Role-Based Access Control Mechanisms

Distributed, Statically Implemented and Driven by CRUD Expressions

Óscar Mortágua Pereira
Instituto de Telecomunicações
DETI – University of Aveiro
3810-193 Aveiro, Portugal
omp@ua.pt

Diogo Domingues Regateiro, Rui L. Aguiar
Instituto de Telecomunicações
DETI – University of Aveiro
3810-193 Aveiro, Portugal
{diogoregateiro, ruilaa}@ua.pt

***Abstract*— Most of the security threats in relational database applications have their source in client-side systems when they issue requests formalized by Create, Read, Update and Delete (CRUD) expressions. If tools such as ODBC and JDBC are used to develop business logics, then there is another source of threats. In some situations the content of data sets retrieved by Select expressions can be modified and then committed into the host databases. These tools are agnostic regarding not only database schemas but also regarding the established access control policies. This situation can hardly be mastered by programmers of business logics in database applications with many and complex access control policies. To overcome this gap, we extend the basic Role-Based Access policy to support and supervise the two sources of security threats. This extension is then used to design the correspondent RBAC model. Finally, we present a software architectural model from which static RBAC mechanisms are automatically built, this way relieving programmers from mastering any 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; distributed systems; middleware; databases.

I. INTRODUCTION

Information systems, including the ones of telecommunication operators, 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 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 presents a typical usage of JDBC. Similarly to the other tools, JDBC is agnostic regarding the schema of databases and also regarding the schema of access control mechanisms. Programmers can write any CRUD expression (line 100) and execute it (line 100-103). In this case it is a Select expression and, therefore, a LDS is instantiated (line 103). Once again, programmers can read attributes (line 105,106), update rows (line108-110), insert new rows (line 112-115) and, finally, delete rows (line 117). After being committed, these update, insert and delete protocols are replicated in the host database. 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, security components are automatically built to statically enforce the established RBAC policies. This way,

```
100    sql="SELECT * FROM dbo.Customers WHERE Country = ?";
101    ps=conn.prepareStatement(sql);
102    ps.setString(1, country);
103    lds=ps.executeQuery();
104    if(lds.next()) {
105        custName=lds.getString("CustomerName");
106        // read more attributes
107        // ... code
108        lds.updateString("CustomerName", custName);
109        // update more attributes
110        lds.updateRow();
111        // ... code
112        lds.moveToInsertRow();
113        lds.updateString("CustomerName", custName);
114        // insert more attributes
115        lds.insertRow();
116        // ... code
117        lds.deleteRow();
118    }
```

Figure 1. Typical usage of JDBC.

programmers are relieved from mastering any schema.

This paper is organized as follows: section II presents the related work; section III presents our conceptual proposal; section IV presents our implementation proposal; section V discusses some aspects of the presented solution and, finally, section VI presents the conclusion.

II. Related Work

Chlipala et al. [12] 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 assure 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. [13] 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 [12], the validation process takes place only at compile time, this way entailing programmers to master the established access control policies.

Zarnett et al. [14] present a different solution, which can be applied to control the access to methods of remote objects via Java RMI [15]. 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. [16] 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. [17], 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.

There are other works related to access control: a distributed enforcement of the RBAC policies is proposed by Komlenovic et al. in [18]; a new technique and a tool to find errors in the RBAC policies are presented by Jayaraman et al. in [19] and, finally, Wallach et al. in [20] 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. Our work complements these, regarding the access to relational databases, by generating static access control mechanisms automatically and accordingly with the established RBAC policies, this way relieving programmers from mastering them.

The works presented in [21][22] 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 [22] 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 both access modes but does not address how to relate CRUD expressions and policies based on RBAC. The work presented in [22] also leverages [21] 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.

III. 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.

A. RBAC Policy

In this sub-section we present an extension to the basic RBAC policy that is 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. At the end, actions are the four main operations on database objects (tables and views): read, 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 and also the operations that can be done on LDS. 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, each role comprises a set of un-ordered CRUD expressions.

