqlCommandBuilder(da);
 DataSet ds = new DataSet();
 da.Fill(ds, "Products");
 DataRow dr = ds.Tables["Products"].Rows[0];
 productName = (String) dr["productName"];
 // ... more code
 dr["productName"] = productName;
 cb.GetUpdateCommand();
 da.Update(ds, "Products");
 // ... more code
}
```

Figure 2. *Typical usage of ADO.NET.*

```
private void useLINQ()
{
 Product prd=dc.Products.Single(p=>p.CategoryID==10);
 productName=prd.ProductName;
 // ... more code
 prd.ProductName = productName;
 dc.SubmitChanges();
 // ... more code
}
```

Figure 3. *Typical usage of LINQ.*

2. Related Work

This paper is an extension of [12]. The authors of this paper have also been addressing the research issue of this paper for some time [13][14][15]. While in [13] the focus is centered on reusable business tier components, in [14][15] the presented works deal with the direct and the indirect access modes, but none of them is focused on how to enforce RBAC policies based on CRUD expressions. The work presented in [15] can be seen as the first step to achieve the objectives of the work presented in this paper. Basically, it deals with CRUD expressions and also with both access modes but does not address how to relate CRUD expressions and policies based on RBAC. The work presented in [15] also leverages [14] but it is mainly focused on addressing a different security key aspect: the enforcement of access control policies to the runtime values used on the direct and on the indirect access modes.

For the best of our knowledge no similar work has been or is being done around distributed and typed RBAC driven by CRUD expressions. Therefore, in the remaining of this section we present some of the most relevant works around access control for relational database applications.

Chlipala et al. [16] present a tool, *Ur/Web*, that allows programmers to write statically-checkable access control policies as CRUD expressions. Basically, each policy determines which data is accessible. Then, programs are written and checked to ensure that data involved in CRUD expressions is accessible through some policy. To allow policies to vary by user, queries use actual data and a new extension to the standard SQL to capture '*which secrets the user knows*'. This extension is based on a predicate referred to as '*known*' used to model which information users are already aware of to decide upon the information to be disclosed. The validation process takes place at compile time, this way not relieving programmers from mastering database schemas and security policies while writing source code.

Abramov et al. [17] present a complete framework that allows security aspects to be defined early in the software development process and not at the end. They present a model from which access control policies can be inferred and applied. Nevertheless, similarly to [16], the validation process takes place only at compile time, this way entailing programmers to master the established access control policies.

Zarnett et al. [18] present a different solution, which can be applied to control the access to methods of remote objects via Java RMI [19]. The server that hosts the remote objects uses Java Annotations to enrich methods and classes with metadata about the roles to be authorized to use them. Then, RMI Proxy Objects are generated in accordance with the established access control policies (they contain the authorized methods only). Fischer et al. [20] present a more fine-grained access control, which uses parameterized Annotations to assign roles to methods. These approaches, in contrast with our concept, do not facilitate the access to a relational database because the developers still need to have full knowledge of the database schema and also the authorized accesses to database objects. A similar approach was presented by Ahn et al. [21], where a tool is used to generate, from a security model, source code to check if there is any security violation. The verification process takes place only after writing the source code, this way not addressing the key aspects of our work.

Oracle, in the Oracle DB, addressed access control by in-

troducing the *Virtual Private Database* [22] technology. This technology is based on query-rewriting techniques, which means that CRUD expressions are rewritten before their execution and in accordance with the established access control policies. Authorization policies are encoded into functions defined for each relation, which are used to return *where* clause predicates to be appended to CRUD expressions, this way limiting data access at the row level. Virtual Privacy Database is an alternative to database views by avoiding some of their drawbacks such as the need for an additional view for each additional policy. With the Virtual Private Database technique, the same CRUD expression is shared by all users and automatically modified in accordance with permissions of each user.

LeFevre et al. [23] propose a technique to control the disclosing data process in Hippocratic databases. The disclosing process is based on the premise that the subject has control over who is allowed to see its protected data and for what purpose. It is based on the query rewriting technique. Policies are defined using P3P [24] or EPAL [25] and comprise a set of rules that describe to whom the data may be disclosed and how the data may be used. Two disclosure models are supported for cells: at the table level - each purpose-recipient pair is assigned a view over each table in the database and prohibited cells are replaced with null values; at the CRUD expressions level - protected data are removed from the returned relations of Select expressions, in accordance with the purpose-recipient constraints. Rules are stored as meta-data in the database. CRUD expressions must be associated with a purpose and a recipient, and are rewritten to reflect the ACP.

SELINKS [26] is a programming language in the type of LINQ and Ruby on Rails which extends LINKS [27] to build secure multi-tier web applications. LINKS aims to reduce the impedance mismatch between the three tiers. The programmer writes a single LINKS program and the compiler creates the byte-code for each tier and also for the security policies (coded as user-defined functions on RDBMS). Through a type system object named as Fable [28], it is assured that sensitive data is never accessed directly without first consulting the appropriate policy enforcement function. Policy functions, running in a remote server, check at runtime what type of actions users are granted to perform. Programmers define security metadata (termed labels) using algebraic and structured types and then write enforcement policy functions that applications call explicitly to mediate the access to labeled data.

Rizvi et al. [29] present a query rewriting technique to determine if a CRUD expression is authorized but without changing the CRUD expression. It uses security views to filter contents of tables and simultaneously to infer and check at runtime the appropriate authorization to execute any CRUD expression issued against the unfiltered table. The user is responsible for formulating the CRUD expression properly. They call this approach the Non-Truman model. Non-Truman models, unlike Truman models, do not change the original CRUD expression. The process is transparent for users and CRUD expressions are rejected if they do not have the appropriate authorization. The transparency of this technique is not always desirable particularly when it is important to understand why authorization is not granted so that programmers can revise their CRUD expressions more easily.

