Enforcement of Purpose Based Access Control within Relational Database Management Systems

Pietro Colombo and Elena Ferrari, *Fellow, IEEE*

Abstract—Privacy is becoming a key requirement for ICT applications that handle personal data. However, Database Management Systems (DBMSs), which are devoted to data collection and processing by definition, still do not provide the proper support for privacy policies. Policies are enforced by ad-hoc programmed software modules that complement DBMS access control services. This practice is time consuming, error prone, and neither general nor scalable. This work does a first step to overcome these limits. We propose a systematic approach to the automatic development of a monitor that regulates the execution of SQL queries based on purpose based privacy policies. The proposed solution does not require programming, it is general, platform independent and usable with most of the existing relational DBMSs.

Index Terms—Privacy policies, enforcement, monitor, relational database management systems, aspect oriented programming, model driven engineering

1 Introduction

Nowadays the number of ICT applications that collect and process personal and sensitive data, is rapidly growing. Organisations that handle personal data need to take care of the privacy of individuals belonging to a broad range of categories, such as customers, employees, citizens, or patients. As a consequence, privacy-preserving data management has become a requirement of paramount importance for both the development of new applications and the upgrade of existing ones.

Research efforts have been devoted to address different issues related to the development of privacy-preserving data management techniques. We can roughly classify them into two categories. The first includes privacy-preservation techniques, such as anonymization and encryption techniques [1], which deal with data release to third parties. The second groups all approaches to integrate privacy support in a conventional DBMS.

The goal of this work is the enhancement of existing relational DBMSs with privacy policy enforcement.

Current database technology supports access control according to a variety of options, ranging from discretionary, mandatory to role-based access control (RBAC). Additionally, they support the possibility of enforcing fine-grained and context dependent policies (e.g., Oracle VPD [1]). In contrast, the enforcement of privacy policies is still poorly considered. Very often the enforcement is achieved by ad-hoc programmed software modules that interact with the access control services provided by the target DBMS. We believe that what is needed is a systematic approach to reduce as much as possible the development effort of these modules.

什么是非功能性需求

Within DBMSs, privacy policies are non functional requirements that regulate data collection and use. Policies can involve numerous concepts such as users, roles, authorizations, actions, purposes, retention, conditions, and obligations. However, purposes have been largely considered the essence of any privacy policy [4]. Therefore, in this work we start focusing on policies built around this concept. More precisely, this paper describes an approach to the definition of a monitor, called PuRBAC (Purpose and Role-based Access Control), which regulates the execution of SQL queries based on purpose and role based privacy policies. PuRBAC enforcement mechanisms are defined along the line of the purpose-based access control model proposed by Byun and Li in [4]. The approach to the definition of PuRBAC is built on top of MAPaS, a model-based framework for the specification and analysis of privacy-aware systems that we have presented in [5], [6]. In this work we extend MAPaS with a component responsible for generating PuRBAC starting from a set of privacy policies.

什么是基于模型的框架

? ?

Privacy policies are specified with PaML, a modeling language for privacy-aware systems, using MAPaS editing tools. The modeling activity results in a PaML model that includes all information required for the definition of PuRBAC. PuRBAC is a Java application that operates in between a relational DBMS and the users, and filters user requests and DBMS responses based on the specified policies. It performs runtime privacy policies enforcement through pre-filtering and query rewriting mechanisms. The generation of PuRBAC exploits model driven

1. http://www.oracle.com/technetwork/database/security/index-088277.html.

• *The authors are with the Dipartimento di Scienze Teoriche e Applicate, Università degli Studi dell'Insubria, Varese, Italy.*
E-mail: {pietro.colombo, elena.ferrari}@uninsubria.it.

Manuscript received 14 Dec. 2012; revised 4 Nov. 2013; accepted 15 Nov. 2013. Date of publication 13 Mar. 2014; date of current version 26 Sept. 2014.
Recommended for acceptance by L. Khan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2312112

engineering and aspect oriented programming techniques, and is completely automated. System administrators, after specifying few configuration and interaction options, simply have to invoke the PuRBAC generation functionality provided by MAPaS. The proposed approach is general, platform independent and it can be used with most of the existing relational DBMSs. No programming activity is required to define PuRBAC. This has the potentiality to speedup the development and to reduce mistakes introduced with ad-hoc programming. Although in the literature there exists some work related to our proposal, mainly in the field of access control (see Section 2 for an overview), to the best of our knowledge we are not aware of a system allowing the automatic generation of a privacy-aware monitor for a relational DBMS. Therefore, this work represents a first step to fill this void.

Our work is related to the Privacy by design (PbD) [2] principle, an emerging development approach for privacy aware software. According to PbD, privacy affects system design and implementation choices since the beginning, and privacy preserving mechanisms are designed together with application functionalities. PbD allows developing software that natively enforces privacy. This potentially overcomes in terms of efficiency and efficacy the approaches in which privacy is enforced through the integration of complementary modules [3]. However, PbD is applicable when developing applications from scratch. Although we are not developing DBMSs with PbD, since the goal of this work is the enhancement of existing DBMSs with privacy policies enforcement monitors, we believe that our framework complies with some of the foundational principles of PbD [2]. Privacy is considered all through the life cycle of PuRBAC. PuRBAC is automatically derived from privacy requirements (i.e., policies). The essence of the considered policies is the user consent (to the access) based on given purposes, therefore, user privacy is the prime requirement for the definition of PuRBAC. Moreover, PuRBAC achieves a proactive protection: queries are not executed if they are not compliant with policies, and they are rewritten to prevent purpose violating accesses.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 shortly presents MAPaS elements involved in this work. Section 4 introduces the running example we use throughout the paper. Section 5 provides an overview of the approach to generate PuRBAC, whereas Section 6 gives a more detailed description of the approach and of the generated monitor. Section 7 explains the privacy-aware monitoring techniques used by PuRBAC. Section 8 concludes the paper. The paper contains an appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2014.2312112, reporting the proofs of the security and completeness properties of PuRBAC, and some additional details about query rewriting.

2 Related Work

In the literature, most of the work done on policy enforcement are focused on security and access control. A compendium of runtime enforcement techniques of security policies is presented in [9]. Data privacy enforcement is a more recent research topic, and only few papers propose privacy policy enforcement techniques designed for DBMSs (see [10] for a survey).

A seminal paper by Agrawal et al. [11] discusses implementation strategies for "Hippocratic" DBMSs. The work presents software modules that should be included in a DBMS to be privacy aware. However, this work specifies high level requirements without discussing how this module can be implemented.

Massacci et al. [12] propose an approach to automatically derive privacy requirements from privacy policies specified for Hippocratic databases. The requirements are mapped to models for the Secure Tropos framework, which allows checking their correctness and consistency. In a previous work [6] Colombo and Ferrari presented MAPaS functionalities supporting the analysis of privacy policies specified in PaML models. The goal of this work is different. Privacy policies are used as requirements for the automatic definition of an enforcement monitor.

Based on [11], Byun and Li [4] propose a model and a set of guidelines to integrate purpose-based access control in a relational DBMS, and a query rewriting approach to enforce purpose based access control policies.

In previous work [5], [6] Colombo and Ferrari describe how MAPaS complements and enhances [4]. This work provides an additional enhancement. MAPaS is used to generate a monitor that enforces privacy policies within relational DBMS. With difference to [4], no programming activity is required, and the generated monitor supports both pre-filtering and query rewriting enforcement.

In the literature, there also exist several model based [13] frameworks to develop software systems with role-based access control [15] features. For instance, Basin et al. [14] propose a model based security framework that consists of a UML Profile for modeling RBAC authorization policies, and a set of analysis tools for checking and querying design models of RBAC systems. However, such a framework is not tailored to the specification of privacy policies and no support is provided for the generation of enforcement monitors.

Although data privacy represents a significant requirement for the definition of secure systems, few work have considered this aspect within security aware development. Some formal frameworks (e.g., [16]) support the specification of privacy requirements and provide mechanisms to check correctness of the specified system with respect to these requirements. However, different from our proposal, they do not provide mechanisms for automatically deriving a monitor for privacy policies.

Our runtime enforcement approach exploits query rewriting. In the literature, program rewriting techniques are classified into dynamic and static categories [17]. Dynamic techniques perform rewriting on the fly at runtime. This allows for dealing with queries that are only known at execution time. In contrast, with static techniques one has to specify the queries he/she wants to execute in advance. The rewriting is executed after the specification, and the rewritten queries are usually permanently stored. Static techniques allow for suppressing derivation overheads suffered by the dynamic ones. However, some overhead is required all the same. In fact, when the user invokes the execution of a query it is required to check whether a modified version has already been stored and to load the modified query. So far our framework implements dynamic rewriting, but we are planning to support the static approach too.

The monitor generated with MAPaS enforces policies by means of dynamic program analysis [18]. Our approach uses an aspect oriented programming technique [19]. More precisely it uses AspectJ [8] and expoits a weaving technique based on compile-time binding. In the literature some work propose a dynamic binding technique that weaves access control aspects to the byte code of classes (e.g., [20]). Our approach is different since the generated monitor intercepts query execution requests and checks whether these queries can be effectively executed blocking their execution in case of incompatibility. The rewriting is executed only if the queries pass this check.

