Enforcing Obligations within Relational Database Management Systems

Pietro Colombo and Elena Ferrari, *Fellow, IEEE*

Abstract—Within Database Management Systems (DBMS), privacy policies regulate the collection, access and disclosure of the stored personal, identifiable and sensitive data. Policies often specify obligations which represent actions that must be executed or conditions that must be satisfied before and/or after data are accessed. Although numerous policies specification languages allow the specification, no systematic support is provided to enforce obligations within relational DBMS. In this paper, we make a step to fill this void presenting an approach to the definition of an enforcement monitor which handles privacy policies that include obligations. Such a monitor is derived from the same set of policies that must be enforced, and regulates the execution of SQL code based on the satisfaction of a variety of obligation types. The proposed solution is systematic, has been automated, does not require any programming activity and can be used with most of the existing relational DBMSs.

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

1 Introduction

Obligations represent a significant component of privacy policies and specify how data should be handled in terms of conditions that must be satisfied and actions that must be executed.

Obligations started to be considered in access control systems in the recent years only. Due to the lack of proper specification tools, system administrators modify the policy enforcement engines of their access control systems in such a way that they can handle obligations. The bad effects of this ad-hoc programming activity are manifold. First of all there is no clear trace of which obligations are actually handled by the system and in which measure. Second, this development practice is error prone and due to the lack of specification languages, analysis frameworks capable of verifying the correctness of the implemented systems are missing. Moreover, ad hoc programming does not represent a flexible solution. Small changes in the policies may require consistent reprogramming of the enforcement engines.

However, quite recently, standard languages for the specification of policies such as XACML[1] and EPAL[2] have been extended to support the specification of obligations, even if the provided support is only partial. Researchers have also proposed several policy languages that support the specification of obligations and frameworks capable of monitoring their satisfaction (Section 6 shortly discusses main proposals). Such obligations enforcement frameworks could be adapted to work with relational Database Management Systems (DBMSs), however this activity is not straightforward. A possible adaptation strategy consists in implementing adapter modules that interact with the obligation enforcement framework and the DBMS. Such modules should monitor, block or modify the execution of SQL commands, in such a way that the access complies with the obligations defined for the accessed data. Obligations can require to analyze the current state of the accessed database as well as the activities performed within the DBMSs. As such, it is required to introduce an infrastructure that 1) keeps track of the actions performed within the DBMS, and 2) provides analysis tools for commands execution history and the database state. Additionally, obligations are a single aspect of the privacy policies regulating data access and collection within DBMS. Privacy policies are also specified in terms of users/roles, actions, purposes and conditions. As such, enforcing privacy policies within relational DBMSs requires to consider all these elements.

The work in the literature only focuses on subsets of these elements, and only a few of the proposed frameworks have been integrated into DBMSs. What is missing is a privacy policies enforcement framework that comprehensively considers all the above-mentioned concepts and can be smoothly used within DBMSs.

In this work we make a step to fill this gap proposing an approach to: 1) specify privacy policies that include obligations and 2) enforce them in relational DBMSs.

In a previous work [1] we proposed an approach to automatically define an enforcement monitor called Purpose and Role Based Access Control (PuRBAC) starting from PuRBAC policies. PurBAC ensures that SQL queries are executed in such a way that the purposes for which data are processed comply with the purposes for which they are collected, and the user who requests the query execution belongs to a role that have been authorized to the processing. The goal of this work is the definition of

1. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf.
2. http://www.w3.org/Submission/EPAL/.

• *The authors are with Dipartimento di Scienze Teoriche e Applicate Università degli Studi dell'Insubria via Mazzini, 5, Varese 21100, Italy. E-mail: {pietro.colombo, elena.ferrari}@uninsubria.it.*

Manuscript received 3 May 2013; revised 11 Sept. 2013; accepted 14 Oct. 2013. Date of publication 3 Nov. 2013; date of current version 16 July 2014. 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/TDSC.2013.48

extended PuRBAC (ePuRBAC), a monitor that adds to PuRBAC [1] the capability to enforce obligations.

The specification of privacy policies and the definition of the enforcement mechanisms are supported by Modeling and Analysis of Privacy aware Systems (MAPaS), a framework for the development of privacy aware systems that we introduced in [2], [3].

The generation of ePuRBAC exploits model driven engineering technologies [4] and an aspect oriented development [5] approach. No programming activity is required to system administrators, who only need to specify privacy policies and few additional configuration and interaction options.

The proposed approach features several benefits. Differently from other work in the literature, ePuRBAC considers into a unified framework all the building blocks of privacy policies (i.e., obligations, access purposes, roles, authorizations and conditions). Moreover, ePuRBAC is automatically generated. This avoids issues due to *ad hoc* programming and speeds up the development. Furthermore, ePuRBAC supports a very powerful obligation model, able to enforce recursive obligations, obligations that check the current state of the system, as well as obligations that check the system evolution in specific time intervals. It also allows for orchestrating the execution of system actions based on specified temporal patterns. ePurBAC can be used within most of the existing relational DBMSs. We are not aware of other frameworks supporting the generation of a privacy enforcement monitor for relational DBMSs with these capabilities.

The remainder of this paper is organized as follows. Section 2 shortly presents relevant aspects of PaML and the enforcement mechanisms implemented by PuRBAC. Section 3 introduces the concept of obligation. Section 4 presents an extension of PaML to support the specification of obligations, and the specification model of ePuRBAC. Section 5 provides an overview of the approach to generate ePuRBAC. Section 6 surveys related work. Section 7 concludes the paper. Finally, Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2013.48 summarizes the notations that have been used all throughout the paper. Appendix B, available in the online supplemental material, presents the behavioral aspects of the specification model of ePuRBAC, whereas Appendix C, available in the online supplemental material, proves properties of ePuRBAC.

2 Background

In this section, we shortly present PaML and the privacy policy enforcement mechanisms implemented by PuRBAC. More details can be found in [1].

2.1 PaML

PaML is a modeling language that allows the specification of privacy policies that are then enforced by PuRBAC. PaML has been defined as a UML Profile.[3] Therefore, privacy policies are defined by instantiating and connecting PaML stereotypes within a UML model. PaML is mainly based on the PuRBAC model proposed by Byun and Li in [6]. The class diagram in Fig. 1 shows the core elements of PaML which are shortly described in the rest of this section.

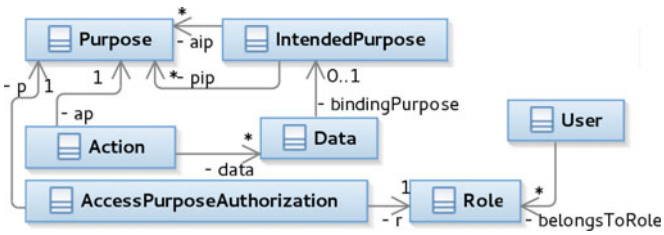


Fig. 1. A high level view on PaML meta-model.

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 into tree structures. In contrast, IntendedPurpose binds the reasons for which data can and cannot be accessed to data themselves. An intended purpose is composed of a set of allowed intended purposes (association pip 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.[5]

Authorized subjects are modelled through some concepts. First, element User is used to model stakeholders characterized by a set of properties, who aim at accessing and processing data. Element Role models functionalities and responsibilities that can be played by users.

Roles can be assigned to Users meaning that the involved stakeholders covers the role. A Role aggregates a set of attributes that specify its intrinsic characteristics and it 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, built starting from the Role attributes.

Roles are hierarchically organized as inverted tree structures. Role properties defined by a role R are therefore inherited by all roles that descend from R in the hierarchical structure. A user U to whom a role RD has been assigned which dominates R, belongs to R when the role constraint is satisfied.

Authorizations are modelled through element AccessPurposeAuthorization, which models the assignment of an access purpose p to a role r, meaning that all users that belong to r are authorized to perform actions with the specified access purpose p. Finally, element Action models an SQL operation that accesses data stored in a database for a specific purpose.

2.2 Privacy Policy Specification and Enforcement

Privacy policies specification within PuRBAC requires to first define a PaML model for the target scenario. A PaML model specifies: 1) hierarchies of purposes and roles,

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

4. http://www.w3.org/TR/P3P11/.

5. PAML has been defined having as target a relational DBMS, however, it can be easily customized for other data models.

2) intended purposes and their associations with data, 3) users who are allowed to interact with the system, and 4) access purpose authorizations.

A PaML model contains all the information required to generate PurBAC, a monitor that enforces all the included policies (an automatic approach which generates PuRBAC starting from a PaML model has been described in [1]). PuRBAC intercepts all SQL query execution requests issued by users and regulate their execution based on the policies specified by means of a PaML model. PurBAC adopts a two stage enforcement process according to which when a user requests the execution of an action it is first checked if there exists an access purpose authorization such that the requester user belongs to its authorized role and the purpose associated with the action descends from the purpose specified in the authorization. In case this check succeeds, the action is allowed only if the intended purposes for which the requested data have been collected comply with the access purpose specified by the action.