B. RBAC Model

In this sub-section we present a model to formalize the extension presented to the RBAC policy. The extension can be formalized by several approaches, depending on the practical scenarios where they are going to be used. 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 up 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 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 2 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.

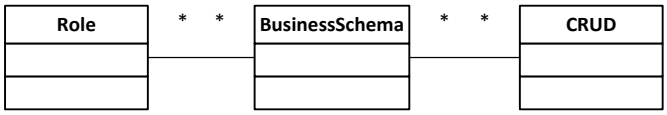

Figure 2. Extension for the RBAC model.

C. Software Architectural Model

In this sub-section we present the software architectural model, shown in Figure 3, 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.

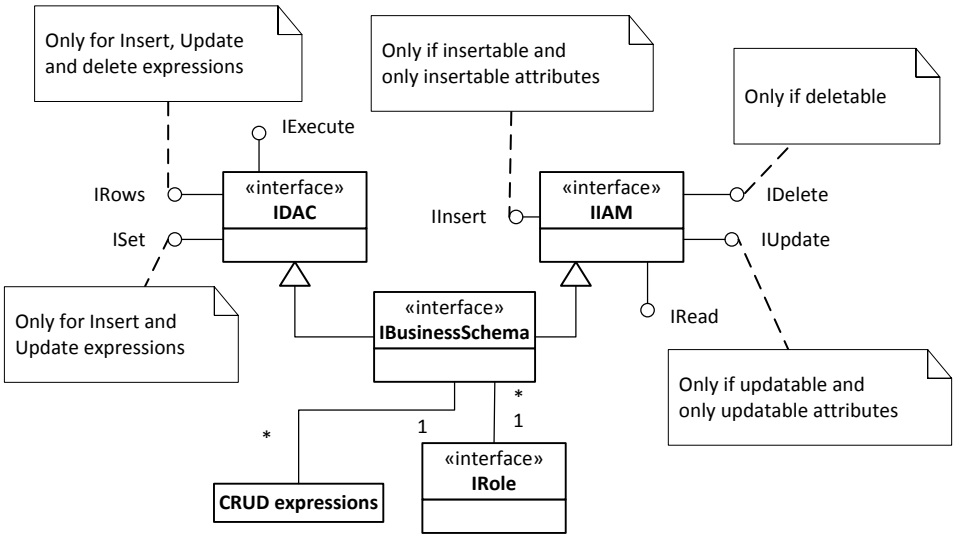

Figure 3. Software architectural model for one role.

Regarding the relation between Business Schemas and, Roles and CRUD expressions, we can see from Figure 3 that the architectural model is consistent with the RBAC model. Please remember 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 4. 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.

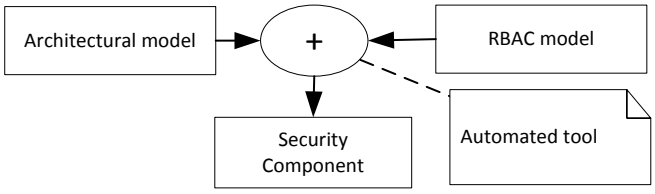

Figure 4. Automated building process of security components.

IV. OUR PROPOSAL: IMPLMENTATION PERSPECTIVE

In this section we present our implementation perspective, which consists of several different components, as shown in Figure 5. 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 aimed at conveying to programmers awareness of the established policies. These data structures are built from data extracted from the Policy Server and also from the software architectural model. They 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