Different from our approach, static program analysis (e.g., [22]) checks if potential security violations and runtime errors are met by execution paths possibly covered by the program under analysis. This technique requires to derive a static model of the program and to verify properties of this model. The approach is not applicable at runtime and requires to analyze the program before the execution. In our case, SQL queries that have to be executed are only known at runtime, therefore this approach cannot be used.

3 The MAPaS Framework

The Modeling and Analysis of Privacy-aware Systems (MAPaS) framework [5], [6] is a model-based development framework for privacy aware system development. The framework is based on a set of privacy concepts selected and formalized by Byun and Li [4]. These concepts have been integrated to form the Privacy-aware Modeling Language (PaML), a modeling language that supports the modeling and analysis of privacy aware systems. The entire MAPaS framework is built around the elements of this language.

MAPaS allows one to create and edit PaML models (i.e., system models defined with PaML language), and to assess characteristics of the edited PaML models through a rich set of analysis tools.

The current paper extends MAPaS with an additional component allowing the automatic generation of a purpose based access control module for relational databases (see Section 5). In what follows, we shortly introduce the elements of PaML. We refer to [6] for more details.

PaML is a domain specific modeling languages tailored to the privacy domain. We defined PaML through the extension of the Unified Modeling Language (UML),[2] a notation that has achieved the status of standard for the modeling of general purpose software systems. More precisely, PaML is a UML Profile, a language that extends UML with a set of domain specific modeling constructs, called UML stereotypes. Each stereotype is defined by customizing properties and semantics of existing UML elements. This is achieved by introducing: 1) structural

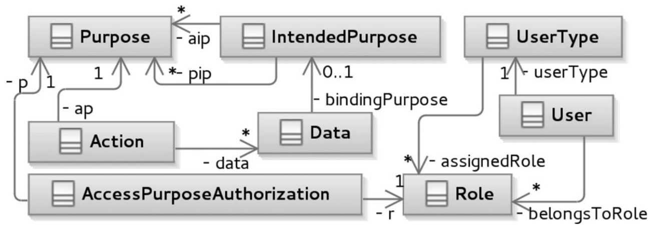


Fig. 1. An high level view on PaML meta-model.

properties that are not specified by the customized UML elements, and 2) constraints[3] that restrict the semantics of these UML elements.

The set of all PaML elements along with their properties, relationships and constraints form the PaML meta-model, i.e., the model that defines the core elements of the language. A partial view of the model is shown in the UML Class of Fig. 1, which introduces a set of privacy domain concepts, along with their characteristics and the relationships among them.

Element Purpose specifies *"the reasons for data collection and use"*[4] and therefore represents the essence of any privacy policy. Purpose elements are hierarchically organized in tree structures. The IntendedPurpose concept binds the reasons for which data can and cannot be accessed to data themselves. As such an IntendedPurpose is composed of a set of allowed intended purposes (association aip in Fig. 1), which specifies the purposes for which data can be accessed, and a set of prohibited intended purposes (association pip in Fig. 1), which specifies the reasons for which data cannot be accessed. IntendedPurpose elements can be assigned to data at scheme level, i.e., to a whole table or to table attributes, or at instance level, i.e., to a tuple or to selected components within a tuple.

Element Action represents an SQL operation that processes data for a specific access Purpose ap (see the association between Action and Purpose in Fig. 1).

Element UserType models a group of stakeholders characterized by the same type of properties, whereas element User models a single stakeholder of a UserType, who requires to access data. For instance, suppose that UserType *UT* models employees who work in a given department. *UT* is characterized by attributes *name* and *office*. The latter is constant and is set to '15-384'. Users *U1* and *U2* are two instances of *UT* such that, attribute *name* of *U1* is set to *'Bob'*, whereas the one of *U2* to *'Mary'*, and for both *office* is equal to *'15-384'*.

Element Role models functionalities and responsibilities that can be played by users. Roles can be assigned to UserType, meaning that users that are instances of that type cover the same role. A role can also be assigned to a user, meaning that the involved single user covers the role. Role aggregates a set of attributes that specify role's intrinsic characteristics. A role is also characterized by a constraint that specifies the conditions under which the role can be assigned. Such a constraint is a propositional logic formula

2. http://www.omg.org/spec/ UML/2.4.1.

3. The constraints are specified using the Object Constraint Language (OCL) - http://www.omg.org/spec/OCL/2.3.1.
4. http://www.w3.org/TR/P3P11/.

Customer

id	name	intolerance
01	John	gluten
02	Alice	none
03	Bob	lactose

Address

id	street	city	state	zip
01	3,St. Paul	Montreal	CA	1234
02	4,3rd Avenue	Montreal	CA	2345
03	1,1st Avenue	Montreal	CA	3456

Order

id	cid	product	credit	date	state
01	01	P123	V123456	01/02/12	shipped
02	02	P234	V234567	02/02/12	packaged
03	03	P345	M123456	31/01/12	ordered

Fig. 2. Selection of tables in MyCompany's database.

built starting from role attributes. Attributes values of a role *R* are derived from those of a user U, when *U* is authorized to play *R*. Roles are hierarchically organized as inverted tree structures and grouped into RoleSets. Role properties defined by a role R are inherited by all roles that descend from R in the hierarchical structure.

A user *U* to whom role *RD* has been assigned, belongs to a role *R* dominated by *RD* when the role constraint is satisfied.

Both UserType and Role identify groups of users, however they have a different meaning. A User is considered an instance of a UserType and it will be always characterized by the same type of attributes that have been specified at definition time. In contrast, roles can be assigned to users at system execution time and users can belong to multiple roles at the same time.

Element AccessPurposeAuthorization models the authorization of an access purpose *p* to a role *r*, meaning that all users that belong to *r* (and thus satisfy the role constraint) are authorized to perform actions with the specified access purpose *p*.

Privacy policies are specified through the definition of a PaML model in which the aforementioned elements (e.g., purposes, roles) are instantiated and connected. The modeled policies specify that users, who cover given roles, are authorized to execute actions that access data with specific access purposes when these comply with the purposes for which the accessed data have been stored. MAPaS supports the specification with dedicated editing tools.

4 Running Example

Throughout the paper, we use a running example from the business domain.

Let us consider MyCompany, a public catering company that works on behalf of the welfare services of a Canadian city. MyCompany cooks and distributes foods to elderly people who cannot leave their houses. These people, which are MyCompany customers, may suffer from food intolerances. MyCompany keeps track of all its customers, their data, and the delivered dishes by means of a database. Such a database includes, among others, tables Customer, Address and Order, whose content is shown in Fig. 2.

The considered database includes personal and sensitive information that should be processed under specific privacy policies. Data stored in MyCompany should only be accessed for specific purposes by MyCompany employees based on the covered roles. However, the relational DBMS used by MyCompany does not feature any specific privacy aware functionality. As such, in the rest of the paper, we show how MAPaS supports the generation of a software module that complements the functionalities of the MyCompany DBMS with an access control mechanism that enforce purpose and role based privacy policies. Roles and purposes used to specify the privacy policies enforced by MyCompany will be introduced in the next sections.

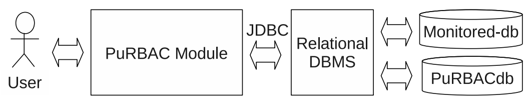


Fig. 3. The role of PuRBAC.

5 Overview of the Approach

A privacy aware DBMS should regulate accesses to data based on the compliance of the purposes for which data are processed with those for which they are collected. In this paper, we propose an approach to enhance an existing relational DBMS to be privacy aware. We achieve privacy awareness through the automatic definition of the PuRBAC module that operates in between the user and the existing DBMS.

The generated module is a Java application that interacts with the DBMS by means of the Java Database Connectivity (JDBC)[5] API [23] (see Fig. 3).

Such a module represents the DBMS front-end, which intercepts all kinds of storage and processing requests issued by users and handles them according to purpose based privacy policies. For the sake of simplicity, in this paper, we describe the enforcement mechanism for queries only. However, other SQL operations (e.g., insert/update) can be handled similarly.

In our approach, policy enforcement is achieved combining a pre-filtering and a query rewriting approach. This complies with the guidelines proposed by Agrawal et al. in [11].

The pre-filtering technique blocks the execution of queries where: 1) the invoker belongs to a role that is not authorized for the access purpose of the query, and 2) the intended purposes assigned at scheme level to tables accessed by the query do not comply with the access purpose of the query. However, if the invocation request passes this check, this does not necessarily mean that the involved query can be executed as it is. In fact, the access purpose of the query may not comply with all the intended purposes associated with the processed data items. This may happen since intended purposes can be assigned to data not only at scheme level (i.e., to tables and attributes) but also at instance level (i.e., to tuples or selected components within a tuple), while pre-filtering operates at scheme level only. Therefore, in our approach a query that has passed the pre-filtering check is rewritten with new constraints that restrict the data set operated by the original query. More precisely, the query is rewritten in such a way that the data set resulting from its execution only contains data items whose

5. Among the most significant reasons for which we have chosen Java and JDBC technologies we mention code portability and the possibility to interact with almost any kind of relational DBMS.