Morin et al. [30] use a security-driven model-based dynamic adaptation process to address access control and software evolution simultaneously. The approach begins by composing security meta-models (to describe access control policies) and architecture meta-models (to describe the application architecture). They also show how to map (statically and dynamically) security concepts into architectural concepts. This approach is mainly focused on how to dynamically establish bindings between components from different layers to enforce security policies. They did not address the key issue of how to statically implement dynamic security mechanisms in software artifacts, in our case business tiers based on CLI.

There are several other works related to access control: a distributed enforcement of the RBAC policies is proposed by Komlenovic et al. in [31]; a new technique and a tool to find errors in the RBAC policies are presented by Jayaraman et al. in [32] and, finally, Wallach et al. in [33] propose new semantics for stack inspection that addresses concerns with the traditional stack inspection, which is used to determine if a dangerous call (e.g. to the file system) is allowed.

3. Our Proposal: Conceptual Perspective

Access control is usually implemented in a three phase approach [1]: security policy definition, security model to be followed and security enforcement mechanisms. The organization of this section is also organized in three sub-sections, each one addressing one implementation phase.

3.1 Distributed RBAC Mechanisms

This paper is focused on distributed access control mechanisms. Therefore, before presenting the solution for their implementation it is advisable to clarify what are "*distributed access control mechanisms*".

Access control mechanisms are entities that act at runtime and, therefore, before advancing with deployment architectures it is important to find out if they can be represented by a general model. From the surveyed commercial and scientific literature, we can state that independently from any technique or solution, access control mechanisms can always be represented by two distinct processes: the enforcement process and the decision process. Figure 4 presents a simplified block diagram for access control mechanisms and their internal and external interactions. The basic operation is as follows: (1) client-side applications request the enforcement process to access to a protected resource; (2) enforcement process asks the decision process to evaluate if the request is authorized; (3) the decision process answers; (4) if authorization is denied, the client application is notified; (5) if authorization is granted, the request is executed by the enforcement process and, finally (6) the result is delivered to client-side applications. This block diagram and its operation is clearly the approach followed by XACML. Anyway, as we will see, it can also be used as the basic block diagram to represent other solutions. We have also intentionally used a similar XACM terminology (enforcement and decision) in order not to introduce

a new complete and different one. The enforcement process is the process being used to enforce a decision about granting or denying the access to the protected resource. The decision process is the process being used to decide if the authorization to access a protected resource is granted or denied. XACML Policy Enforcement Points and XACM Policy Decision Points are the equivalent entities for our enforcement and decision processes, respectively.

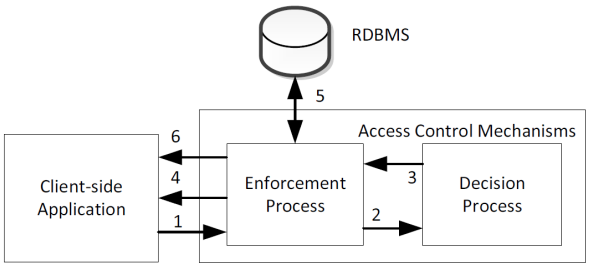

Figure 4. *Access control mechanisms block diagram and their interactions.*

Several architectural solutions are available to implement access control mechanisms. Some are provided by vendors of RDBMS and others have been proposed by the scientific community. Basically there are three main architectural solutions: 1) centralized architectures (such as: vendors of RDBMS, the use of views and parameterized views [34], query rewriting techniques [22][23][29][35][36], extensions to SQL [16][37]); 2) distributed architectures [14][15][38][39][40] and 3) mixed architectures [26][41].

In this paper we are interested in the distributed architecture only, which is presented in Figure 5.

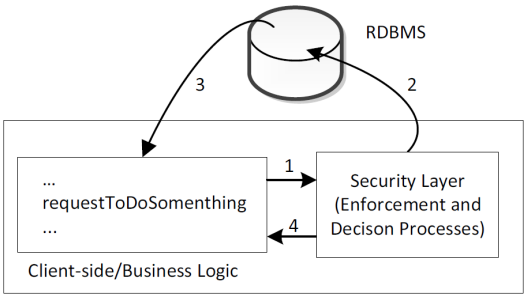

Figure 5. *Block diagram for the distributed architecture.*

In the distributed architecture, decision processes and enforcement processes are both locally managed on client-side systems, see Figure 5. A security expert crafts a security layer to be deployed in each client system. Then, every request to access the RDBMS is captured by the security layer (1) to evaluate if authorization is granted or denied. If granted, the request is sent to the RDBMS (2) and results returned to client-side applications (3). In case of being denied, the request is prevented from being well succeeded (4).