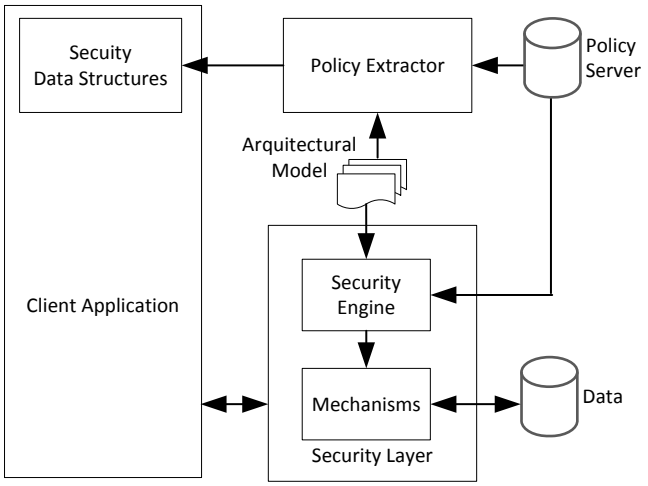

Figure 5. Proposed implementation architecture.

mechanisms. 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 (instances of classes that implement Business Schemas) effectively control users' requests, at runtime, when they issue requests through the direct and the indirect access modes.

A. *Policy Server*

The Policy Server contains a realization of the proposed extension (shown in Figure 2) for a simplified RBAC model, see Figure 6. Our model uses some of the most relevant features of RBAC models: subjects (users), applications, sessions, permissions and delegations. A user can play 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 and 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.

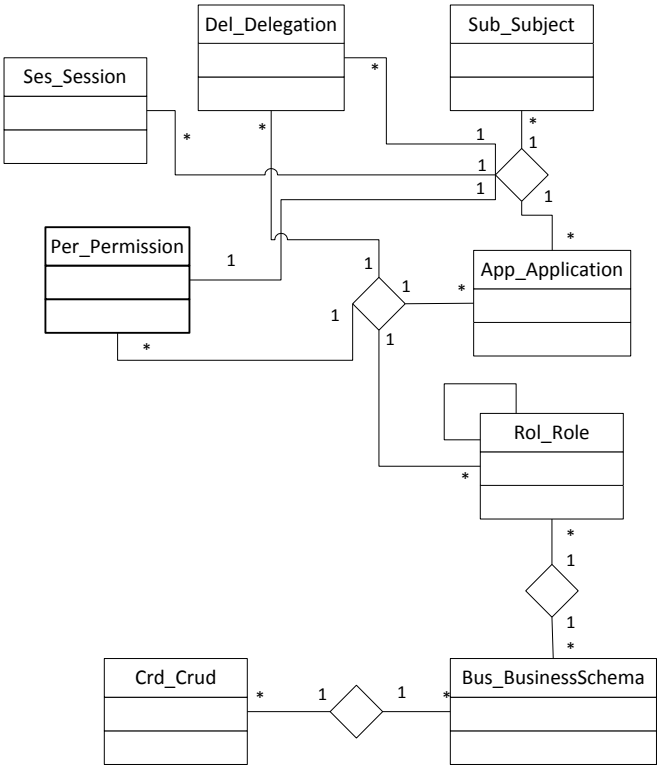

Figure 6. Simplified security model.

B. *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 7 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 8. 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 of t hose 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 level (line 56-68). Some readable,

```
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 7. 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 8. Environment conveyed to programmers.

```
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 9. Example of Figure 1 based on our proposal.

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 9 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.

C. Security Layer

Our security layer comprises three sub-components: 1) a general manager, which is responsible for providing client applications with standard interfaces to access internal functionalities; 2) security engine, which is responsible for building at runtime the necessary access control mechanisms, always in accordance with the established policies to the running user and, finally, 3) the implemented mechanisms, 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 8 (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 more authorized to play that role. In this case an exception is raised.

V. 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 several other approaches, the authorization processes are completely distributed, this way avoiding any scalability problem.

Maintainability: Security layers are automatically built and updated. 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.

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 [25] 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 middle-wear 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.

VI. 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 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.

Future work is organized around a key objective. The key objective is to design a new version where security is completely ensured without the need of any additional security layer. This work is already in progress.