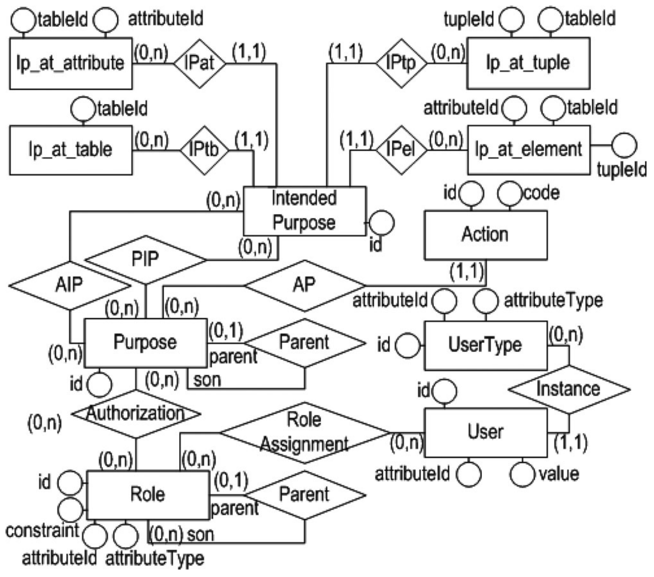


Fig. 4. The ER diagram of PuRBACdb.

intended purposes comply with the access purpose of the query. Further details will be provided in Section 7.3.

The definition of this enforcement mechanism requires to properly configure the monitored database, to introduce a framework supporting the specification of purpose and role based access control policies and to automatically enforce them in the target database. As introduced in Section 3, in MAPaS the specification of policies is achieved with PaML. The PuRBAC module needs to retrieve the constituent elements of privacy policies (e.g., authorizations, purposes and roles) to grant or deny the execution of queries and also for query rewriting. In order to make these elements permanently accessible by PuRBAC, we introduce an auxiliary database, called PuRBACdb, in which we store all the needed metadata. However, data stored in PuRBACdb simply provides a static description of the instances of PaML elements that should be updated to reflect their dynamic modification. For instance, new roles can be assigned to users. Therefore, PuRBAC provides API to handle behavioral aspects of the control module. PuRBAC includes a Java class for each element introduced in the PaML meta-model. Each of these classes provides the operations that are required to modify the state of instances of the corresponding PaML element, as well as the above mentioned pre-filtering and query rewriting mechanisms.

6 The PuRBAC Module

In this section, we describe the process to generate the PuRBAC module for a target database D. The process is articulated into two phases. The first targets the definition of the data layer required for the functioning of PuRBAC, whereas the second concerns the generation of the Java Classes that compose PuRBAC.

6.1 The PuRBAC Data Layer

The main goal of this phase is to introduce an auxiliary database, which stores all the metadata required by PuRBAC to enforce purpose based privacy policies.

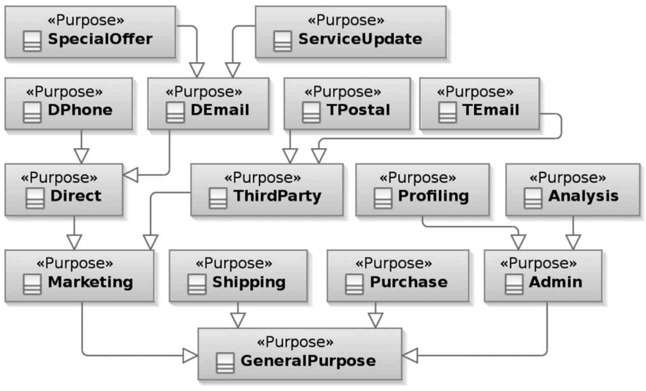


Fig. 5. Purpose tree.

Additionally, we slightly modify the original database by adding to each table an identifier, called tId, to unambiguously identify tuples. This because the enforcement approach very often links tuples in different tables and relying on the primary key can be inefficient if the key consists of many attributes.

The PuRBACdb scheme is derived from the PaML meta-model (cfr. Fig. 1). The relationships among the PaML elements and their structural properties are used to derive the scheme of PuRBACdb tables. More precisely, the PuRBACdb scheme is generated by means of a model to text transformation[6] [24] that takes as input the PaML metamodel (see Section 3), and derives SQL code that generates the PuRBACdb scheme. The derivation is executed when the system administrator invokes the MAPaS functionality “PuRBACdb Scheme Definition”. At invocation time, a wizard allows the user to specify configuration parameters required to properly interact with the DBMS that will host PuRBACdb. Once these parameters have been specified, it executes the transformation, and, finally, it executes the derived SQL code that generates the PuRBACdb scheme. Fig. 4 shows the entity relationship diagram of PuRBACdb.

Once generated, PuRBACdb must be populated. Data that will be used to populate the database are derived from a PaML model. It is worth mentioning that while elements of the PaML meta-model specify the scheme of the tables, the instances of these elements represent data to be stored in PuRBACdb. As such, PuRBACdb population is achieved by means of a model to text transformation that takes as input the edited PaML model and generates the SQL code that inserts data items that specify the properties of its elements in the proper PuRBACdb tables. Editing of the PaML model exploits easy to use visual tools of MAPaS (e.g., Class and Object diagrams). By using these tools, the system administrator introduces the set of purposes that describe the reasons for which data contained in the original database D can be collected and accessed. The hierarchical relationships among these elements are defined by means of generalization relationships. For instance, the purpose set introduced for the running example is shown in the Class diagram of Fig. 5. The generalization relationships among purpose elements specify the purpose hierarchy.

6. A code generation mechanism that derives SQL source code from a model.

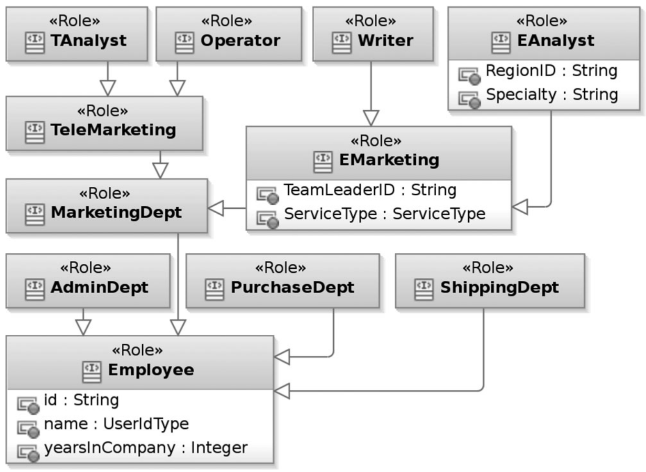


Fig. 6. Role tree.

The definition of the IntendedPurposes is performed in a similar way. The system administrator specifies the set of allowed and prohibited purposes by selecting the proper elements from the previously defined set of purposes. Afterwards, the administrator introduces the set of roles. For each role he/she specifies the attributes that characterize the role, the OCL constraint that specifies when the role can be assigned to a user and the hierarchical relationships with other roles. Fig. 6 shows the Role tree of MyCompany.

The definition of UserTypes is carried out following the same procedure as the one used for roles, while the generation of users requires to select a UserType and to instantiate it by initializing the value of its attributes. Finally, AccessPurposeAuthorizations are defined as associations that connect a Role to a Purpose element (selected from those previously introduced).

Once the modeling phase has been completed, the administrator invokes the MAPaS functionality “Populate PuRBACdb”. This functionality executes the previously mentioned model to text transformation, which derives the SQL operations required for populating PuRBACdb. Afterwards, it invokes the execution of the generated SQL code.

The model to text transformations that allow creating PuRBACdb scheme and populating its tables are both defined by means of an ATL query. More precisely, an ATL query is defined to exhaustively access all the elements included in the analyzed model (the PaML metamodel or a PaML model). Let e be one of these elements. Based on the properties of e's type (e.g., a UML InstanceSpecification that refers to a UML class to which the PAML Stereotype *Purpose* has been applied), a dedicated helper function h_{t_e} is invoked (e.g., derivePurposeTuple). The *context* of the helper is t_e, which represents the type of the considered element (e.g., UML InstanceSpecification). The helper extracts

id	type	purpose
ip0	AIP	GeneralPurpose
ip0	PIP	Admin
ip0	PIP	Marketing
ip0	PIP	Purchase

Fig. 7. An extract of IntendedPurpose table.

id	code	apId
Q01	select name from customer	Marketing
Q02	select name, intolerance, city from customer as c, address as a where c.id=a.id	Shipping
Q03	select product from order where id=01	Profiling

Fig. 8. Queries accessing MyCompany's db.

the structural properties of the element (e.g., ’p1’ and ’p2’, which represent values of the Purpose’s attributes *id* and *parent*, respectively) serializing them as a ordered set of strings (e.g., {*’p1’*, *’p2’*}). Finally, h_{t_e} invokes an auxiliary helper function a_{t_e} that composes the strings in the collection by forming an SQL command (e.g., *’insert into Purpose values(’p1’,’p2’)’*, returning the derived string to h_{t_e}. This helper returns the string to the ATL query that has invoked it. Within that query, the strings returned by all helpers are composed as a sequence of SQL commands that allow the creation/population of PuRBACdb.