3.2 RBAC Policy

In this sub-section we present an extension to the basic RBAC policy that will be used to supervise requests to access data stored in Relational Database Systems (RDBMS). The extension is aimed at defining new properties to be supported by RBAC policies. Traditionally, among other concepts, RBAC policies comprise: users, roles (they can be hierarchized), permissions, delegations and actions. Basically, legitimate (authenticated) users can only execute some action if he has been authorized to play the role that rules that action. In the end, in the last final stage, actions are the four main operations on database objects (tables and views): select, insert, update and delete. Depending on the granularity, these actions can be defined at the level of database objects, at the level of columns, at the level of rows and at the level of cells. There are several approaches to authorize or deny these actions, among them: constraints can be defined directly on database objects and also by using query re-writing techniques. In our case actions are formalized by what can be done on the direct and on the indirect access modes. In other words, actions are the CRUD expressions that can be used (direct access mode) and also the operations that can be done on LDS (indirect access mode). The granularity of the direct access mode is defined by each CRUD expression. The granularity of the indirect access mode must be defined at the protocol level (read, insert, update and delete) and also at the attribute level (except for the delete protocol, which is always at the row level). The granularity at LDS level provides a full control to define which protocols are to be made available. This granularity when combined with the granularity at the attribute level provides, for each LDS, the full control to define which attributes are to be made available for each protocol. In terms of cardinality, we define that each role comprises a set of un-ordered CRUD expressions.

3.3 RBAC Model

In this sub-section we present a model to formalize the extension to be supported by the RBAC policy. The extension can be formalized by several approaches, depending on the practical scenarios where they are going to be used to enforce RBAC policies. The model herein presented is tailored to scenarios where a tool is available to help and minimize the effort in defining the policies to be enforced. We start by analyzing CRUD expressions because every access to data starts through the direct access mode and only then the indirect access mode can be used (only with Select expressions). Each CRUD expression type (Select, Insert, Update and Delete) can be expressed by general schemas but each individual CRUD expression is represented by specializing one of the general schemas. During the assessment we made to Call Level Interfaces (CLI), in which JDBC is included, we found out that the schema of each expression type can be built from a small set of smaller schemas. The functionalities expressed by the smaller schemas are: only Select expressions return relations; all CRUD expressions types can use runtime values for clause conditions; some CRUD expressions return the number of affected rows (Insert, Update and Delete) and, finally, some CRUD expressions use runtime values for column values (Insert and Update). We can also elicit other perspectives for LDS, such as some LDS are

scrollable (there are no restrictions on choosing which row is the next selected row) while others are forward-only (only the next row can be selected). To address this bundle of different smaller schemas, the schema needs to be flexible and adaptable. This challenge is addressed through the design of entities, herein referred to as Business Schemas. Business Schemas are responsible for hiding the actual direct and indirect access modes and also for providing new direct and indirect access modes driven by access control policies. Additionally, after some research we came to the conclusion that the relationship between Business Schemas and CRUD expressions is many to many. This means that one Business Schema can manage one or more CRUD expressions and one CRUD expression can be managed by one or more Business Schemas. Now we give one example for each case. Let us consider the next two Select expressions:

1. Select * from table;
2. Select * from table where col>10;

First we analyze the direction "*one Business Schema -> many CRUD expressions*". From the direct access mode perspective, there is no difference between the two expressions. Both are Select expressions and both have zero runtime values. Additionally, the schema of the returned relations is equal in both cases. Then, the same Business Schema can be shared by both expressions if the security policy to be applied on the indirect access mode is the same for both cases. Now we analyze the direction "*one CRUD expression -> many Business Schemas*". This case is simpler to explain. We can use any of the two Select expressions. In cases where different security policies are applied to the same Select expression, then we can use it in in more than one Business Schema. For example, the same CRUD expression is managed by two Business Schemas where the updated protocol is provided only in one of them. Finally, Figure 6 presents the general extension to be included in concrete RBAC models. This extension does not need to be exactly as presented. The only important issues are the relationships and cardinalities between roles, Business Schemas and CRUD expressions. By this we mean that it is not compulsory to keep them adjacent as presented. Other entities can be included between them. Moreover, the policies to be followed to authorize or not to authorize roles are also out of scope of this paper. It is up to the security expert to decide the granting and the denying models to be followed.

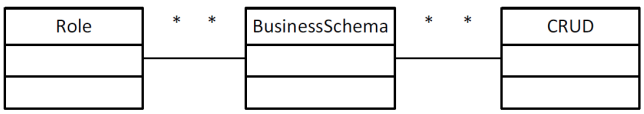

Figure 6. *Extension for the RBAC model.*

3.4 Software Architectural Model

In this sub-section we present the software architectural model, shown in Figure 7, for building the enforcement mechanisms from the extended RBAC model. The presented architectural model represents the implementation of one role. It is up to each system architect to decide how to expand it to support several roles. Moreover, it is focused on how to implement RBAC mechanisms and not how to build complete and feasible implementations. For example, the architectural model does not address key issues such as the scrolling policy on LDS and database transactions. These and other issues are out of the architectural model context. We start by describing the Business Schema interface, herein known as IBusinessSchema, which is the most complex entity. From it we will present and describe the architectural model. This interface, as we can infer from what has been already presented, needs to cope with the two access modes. The functionalities to be provided depend mainly on the CRUD expressions type and on the necessary runtime values. This is translated into the architectural model this way: IBusinessSchema extends two interfaces IDAC (direct access mode) e IIAM (indirect access mode).

IDAC

This interface manages the direct access mode. Depending on the type of CRUD expressions and on the runtime values, it can extend 1, 2 or 3 interfaces:

- IExecute - This interface is mandatory. It is responsible for the execution of CRUD expressions of any type and also for setting the runtime values for clause conditions.
- ISet – This interface is used with Insert and Update expressions when there is the need to set runtime values for columns.
- IRows – This interface is used only with Update, Insert and Delete expressions to notify applications about the number of affected rows

IIAM

This interface manages the indirect access mode. Depending on the mechanisms to be implemented, it can extend at most four interfaces:

- IRead – This interface is mandatory. It can comprise services to read any sub-set of attributes of returned relations.