The enforcement is achieved by 1) preventing the execution of unauthorized actions and 2) rewriting queries to be executed so that they only access data that they are authorized to process.

3 Obligations

PuRBAC [1] allows one to specify and enforce privacy policies that regulate access to data based on purpose and role-based authorizations. Although PuRBAC can express a wide range of privacy policies, it does not cover the specification of all kinds of privacy requirements. In particular, it does not allow the specification of *obligations* that privacy laws and privacy regulations often express to constrain the execution of actions.

Here and in what follows we will use a running example from the business domain to ease the presentation of the introduced concepts. We consider MyBank, a bank that is investing to protect the privacy of its customers by means of an information system that ensures that the provided banking services comply with the privacy policies of the bank. Our focus will be on services concerning banking accounts and loans.

Obligations can be defined as constraints that refer to the state of the data stored in the database at the time in which the execution of an action (i.e., SQL code) is invoked.[6] For instance, for MyBank accounts with no overdraft facility, required condition to consent withdrawing money from an account is that the balance after the drawing is still positive. In this case, the constrained action is the withdrawal, whereas the obligation is a constraint that specifies $balance - withdrawal > 0$.

Obligations can also specify: 1) the expected state of the data stored in the DB in specific time intervals, or 2) the need that other actions are executed at a given time.

6. In the literature these constraints have been referred to as conditions in [7]. However, in our previous work [1] the term *condition* denoted a constraint that regulates the assignment of users to roles. The framework described in this paper extends [1] inheriting concepts and mechanisms presented in [1]. To avoid ambiguity, we decided to denote this type of obligations as *simple obligations*.

Example 1. Let us consider two policies of MyBank's regulation. The first one introduces an obligation that refers to case 1 above. It specifies that required condition to levy the overdraft tax is that a customer has overcome the credit limit in the last month. The amount to be drawed is proportional to the number of days spent with the overdraft. The second policy refers to case 2 and it is about an optional service activated only when requested by customers: whenever a credit card is used, the owner of the card must be informed via sms of the payment within 30 minutes from the transaction. In this case, the action "payment with credit card" is performed under the obligation of executing the action "send the sms" within 30 minutes from the transaction.

Checking an obligation therefore requires to verify that the expressed condition holds within the specified time intervals. In case the obligation specifies the execution of an action, the satisfaction requires the involved action to be executed according to the provided temporal specification. For instance, in the previous example, the temporal specification for the notification is expressed with respect to the time at which the action is executed. Additionally, obligations may be characterized by a *temporal pattern* that requires the execution of the specified action to be repeated multiple times. For instance, the bank must issue a privacy notice every year to its customers as long as they own an account in the bank.

Based on their relationships with the execution of the actions that constrain, obligations are classified into pre and post-obligations [8]. Pre-obligations are those whose satisfaction is a pre-condition to the execution of the action to which they refer to. The second category includes all obligations that must be fulfilled after the execution of the action to which they refer to.

Obligations of both categories must be verified at the end of the associated time interval. For instance, for the case of the sms, the condition is verified 30 minutes after the transaction. Indeed, if the obligation is not verified 5 minutes after the transaction it does not mean that it will not be verified in the following 25 minutes.

Obligations with a bounded periodic pattern must be evaluated at the end of the last time interval specified by this pattern.

Example 2. Suppose that an internal policy of MyBank requires that a customer profile is reported to the legal office if during the last six months the customer account has continuously overcome the credit limit and even though the customer has been informed of this situation no deposit has been done. Based on the regulation, the bank must warn the customer every two weeks in the last two months before reporting the case to the legal office. The legal office can take action only if the obligation has been satisfied. The obligation can be evaluated at the end of the last interval of two weeks, and, to be satisfied, there must be trace of the notification to the customer in each interval of two weeks, for a total of four notifications.

In some cases there is no limit to the repetitions specified by a periodic pattern. This requires to verify the imposed conditions in all intervals that have been passed so far since the first interval. For instance, the bank must issue an annual account state report to its customers. The

obligation is currently satisfied for a given customer if one report has been issued every year since the year in which the account has been activated.

An obligation is defined to constrain the execution of an action, but in turn it can also require to execute an action. To distinguish between these two kinds of actions, we denote with *supplier*, the action which is constrained by the obligation, (e.g., credit card payment), whereas the term *compulsory* refers to the action to be executed by the obligation (e.g., sending the sms). A compulsory action can be in turn supplier of another obligation. For instance, the sms can be issued only if the consent to access personal information has been granted by the customer that will receive the notification.

A further dimension to be considered in defining a technique for obligation enforcement, is that, whenever an obligation specifies an action, the execution of such an action should be handled in a privacy-preserving way. The same privacy checks are required for supplier and compulsory actions. As underlined by Ni et al. in [8], numerous work in the literature do not consider this aspect and assume that obligations are only system actions. We believe that users may be required to execute these actions as well, as the following example shows.

Example 3. MyBank grants a loan if the risk factor of the requesting customer is below a specified threshold. Risk analysis is achieved by a bank employee who elaborates personal and sensitive data of the customer who asked for the loan. The bank director is responsible of the procedure. As such, in order to grant a loan the director must have authorized it based on the risk factor.

Therefore, into our framework access purpose authorizations and purpose compliance are verified for both supplier and compulsory actions.

4 Obligation Management

Obligation management and action handling requires proper specification and enforcement support. Actions must be handled so that they are executed at a given time, possibly repetitively and under obligations.

To address these issues we propose an extension of PaML and the specification of a monitor that enforces policies that include obligations.

4.1 PaML Extensions

The concepts of action and obligation are strictly connected. Therefore, to proper model the concept of obligation, we first specify the concept of action introducing the basic conceptual elements that concur to its definition. We start to introduce these building elements. Later on we will exemplify how these concepts are used.

The first building blocks are *operations*, parametric SQL code that accesses data, and *access purposes*, that is, the purposes for which the operation aims to access data. Another element involved in the definition of an action is the *temporal pattern*, that is, the temporal specification of when the operation must be executed.

Definition 1 (Temporal pattern). *A temporal pattern tp is a tuple $\langle d, p, n\rangle$, where:*

- *$d : interval$ [7] specifies the time between the activation of the action and the first execution (e.g., 2h 34′ 10″);*
- *$p : interval$[7] expresses the time between the starting of each couple of subsequent executions;*
- *$n : int$ is the number of execution repetitions.*

It is worth noting that Definition 1 also supports the specification of actions instantaneously executed without delay or repetitions. In this case the temporal pattern is $\langle 0, 0, 1\rangle$.

Actions are specified composing all previously introduced elements.

Definition 2 (Action). *An action a is a tuple $\langle op, ap, ps, tp, at, bo, ao\rangle$, where op identifies an SQL operation that accesses data stored in the target database, ap specifies the purpose for which op aims to process data, ps specifies the parameters used by op, tp is a temporal pattern, $at : \{sys, usr\}$ specifies if a is user or system-defined, whereas bo and ao are optional pre and post obligations.*

Example 4. Let us consider a service provided by MyBank that sends the account statement via email[8] to all customers that have activated this service. It requires to access the customer accounts for the purpose of *informing* the subscribers of their current account balance. Let us assume that account data are retrieved by means of *q1*, an SQL query that specifies *customer* as parameter.

Let us also suppose that the activation takes one day and the statement is then issued once every 30 days in the following year (i.e., the temporal pattern is $\langle 1dd, 30dd, 12\rangle$).

The user action *notification* that notifies the account statement to the user is defined through the composition of the above introduced elements: $\langle q1$, *Informing*, *{customer}*, $\langle 1dd, 30dd, 12\rangle$, *usr*, *null*, *null*$\rangle$.

To model the variety of obligation types discussed in Section 3, we introduce here two types of obligations. We denote as *simple* those obligations that define conditions specifying the expected state of the data stored in the database at the time in which the operation is invoked for execution, whereas we denote with *complex* those that specify conditions that must be evaluated or actions that must be executed in given time intervals.

Definition 3 (Simple obligation). *A simple obligation sb is a tuple $\langle vars, exp, ot\rangle$, where $vars$ is a set of variables of a given data type, exp is a logic predicate defined on such variables, and $ot : \{pre, post\}$ specifies whether the obligation must be evaluated before or after the execution of the supplier action. Each variable $v \in vars$ models a property whose value is derived from data collected in the database through the execution of an SQL operation. A variable v is a pair $\langle dt, de\rangle$, where dt represents the data type of v, and de is the identifier of an SQL query whose evaluation results in the value that must be used to initialize v.*

7. Type *interval* corresponds to the homonymuous SQL type. It is characterized by integer fields that define the length of the time interval in terms of days, months, years, hours, minutes and seconds.