Fig. 7 shows an extract of table IntendedPurpose defined for our case study.

However, the PaML model does not include all data that should be inserted in PuRBACdb. In fact, this model does not contain any element that directly relates to data stored in D. Therefore, the administrator has to further specify the purposes for which data in D can be collected and processed, and the SQL operations that will access data in D along with their allowed access purposes. This is achieved through a set of input masks that are used to specify the purpose assignments and the definition of SQL operations to be executed on D. It is worth noting that the SQL queries that will be executed on D may not be known at PuRBAC generation time. As such, these elements may be introduced in PuRBACdb even after the generation of PuRBAC.

Fig. 8 shows the simple queries we consider for our running example as well as the associated purpose.

The assignment of intended purposes to data can be specified at different granularity levels, i.e., to tables, attributes, tuples and selected components within a tuple. As such, PuRBACdb includes four tables, denoted respectively IP_AT_TABLE, IP_AT_ATTRIBUTE, IP_AT_TUPLE, IP_AT_ELEMENT, each of which is used to model the binding at a different level. For instance, the scheme of IP_AT_ELEMENT includes attributes tableId, tupleId, and attributeId, which allow the identification of a data item contained into a specific tuple of a given table, and attribute ipId, that allows the specification of the assigned IntendedPurpose.

Example 1. Consider our running example and suppose that intended purposes are assigned at element level to table Customer, whereas they are assigned to table Address at tuple level, and to table Order at attribute and table level. These assignments are shown in Fig. 9.

MAPaS also supports a variety of assignment options [6], namely No-change, Substitution, Weakening, Strengthening, and Inheritance, to regulate how and if purpose assignments propagate to finer grained data levels. For instance, the option Substitution specifies that assignments at finer grained data levels substitute those at higher levels regardless of the involved IntendedPurpose elements.

IP_AT_ELEMENT

tableId	tpId	attributeId	ipId
Customer	01	id	ip1
Customer	01	name	ip2
Customer	01	intolerance	ip3
Customer	02	id	ip1
Customer	02	name	ip1
Customer	02	intolerance	ip1
Customer	03	id	ip1
Customer	03	name	ip1
Customer	03	intolerance	ip4

IP_AT_TUPLE

tableId	tupleId	ipId
Address	01	ip6
Address	02	ip1
Address	03	ip4

IP_AT_ATTRIBUTE

tableId	attributeId	ipId
Order	product	ip7
Order	credit	ip8
Order	date	ip9
Order	status	ip7

IP_AT_TABLE

tableId	ipId	option
Customer	ip0	substitution
Address	ip0	substitution
Order	ip5	substitution

Fig. 9. Assignment of IntendedPurposes to data stored in MyCompany's database.

Example 2. With reference to Fig. 9, IntendedPurpose *ip5* is assigned to table *Order*, whereas *ip8* to attribute *credit* of table *Order*. Since option *substitution* is specified for *Order*, the intended purpose assigned to attribute *credit* is *ip8*.

Due to the variety of assignment options, the runtime derivation of the actual IntendedPurpose assigned to data may be a complex and time consuming task. In order to minimize the processing time, we pre-calculate the intended purposes and we store them into temporary tables. More precisely, for each table T_i in D, we generate a table T_i^{ip}, characterized by the same set of attributes and the same number of tuples of T_i. For instance, starting from table *Customer(tId, id, name, intolerance)* we generate table *Customer^{Ip} (tId, id, name, intolerance)*. Each element of this new table stores the IntendedPurpose assigned to the corresponding element of table T_i.

The tuples to be included in T_i^{ip} are derived from the items stored into the four PuRBACdb tables previously described (e.g., IP_AT_TUPLE) by applying the derivation procedure required for the specified assignment option. The derivation process is illustrated by the pseudocode shown in Listing 1.

```
List tableList= show tables in D
for(i:1..|tableList|){
 ipAtTb= select INTENDED_PURPOSE_ID
  from IP_AT_TABLE  where TABLE_ID=$T_i$
 List attributeList= show columns in $T_i$
 List tupleList= select TUPLE_ID from $T_i$
 for(tp in tupleList){
  ipAtTp=select INTENDED_PURPOSE_ID from IP_AT_TUPLE
   where TABLE_ID=$T_i$ and TUPLE_ID=tp
  List rowIP= new List()
  rowIP.add(tp)
  for (at in attributeList){
   ipAtAt=select INTENDED_PURPOSE_ID
    from IP_AT_ATTRIBUTE
    where TABLE_ID=$T_i$ and ATTRIBUTE_ID=at
   ipAtEl=select INTENDED_PURPOSE_ID
    from IP_AT_ELEMENT
    where TABLE_ID=$T_i$ and ATTRIBUTE_ID=at
     and TUPLE_ID=tp
    resIP=derive(opt,ipAtTb,ipAtAt,ipAtTp,ipAtEl)
    rowIP.add(resIP)}
 } insert into $T_i^{ip}$ (rowIP)}                              (1)
```

The process starts by deriving the list of tables in D. For each table T_i in the list, we look at IP_AT_TABLE to check whether an IntendedPurpose is specified for T_i. We also derive the list of the attributes and the list of the tuple identifiers of T_i (*attributeList* and *tupleList*, respectively). For each tuple *tp* in the list we look for a possibly associated IntendedPurpose in the table IP_AT_TUPLE (the derived element is stored into variable *ipAtTp*). Similarly, by analyzing tables IP_AT_ELEMENT and IP_AT_ATTRIBUTE, we derive the IntendedPurpose associated with each attribute of T_i and with each component of tuple *tp*, respectively. Variables *ipAtAt* and *ipAtEl* keep track of these assignments. The resulting intended purpose elements derived at scheme and instance level are provided as input to function *derive* that calculates the actual IntendedPurpose to be assigned to each data item. Based on the assignment option *opt* (e.g., substitution, no change), different derivation criteria can be used. Assignment options and the corresponding derivation criteria are thoroughly discussed in [6].

The derived intended purpose is stored in the list of values to be associated to a tuple of the newly generated table. Once the row is completed, it is inserted into table T_i^{ip}.

6.2 The PuRBAC Module Components

As explained before, PuRBAC acts as a broker in between the user and the DBMS that hosts the target database D. The challenging aspect of the definition of PuRBAC concerns the definitions of the enforcement mechanism that verifies that: 1) the requester users are authorized to execute the submitted queries; and 2) the involved queries are actually allowed to access data based on purpose compliance.

PuRBAC is entirely defined using MAPaS, therefore no programming activity is required. All information required for the definition of PuRBAC is extracted from PuRBACdb. The invocation of the MAPaS functionality "Generate PuRBAC", triggers the generation of the Java classes that compose PuRBAC. Most of the structural features of PuRBAC classes are derived from PuRBACdb tables. Other features, such as the query rewriting mechanism, which are independent from the set of privacy policies that constrain the behavior of the module are statically defined. PuRBAC can be seen as a model composed of classes and objects that map elements of the PaML model that has been used to define PuRBACdb. The PuRBAC model represents the context with respect to which the purpose and role based access control policies specified in the PaML model must be enforced.

A set of OCL constraints specifies conditions, such as purpose compliance criteria and belonging to roles, which should be satisfied at query execution time. As such, the enforcement requires to verify these OCL constraints at runtime. Verification is performed by means of an aspect oriented approach. AspectJ code is derived from the OCL constraints by exploiting the functionalities of the Dresden OCLToolkit project's library.[7]

In the next sections, we describe the most relevant properties of the classes that compose PuRBAC.

6.2.1 Classes Purpose, IntendedPurpose, Action and AccessPurposeAuthorization

These classes are derived from the schema of the homonymous PuRBACdb tables. The classes include the same set of attributes that characterize the tables schema and methods that allow for accessing and modifying the value of the attributes.

For instance, similar to table Purpose, Class *Purpose* is characterized by the attributes *id* and *parent* and by the getter and setter methods, while Class *IntendedPurpose*, besides the identifier, includes two lists of purpose elements that collect allowed and prohibited purposes, respectively. Class *Action* models the concept of SQL query. It includes attribute *accessPurpose* that keeps track of the access purpose associated with the modeled query, and *code*, that stores the source code of the query. Finally, class *AccessPurposeAuthorization* includes attributes *RoleId* and *PurposeId*, which identify a role authorized to execute actions for a specific purpose.