- IUpdate – This interface is only available if the established access control policies authorize the attributes of LDS to be updated. In this case, only the updatable attributes can be updated.
- IInsert - This interface is only available if the established access control policies authorize the insertion of new rows on LDS. In this case, only the insertable attributes can be inserted.
- IDelete – This interface is only available if the established access control policies authorize the rows of LDS to be deleted.

Regarding the relation between Business Schemas and, Roles and CRUD expressions, we can see from Figure 7 that the architectural model is consistent with the RBAC model. Please remember

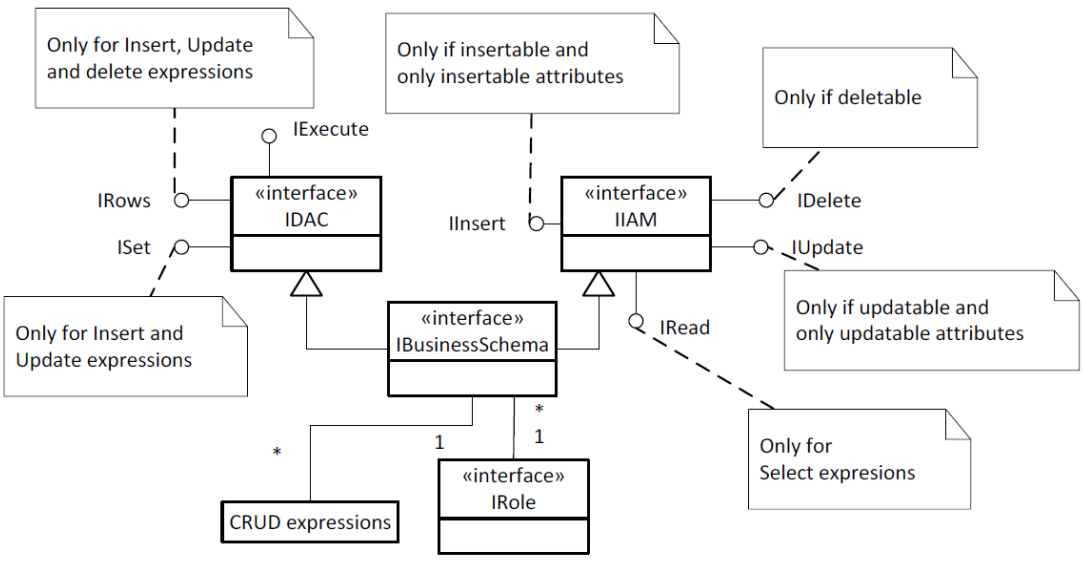

Figure 7. *Extension for the RBAC model.*

that the architectural model represents the implementation of one role only. The model says that one role comprises one or more Business Schemas and each Business Schema comprises one or more CRUD expressions. From the presented architectural model and also from the RBAC model, security components can be automatically built, see Figure 8. To achieve this goal, a tool is necessary to automate the process. It is not part of our proposal but the tool is a key component to transform modeled RBAC policies into security components.

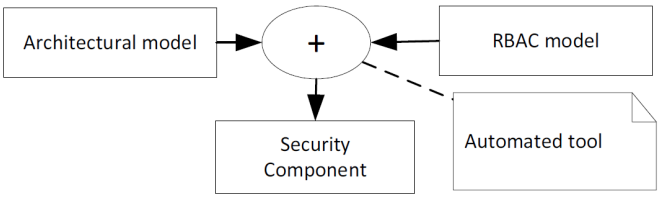

Figure 8. *Automated building process of security components.*

4. Our Proposal: Implementation Perspective

In this section we present our implementation perspective, which consists of several different components, as shown in Figure 9. Next we present a brief explanation about our proposal. Then, we will present a more detail explanation about each component.

The Policy Server is a relational database that contains a realization of the proposed extension to the RBAC model. The Policy Extractor is an automated tool responsible for building automatically Security Data Structures. These data structures are built from data extracted from the Policy Server and also from the software architectural model. The data structures are responsible for relieving programmers from mastering any database schema and any RBAC policy while they are writing source code. The Security Layer is responsible for implementing the access control mechanisms (enforcement and decision processs). It comprises a component, herein known as the Security Engine, that builds the mechanisms at runtime from Policy Server and also from the software architectural model. These mechanisms effectively control users' requests, at runtime, when they issue requests through the direct and the indirect access modes. The Policy Manager is a component that receives commands from the Security Engine and retrieves the relevant information from the Policy Server, e.g. the list of Business Schemas authorized for the client application's role. The Policy Watcher is a DLL that resides in the DBMS and allows the Policy Server database to send messages when changes occur to the established access control policies. These messages are then resent to the Policy Manager, which updates the access control mechanisms of the affected clients.