8. In this paper, we focus on actions executed within a DBMS. Therefore, we can imagine that at the time in which the email should be sent some data are added to a dedicated table. The insertion event triggers a stored procedure in charge of issuing the email.

Example 5. Suppose that Bob wants to be notified by email of the balance of his account (see Example 4). The notification can be issued only if Bob has requested the activation of the service. This simple obligation is modelled as $\langle\{activation\}, activation = true, pre\rangle$, where $activation$ is a variable that specifies whether the service has been enabled by a given user. $activation : \langle boolean, d1\rangle$ is a boolean variable initialized with the value resulting from the execution of $d1$, where $d1$ identifies the query *select activation from services where sid= 'statement_notification' and userid='Bob'*.

Complex obligations must be evaluated in time intervals defined by a temporal constraint, which is formally defined as follows.

Definition 4 (Temporal constraint). *A temporal constraint tc is a tuple $\langle ts, te, tw, cnt\rangle$, where $ts : interval^7$ and $te : interval^7$ identify the starting and ending points of a time interval, referred to as evaluation interval, within which the obligation must be satisfied, $tw : interval^7$ represents the time that elapses between two subsequent evaluation intervals, whereas $cnt : int$ specifies how many subsequent intervals must be considered.*

ts and te are specified with respect to the time at which the operation specified by the supplier action is invoked: negative values specify instants that precede the execution, whereas positive values those that come after the execution. The combined use of ts, te, tw, and cnt allows the definition of multiple evaluation intervals. Evaluation intervals are defined as follows: $ts_1 = ts$, $te_1 = te$ and $\forall i(1 < i \leq cnt) ts_i = ts_{i-1} + |ts - te| + tw$, $te_i = ts_i + |ts - te|$.

Some constraints restrict the admissible value of tc components. Obviously the ending point of an evaluation interval must not precede its starting point, as such $ts \leq te$. Furthermore, the obligation should be evaluated at least wrt a time interval, and thus $cnt > 0$. Moreover, if multiple evaluation intervals are defined, they must be disjoint, therefore $cnt > 1 \rightarrow tw > 0$, whereas if no repetition is specified, $tw = 0$ and $cnt = 1$.

Obligations may in turn require compulsory actions to be executed. In Example 2, the action consists in the notification to the customer. Compulsory actions are defined according to Definition 2, as such, the included operations can be repetitively executed according to a specific temporal pattern, and they can also be defined as parametric. Complex obligations include an optional expression that binds the parameters of the compulsory action to those of the supplier action. For instance, let us suppose that in Example 2, both the supplier action *legal_report* and the compulsory action *notification* specifies *customer* as parameter. The obligation requires that *notification.customer = legal_report.customer*, since the only customers that can be notified to the legal office are those that have been informed of the overdraft.

Combining all the above discussed elements, complex obligations are formalized as follows.

Definition 5 (Complex obligation). *A complex obligation co is a tuple $\langle ac, be, ot, tc, min, max\rangle$, where ac is a compulsory action, be is a binding expression, $ot : \{pre, post\}$ specifies whether the obligation must be evaluated before or after the execution of the supplier action a; tc is a temporal constraint; $min : int$ and $max : int$ specify how many times ob must be satisfied within the intervals specified by tc. If min/max is set to -1 no lower/upper limit is specified.*

Example 6. Based on Example 2, there must be a notification every 15 days starting from 60 days before the legal office is informed. Therefore, the evaluation interval has a length of 15 days and the first interval starts 60 days before the possible deactivation. As such, $ts = -60dd$ and $te = -46dd$. The associated temporal constraint is $\langle -60dd, -46dd, 1dd, 4\rangle$, and thus, four evaluation intervals are defined: $[-60dd, -46dd]$, $[-45dd, -31dd]$, $[-30dd, -16dd]$ and $[-15dd, -1dd]$. As such, based on Definition 5, the obligation discussed in Example 2 is defined as: $\langle$*notification, notification.customer = legal_report.customer, pre,* $\langle -60dd, -46dd, 1dd, 4\rangle, 1, 1\rangle$, where *notification* is the action that handles the e-mail notification, whereas *legal_report* the one that issues the report to the legal office.

4.2 ePuRBAC Specification

ePuRBAC (extended PuRBAC) must be defined in such a way that, whenever an action *a*, which specifies an access purpose *ap*, a pre obligation *bo* and a post obligation *ao*, is scheduled for execution as requested by a user *u*, *a* is executed if: 1) the pre obligation *bo* is satisfied; 2) *u* belongs to a role that has been authorized to execute actions with an access purpose from which *ap* descends from; and 3) the intended purposes of data accessed by *a* comply with the access purpose *ap*.

The above checks represent *pre conditions* to the action execution. The privacy aware and correct execution also impose as *post condition* that the post obligation *ao* is satisfied after *a* has been executed. Post obligations cannot block the execution of the corresponding action, however in case a post obligation is not satisfied ePuRBAC must find out the policy violation and keep track of it.[9]

In the remainder of this section, we present the specification model of ePuRBAC that fulfils these requirements.

4.2.1 Structural Aspects

For the sake of simplicity, we describe the model using the object oriented paradigm. This allows us to reduce the notational gap between the specification and the implementation of ePuRBAC (presented in Section 5). The model is composed of classes, characterized by attributes and operations. The class diagram in Fig. 2 shows an overview of the model.

One class is introduced for each construct of PaML. Class Action models a process within which an SQL operation is possibly executed multiple times. In accordance with Definition 2, Action is characterized by the attributes *op*, *ps*, *tp*, *ap*, *bo* and *ao* which specify the associated operation, parameters, temporal pattern, access purpose, pre and post obligations. It also provides the operations *execute(u)*, *eval_pc()* and *eval_au(u)*, which will be presented in Section 4.2.2.

9. We are working to integrate compensatory actions that will be executed when post obligations are not fulfilled. This feature will be provided by a future version of ePuRBAC.

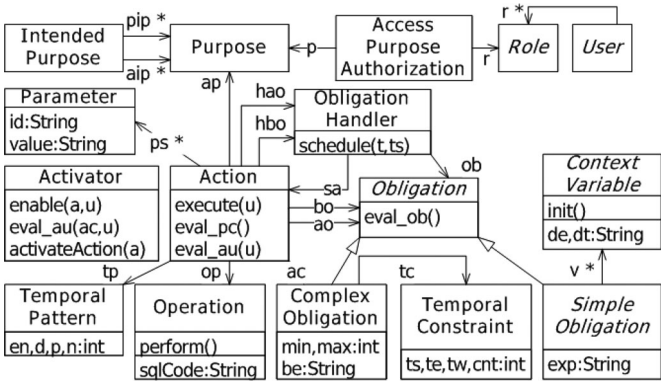


Fig. 2. A high level view of the classes that compose ePuRBAC.

Class Operation models the basic behavioral unit that composes an action. It includes the attribute *sqlCode* that specifies the code of an SQL operation and *perform*, an operation that executes such a code.

The abstract class Obligation models a generic obligation that only introduces the abstract operation *eval_ob()*. According to Definitions 3 and 5, the aspects that characterize simple and complex obligations are defined by the classes *SimpleObligation* and *ComplexObligation*, which specialize *Obligation*.

Class ContextVariable models a variable involved in the definition of a simple obligation.

Class TemporalPattern models a scheduler that invokes the execution of actions at the time instants specified by the attributes d, p and n (cfr. Definition 1).

Class Activator models a handler that regulates the activation of actions ac requested by a user u. Activation requests are handled by operation *enable*, whereas the activation execution by operation *activateAction()*.

Class ObligationHandler deals with the obligations associated with an action. It includes a reference *ob* to the handled obligation, and one reference *sa* to the corresponding supplier action. The operation *schedule* allows configuring ObligationHandler to evaluate the obligation *ob*, t time after the invocation.

An Action a is associated with two obligation handlers, denoted *hbo* and *hao*, respectively, which handle the pre and post obligations *bo* and *ao*, specified by a.

The model also includes classes like User, Role, Purpose, IntendedPurpose and AccessPurposeAuthorization that represent elements of the original version of PaML (see Section 2.1). We refer to [1] for a description of the structural and behavioral features of these classes.

4.2.2 Dynamic Aspects

In this section we shortly discuss at a high level of abstraction the behavioral aspects of ePuRBAC classes. A more detailed description is provided in Appendix B, available in the online supplemental material.

A user requests the activation of a given Action (whose specification is provided in a PaML model) invoking operation *enable* of Activator. If there exists an access purpose authorization which is valid for the user, *activateAction* is invoked. This operation generates an instance of Action, one of TemporalPattern, and two instances of ObligationHandler that will handle the pre and post obligations specified by the action.