6.2.2 Abstract Classes Role and UserType

Abstract classes Role and UserType model features shared by all roles and users. A *concrete* role class extends the abstract class Role with a user assignment constraint, and with attributes that characterize a specific role. Similar to roles, a concrete UserType class introduces the attributes that characterize a group of users. The access and modification of role/user attributes is achieved by operations provided by the abstract class *Role/UserType*. Names and types of the attributes are resolved at runtime by means of reflection mechanisms.[8]

The abstract class *UserType* provides mechanisms to instantiate concrete role classes and to handle role assignments. At assignment time, Role attributes are initialized to the same values of the homonymous attributes of the involved instance of concrete UserType.

Example 3. Let us suppose that MyCompany includes the class *PersonnelOffice23*, a concrete UserType class that models a group of users who work in office 23 of the personnel department. *PersonnelOffice23* is characterized by attributes *name*, *id*, *officeId*, *serviceType*, *managerId*, *expLevel*, *teamLeaderId*, and *yearsInCompany*. Consider an instance of *PersonnelOffice23* denoted as *Jack*, such that, *Jack* = ⟨*name='Jack', id=123, officeId=23, serviceType='ReadInfo', managerId=2,345, expLevel='low', teamLeaderId=234, yearsInCompany=2*⟩. Consider also role *Employee*, which contains attributes *id*, *name*, *yearsInCompany* (see Fig. 6), and specifies the constraint *yearsInCompany<5 and yearsInCompany>0*. When Employee is assigned to Jack, its attributes are initialized to ⟨*id*=123, *name*='Jack', *yearsInCompany*=2⟩.

6.2.3 Derivation of "concrete" UserType Classes

Concrete classes representing groups of users are derived from data stored in the PuRBACdb tables UserType.

The definition of concrete UserType classes first requires to incrementally build abstract representations starting from the information specified in each tuple of UserType. These representations are defined using Javassist,[9] a toolkit that allows one to create abstract representation of Java classes and generate their bytecode at runtime. These representations are introduced as instances of class CtClass (Compile-time Class). For each user attribute specified in a UserType tuple, an attribute is added to the involved instance of CtClass. The abstract class UserType is specified as the supertype of the Java Classes that will be derived from these CtClass instances. Then, the bytecode of new Java Classes is automatically derived from the previously generated instances of CtClass exploiting Javassist functionalities.

The previously illustrated concrete UserType classes introduce structural properties shared by multiple users. Users are therefore introduced as instances of concrete UserType classes. The information stored in table User is used to initialize the value of UserType attributes.

6.2.4 Derivation of "concrete" Roles

The derivation of concrete Role classes is carried out starting from the PuRBACdb table Role using the same approach applied to concrete UserType classes. However, besides a set of attributes, a Role also includes a condition that specifies whether the user to whom the role is assigned, is actually allowed to play the involved Role. The conditional expression is an OCL constraint built on Role's attributes.

The verification is achieved by a dedicated Aspect, which complements the functionalities of the Role class with a new operation that checks the constraint. The Aspect is automatically derived starting from each assignment constraint using the Dresden OCL Toolkit. The generated Aspect allows checking the constraint at role assignment time and whenever the security administrator desires to check it.

6.2.5 Definition of the Application Scenario

Class ModelProvider specifies the Java classes that compose PuRBAC. ModelProvider consists of a set of attributes each of which of a type that corresponds to one of the Java classes introduced so far.

The generation of ModelProvider uses Javassist functionalities. As such similar to the UserType case, we build an abstract representation of ModelProvider as an instance of a CtClass. Afterwards, we add attributes for each of the previously analyzed PuRBACdb tables, and each concrete UserType and Role classes previously introduced. Once the definition of this abstract representation has been completed, the bytecode of ModelProvider is automatically derived using Javassist functionalities.

7 Execution of Privacy Aware Queries

Our purpose based access control enforcement mechanism is based on the compliance analysis between the purposes

7. http://www.dresden-ocl.org/.
8. http://docs.oracle.com/javase/tutorial/reflect/index.html.
9. http://www.csg.ci.i.u-tokyo.ac.jp/ chiba/javassist/.

associated with query Q and the purposes associated with the data accessed by Q. Compliance analysis follows the guidelines proposed by Byun and Li in [4]. Before presenting the details, we introduce the needed notations.

7.1 Notations

Given a purpose P, we denote with $P\downarrow$ / $P\uparrow$ all purposes that descend from / precede P in the purpose tree that includes P itself. Given an IntendedPurpose IP, $IP.aip\downarrow$ denotes the set of all purposes that descend from the purposes included in the aip (allowed purposes) component of IP, whereas $IP.pip\updownarrow$ denotes the set of all purposes that precede or descend from the purposes collected in the pip (prohibited purposes) component of IP. Purpose compliance is modeled through a function $c : (IntendedPurpose \times Purpose) \rightarrow Boolean$.

A Purpose P complies with an IntendedPurpose IP, if P is included in the set $IP.aip\downarrow - IP.pip\updownarrow$. As such, the compliance function c is defined as follows:

$$c(IP, P) = \begin{cases} \text{true}, & \text{if} P \in IP.aip\downarrow - IP.pip\updownarrow \\ \text{false}, & \text{otherwise}, \end{cases} \qquad (1)$$

Purpose compliance is one of the key factors that regulate query execution. Enforcement capabilities of PuRBAC exploit a query formalization that keeps track of all information needed in the compliance analysis.

More precisely, in what follows we refer to:

- *standard* query, as the original submitted SQL query that accesses data in D.
- *privacy-aware* query, as an SQL query that, besides accessing data in D, also refers to data in PuRBACdb that specify the purposes for which the accessed data in D can be processed.

We model an SQL query Q as a tuple $\langle Ts, Ts_{Ip}, Ia, Ia_{Ip}, Pa, Cd, Ga, Gc, Ap\rangle$. Q components are totally ordered sets defined as follows:[10]

- Ts is the set of tables of the target database D that are accessed by Q. We denote with T_i the ith table included in Ts.
- Ts_{Ip} is the set of tables that specify the intended purposes associated with data stored into tables in Ts. If Q is a standard query this set is empty.
- Ia is the set of the attributes of tables in Ts which are accessed by Q. This set includes the attributes specified in the *select*, *where*, *group by* and *having* statements of the SQL query. In case the wildcard character * is used either in the *select* or *having* statement, the set includes the attributes of all tables in Ts. We refer to attributes with notation $a_{j,i}$, which specifies jth attribute of T_i.
- Ia_{Ip} is the set of the attributes of tables in Ts_{Ip} that are accessed by Q. If Q is a standard query this set is empty, whereas if Q is a privacy-aware query, Ia_{Ip} is defined as a collection of attributes $a^{Ip}_{j,i}$, each of which specifies the intended purpose associated with the attribute $a_{j,i}$ of a table $T_i \in Ts$.

10. We assume that ordering follows internal criteria of the hosting DBMS.

component	value
Ts	customer, address
Ts_{Ip}	$\emptyset$
Ia	name, intolerance, city, customer.id, address.id
Ia_{Ip}	$\emptyset$
Pa	⟨name, null⟩, ⟨intolerance, null⟩, ⟨city, null⟩
Cd	⟨$\emptyset$,(customer.id=address.id)⟩
Ga	$\emptyset$
Gc	$\emptyset$
Ap	shipping

Fig. 10. Formalization of query Q02.

- Pa is the set of attributes involved in the projection operated by Q. We model each element of this set as a pair $\langle Id, Exp\rangle$, where:
  - Id is either the name of an attribute of a table in $Ts \cup Ts_{Ip}$ or the name of an *alias* associated with the expression Exp.
  - Exp is either an expression built on top of a set of attributes of tables in $Ts \cup Ts_{Ip}$ by means of operators and aggregate functions, or *null* if no expression is defined.
- Cd is a pair $\langle js, w\rangle$, where js is the set of *join* conditions of Q, whereas w is the condition specified in the *where* statement of Q.
- Ga is the set of attributes listed in the *group by* statement of Q.
- Gc is the set of attributes listed in the *having* statement of Q.
- Ap is the access purpose associated with Q.

Fig. 10 shows the formalization of query $Q02$, whose source SQL code is given in Fig. 8.

Hereafter, components of the query model are referred to as $component^{query}$. For instance, Ts^Q denotes the set of tables accessed by query Q.

7.2 Pre-Filtering

Precondition to the execution of Q is that the intended purpose associated with each attribute in Ia complies with the access purpose Ap^Q. An additional condition for the execution of Q is that Ap^Q is authorized for role R to which user U, who aims to execute Q, belongs to. More precisely, Q is authorized for execution if there exists an access purpose authorization apa such that Ap^Q is implied by the purpose component of apa, and user U belongs to a role R, which descends from the role specified by the role component of apa.