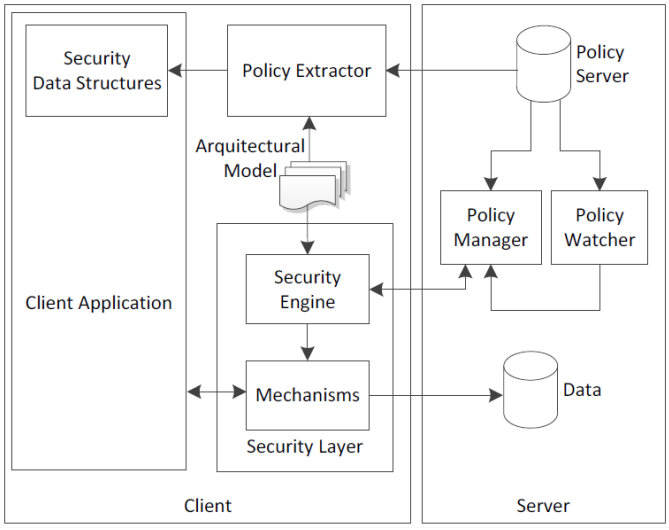

Figure 9. *Implementation architecture.*

4.1 Policy Server

The Policy Server contains a realization of the proposed extension (shown in Figure 6) for a simplified RBAC model, see Figure 10. Our model uses some of the most relevant features of RBAC models: subjects (users), applications, sessions, permissions and delegations. A user can play a role only if that role is explicitly authorized (permitted or delegated) to him when he is running a session of an application. Permissions and delegations can be dynamically modified at runtime. CRUD expressions are kept in Crd_crud and Business Schemas are stored as Java interfaces (based on the architectural model) in Bus_BusinessSchema. This method of storage is not mandatory. Business Schemas can be represented in any other metadata model. Additionally, the Policy Server contains triggers to wake up the Policy Watcher whenever changes occur in the established access control policies (when Business Schemas are added/deleted from roles or delegations are created/deleted).

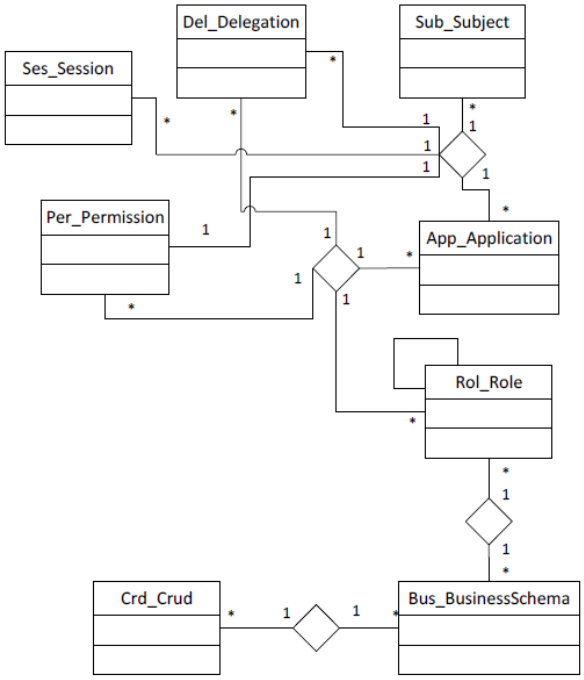

Figure 10. *Simplified security RBAC model.*

4.2 Policy Watcher

The Policy Watcher is a DLL that resides in the DBMS. It is responsible for dealing with the triggers that have been put on the Policy Server. These triggers are responsible for detecting modifications being taken at the level of the established access control policies. This is very important because whenever a modification takes place, it is necessary to update the enforcement and decision processes of client systems affected by the modification. Basically, the Policy Watcher sends a message to the server component responsible for the updating process (Policy Manager).

4.3 Policy Manager

Policy Manager is a component that runs in the server side and handles requests from two different sources: the Security Layer of the client systems and from the Policy Watcher. Messages from clients systems alert the Policy Manager about a client requesting information for its enforcement and decision processes (Mechanisms). The Policy Manager identifies the application and the user profile and from them it knows which security metadata to send to the client system. Messages from the Policy Watcher alert the Policy Manager about modifications in policies. In this case, the Policy Manager determines the list of client systems affected by the modifications and it sends to each client system the list of changes to be implemented in the enforcement and decision processes. Figure 11 shows some of the interfaces implemented by the Policy Manager. For example, the *handleDelegationDelete()* is a method called when the Policy Watcher informs the Policy Manager that a role delegation is no longer in effect, and the clients of the affected role will have the related authorizations revoked.

```
132  handleDelegation(String input)                    void
133  handleDelegationDelete(String[] inputFields)      void
134  handleDelegationInsert(String[] inputFields)      void
135  handleDelete(String input)                        void
136  handleEnd(String input)                           void
137  handleGetBEUrl(String input)                      void
138  handleGetBEsIDs(String input)                     void
139  handleGetBus()                                    void
140  handleGetCRUDs(String input)                      void
141  handleGetJar(String input)                        void
142  handleGetJarForInfo(String input)                 void
143  handleGetRoles(String input)                      void
```

Figure 11. *Policy Manager handle methods.*

4.4 Policy Extractor

In this subsection we will present the Policy Extractor, which is responsible for building automatically the Security Data Structures to convey a complete awareness of the security mechanisms to programmers. We have implemented two different Policy Extractors: one as a standalone application and other based on Java annotations. Independently from the used technique, programmers are always provided with the same Security Data Structures. In our implementation, Security Data Structures are Java interfaces that formalize roles and mechanisms to be implemented on both direct and indirect access modes. Figure 12 shows the data structures for a role identified by *Role_IRole_B1* (line 7). This role is defined as a Java interface, as previously mentioned, that extends the role *Role_IRole_A*. We use this Java property to allow hierarchization of roles. Beyond extending the role *Role_IRole_A*, *Role_IRole_B1* comprises two Business Schemas: *i_orders* (9-10) and *s_customers* (15-16). The first Business Schema manages one CRUD expression identified by *i_orders_I_Orders_withCustomerID* (line 11-12) and the second manages *s_customers_S_Customer_all* (line 17-18). Again, these Business Schemas are formalized through Java interfaces. From these data structures (some not explicitly shown) programmers write source code as the one shown in Figure 13. From this figure

we can see that the Business Schema *Role_IRole_B1.s_customers* is instantiated for a user playing the role B1 (line 53). The CRUD expression is selected by selecting one supported by the selected Business Schemas (line 54). In this case the CRUD expression is identified by the integer *Role_IRole_B1.s_customers_S_Customers_all*. A runtime value is set for a clause condition (line 55) and the CRUD expression is executed (line 55) (this is the direct access mode). Programmers continue to be aware of the policies on the indirect access mode (line 56-68). Some readable, updatable (with prefix u) and insertable (with prefix i) attributes are shown. As a final note, in our implementation, CRUD expressions are identified by integers, this way hiding information about database schemas. This aspect can be very relevant in critical database applications where schemas of databases need to be hidden. CRUD expressions only exist at the level of Security Layers.