References

[1] P. Samarati and S. D. C. di Vimercati, "Access Control: Policies, Models, and Mechanisms," *Found. Secur. Anal. Des.*, vol. 2171, pp. 137–196, 2001.

[2] S. D. C. di Vimercati, S. Foresti, and P. Samarati, "Recent Advances in Access Control - Handbook of Database Security," M. Gertz and S. Jajodia, Eds. Springer US, 2008, pp. 1–26.

[3] R. S. Sandhu and P. Samarati, "Access Control: Principle and Practice," *Commun. Mag. IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

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

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

[6] C. Pablo, M. Sergey, and A. Atul, "ADO.NET entity framework: raising the level of abstraction in data programming," in *ACM SIGMOD International Conference on Management of Data*, 2007, pp. 1070–1072.

[7] M. Erik, B. Brian, and B. Gavin, "LINQ: Reconciling Object, Relations and XML in the .NET framework," in *ACM SIGMOD Intl Conf on Management of Data*, 2006, p. 706.

[8] D. Yang, *Java Persistence with JPA*. Outskirts Press, 2010.

[9] J. O. Elizabeth, "Object/relational mapping 2008: hibernate and the entity data model (edm)," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[10] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *Computer (Long. Beach. Calif).*, vol. 29, no. 2, pp. 38–47, 1996.

[11] L. Fuchs, G. Pernul, and R. Sandhu, "Roles in information security – A survey and classification of the research area," *Comput. Secur.*, vol. 30, no. 8, pp. 748–769, 2011.

[12] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *9th USENIX Conf. on Operating Systems Design and Implementation*, 2010, pp. 1–14.

[13] J. Abramov, O. Anson, M. Dahan, P. Shoval, and A. Sturm, "A methodology for integrating access control policies within database development," *Comput. Secur.*, vol. 31, no. 3, pp. 299–314, May 2012.

[14] J. Zarnett, M. Tripunitara, and P. Lam, "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*, 2010, pp. 79–88.

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

[16] J. Fischer, D. Marino, R. Majumdar, and T. Millstein, "Fine-Grained Access Control with Object-Sensitive Roles," *23rd ECOOP - European Conference on Object-Oriented Programming*. Springer-Verlag, Italy, pp. 173–194, 2009.

[17] G.-J. Ahn and H. Hu, "Towards Realizing a Formal RBAC Model in Real Systems," in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, 2007, pp. 215–224.

[18] M. Komlenovic, M. Tripunitara, and T. Zitouni, "An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC)," *Proc. first ACM Conf. Data Appl. Secur. Priv. - CODASPY '11*, p. 121, 2011.

[19] K. Jayaraman, M. Tripunitara, V. Ganesh, M. Rinard, and S. Chapin, "MOHAWK : Abstraction-Refinement and Bound-Estimation," vol. 15, no. 4, pp. 1–28, 2013.

[20] D. S. Wallach, A. W. Appel, and E. W. Felten, "S AFKASI 1 : A Security Mechanism for Language-based Systems," vol. 1, no. 212, 1998.

[21] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "ACADA - Access Control-driven Architecture with Dynamic Adaptation," in *SEKE'12 - 24th Intl. Conf. on Software Engineering and Knowledge Engineering*, 2012, pp. 387–393.

[22] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Runtime Values Driven by Access Control Policies Statically Enforced at the Level of the Relational Business Tiers," in *SEKE'13 - Intl. Conf. on Software Engineering and Knowledge Engineering*, 2013, pp. 1–7.

[23] Ó. Pereira, R. Aguiar, and M. Santos, "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," vol. 4, no. 1, pp. 158–180, 2011.

[24] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "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.*, vol. 4, no. 1&2, pp. 158–180, 2011.

[25] L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão, "Type-based access control in data-centric systems," *20th European conference on Programming Languages and Systems: part of the joint European conferences on theory and practice of software*. Springer-Verlag, Saarbrucken, Germany, pp. 136–155, 2011.