In PuRBAC, SQL queries are modeled by class Action (see Section 6.2). This class provides an operation, denoted *execute*, which handles the execution of the modeled SQL query. Pre-filtering is achieved by specifying a precondition to the execution of *execute*. Listing 2 shows the OCL specification of the precondition.

```
context Action::execute(u: User):RowSet
pre: deriveIPAtScheme(code).values()-> forAll(ip|
  ap.compliesWith(ip.oclAsType(IntendedPurpose))
  )and u.isAPAuthorized(ap)                          (2)
```

The method *deriveIPAtScheme* of class Action derives the set of IntendedPurpose elements associated at scheme level with data processed by Q. The analysis starts deriving the list of tables and attributes accessed by Q. Then, the IntendedPurposes assigned to the derived elements are: 1) extracted from the PuRBACdb tables IP_AT_TABLE and IP_AT_ATTRIBUTE, 2) combined based on the specified assignment option, and 3) returned back to the caller. Furthermore, the derived IntendedPurpose elements have to be checked against the access purpose associated with the SQL query. This is achieved by means of "auxiliary" operations which are added to those provided by classes Purpose, IntendedPurpose and User. The specification of these operations uses OCL. Listing 3 shows the OCL specification of the operation *compliesWith*, to be included in class Purpose, which carries out the purpose compliance analysis. Such an operation is an OCL implementation of function c introduced in Section 7.1.

```
context Purpose
def: compliesWith(ip: IntendedPurpose): Boolean=
 ip.getImpliedPurposes()->includes(self)                 (3)
```

Operation *compliesWith* is invoked on Ap^Q.

As mentioned in Section 7.1, the compliance of a purpose P with an IntendedPurpose IP requires to calculate the set of purposes that descend from the purposes included in the *aip* component of IP, the set of purposes that precede and descend from the purposes grouped in the *pip* component of IP, and to check that P is part of the first set but not of the second one. This is done by Operation *getImpliedPurposes* shown in Listing 4.

```
context IntendedPurpose
def: getImpliedPurposes() : Set(Purpose)=
let IMP:Set(Purpose)=Set{} in
let AIP:Set(Purpose)=self.aip->asSet()
  ->iterate(el;locSet:Set(Purpose)=Set{}|locSet
  ->union(el.getDescendants()))in
let PIP:Set(Purpose)=self.pip->asSet()->iterate(el;
  locSet:Set(Purpose)=Set{}|locSet->union(
  el.getAncestors()->union(el.getDescendants())))in
IMP->union(AIP->
 symmetricDifference(AIP->intersection(PIP)))            (4)
```

The derivation of purpose ancestors and descendants is carried out by auxiliary operations to be added to class Purpose. More specifically, *getAncestors* derives the list of Purpose elements from which P descends from (see Listing 5), while *getDescendant* calculates all Purpose elements that descend from P.

```
context Purpose def: getAncestors():Set(Purpose)=
let ancEl: Set(Purpose) = Set{} in
if parent.oclIsUndefined() then ancEl->including(self)
else let currSet:Set(Purpose)=ancEl->including(self)
 in currSet->union(parent->iterate(el;
 lSet:Set(Purpose)=Set{}|lSet
 ->union(el.getAncestors())))endif                      (5)
```

Finally, in order to grant the permission to execute an SQL query Q for a given purpose, the requesting user should be effectively authorized to do that. Let *Apa* be an access purpose authorization, and let us denote *Apa.p* and *Apa.r* the access purpose and role authorized by *Apa*, respectively. To allow the execution of Q it is required to: 1) check the existence of an *Apa* such that the descendants of purpose *Apa.p* include the access purpose Ap^Q, and 2) verify that the requester user belongs to *Apa.r* that has been authorized for *Apa.p*. Listing 6 shows the OCL specification of this check, which is implemented by operation *isAPAuthorized* that has to be included in class User.

```
context User
def: isAPAuthorized(ap:Purpose): Boolean=
 AccessPurposeAuthorization.allInstances()->
   exists(apa|apa.p.getDescendants()->includes(ap)
 and self.belongsToRole(apa.r))                           (6)
```

Example 4. Let us suppose that user Jack, instance of UserType PersonnelOffice23 (see Example 3), invokes query Q02 reported in Fig. 8 and Fig. 10. Let us also suppose that there exists an access purpose authorization *APA1* = $\langle$*GeneralPurpose, Employee*$\rangle$, which authorizes all users that cover the role *Employee* to execute actions whose access purpose descends from *GeneralPurpose*. Purpose compliance requires access purpose *Shipping* associated with Q02 to comply with the IntendedPurposes assigned to the attributes accessed by Q02.

Since, based on table IP_AT_ATTRIBUTE in Fig. 9, no intended purpose is explicitly assigned to any attribute of tables Address and Customer, the IntendedPurpose assigned to the attributes of these tables is the one assigned to the tables as a whole. Therefore, based on table IP_AT_TABLE (see Fig. 9), *ip0* is assigned to each attribute accessed by Q02.

For each attribute, *compliesWith* checks that the *Shipping* purpose complies with *ip0*. This requires *getImpliedPurposes* to be invoked on *ip0*. Based on *ip0* components (see Fig. 7), such an invocation results in the set *{Shipping}* and therefore the compliance check is successful.

The second part of the verification process requires to check that *Jack* is authorized for the *Shipping* purpose. The condition specified by role *Employee* is satisfied by *Jack*'s attributes (see Example 3), and thus, *Jack* belongs to role *Employee*. Moreover, since *Shipping* belongs to the descendants of *GeneralPurpose*, and Role *Employee* is authorized to execute actions with purpose *GeneralPurpose*, we derive that *Jack* is authorized to execute *Q02*.

Precondition evaluation exploits aspect oriented programming techniques. A dedicated aspect is used for checking the precondition at runtime. Similar to the case of invariant constraints introduced for role assignments, aspects are automatically derived from the corresponding OCL expressions using the Dresden OCL toolkit. It is worth noting that the derivation is achieved at the level of class, and therefore it is independent from any specific instance of Action. As such, once the aspect that checks the precondition is derived, it is used to verify the execution of any SQL query.

The AspectJ code shown in Listing 7 refers to the Aspect derived from the OCL specification in Listing 2.

```
public privileged aspect Action_PreAspect_execute {
 protected pointcut executeCaller(Action a, User u):
 call(* Action.execute(User))&&target(a)&&args(u);
 before(Action a, User u):executeCaller(a, u){
  Boolean result1; res = true;
  for (Object e:a.deriveIPAtScheme(a.code).values()){
   if (!a.ap.compliesWith(((IntendedPurpose)e))){
    res = false; break;}
  }if (!(res && u.isAPAuthorized(a.ap))) {
   String msg="";throw new RuntimeException(msg);}
 }}                                                      (7)
```

The generated pointcut catches the invocation event of method *execute* of class Action. The advice associated with the pointcut is executed before allowing the execution of the query. In case either the user does not belong to an authorized role, or the intended purposes derived at scheme level of the tables involved in the query execution do not comply with the access purpose of the query, a runtime exception is thrown which prevents the execution of the query. Otherwise, the query is issued to the DBMS, where it is executed.

The Java Classes that compose PuRBAC must be instrumented with the OCL operations introduced for the specification of the precondition. To this aim we use the same technique we have used for the instrumentation of code that checks invariants constraints of Role classes (see Section 6.2.4). As such, Dresden OCL toolkit is used to generate aspects that include the Java implementation of the OCL operations, while the AspectJ framework is used to weave the generated aspects with the existing classes of PuRBAC. For instance, class Purpose is instrumented with the operations *getAncestor*, *getDescendants* and *getAncestorsAndDescendants*.

7.3 Query Rewriting

Post condition to the execution of a query Q is that the intended purposes of each data item contained in the data set resulting from the execution of Q, denoted as Rs^Q, comply with the access purpose of Q.

Rs^Q is either a subset of the data items collected in Ts^Q or it is derived from that set (e.g., through an aggregate operation). All tuples in Rs^Q that include data items derived from elements in Ts whose intended purpose does not comply with Ap^Q must be pruned out. We achieve this by means of a query rewriting approach. In other words, instead of directly executing Q, we derive and execute queries Q' and Q'', which, while carrying out the selection criteria of Q, also achieve purpose-based filtering.

More precisely, query Q' receives as input the data set provided as input to Q and, in addition to the join operations possibly performed by Q, it performs the join of each table accessed by Q with the corresponding table that specifies the assignment of intended purposes (i.e., those in Ts^Q_{Ip}) on attribute tId. The execution of Q' results in the data set $Rs^{Q'}$.

In contrast, query Q'' receives as input $Rs^{Q'}$ and prunes out from this data set all tuples that include at least one data item whose intended purpose does not comply with the access purpose of Q. Moreover, it executes the projection, selection and the aggregations possibly specified by Q on that filtered data set.

Therefore, query Q' is derived from Q as follows:

- $Ts^{Q'} = Ts^Q$.
- $Ts^{Q'}_{Ip}$ includes the intended purpose tables (see Section 6.1) associated with tables in $Ts^{Q'}$.
- $Ia^{Q'}$, includes all the attributes processed by Q plus attribute tId that identifies the tuples of all tables. Therefore, $Ia^{Q'} = Ia^Q \cup \bigcup_{T_i \in Ts^{Q'}} T_i.tId$.
- $Ia^{Q'}_{Ip} = \bigcup_{a_{j,i} \in Ia^{Q'}} a^{Ip}_{j,i}$, where $a^{Ip}_{j,i}$ represents the jth attribute of the intended purpose table T^{Ip}_i, and specifies the intended purpose assigned to instances of $a_{j,i}$ (i.e., the jth attribute of T_i).