Finally, Figure 14 shows the example presented in Figure 1 but now based on our proposal. Unlike Figure 1, now programmers are completely aware of constraints enforced by mechanisms, being relieved from mastering any schema.

```
7  public abstract interface Role_IRole_B1 extends Role_IRole_A {
8      // I_Orders Business Schema and related CRUDs
9      public static final java.lang.Class<II_Orders>
10             i_orders = II_Orders.class;
11     public static final int
12             i_orders_I_Orders_withCostumerID = 1;
13
14     // S_Customers Business Schema and related CRUDs
15     public static final java.lang.Class<IS_Customers>
16             s_customers = IS_Customers.class;
17     public static final int
18             s_customers_S_Customers_byCountry = 2;
19 }
```

Figure 12. *Implemented security data structures.*

```
53     S_Cust = ss.businessService(Role_IRole_B1.s_customers,
54         Role_IRole_B1.s_customers_S_Customers_byCountry);
55     S_Cust.execute(country);
56     S_Cust.
57         iPhone (String arg0)          void
58         iPostalCode (String arg0)     void
59         iRegion (String arg0)         void
60         Phone ()                    String
61         PostalCode ()               String
62         Region ()                   String
63         uAddress (String arg0)        void
64         uCity (String arg0)           void
65         uContactName (String arg0)    void
66         uContactTitle (String arg0)   void
67         uCountry (String arg0)        void
68     Dot, semicolon and some other keys will also close this lookup ar
```

Figure 13. *Environment conveyed to programmers.*

4.5 Security Layer

Our security layer comprises three sub-components: 1) a general manager (not shown in Figure 9), which is responsible for providing client applications with standard interfaces to access internal functionalities; 2) Security Engine, which is responsible

```
100    S_Cust = ss.businessService(Role_IRole_B1.s_customers,
101        Role_IRole_B1.s_customers_S_Customers_byCountry);
102    S_Cust.execute(country);
103    if(S_Cust.moveNext()) {
104        custName= S_Cust.CustomerName();
105        // read more attributes
106        // ... code
107        S_Cust.beginUpdate();
108        S_Cust.uCustomerName(custName);
109        // update more attributes
110        S_Cust.updateRow();
111        // ... code
112        S_Cust.beginInsert();
113        S_Cust.iCustomerName(custName);
114        // insert more attributes
115        S_Cust.endInsert(true);
116        // ... code
117        S_Cust.deleteRow();
118    }
```

Figure 14. *Example of Figure 1 based on our proposal.*

for building at runtime the necessary access control mechanisms (enforcement and decision processes), always in accordance with the established policies for the running user profile and, finally, 3) the Mechanisms (enforcement and decision processes), which comprise: classes that implement Business Schemas and also the authorized CRUD expressions. Unlike Security Data Structures, these mechanisms implement the necessary source code to supervise requests issued through both access modes. If any mismatch exists between what users want to request and the implemented policies, runtime exceptions are raised. In our implementation, security layers provide generic type safe methods to allow application tiers to instantiate Business Schemas and execute CRUD expressions, see Figure 13 (line 53-54). These methods look up in local libraries for the requested Business Schemas and CRUD expressions and, if found, classes that implement the requested Business Schemas are instantiated through reflection. If they are not found, it means that that user, for some security reason, is no longer authorized to play that role. In this case, an exception is raised.

5. Discussion

The approach herein presented was successfully evaluated against the objective initially defined. There are other relevant issues that also deserve to be discussed, although they are not key aspects of this work. As such, a brief description is presented about eight different aspects: scalability, maintainability, autonomic computing, configurability, usability, applicability, separation of concerns and trustworthy.

- *Scalability*: Unlike some approaches to implement access control mechanisms, such as those based on the centralized and mixed architectures, their implementation in our proposal is completely distributed. Each client application is responsible for two fundamental aspects: to decide upon granting or denying the access to protected data and to enforce the decision. There is no central system interfering in this process. It is completely distributed.

- *Maintainability*: Security layers are automatically built and updated at runtime. This is clearly different from what happens with other approaches where maintenance activities are required at the level of client systems whenever modifications occur at the level of business logics or security requirements.

- *Autonomic Computing*: An autonomic system is characterized by making decisions on its own. It permanently checks the context and, based on policies, it automatically adapts itself. Our proposal is not an autonomic system but systems based on our proposal are easily integrated in autonomic systems. An autonomic system prepared to detect situations where policies need to be dynamically adapted can use our proposal to dynamically adapt the implemented mechanisms.

- *Configurability*: The configuration process of metadata is substantially automated if an enhanced tool similar to the one presented in [20][21] is used. The new tool would automatically create the required metadata from CRUD expressions. Moreover, the tool could also automate the process to obtain the basic metadata to access databases on a table basis as O/RM tools and LINQ do. Additionally, tools similar to those presented in [38] could also be used to validate the authorized CRUD expressions.