TemporalPattern is a scheduler that repeatedly invokes the Action for which it has been defined.

Action is the core element of the behavioral specification of ePuRBAC. An action receives execution requests issued by instances of TemporalPattern and ObligationHandler. Once an execution request has been received, Action checks whether the preconditions to the execution are satisfied. This control exploits operation *eval_pc* and *eval_au*, which 1) evaluates the compliance of the purposes of data accessed by the action with the access purpose of the action itself, and 2) checks if the requester user is authorized to the execution. If the action specifies an obligation of type *pre* it immediately invokes *eval_ob* on such an obligation, checking its satisfaction. If the pre obligation cannot be immediately checked, *schedule* is invoked on ObligationHandler *hbo*, requiring to postpone the evaluation (according to the obligation's temporal constraint *bo.tc*). If all the preconditions are satisfied, *activateOperation* is invoked, which generates a new instance of Operation that is executed invoking *perform* on it. Afterwords, the Action invokes *schedule* on the ObligationHandler associated with *ao*, the obligation of type post. In this way it schedules the analysis of the post obligation *del* time after the operation execution is completed (*del* is derived from *ao.tc*).

ObligationHandler receives the request to perform the analysis of obligations after a given time *del*. In case of pre obligations, after waiting for the specified interval, ObligationHandler invokes *execute* on the supplier action. By contrary, if *ob* is a simple obligation of type *post*, the handler immediately invokes *eval_ob* on *ob*. If *ob* is a complex obligation and the associated compulsory action is of type *usr*, the handler waits for the specified interval before invoking *eval_ob* on *ob*. Finally, if *ob* is a complex obligation and the associated compulsory action is of type *sys*, it activates the compulsory action invoking *enable* on Activator, and after waiting for the required interval, it invokes *eval_ob* on *ob*.

The behavioral aspects of Operation are straightforward. When operation *perform* is executed, the code stored by the attribute *sqlCode* is issued to the DBMS, where it is interpreted.

4.2.3 Checking Obligations

The evaluation of obligations requires to analyze data that describe the state of the system when specific operations are executed. As such, to correctly enforce obligations, we trace the execution of the operations of some of the ePuRBAC classes. More precisely, the execution of the monitored operations causes the generation of *execution events* and the storage of some metadata describing the state of given elements of the system when an event takes place. By properly filtering event tuples, one can derive what happened when specific actions have been executed. This also allows evaluating the satisfaction of obligations.

Table 1 introduces the execution events and the corresponding metadata. In what follows we refer to the data associated with an event e as e-tuple, denoted as $e\langle\rangle$.

TABLE 1
Execution Events and Traced Data

Event ⟨tuple⟩	Cause	Components
activate $\langle ai, id, ps, ts, u, au\rangle$	Operation *enable* is invoked on an instance of Activator	ai is the identifier of the action id identifies the instance of the generated action ps is a pair ⟨*name, value*⟩ that specifies the value associated with the action parameter ts represents the time when *enable* has been invoked u is the user that made the activation request au specifies if there exists an authorization for the execution of ai by user u
check $\langle id, ts, pr, ob\rangle$	Operation *execute* is invoked on an instance of Action	id is the identifier of an instance of Action ts models the time at which the invocation of *execute* has been done pr specifies if the purpose and role based preconditions have been satisfied ob indicates whether the obligation associated with the action has been satisfied, if any
start_ex $\langle id, ts, iop, tm\rangle$	Operation *perform* is invoked on an instance of Operation	id and ts identify the homonymous components of the corresponding *check* tuple iop identifies the instance of Operation that is involved in the execution tm specifies the time at which the execution starts.
stop_ex $\langle id, ts, iop, tm\rangle$	Operation *perform* completes the execution	id and ts identify the homonymous components of the corresponding *check* tuple iop identifies the instance of Operation involved in the execution tm specifies the time at which the execution ends
post_ob $\langle id, ts, tm, at, ob\rangle$	Operation *eval_ob*, invoked on an instance of Obligation of type post, completes the execution	id and ts identify the homonymous components of the *stop_ex* tuple that specifies the execution end of the action for which the post obligation has been introduced at represent the time at which *post_ob* has been generated ob the satisfaction value derived by *eval_ob*

Let *ActiveS*, *CheckS*, *StartexS*, *StopexS* and *PostobS* be the set of all *activate*, *check*, *start_ex*, *stop_ex* and *post_ob* tuples, respectively. We denote *execution trace* the set of all *check*, *start_ex*, *stop_ex* and *post_ob* tuples that have been generated during the life of an Action instance.

Definition 6 (Execution trace). *Let aii be an instance of a given action a. The execution trace for aii, denoted as tr_{aii}, is defined as the tuple $\langle fact, FCheckS, FOpExS, FPostobS\rangle$, where:*

- *fact is the* activate *tuple corresponding to* aii *activation, fact $\in$ ActivateS $\wedge$ fact.id $=$ aii;*.[10]
- FCheckS *consists of all* check *tuples that refer to* aii, $FCheckS = \{c \in CheckS | c.id = aii\}$
- FOpExS *is composed of all pairs* $\langle start\_ex, stop\_ex\rangle$ *that refer to* aii*:* $FOpExS = \{\langle s, e\rangle | s \in StartexS \wedge e \in StopexS \wedge s.id = e.id = aii\}$
- FPostobS *consists of all* post_ob *tuples that refer to* aii *and to a* stop_ex *tuple in* FOpExS*:* $FPostobS = \{p \in FPostobS \,|\, p.id = aii \wedge \exists \langle s, e\rangle \in FOpExS \wedge e.id = aii.ts = p.ts\}$.

Tuples that refer to the same action compose the action execution history.

Definition 7 (Execution history). *The execution history of an action a is the set of all the execution traces that refer to a.* $history_a = \{tr_{en.id} \mid en \in ActivateS \wedge en.ai = a\}$.

Let us introduce a function, named *deriveObTs(tc)*, which given an obligation ob derives the time intervals specified by the temporal constraint *ob.tc*. More precisely,

$$deriveObTs(tc) = \{\langle s, e\rangle \mid \forall i \in [1..tc.cnt] \rightarrow s = tc.ts + (i-1)(|tc.ts - tc.te| + tc.tw) \wedge e = tc.te + (i-1)(|tc.ts - tc.te| + tc.tw)\}.$$

Let *deriveExTs(tp)* be a function that calculates the set of time instants specified by the temporal pattern tp of an action a. More precisely,

$$deriveExTs(tp) = \bigcup_{i \in [1..tp.n]} tp.en + tp.d + (i-1)tp.p.$$

The set of time instances at which a must be scheduled for execution is derived by function *getSchedule*.

getSchedule(a) =

$$\begin{cases} t | \forall tt \in deriveExTs(a.tp) \rightarrow t = tt + getExDelay(a.ob.tc), \\ \quad \text{if } a.ob.ac! = null \text{ and } getExDelay(a.ob.tc) > 0 \\ deriveExTs(a.tp), \text{otherwise.} \end{cases}$$

As such, in case *a* specifies a complex obligation *ob* and the terminal instant of the time interval specified by *ob.tc* is greater than 0, the execution must be delayed.

Let us introduce now the concepts of *valid* and *complete* trace, which we use to define the mechanisms that evaluate the satisfaction of the obligations.

The execution trace of an action *a* is *valid* if it is composed of data tuples showing that 1) the operation *op* specified by *a* has been invoked at time instances that comply with the schedule of the action, and 2) the operation has been executed only when the precondition to the execution are satisfied. More precisely, a valid trace includes 1) one *check* tuple for each time instance *t* that belongs to *getSchedule(a)* and precedes *tm*; 2) only *start_ex* tuples that refer to a *check* tuple whose *ob* and *pr* components are set to true.[11]

Definition 8 (Trace validity). *The execution trace tr_{aii} associated with an instance aii of a given action a is valid at time tm iff*

$$\forall t(t \in getSchedule(a) \wedge t < tm) \rightarrow \exists c \in FCheckS \wedge c.ts = t \wedge \neg\exists cc \in FCheckS \wedge cc.ts \notin getSchedule(a) \wedge \forall \langle s, e\rangle \in FOpExS \rightarrow \exists ccc \in FCheckS \wedge ccc.ts = s.ts \wedge ccc.pr = ccc.ob = true.$$

Let *isValid(tr,tm)* be a boolean function that evaluates the validity of the trace *tr* at time *tm* according to Definition 8.

10. Here and in what follows we use the dot notation to refer to selected components within a tuple.

11. According to def.6, ob and pr keep trace of the preconditions to the execution of a when such an action has been invoked. In case these component are set to true, the preconditions are satisfied.