Ts	customer,address
Ts_{Ip}	customer^{Ip},address^{Ip}
Ia	customer.tId, address.tId, customer.name, customer.intolerance, address.city, customer.id, address.id
Ia_{Ip}	customer^{Ip}.tId, address^{Ip}.tId, customer^{Ip}.name, customer^{Ip}.intolerance, address^{Ip}.city, customer^{Ip}.id, address^{Ip}.id
Pa	⟨gtId, customer.tId+address.tId⟩, ⟨customer.tId as customer+'_'+tId,null⟩, ⟨customer^{Ip}.tId as customer^{Ip}+'_'+tId,null⟩, ⟨address.tId as address+'_'+tId,null⟩, ⟨address^{Ip}.tId as address^{Ip}+'_'+tId,null⟩, ⟨customer.name as customer+'_'+name,null⟩, ⟨customer^{Ip}.name as customer^{Ip}+'_'+name,null⟩, ⟨customer.intolerance as customer+'_'+intolerance,null⟩, ⟨customer^{Ip}.intolerance as customer^{Ip}+'_'+intolerance,null⟩, ⟨address.city as address+'_'+city,null⟩, ⟨address^{Ip}.city as address^{Ip}+'_'+city,null⟩, ⟨customer.id as customer+'_'+id,null⟩, ⟨customer^{Ip}.id as customer^{Ip}+'_'+id,null⟩, ⟨address.id as address+'_'+id,null⟩, ⟨address^{Ip}.id as address^{Ip}+'_'+id,null⟩
Cd	⟨{(customer.tId=customer^{Ip}.tId), (address.tId=address^{Ip}.tId)}, ∅⟩
Ga	∅
Gc	∅
Ap	shipping

Fig. 11. Formalization of query Q02'.

- $Pa^{Q'} = \langle gtId, \sum_{T_i \in Ts^{Q'}} T_i.tId\rangle \cup \bigcup_{a_{j,i} \in Ia^{Q'}} \langle a_{j,i}\ as\ T_i + \text{“\_”} + a_j, null\rangle \cup \bigcup_{a^{Ip}_{j,i} \in Ia^{Q'}_{Ip}} \langle a^{Ip}_{j,i}\ as\ T^{Ip}_i + \text{“\_”} + a_j, null\rangle$, where $gtId$ is an attribute that identifies the tuples in $Rs^{Q'}$.

  Based on this definition, instances of $gtId$ will be set to the concatenation of instances of tId attributes included in the tuples of the tables accessed by Q'.

  The Id component of all other elements included in $Pa^{Q'}$ is defined by concatenating the name of the table and that of the attribute, whereas the Exp component of these elements is set to $null$.
- $Cd^{Q'} = \langle js', \emptyset\rangle$, where js' is defined as the union of $Cd^Q.js$ with the join predicates between tables T_i and T^{ip}_i on attribute tId. Therefore, $Cd^{Q'}.js = Cd^Q.js \cup \bigcup_{T_i \in Ts^{Q'}} T_i.tId = T^{Ip}_i.tId$
- $Ga^{Q'} = Gc^{Q'} = \emptyset$,
- $Ap^{Q'} = Ap^Q$.

Fig. 11 shows the formalization of query $Q02'$ derived from $Q02$ in Fig. 10.

Let Q^* be a modified version of query Q, which only differs from Q since it does not include any *where* and *group by* clause. Based on the definition of Q', $Rs^{Q'}$ includes a number of tuples which equals the cardinality of the result set derived from the execution of Q^*. Q' differs from Q^* only for the additional join operations, however such operations do not add tuples to the result set. In fact, $|T_i| = |T^{ip}_i|$ and $\forall tp \in T_i \rightarrow \exists tp' \in T^{ip}_i \wedge tp.tId = tp'.tId$, by construction. Therefore, it can be proved that $|T_i| = |T_i \bowtie_{T_i.tId = T^{ip}_i.tId} T^{ip}_i|$, and thus $|Rs^{Q'}| = |Rs^{Q*}|$.

Finally, purpose-based filtering is operated by query Q'' which is derived from Q and Q' according to the following criteria:

- $Ts^{Q''}$ includes a unique table corresponding to the one generated by Q'. Therefore, $Ts^{Q''} = Rs^{Q'}$

Ts	$Rs^{Q02'}$
Ts_{Ip}	$\emptyset$
Ia	customer_tId, address_tId, customer_name, address_city, customer_id, address_id, customer^{Ip}_tId, address^{Ip}_tId, customer^{Ip}_name, address^{Ip}_city, customer^{Ip}_intolerance, customer^{Ip}_id, address^{Ip}_id
Ia_{Ip}	$\emptyset$
Pa	⟨customer_id,null⟩, ⟨address_id,null⟩, ⟨gtId,null⟩
Cd	⟨$\emptyset$, customer_id=address_id $\wedge$ c(customer^{Ip}_tId,shipping) $\wedge$c(address^{Ip}_tId,shipping)$\wedge$c(customer^{Ip}_name,shipping) $\wedge$c(customer^{Ip}_intolerance,shipping) $\wedge$ c(address^{Ip}_city,shipping) $\wedge$c(customer^{Ip}_id,shipping)$\wedge$c(address^{Ip}_id,shipping)⟩
Ga	$\emptyset$
Gc	$\emptyset$
Ap	shipping

Fig. 12. Formalization of query Q02“.”

- $Ts_{Ip}^{Q''} = \emptyset$, since intended purposes are specified in $Rs^{Q'}$.
- $Ia^{Q''}$ includes the attributes of $Rs^{Q'}$.
- $Ia_{Ip}^{Q''} = \emptyset$, since intended purposes are specified in $Rs^{Q'}$.
- $Pa^{Q''}$ corresponds to the union of Pa^{Q} and the pair $\langle gtId, Exp\rangle$, which keeps track of the tuples in $Rs^{Q'}$ from which each tuple in $Rs^{Q''}$ is derived. If there does not exist an attribute in Pa^{Q} whose Exp component specifies an aggregate function, the Exp component of $\langle gtId, Exp\rangle$ is set to $null$. In this case, each tuple in $Rs^{Q''}$ is derived from a single tuple in $Rs^{Q'}$. Therefore, the value of the $gtId$ component that identifies each tuple t in $Rs^{Q''}$ corresponds to the one of the tuple identifier $gtId$ of $Rs^{Q'}$ from which t is derived. Otherwise, Exp is set to *Group_Concat* *($Rs^{Q'}$.gtId)*.[11] In fact, in case one of the element in Pa^{Q} specifies an aggregate function, each tuple in $Rs^{Q''}$ is derived from multiple tuples in $Rs^{Q'}$. More precisely, in this case $gtId$ is set to the concatenation of the tuple identifiers in $Rs^{Q'}$ from which the involved tuple is derived.
- $Cd^{Q''}\langle\emptyset, wcd\rangle$, where wcd is defined as the conjunction of the filtering conditions in Q with a set of conditions that select proper tuples based on purpose compliance. More specifically, $Cd^{Q''}.w = Cd^{Q}.w \wedge \bigwedge_{a_{j,i}^{Ip}\in Ia_{Ip}^{Q'}} c(a_{j,i}^{Ip}, Ap^{Q''})$.
- $Ga^{Q''} = Ga^{Q}$.
- $Gc^{Q''} = Gc^{Q}$.
- $Ap^{Q''} = Ap^{Q}$.

The formalization of query $Q02''$ derived from $Q02$ and $Q02'$ is shown in Fig. 12.

7.3.1 Evaluating Purpose Compliance

Function c specifies the logic of the purpose compliance check (see Equation (1)). Such an abstract specification must be properly implemented by the rewritten query Q″, which is responsible for selecting data items based on purpose compliance.

As introduced in Section 7.1, purpose compliance analysis requires to calculate ancestors and descendants of the intended purpose elements associated with data accessed by Q. The derivation requires to visit the tree structure that models the hierarchy of purpose elements.

We implement the derivation by means of a set of recursive queries[12] that are used in the definition of query Q″. The complete SQL code that supports the compliance analysis is reported in the Appendix.

```
with recursive
   ...
compliesWith(id, ip_id, val) as (
 select distinct p.id, ip.id,
  case when p.id in( select distinct gip.id
  from getImpliedPurposes gip where gip.ip_id = ip.id)
  then true else false end
 from Purpose p, IntendedPurpose ip)                    (8)
```

The alias associated with the recursive queries directly describes what is achieved by the query itself. For instance, query *compliesWith*, whose specification is shown in Listing 8, checks the compliance of the Purpose/IntendedPurpose elements. Query *compliesWith* returns a result set characterized by attributes *id*, *ip_id* and *val*. Attribute *id* specifies the identifier of a Purpose, *ip_id* specifies the identifier of an IntendedPurpose, whereas *val* is a boolean attribute that specifies the compliance. The compliance of a specific Purpose/IntendedPurpose couple must be selected from the result set of *compliesWith*. For instance, let us suppose to check the compliance of purpose *Admin* with IntendedPurpose *IP1*. The compliance value is extracted from the result set of query *compliesWith* as shown in Listing 9.

```
select every(val) from compliesWith cw
where cw.ip_id='IP1' and cw.id='Admin'                  (9)
```

7.3.2 The Rewriting Process

As previously described, the execution of SQL queries is handled by means of method *execute* of class Action. Such a method is responsible for forwarding to the DBMS the SQL code of the query that must be executed. Based on the proposed rewriting criteria, the code to be issued to the DBMS is not the one of the original “standard” query Q, but the one of the two privacy aware queries Q′ and Q″ derived from Q.