- *Usability*: tools similar to JDBC are very poor regarding their usability [20][21]. Our solution overcomes some of the most relevant aspects of their lack of usability. For example, unlike JDBC, our solution transforms runtime errors of getter and setter methods into compile errors.

- *Applicability*: JDBC was the main API used in our solution. In order to evaluate the possibility of using other tools than JDBC, a successful attempt was achieved with ADO.NET. The implementation in ADO.NET was mainly carried out to evaluate if the main aspects of the software architectural model are flexible enough to be used with different middlewear tools and frameworks. There were some technical implementation aspects that needed some adjustments but the final result is a fully functional security layer based on ADO.NET. Nevertheless, some paradigms, such as O/RM, can be used but should not be considered as an option. O/RM tools are mostly oriented to handle database tables as entity classes which is too restrictive to most database applications. CRUD expressions can also be handled by O/RM tools but that is not the focus of O/RM.

- *Separation of Concerns*: the architecture here presented, clearly separates the roles played by programmers of client systems from roles played by security experts. Security experts act at the level of the policy model while programmers act only at the level of application tiers. Eventually, for some organizational reasons, the two roles can be played by the same person or group of persons during the development process. Anyway, security experts can always have the last word by inspecting and validating the content of security models, which can be an automated process.

- *Trustworthy*: From a security perspective, our solution, in this current version, by itself cannot be used in practice. We emphasize that it is not aimed at providing a reliable access control. It is aimed at easing programmers work during the development process of client systems in database applications protected by access control policies.

6. Conclusion and Future Work

In this paper we addressed the key issue of easing programmers work when they develop source code for client systems of relational database applications with complex schemas and/or complex access control policies. A solution was presented for distributed and typed RBAC policies when programmers use tools, such as JDBC, Hibernate, ADO.NET. We started by defining an extension to traditional RBAC policies, then we defined the respective extension to traditional models and, finally, we described how to enforce policies. In our solution, each role comprises a set of CRUD expressions and the authorized actions on LDS of each Select expression. Thus, access control mechanisms act at the level of the direct and also at the level of the indirect access modes, this way covering the two most used access modes. A proof of concept based on JDBC was also presented. From it, we can realize that programmers are now relieved from mastering not only any RBAC policy but also any database schema. Access control mechanisms are automatically built and statically implemented at the level of business logics of relational database applications.

The work here presented cannot by itself ensure that the mechanisms are completely safe. We have already developed a new security layer, which seats above this one, to ensure that the distributed mechanisms are completely secure. This work is already concluded and ready to be published in the near future.

References

[1] SAMARATI, Pierangela et DE VIMERCATI, Sabrina Capitani. Access control: Policies, models, and mechanisms. In : *Foundations of Security Analysis and Design*. Springer Berlin Heidelberg, 2001. p. 137-196.

[2] DE VIMERCATI, Sabrina De Capitani, FORESTI, Sara et SAMARATI, Pierangela. Recent Advances in Access Control - Handbook of Database Security. In : *Handbook of Database Security*, Gertz, M. et Jajodia, S., Eds. Springer, 2008. p. 1–26.

[3] SANDHU, Ravi S. et SAMARATI, Pierangela. Access control: principle and practice. *Communications Magazine*, IEEE, 1994, vol. 32, no 9, p. 40-48.

[4] Microsoft, *Microsoft Open Database Connectivity*, 1992. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.

[5] PARSIAN, Mahmoud. *JDBC Recipes: A Problem-Solution Approach*. NY, USA: Apress, 2005.

[6] CASTRO, Pablo, MELNIK, Sergey, et ADYA, Atul. ADO. NET entity framework: raising the level of abstraction in data programming. In : *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007. p. 1070-1072.

[7] MEIJER, Erik, BECKMAN, Brian, et BIERMAN, Gavin. Linq: reconciling object, relations and xml in the. net framework. In : *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006. p. 706-706.

[8] YANG, Daoqi. Java Persistence with JPA. Outskirts Press, 2010.

[9] O'NEIL, Elizabeth J. Object/relational mapping 2008: hibernate and the entity data model (edm). In : *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008. p. 1351-1356.

[10] SANDHU, Ravi S., COYNE, Edward J., FEINSTEIN, Hal L., et al. Role-based access control models. *Computer*, 1996, vol. 29, no 2, p. 38-47.

[11] FUCHS, Ludwig, PERNUL, Günther, et SANDHU, Ravi. Roles in information security–a survey and classification of the research area. *Computers & Security*, 2011, vol. 30, no 8, p. 748-769.

[12] PEREIRA, Óscar M., REGATEIRO, D. D., et AGUIAR, Rui L. Role-Based Access Control Mechanisms Distributed, Statically Implemented and Driven by CRUD Expressions. In : *ISCC'14 - 9th. IEEE Symposium on Computers and Communications*. 2014.

[13] PEREIRA, Óscar Mortágua, AGUIAR, Rui L., et SANTOS, Maribel Yasmina. Reusable Business Tier Components: based on CLI and driven by a single wide typed service. *International Journal of Software Innovation (IJSI)*, 2014, vol. 2, no 1, p. 37-60.

[14] PEREIRA, Óscar M., AGUIAR, Rui L., et SANTOS, Maribel Yasmina. ACADA - Access Control-driven Architecture with Dynamic Adaptation. In : *SEKE'12 - 24th International conference on Software Engineering and Knowledge Engineering*. 2012. p. 387–393.