An execution trace of an action *a* is said *complete* at time *tm* if it is valid and in case *a* specifies a post obligation *oa* that can be evaluated at time *tm*, this has been satisfied for each execution of *a*. This requires that for each *stop_ex* tuple *ee* in the *FOpExS* component of the trace there exists a *po* tuple in *FPostobS* whose *ob* component is set to true and whose *po.tm* $\geq$ *ee.tm*+*getExDelay(oa.tc)*.

Definition 9 (Trace completeness). *tr_{aii} is complete iff*

$$isValid(tr_{aii}, tm) \wedge \forall po \in FPostobS \rightarrow \exists \langle s, e\rangle \in FOpExS \wedge po.ts = e.ts \wedge po.id = e.id \wedge po.ob = true \wedge po.tm \geq stop\_ex.tm + getExDelay(po.tc) \wedge \forall \langle ss, ee\rangle \in FOpExS(ee.tm < t - getExDelay(ob.tc)) \rightarrow \exists po \in FPostobS \wedge po.ts = e.ts \wedge po.id = e.id \wedge po.ob = true.$$

Let *isComplete(tr,tm)* be a function that evaluates the completeness of the trace *tr* at time *tm* based on Definition 9.

Let *a* be a supplier action that specifies an obligation *ob* (either of type pre or post) that introduces a compulsory action *ca*. The satisfaction of *ob* requires *ca* to be activated initializing the action parameters with values that satisfy the binding expression of *ob*. An execution trace of *a* is said *bound* to an execution trace of *ca* if the binding expression of *ob* initialized with the value of the parameters of *a* and *ca* is satisfied.

In what follows we assume that ais is an instance of *a*, aic is an instance of *ca*, and tr_{ais} and tr_{aic} are the execution traces of ais and aic, respectively.

Definition 10 (Trace connection). *tr_{ais} is said to be bound to tr_{aic} if the binding expression of ob, which is initialized with the value of the parameters specified when aic and ais are activated, is satisfied.*

Let *isBound(trs,trc,ob)* be a boolean function that evaluates whether the traces *trs* and *trc* are bound based on the binding expression specified by *ob* according to Definition 10.

The evaluation of an obligation *ob* wrt *ais* (the execution trace of *a*) requires to analyze the existing traces of *ca* which are complete and bound to *ais*. In each interval of the temporal constraint of *ob*, the number of tuples showing that the pre and post conditions to the execution of *ca* are satisfied must be in the range specified for *ob*. Let us denote as *model* the set composed of execution traces of *ca* that satisfy this condition.

Definition 11 (Obligation model). *Let $cas_{ob}^{tr^{ais}}$ a set composed of execution traces of ca which are complete and, based on ob, bound to tr^{ais}.*

$$cas_{ob}^{tr^{ais}} = \{trc | trc \in history(ob.ca) \wedge isComplete(trc, now) \wedge isBound(trc, tr^{ais}).$$

$cas_{ob}^{tr^{ais}}$ is said a model *for ob if, in each interval of the temporal constraint ob.tc the number of* check *tuples collected by the grouped traces of* ca*, which prove that the pre and post conditions to the execution of ca are verified, is in the range* [ob.min..ob.max]

$$\forall i \in deriveObTs(ob.tc) \rightarrow min \leq \sum_{trc \in cas_{ob}^{tr^{ais}}} \sum_{p \in trc.FPostexS} v \leq max,$$

$$where, v = \begin{cases} 1, & if \quad i.s \leq p.ts \leq i.e \wedge p.ob = true. \\ 0, & else. \end{cases}$$

Since $cas_{ob}^{tr^{ais}}$ is composed of traces which are complete, the existence of *p* in *trc.FPostexS* implies the existence of a tuple *c* in *trc.FPostexS* such that *c.ob* = *c.pr* = *true*.

Let *isModel(trcaS,trsa,ob)* be a boolean function built on previous definition, which evaluates whether *trcaS*, the set of traces of *ca*, is a model for *ob* with respect to *trsa*, the trace of *a* that specifies *ob* as obligation.

Definition 12 (Satisfiability of complex obligations). *A complex obligation co is satisfied wrt tr^{ais} if there exists a set of execution traces of action co.ca which is a model for co.*

The operation *eval_ob* is therefore specified as follows:

$$eval\_ob(tr^{ais}) = \exists trsc \subseteq history(this.ac) \wedge isModel(trsc, tr^{ais}, this).$$

Label *this* refers to the instance of Obligation on which *eval_ob* is invoked, as such, *this.ac* refers to the compulsory action introduced by the obligation.

Example 7. Let us consider Example 6. Action *notification* must be executed once every 15 days in the 60 days that precede the notification to the legal office. Let us suppose that today, March 1, 2013, Bob, who is an employee of MyBank, wants to report Mary's profile to the legal office by executing action *legal_report*. Let us assume that Bob has requested the activation of *legal_report* by invoking the operation *enable* and he has been authorized to activate this action. Based on the proposed specification, there exists an execution trace *t0* of *legal_report* that keeps track of this activation. The trace only includes one *activate* tuple, whose *ps* component includes the parameter ⟨*customer, Mary*⟩. Let us suppose that Alice, who is another employee of MyBank, notified the overdraft condition to Mary four times. The execution traces *t1*, *t2*, *t3*, and *t4* keep track of these notifications. Let us assume that the *ps* field of the *activate* tuple included in each execution trace of *notification* specifies the pair ⟨*customer, Mary*⟩. Based on obligation *ob1*, the execution trace of *legal_report* is bound to a trace of *notification* if the binding expression *notification.customer=legal_report.customer* is satisfied. This requires to evaluate the expression that one gets substituting *legal_report.customer* with the value of the parameter *customer* which is enclosed in the *activate* tuple of the execution trace of *legal_report*, and *notification.customer* with the value of the parameter *customer* in the *activate* tuple of *notification*. We get the expression *Mary=Mary*, which evaluates true, as such the considered couples of traces *t0-t1*, *t0-t2*, *t0-t3*, *t0-t4* are bound.

The satisfaction of the obligation requires to analyze the execution history of action *notification* to see whether it includes the proper execution traces. Let us assume that the history only includes *t1*, *t2*, *t3* and *t4*, and that these traces are valid. Each trace aggregates 1 *check* tuple and 1 pair of *start_ex*, *stop_ex* tuples that refer the *check* tuple. The valid traces prove that each execution of *notification* has been authorized. Let us suppose that the *ts* fields of the *check* tuple in *t1*, *t2*, *t3* and *t4* are set to 1/1/2013, 10/1/2013, 5/2/2013 and 25/2/2013, respectively. As such, 2 executions

happened in the first interval of the temporal constraint of *ob1* while 0 in the second. The *max* and *min* field of *ob1* are both set to 1 (see Example 6), therefore, even though all traces are valid the obligation is not satisfied, and consequently *legal_report* cannot be executed.

Note that, if *co.ac* does not specify an SQL operation (i.e., $co.ac.op = null$), the corresponding execution trace does not include any event that refers to the execution of $co.ac.op$, that is, $FOpExS = \emptyset$. Actions with no operation are introduced to specify obligations that consist of a condition that must be checked at given time intervals. The condition is specified as a simple obligation which is defined for the compulsory action $co.ac$. In other words $co.ac$ is supplier of this simple obligation. The specified condition is checked when $co.ac$ is scheduled for execution.

Function *eval_ob* evaluates both pre and post obligations. The analysis of pre and post obligations only differs for the time at which the evaluation is performed.

Example 8. Let us consider again Example 1. An sms must be issued within 30 minutes from a credit card transaction. Let us denote this obligation as *ob2*. In this case the supplier action *payment* specifies *ob2* as post obligation, which in turn specifies *send_sms* as compulsory system action. The obligation is defined as $ob2 = \langle send\_sms,\ send\_sms.customer{=}payment.customer,\ post,\ \langle 1dd, 30dd, 0, 1\rangle\ 1,\ 1\rangle$. Let us consider a scenario in which John did a payment with a credit card one hour ago. The satisfaction of *ob2* is verified if the execution trace of payment includes a single *post_ex* tuple whose ob component is set to true. Such a component is true if there exists a set of execution traces of *send_sms* which is valid, bound to the trace of *payment* under analysis, and the included number of *check* components comply with *ob2* in each interval of the temporal constraint. Based on the specification, the evaluation is done 30 minutes after completing the execution of *payment*. The evaluation process follows the same scheme presented in Example 7.

The satisfaction of simple obligations does not require the analysis of execution traces and time intervals. It only relies on the evaluation of the conditional expression.

Definition 13 (Satisfiability of simple obligations). *A simple obligation so is satisfied if substituting within so.exp each occurrence of a variable v in so.vars with the value resulting from the interpretation of the derivation expression v.de, so.exp is satisfied.*

Let $\underline{v}$ be a vector which is defined such that each of its components is a variable included in *so.vars*, and let *eval(exp)* be a function that evaluates the value of the boolean expression *exp*.

eval_ob is specified in such a way that:
$eval\_ob() = eval(exp[init(\underline{v})/\underline{v}])$.