The rewriting task is achieved by methods of class Rewriter. More precisely, method *derivePrivacyAwareQuery* orchestrates the derivation of the source code of Q′ and Q″, which is achieved by the methods *deriveFirstQuery* and *deriveSecondQuery*, respectively.

The method *deriveFirstQuery* parses the code of the original query and derives the list of tables and attributes that must be accessed by query Q′ according to the requirements defined for Q′ components (see Section 7.1). In order to make the result set of Q′ accessible by Q″, such a dataset is stored in a temporary table $Rs^{Q'}$. The code of Q′ is derived from the information specified in Q′ components. For

11. Group_Concat() is an SQL aggregate function that concatenates the values of the grouped data items.

12. See ISO/IEC 9075 standard: “Information technology - Database languages - SQL”.

instance, Listing 10 shows an extract of the SQL code generated for query Q02′.

```
create temporary table rsq1
select (customer.tId+address.tId) as gtId,
 customer.tId as customertId,
 customerIp.tId as customerIp.tId,
 ....
 address.id as address_id,
 addressIp.id as addressIp_id
from customer join customerIp on
 customer_tId=customerIp_tId,
 address join addressIp on
 address_tId=addressIp_tId                        (10)
```

The method *deriveSecondQuery* parses the source code of Q and Q′ and derives Q″. Listing 11 shows an extract of the source code of Q02″, which has been derived from Q02′ and Q02.[13]

```
select gtId, customer_id, address_id from rsq1
where customer_id=address_id and (
  select every(val) from compliesWith cw
  where cw.ip_id=customerIp_tId and
    cw.id='Shipping')=true and (
  ...
  select every(val) from compliesWith cw
  where cw.ip_id=addressIp_id and
    cw.id='Shipping')=true                        (11)
```

Finally, the method *derivePrivacyAwareQuery* simply concatenates the source code generated for Q′ and Q″.

7.4 Correctness and Completeness of the Approach

The correctness analysis of the pre-filtering approach requires to check whether the Java code of PuRBAC instrumented with the OCL specifications presented in Section 7.2 satisfies the OCL contract. The OCL toolkit automatically derives the AspectJ code from the OCL specifications, while Aspect are weaved together with code by the AspectJ framework. As such, the correctness of the prefiltering completely relies on the work performed by these tools. We assume that both the derivation and weaving are performed correctly.

The correctness and completeness of the rewriting approach is proved by the following theorems, whose proofs are presented in the Appendix.

Theorem 1. *Security*

Let Q be an SQL query and let $Rs^{Q''}$ be the result set returned by Q after the query rewriting described in Section 7.3. For each tuple $t \in Rs^{Q''}$ the intended purpose associated with each component of t complies with the access purpose of Q.

Theorem 2. *Completeness*

Let Q be an SQL query and let $Rs^{Q''}$ be the result set returned by Q after the query rewriting described in Section 7.3. Each tuple $t \in Rs^{Q}$ which is derived from data items whose intended purposes comply with the access purpose of Q, is also included in $Rs^{Q''}$.

8 Conclusions

This paper presented an approach to the definition of a monitor called PuRBAC which regulates the execution of SQL queries based on purpose and role based privacy policies. The work is a first step towards the definition of privacy aware DBMSs built following the guidelines presented in [11]. The whole definition approach is defined on top of MAPaS, a model-based framework for the specification and analysis of privacy-aware systems. A dedicated module of this framework achieves the derivation of PuRPBAC.

Our work is progressing in several directions. First, we are working to introduce in PuRBAC mechanisms to enforce privacy policies that include obligations. We are also investigating the definition of post execution enforcement capabilities that would allow one to identify policy violations based on the analysis of execution traces. As future work we also plan to thoroughly analyze performance aspects of PuRBAC by testing it on different platforms and with different DBMSs.

13. The code of Q02″ also includes the declaration of the recursive queries shown in Listing 8, but such code has been omitted here for space limitations.

References

[1] F. Bonchi and E. Ferrari, *Privacy-Aware Knowledge Discovery: Novel Applications and New Techniques*. Boca Raton, FL, USA: CRC Press, 2009.

[2] F. Bonchi and E. Ferrari, “Privacy by design: Leadership methods, results,” *European Data Protection: Coming of Age*, S. Gutwirth, R. Leenes, P. de Hert, and Y. Poullet, eds., New York, NY, USA: Springer-Verlag, 2013.

[3] S. Gürses, C. Troncoso, and C. Diaz, “Engineering privacy by design,” in *Computer, Privacy & Data Protection*. New York, NY, USA: Springer-Verlag, 2011.

[4] J. Byun and N. Li, “Purpose based access control for privacy protection in relational database systems,” *The Int. J. Very Large Data Bases*, vol. 17, no. 4, pp. 603–619, 2008.

[5] P. Colombo and E. Ferrari, “Towards a framework to handle privacy since the early phases of the development: Strategies and open challenges,” in *Proc. IEEE 6th Int. Conf. Digital Ecosystems Technol.*, 2012, pp. 1–6.

[6] P. Colombo and E. Ferrari, “A modeling and analysis framework for privacy-aware systems,” in *Proc. IEEE Int. Conf. Privacy, Security, Risk Trust Int. Conf. Social Comput.*, 2012, pp. 81–90.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proc. Eur. Conf. Object-Oriented Program.*, 1997, pp. 220–242.

[8] R. Laddad, *Aspectj in Action: Enterprise AOP with Spring Applications*. Greenwich, CT, USA: Manning Publications Co., 2009.

[9] G. Gheorghe and B. Crispo, “A survey of runtime policy enforcement techniques and implementations,” Univ. of Trento, Trento, Italy, Tech. Rep. DISI-11-477, 2011.

[10] S. Nabar, K. Kenthapadi, N. Mishra, and R. Motwani, “A survey of query auditing techniques for data privacy,” *Privacy-Preserving Data Mining*, vol. 34, pp. 415–431, 2008.

[11] R. Agrawal, J. Kiernan, R Srikant, and Y. Xu, “Hippocratic databases,” in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 143–154.

[12] F. Massacci, J. Mylopoulos, and N. Zannone, “From hippocratic databases to secure tropos: A computer-aided re-engineering approach,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 17, no. 2, pp. 265–284, 2007.

[13] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *Proc. Future of Softw. Eng.*, 2007, pp. 37–54.

[14] D. Basin, J. Doser, and T. Lodderstedt, “Model driven security: From uml models to access control infrastructures,” *ACM Trans. Softw. Eng. Methodology*, vol. 15, no. 1, pp. 39–91, 2006.

[15] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, “Proposed NIST standard for role-based access control,” *ACM Trans. Inf. and Syst. Security*, vol. 4, no. 3, pp. 224–274, 2001.

[16] D. Garg, L. Jia, and A. Datta, “Policy auditing over incomplete logs: Theory, implementation and applications,” in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 151–162.

[17] J. Zhou and G. Vigna, “Detecting attacks that exploit application-logic errors through application-level auditing,” in *Proc. 20th Ann. Comput. Security Appl. Conf.*, 2004, pp. 68–178.

[18] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, Sep./Oct. 2009.

[19] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Commun. ACM*, vol. 44, no. 10, pp. 29–32, Oct. 2001.

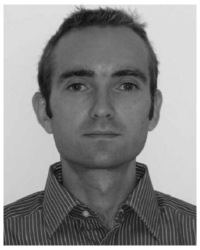
[20] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *Proc. 3rd ACM SIGPLAN Workshop Program. Languages Anal. Security*, 2008, pp. 11–20.

[21] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 109–124.

[22] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Syst. J.*, vol. 46, no. 2, pp. 265–288, Apr. 2007.

[23] M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*. Reading, MA, USA: Addison-Wesley, 2003.

[24] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, Jul. 2006.



Pietro Colombo received the PhD degree in computer science from the University of Insubria, Italy, in 2009. He is a post doctoral fellow at the University of Insubria, Italy. He was with the STRICT SociaLab, investigating the definition of privacy-aware data management systems. His research interests include data privacy and model driven engineering.

Elena Ferrari is a full professor of computer science at the University of Insubria, Italy and a scientific director at the K&SM Research Center. Her research activities include access control, privacy and trust. In 2009, she received the IEEE Computer Society's Technical Achievement Award for "outstanding and innovative contributions to secure data management". She is a fellow of the IEEE and an ACM distinguished scientist.