[15] PEREIRA, Óscar Mortágua, AGUIAR, Rui L., et SANTOS, Maribel Yasmina. Runtime Values Driven by Access Control Policies Statically Enforced at the Level of the Relational Business Tiers. In : *SEKE'13 - International Conference on Software Engineering and Knowledge Engineering*. 2013. p. 1–7.

[16] CHLIPALA, Adam et IMPREDICATIVE, L. L. C. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In : *The USENIX Conference on Operating Systems Design and Implementation*. 2010. p. 105-118.

[17] ABRAMOV, Jenny, ANSON, Omer, DAHAN, Michal, et al. A methodology for integrating access control policies within database development. *Computers & Security*, 2012, vol. 31, no 3, p. 299-314.

[18] ZARNETT, Jeff, TRIPUNITARA, Mahesh, et LAM, Patrick. Role-based access control (RBAC) in Java via proxy objects using annotations. In : *Proceedings of the 15th ACM symposium on Access control models and technologies*. ACM, 2010. p. 79-88.

[19] *RMI-Remote Method Invocation*. [Online]. Available: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[20] FISCHER, Jeffrey, MARINO, Daniel, MAJUMDAR, Rupak, et al. Fine-grained access control with object-sensitive roles. In : *ECOOP 2009–Object-Oriented Programming*. Springer Berlin Heidelberg, 2009. p. 173-194.

[21] AHN, Gail-Joon et HU, Hongxin. Towards realizing a formal RBAC model in real systems. In : *Proceedings of the 12th ACM symposium on Access control models and technologies*. ACM, 2007. p. 215-224.

[22] ORACLE. *Using Oracle Virtual Private Database to Control Data Access*. 2011. [Online]. Available: http://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm#CIHBAJGI.

[23] LEFEVRE, Kristen, AGRAWAL, Rakesh, ERCEGOVAC, Vuk, et al. Limiting disclosure in hippocratic databases. In : *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004. p. 108-119.

[24] W3C. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*. 2002. [Online]. Available: http://www.w3.org/TR/P3P/.

[25] W3C. *Enterprise Privacy Authorization Language (EPAL 1.2)*. 2003. [Online]. Available: http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/.

[26] CORCORAN, Brian J., SWAMY, Nikhil, et HICKS, Michael. Cross-tier, label-based security enforcement for web applications. In : *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009. p. 269-282.

[27] COOPER, Ezra, LINDLEY, Sam, WADLER, Philip, et al. Links: Web programming without tiers. In : *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2007. p. 266-296.

[28] SWAMY, Nikhil, CORCORAN, Brian J., et HICKS, Michael. Fable: A language for enforcing user-defined security policies. In : *IEEE Symposium on Security and Privacy*, 2008. p. 369-383.

[29] RIZVI, Shariq, MENDELZON, Alberto, SUDARSHAN, Sundararajarao, et al. Extending query rewriting techniques for fine-grained access control. In : *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004. p. 551-562.

[30] MORIN, Brice, MOUELHI, Tejeddine, FLEUREY, Franck, et al. Security-driven model-based dynamic adaptation. In : *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010. p. 205-214.

[31] KOMLENOVIC, Marko, TRIPUNITARA, Mahesh, et ZITOUNI, Toufik. An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). In : *Proceedings of the first ACM conference on Data and application security and privacy*. ACM, 2011. p. 121-132.

[32] JAYARAMAN, Karthick, TRIPUNITARA, Mahesh, GANESH, Vijay, et al. Mohawk: abstraction-refinement and bound-estimation for verifying access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 2013, vol. 15, no 4, p. 18.

[33] WALLACH, Dan S., APPEL, Andrew W., et FELTEN, Edward W. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2000, vol. 9, no 4, p. 341-378.

[34] ROICHMAN, Alex et GUDES, Ehud. Fine-grained access control to web databases. In : *Proceedings of the 12th ACM symposium on Access control models and technologies*. ACM, 2007. p. 31-40.

[35] WANG, Qihua, YU, Ting, LI, Ninghui, et al. On the correctness criteria of fine-grained access control in relational databases. In : *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007. p. 555-566.

[36] BARKER, Steve. Dynamic meta-level access control in SQL. In : *Data and Applications Security XXII*. Springer Berlin Heidelberg, 2008. p. 1-16.

[37] CHAUDHURI, Surajit, DUTTA, Tanmoy, et SUDARSHAN, S. Fine grained authorization through predicated grants. In : *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007. p. 1174-1183.

[38] CAIRES, Luís, PÉREZ, Jorge A., SECO, João Costa, et al. Type-based access control in data-centric systems. In : *20th European conference on Programming Languages and Systems: part of the joint European conferences on theory and practice of software*. Springer Berlin Heidelberg, 2011. p. 136-155.

[39] ZHANG, D., ARDEN, O., VIKRAM, K., et al. Jif: Java + information flow (3.3). 2012. [Online]. Available: http://www.cs.cornell.edu/jif/.

[40] RIBEIRO, Carlos, ZUQUETE, Andre, FERREIRA, Paulo, et al. SPL: An Access Control Language for Security Policies and Complex Constraints. In : *Network and Distributed System Security Symposium*. San Diego, CA,USA. 2001. p. 89–107.

[41] OASIS. *XACML - eXtensible Access Control Markup Language*. 2012. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[42] PEREIRA, Oscar M., AGUIAR, Rui L., et SANTOS, Maribel Yasmina. CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms - an Enhanced Performance Assessment Based on a case Study. *Int. J. Adv. Softw.*, 2011, vol. 4, no. 1&2, p. 158–180.