Example 9. Let us consider the evaluation of the simple obligation presented in Example 5. *exp* is set to *activation= true* and the only defined variable is *activation*, whose *de* component is set to *d1*. *d1* identifies the query *select activation from services where sid='statement_notification' and userid='Bob'*. Therefore, $\underline{v}$ is composed of a single variable *activation*. The execution of *init* causes the execution of *d1*. Let us suppose that this query returns *true*. This value substitutes each occurrence of *activation* in *exp*. Due to the substitution, *exp* becomes *true=true*, which evaluates to *true*. As such, the obligation is satified.

The specification of *eval_ob* is based on the definition of satisfiability of simple and complex obligations. The function returns *true* iff the corresponding obligation is satisfied.

The correctness of the obligation management performed by ePuRBAC is stated by Theorems 1 and 2. Theorem 1 states the correctness of the definition of *eval_ob()*, whereas Theorem 2 states that ePuRBAC properly handles the scheduling of the obligations analysis.

Theorem 1. *Let ob be an obligation defined by an action* a. *ob is satisfied iff ob.eval_ob()=true.*

Theorem 2. *Let a be an action that specifies bo and ao as obligation of type pre and post, respectively.*

1. *ePuRBAC evaluates* bo *every time* aii *is scheduled for execution.*
2. *The obligation ao is evaluated t time after each execution of a, where t is set to 0 if ao is a simple obligation, whereas t corresponds to the ending time of the time interval of ao.tc, if ao is a complex obligation.*

The theorems proofs are presented in Appendix C, available in the online supplemental material.

5 ePuRBAC

ePuRBAC is defined as a Java application that implements the specification model presented in Section 4.2. ePuRBAC is partially derived from the specification model, and partially generated starting from the PaML model that includes the policies that regulate the execution of SQL code on the target DBMS. Let us refer to such a PaML model as the *reference* model.

ePuRBAC functionalities can be classified into two distinct groups: *interaction* and *control*.

The *interaction* functionalities are the standard ones required to support the execution of SQL code, such as: loading vendor specific driver to communicate with a DBMS, creating data definition and data manipulation commands to be issued to the DBMS, executing these commands on the target DBMS, and so on.

The *control* functionalities are those required to verify if the execution of SQL code requested by users complies with properties specified by the privacy policies, and block the execution in case of non-compliance.

JDBC[12] provides a rich API supporting the straightforward definition of interaction functionalities. In contrast, due to the complex nature of privacy policies, which typically involve numerous aspects (i.e., roles, purposes, actions, conditions, obligations), the implementation of control mechanisms is a complex activity (e.g., authorization analysis require to consider role hierarchies and the inheritance of attributes and conditions). Due to space constraints, we present here only the implementation of control mechanisms that allow evaluating the satisfaction

12. http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/.

of obligations (purpose compliance and authorization analysis have been discussed in [1]).

Control functionalities are introduced by means of a formal approach. We consider the set of the Java classes of ePuRBAC as a model that represents the context with respect to which the satisfaction of the obligations must be evaluated. Based on this model, we provide a formal specification of the control mechanisms using the Object Constraint Language (OCL).[13] The code that implements the specified control functionalities is automatically derived from this OCL specification. More precisely, AspectJ [9] code is derived from the OCL specification using the Dresden OCL Toolkit.[14]

The definition of an ePuRBAC monitor for a given reference model uses an automatic generation approach, which can be sketched as a sequential process articulated into 3 steps. It first derives from the PaML model the byte code of concrete Roles, Users, ContextVariables and SimpleObligations classes and the OCL code representing the conditions introduced in the definition Roles and SimpleObligations. Afterwords it takes as input the OCL constraints and generates the corresponding AspectJ code. Finally it instantiates both the statically provided and dynamically generated classes initializing the properties of the generated objects based on the information included in the reference model.

The resulting application is capable of enforcing the policies included in the reference model.

No programming activity is required to system administrators. They only have to specify a PaML model exploiting the editing functionalities of MaPAS and to invoke the generation of ePuRBAC for the edited model.

5.1 The ePuRBAC Skeleton

ePuRBAC includes a Java class for each of the classes of the specification model, as such it is composed of the classes Action, Obligation, ContextVariable, TemporalPattern, TemporalConstraint, User, Role, Purpose, IntendedPurpose and AccessPurposeAuthorization. It also includes Java classes that implement those elements of the reference model, such as roles, users, obligations and context variables, whose structural aspects cannot be a priori determined since are different for each application scenario. Classes belonging to this category are defined as concrete classes that extend the abstract classes Role, User, Obligation and ContextVariable.

The definition of concrete classes starting from elements of a PaML model has been automated using functionalities of the Javassist project.[15] The generation uses the same technique that we used for PuRBAC [1], therefore we do not discuss it here. In the remainder of this section we shortly introduce classes and features required for evaluating obligations.

The first class is *Action*, whose method *execute* invokes *perform()* on the Operation referred to by the attribute *op*.

Class Operation handles the interpretation of the SQL code specified by attribute *sqlCode*. Operation extends Thread and its method *run()* issues the SQL code that must be executed in the DBMS by means of a JDBC connection. Method *perform* generates an instance of Operation (a new Thread) and invokes *start()* on it.

Class TemporalPattern handles the scheduling of the activities carried out by the Action to whom it is associated with. It relies on ScheduledExecutorService and ScheduledThreadPool objects, which are set to invoke *execute* on the handled Action at the time specified by attributes *d*, *p* and *n* (cfr. Section 4.1).

Abstract class *SimpleObligation* models an abstract condition that must be satisfied to allow the execution of the supplier action. Concrete extensions of this class include a set of context variables and a conditional expression.

Context variables are defined as concrete extension of the abstract class *ContextVariable*. The extension introduces the attributes: 1) *dt*, which specifies the type of the context variable; 2) *value*, which keeps track of its value; and 3) *de*, the identifier of the SQL query that specifies the derivation criteria. The extension also defines *init()* in such a way that the query referred to by *de* is executed and its result set is used to initialize *value*.[16]

The conditions defined by SimpleObligations are logic formulas specified using OCL syntax, which predicate on the expected value of context variables.

Example 10. Let us consider the simple obligation in Example 5. Based on the proposed strategy, we introduce class *Activation* as an extension of ContextVariable characterized by attribute *value* of type *boolean*, *dt* set to *"boolean"*, and *de* to *"d1"*. *init()* is defined in such a way to execute *d1* and to initialize *value* with the returned result set. We also introduce class *ConditionActivation*, which extends SimpleObligation with the attribute *activation* of type *Activation* and *exp* that is set to *activation.value=true*.

The definition of classes TemporalConstraint and ComplexObligation is straightforward as they are characterized by the same set of attributes defined by the homonymous classes of the specification model.

Finally, the definition of class ObligationHandler relies on ScheduledExecutorService and ScheduledThreadPool objects to invoke the analysis of the associated Obligation *t* time after *schedule* is invoked.

5.2 Development of Control Mechanisms

The controls mechanisms, which are specified with OCL, consists of operations provided by ePuRBAC classes, pre and post conditions to the execution of operations and invariant constraints. The OCL specification is provided as input to a code generation approach that complements the ePuRBAC skeleton with control functionalities.

5.2.1 OCL Specification

In ePuRBAC, a simple obligation is a class that extends SimpleObligation with a set of context variables and a conditional expression expressed using OCL syntax (cfr. Section 5.1). We use this expression for deriving an invariant constraint to be assigned to the involved class.

13. http://www.omg.org/spec/OCL/.
14. http://www.dresden-ocl.org.
15. http://www.csg.is.titech.ac.jp/ chiba/javassist/.
16. The extension is required since it is not possible to a priori determine the type of value.

Example 11. Let us consider the classes ConditionActivation and Activation specified in Example 10. Constant *exp* of ConditionActivation is set to *"activation.value=true"*. The OCL invariant constraint derived from *exp* and assigned to ConditionActivation is: *context ConditionActivation inv: activation.value=true*

Operation *eval_ob*, to be included in SimpleObligation, is defined in such a way that, after invoking *init()* on all the grouped ContextVariables, it checks the invariant constraint[17] (see Listing 1).

```
context SimpleObligation def: eval_ob():boolean=
  init_cd() and checkInvariants()
context SimpleObligation def: init_cd():boolean=
ContextVariable.allInstances()->
 select(cv|cv.belongsTo(self))->
  iterate(cv:ContextVariable;r:boolean=true|
  r and cv.init())                                    (1)
```

An obligation is evaluated invoking *eval_ob()*.

Example 12. Suppose to define an obligation that prevents MyBank services to process personal data of a minor without parental consent. The obligation is expressed as $age < 18$ *implies parentalConsent=true*. Concrete ContextVariable classes denoted *AgeCV* and *ParentalConsentCV*, are defined for *age* and *parentalConsent*. The *de* attribute of these classes refers to an SQL query. The value derived from the execution of this query is used to initialize the variable. Let us suppose that query *"select age from subjects where..."* is defined for *AgeCV*. The obligation is defined as the class *ProcessingConsentC*, which extends *SimpleObligation* with the attributes *age* of type *AgeCV*, *parentalConsent* of type *ParentalConsentCV*, and *exp* which is set to *"age.value* < 18 *implies parentalConsent.value=true"*. The OCL invariant constraint derived from *exp* is:

context ProcessingConsentC inv: age.value<18 implies parentalConsent.value=true.

The evaluation of a complex obligation *ob* requires to check that the specified compulsory action *ca* is executed in each time interval of the temporal constraint *tc*.

```
context ComplexObligation
def:eval_ob(invTime:Integer):Boolean=
let start:Integer=invTime+ts in
let stop:Integer=invTime+ts+
  cnt*((te-ts).abs()+tw)-tw in
let checkS:Set(Check)=
  LogHandler::getCheck(ac.id,start,stop,self) in
if ac.oa.oclIsUndefined()
then let crange:Set(Integer)=Set{1..cnt}in
 crange->forAll(j|
  let cnum:Integer=count_check(checkS,j) in
  cnum<=max and cnum>=min)
else let postObS:Set(Check)=
 LogHandler::getPostob(ac.id,start,stop,self) in
 let prange:Set(Integer)=Set{1..cnt}in
 prange->forAll(j|
  let bn:Integer=count_check(checkS,j) in
  let an:Integer=count_postob(postobS,j) in
  bn<=max and an<=max and an>=min and bn>=min
  and an<=bn)
endif                                                 (2)
```

17. To this aim we use a functionality of the Dresden OCLToolkit which allows evaluating OCL invariants on demand.

Listing 2 shows the OCL specification of *eval_ob*, which is defined as an operation of ComplexObligation.

eval_ob first calculates the ending points *start* and *stop* of the macro interval that groups all the intervals specified by the temporal constraint *tc*. Then, it calculates *checkS*, the set of *check* tuples that refer to *ac*.

The resulting set is further refined selecting those tuples whose *ts* components specify a timestamp that falls in *[start,stop]*, the macro interval of *ob.tc*.

The analysis proceeds invoking the operations *count_check* and *count_postob* that count the *check* and *post_ob* tuples (*post_ob* are considered only if the compulsory action specifies a post obligation) included in each interval specified by *ob.tc*. The obligation is satisfied if the number of *check*/*post_ob* tuples in each interval, whose *ob* and *pr* components are set to true (*pr* is considered only for *check* tuples), is in the range *[min,max]*.

Example 13. Let us consider Example 6 and suppose that Bob wants to report Mary's profile to the legal office and in the past he has notified the overdraft to Mary and John. *getCheckTuples()* extracts all *activate* tuples grouped by traces of *notification*, whose parameters satisfy the binding expression *notification.customer=legal_report.customer*. The expression is evaluated wrt the execution trace of *legal_report* which is currently constrained by the obligation and whose *activate* tuple specifies as parameter $\langle customer, Mary \rangle$. The binding analysis substitutes *legal_report.customer* with the value "Mary" of the parameter *customer* specified by the *activate* tuple in the involved trace of *legal_report*, and *notification.customer* with the values specified in the *activate* tuples of *notification*, i.e., with "Mary" and "John". Finally, *getCheckTuples* selects those tuples that satisfy the binding, i.e. those that refer to Mary.

The variables *start* and *stop*, which delimit the macro interval of the temporal constraint introduced by *ob1*, are initialized to 30/12/2012 and 1/3/2013, respectively. *getCheckTuples()* extracts all *check* tuples that refer to a previously selected *activate* tuple of *notification*, whose *ts* components specify a date in the range [30/12/2012..1/3/2013].

The extracted tuples are partitioned into the 4 intervals specified by *tc* and counted to check that they are in the range [1..1]. The obligation is not satisfied since the interval [30/12/2012..12/1/2013] includes two tuples.

Based on their type, the satisfaction of obligations is either a pre or a post condition to the execution of the supplier action that specifies them. Listing 3 specifies the precondition based on the satisfaction of obligations of type *pre*. A simple obligation is evaluated as soon as *execute* is invoked. In contrast, as discussed in Section 4.2 and Appendix B, available in the online supplemental material, the temporal constraint of a complex obligation may require to postpone its evaluation to the ending time of the macro interval specified by the temporal constraint (*stop* in Listing 3).

The evaluation of the post obligation *ao* follows the same scheme.

5.2.2 Generating Code from the OCL Specification

The OCL specification presented in Section 5.2.1 are used to derive control functionalities that will be integrated into

ePuRBAC. The proposed approach uses AspectJ code generation functionalities of Dresden OCLToolkit.

OCL Operations (see e.g., Listing 2) are automatically converted into AspectJ code. AspectsJ code is also derived to check OCL pre and post conditions and invariant constraints (e.g., Listing 4 shows the code derived from the invariant contraint *"context ConditionActivation inv: activation.value=true"*).

```
context Action::execute(User u) pre:
let isSO:Boolean=bo.oclIsTypeOf(SimpleObligation) in
let now:Integer=getCurrentTimestamp() in
 if(isSo)
  then bo.eval_ob()
  else let stop:Integer=bo.ts+bo.cnt*
   ((bo.te-bo.ts).abs()+bo.tw)-bo.tw in
    if (stop<=0) then bo.eval_ob()
    else hbo.schedule(stop,now) endif
 endif
```
(3)

```
public privileged aspect ConditionActivation{
 declare parents: ConditionActivation
 implements OclCheckable;
public void ConditionActivation.checkInvariants(){}
protected pointcut checkInvariantsCaller(
  ConditionActivation aClass):
  call(void checkInvariants())
  && target(aClass);
 after(ConditionActivation aClass) :
  checkInvariantsCaller(aClass) {
   if (aClass.getClass().getCanonicalName()
   .equals("ConditionActivation")) {
    if (!(aClass.activation.value ==
    new Boolean(true))) {
     String msg = "Constraint violation"
     + aClass.toString() + ".";
     throw new RuntimeException(msg);}}}}
```
(4)

The bytecode of the ePuRBAC classes is weaved together with the one of the generated AspectJ code. As a result of this integration, ePuRBAC provides complementary control operations.

5.3 Early Assessment

In this section, we discuss preliminary results of the ePuRBAC validation process. The thorough validation will be one the goals of future work. Four versions of ePuRBAC have been derived from application scenarios characterized by different sets of privacy policies regulating the activities performed within a MySQL DBMS. The considered policy sets differ for the specified obligations and the scheduling of actions. The involved DBMS manages a database with four tables, each of which stores around 150 tuples. The scenarios share a purpose set characterized by 16 purposes. Each application scenario considers the same eight predefined SQL queries that access columns of the DB tables. Such operations are composed to form 12 actions characterized by different temporal patterns.

In our analysis we checked simple and composite obligations of type pre and post, by varying temporal constraints and the scheduling of these actions. Table 2 summarizes relevant aspects of the validation scenarios.[18] Such a counting includes both supplier and compulsory actions. The obligation columns report the number of considered obligations, specifying how many of them are of type *pre*.

TABLE 2
Early Validation Scenarios

Case	Max act (compulsory)	Avg act (compulsory)	Simple obl (pre)	Complex obl (pre)
#1	3 (1)	2 (1)	10 (5)	2 (1)
#2	8 (3)	3 (2)	2 (1)	10 (5)
#3	10 (6)	8 (4)	4 (2)	8 (4)
#4	9 (6)	7 (5)	2 (1)	10 (5)

The generated versions of ePuRBAC performed correctly in all tests: obligations have always been evaluated correctly and compulsory actions have always been executed. However, this preliminary validation allowed us to identify possible performance improvements. Indeed, in the scenarios #3 and #4, actions were scheduled for executions and obligations were evaluated a few milliseconds later than what specified by privacy policies. Although we believe that delays of milliseconds are negligible, we investigated the reasons of this behavior. These scenarios were characterized by numerous actions that were active at the same time (see Table 2). In both the considered cases, two of the compulsory actions were defined in such a way that in every period specified by the temporal pattern another action had to be executed according to a further temporal pattern. We think the delays are due to memory management of numerous threads executed in parallel. Indeed, in the current approach a thread that handles the execution of actions is activated for every specified temporal pattern. Therefore, if a temporal specification involves multiple patterns, multiple threads are executed at the same time. Whenever a thread completes its execution the memory handled by the thread should be freed. This activity is performed by the garbage collector, a tool of the Java virtual machine. However, the scheduling of the activities of this tool is non deterministic, and as a consequence, unexpected delays can be introduced and can affect the scheduling of other activities in the system. To address this issue, we are considering to modify the approach in such a way that within ePuRBAC a unique thread is generated for each composite temporal specification. By diminishing the number of threads that are concurrently executed we optimize the memory consumption, allowing the parallel management of a bigger number of actions and obligations. This choice should reduce the number of non deterministic delays introduced by the garbage collector. To optimize the accuracy of the scheduling we are also considering the use of a real-time Java virtual machine. However, since a real-time virtual machine uses a different memory model and thread management system, this choice would require to significantly modify the ePuRBAC code.

6 Related Work

In the literature, obligations have been recognized as important requirements for the development of privacy and security aware systems. However, there is still no consensus on the precise meaning of the term obligation, with the result that different authors assign different meaning to it. For instance, Park, and Sandhu [7] propose $UCON_{ABC}$, an

18. In the table, Max and Avg specify the maximum and average number of actions that are active at the same time, respectively.

access control model that formalizes the concept of *pre-obligations* and *ongoing obligations*, namely obligations that specify actions to be executed during the access. Katt et al. in [10] propose an extended version of the UCON model [7] which introduces post-obligations.

In other work [11], [12], [13], obligations mean those requirements that must be fulfilled after data access has been done. Therefore, different from ePuRBAC, no support is provided to pre-obligations. The authors of [13], [14] present a framework supporting the enforcement of obligation-based policies. This framework consists of: 1) xSPL, a language that supports the specification of obligation based, history based and RBAC policies, and 2) Heimdall [13], a middleware in charge of policy enforcement. Compensatory actions are executed in cases the actions do not fulfill the obligations. Besides allowing one to implement this behavior by means of post obligations, ePuRBAC also allows one to execute actions only when pre-obligations are satisfied. Bettini et al. in [11] denominate pre-obligations as provisions, and describe a formalization of provisions and (post) obligations based on Horn clauses. The authors also propose a framework to control them and to properly react (e.g., modifying users trustworthiness) in case they are not fulfilled. However, different from ePuRBAC, no support is given for obligations characterized by temporal constraint and periodic patterns.

Ni et. al in [8] propose a model that merges some of the previously introduced concepts. They consider both pre and post obligations and provide support to specify temporal constraints that denote when obligations must be evaluated. Our obligation model is built on top of the model proposed by Ni et al. [8]. Our model is recursive, it allows the definition of chain of obligations. We consider our model more effective than the one proposed in [8] for a number of reasons. Indeed, in [8] the execution of the actions introduced by the obligation (i.e., those that correspond to the compulsory actions in our model) only starts at the time in which the action for which the obligation has been specified is invoked for execution (the supplier action in our model). The model does not catch a quite common situation in which the execution of compulsory actions is independent from the execution of supplier actions. This causes the systematic delayed execution of the supplier action.

Irwin et al. [15] consider obligations as a contract between the subject and the system. They propose a framework that derives the reasons for which some obligations are not satisfied. This allows administrators to take countermeasures in case the unsatisfiability is due to user negligence or insufficient authorization. Based on [15], a fault analysis approach has been proposed in [16], which aims at identifying who was at fault when obligations have been violated. However, the framework does not support pre-obligations and periodic obligations.

A recent work by Jafari et al. [17] proposes a framework supporting the specification and enforcement of purpose based privacy policies. Purposes are defined as activity happening in the future for which the execution of an action is a prerequisite. Some similarities can be found with obligations, but the formalization proposed in [17] does not explicitly introduce the obligation concept. Moreover, even though basic pre-obligations can be expressed with the pair "action—future activity", this model cannot express post obligations, temporal-based and conditional obligations.

Li et al. in [18] propose an XACML-based[1] approach for specifying and enforcing obligations. The authors also present ExtXACML, a prototype tool which has been built on top of SUN's XACML implementation.[19] Obligation modules can be defined and integrated into ExtXACML to handle different types of obligations. However, these modules must be developed, whereas ePuRBAC is automatically generated.

In the recent years policy languages, such as XACML[1] and EPAL[2], have been extended to support the specification of security policies that include obligations. Nevertheless, the expressive capabilities of these languages is still minimal. No proper support is given to temporal contraints and repeating obligations. Moreover, most of them allow only the specification of system obligations.

7 Conclusions

In this paper, we have presented an approach to the definition of ePuRBAC, a monitor that performs runtime enforcement of privacy policies that include obligations within relational DBMSs. The automated approach that generates ePuRBAC uses principles and technologies of model driven engineering and aspect oriented programming, without requiring programming actvities.

As future work we plan to cooperate with companies and public institutions with the aim to thoroughly analyse ePuRBAC performances through case studies. We plan to specify the privacy policies introduced by several privacy regulations and privacy laws and to test the derived monitor on different DBMSs. We plan also to extend the obligation language to the support of on going obligations [7].

References

[1] P. Colombo and E. Ferrari, "Enforcement of Purpose Based Access Control within Relational Database Management Systems," *IEEE Transaction on Knowledge and Data Eng. (IEEE TKDE)*, to appear.

[2] P. Colombo and E. Ferrari, "Towards a Framework to Handle Privacy Since the Early Phases of the Development: Strategies and Open Challenges," *Proc. IEEE Sixth Int'l Conf. Digital Ecosystems Technologies (DEST)*, 2012.

[3] P. Colombo and E. Ferrari, "Towards a Modeling and Analysis Framework for Privacy-Aware Systems," *Proc. Int'l Conf. Privacy, Security, Risk and Trust and Int'l Conf. Social Computing (PASSAT)*, 2012.

[4] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," *Proc. Future of Software Eng. (FOSE)*, 2007.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming (ECOOP '97)*, pp. 220-242, 1997.

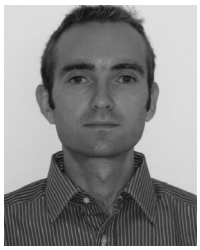
[6] J. Byun and N. Li, "Purpose Based Access Control for Privacy Protection in Relational Database Systems," *The Int'l J. Very Large Data Bases*, vol. 17, no. 4, pp. 603-619, 2008.

[7] J. Park and R. Sandhu, "The UCON_{ABC} Usage Control Model," *ACM Trans. Information and System Security*, vol. 7, no. 1, pp. 128-174, 2004.

[8] Q. Ni, E. Bertino, and J. Lobo, "An Obligation Model Bridging Access Control Policies and Privacy Policies," *Proc. 13th ACM Symp. Access Control Models and Technologies (SACMAT)*, 2008.

19. http://sunxacml.sourceforge.net.

[9] R. Laddad, *Aspectj in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., 2009.
[10] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert, "A General Obligation Model and Continuity: Enhanced Policy Enforcement Engine for Usage Control," *Proc. 13th ACM Symp. Access Control Models and Technologies (SACMAT)*, 2008.
[11] C. Bettini, S. Jajodia, X.S. Wang, and D. Wijesekera, "Provisions and Obligations in Policy Rule Management," *J. Network and Systems Management*, vol. 11, no. 3, pp. 351-372, 2003.
[12] M. Hilty, D. Basin, and A. Pretschner, "On Obligations," *Proc. European Symp. Research in Computer Security (ESORICS '05)*, pp. 98-117, 2005.
[13] P. Gama, C. Ribeiro, and P. Ferreira, "Heimdhal: A History-Based Policy Engine for Grids," *Proc. IEEE Sixth Int'l Symp. Cluster Computing and the Grid (CCGRID)*, 2006.
[14] C. Ribeiro and P. Ferreira, "A Policy-Oriented Language for Expressing Security Specifications," *Int'l J. Network Security*, vol. 5, no. 3, pp. 299-316, 2007.
[15] K. Irwin, T. Yu, and W.H. Winsborough, "On the Modeling and Analysis of Obligations," *Proc. 13th ACM Conf. Computer and Comm. Security (CCS)*, 2006.
[16] W.H. Winsborough, T. Yu, and K. Irwin, "Assigning Responsibility for Failed Obligations," *Trust Management II*, vol. 263, no. 1, pp. 327-342, 2010.
[17] M. Jafari, P. Fong, R. Safavi Naini, and K. Barker, *A Framework for Expressing and Enforcing Purpose-Based Privacy Policies*, 2013.
[18] N. Li, H. Chen, and E. Bertino, "On Practical Specification and Enforcement of Obligations," *Proc. Second ACM Conf. Data and Application Security and Privacy (CODASPY)*, 2012.



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. He works within 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 of the K&SM Research Center. Her research activities are related to 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.