Model-Driven Run-Time Enforcement of Complex Role-Based Access Control Policies*

Ameni Ben Fadhel
University of Luxembourg
Luxembourg
ameni.benfadhel@uni.lu

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Lionel C. Briand
University of Luxembourg
Luxembourg
lionel.briand@uni.lu

ABSTRACT

A Role-based Access Control (RBAC) mechanism prevents unauthorized users to perform an operation, according to authorization policies which are defined on the user's role within an enterprise. Several models have been proposed to specify complex RBAC policies. However, existing approaches for policy enforcement do not fully support all the types of policies that can be expressed in these models, which hinders their adoption among practitioners.

In this paper we propose a model-driven enforcement framework for complex policies captured by GemRBAC+CTX, a comprehensive RBAC model proposed in the literature. We reduce the problem of making an access decision to checking whether a system state (from an RBAC point of view), expressed as an instance of the GemRBAC+CTX model, satisfies the constraints corresponding to the RBAC policies to be enforced at run time. We provide enforcement algorithms for various types of access requests and events, and a prototype tool (MORRO) implementing them. We also show how to integrate MORRO into an industrial Web application. The evaluation results show the applicability of our approach on a industrial system and its scalability with respect to the various parameters characterizing an AC configuration.

CCS CONCEPTS

• **Security and privacy → Access control**; • **Software and its engineering → Model-driven software engineering**;

KEYWORDS

role-based access control, enforcement, policies, model-driven engineering

ACM Reference Format:

Ameni Ben Fadhel, Domenico Bianculli, and Lionel C. Briand. 2018. Model-Driven Run-Time Enforcement of Complex Role-Based Access Control Policies. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238167

*This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), from the Luxembourg National Research Fund (FNR) (grant No FNR/P10/03), and from the University of Luxembourg (grant "ReACP").

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238167

1 INTRODUCTION

Access control (AC) systems have been used to restrict a user to access critical resources within an enterprise. One of the most used AC models is Role-based Access Control (RBAC), which allows a user to access a resource or to perform an operation based on her role, e.g., her job position within an enterprise.

The first RBAC model (RBAC96 [43]) defines *(authorization) policies* by mapping each user to a set of roles and each role to a set of permissions; a permission is defined as an abstraction of a set of operations that can be performed on a set of objects, i.e., resources. Therefore, a user is allowed to perform only the operations of the permissions associated with her role. Several proposals have extended RBAC96 to support new types of policies, such as delegation [18, 46, 54, 56] and contextual [3, 9, 10, 28, 42]. In addition to extended RBAC models, several policy languages have been proposed to ease the specification of complex RBAC policies on top of these models. In this paper we consider our recent proposal of an RBAC model, called GemRBAC+CTX [6, 8], which has been designed to be very expressive, by seamlessly integrating the various types of authorization policies surveyed in the literature and classified in a taxonomy [6]. To the best of our knowledge, the GemRBAC+CTX model is the only one supporting *all* the policies types classified in [6] (prerequisite [2, 43], cardinality [1], precedence and dependency [44], role hierarchy [43], separation of duty (SoD) [45], binding of duty (BoD) [48], delegation and revocation [18, 54], contextual [10, 28]) and their different facets. Moreover, GemRBAC+CTX is paired with a high-level policy specification language, GemRBAC-DSL [7], to encourage its adoption among practitioners. We formalized the policies supported by GemRBAC+CTX as OCL (Object Constraint Language [39]) constraints [6, 8], to enable their operationalization. Following a model-driven approach, we defined the semantics of GemRBAC-DSL by mapping the language constructs to the OCL constraints presented in [6, 8].

A big gap between the definition of new, richer RBAC models and languages and their adoption in practice is the availability of an *enforcement mechanism*: the latter is a component that receives a user access request (hereafter referred to as "AC request") at run time and makes an access decision (*allow*/*deny*) based on the policies configured for a system. The lack of enforcement mechanisms for more expressive RBAC models has favored the adoption on a large scale [38] of the standard RBAC96 model, which is, however, the least expressive model. For instance, one of our industrial partners—developing communication solutions for manipulating sensitive data in critical situations such as natural disasters or wars—needs *to* *specify and enforce* complex policies that involve the user's context (space and time), the history of operations performed by the user, and role delegations. An example of such a policy, in the context of asylum seekers assistance, is: *an operator with the role "coordinator" can register asylum seekers only if he is located within a 20 miles radius from the base camp, from 8am to 7pm.*

Such policies are not supported by RBAC96, but can be specified using GemRBAC+CTX (and expressed using GemRBAC-DSL). However, they cannot be enforced because *there is no enforcement mechanism for checking the complex AC policies that are supported by the GemRBAC+CTX model.* Indeed, state-of-the-art enforcement mechanisms (such as [27, 29, 31, 33–35, 41, 47, 54]) support AC models which are much less expressive than GemRBAC+CTX.

In this paper, we aim to fill this gap, by proposing an automated mechanism to enforce complex AC policies defined on top of the GemRBAC+CTX model and expressed in the GemRBAC-DSL language. We follow a model-driven engineering (MDE) [15] approach for enforcement based on standardized technologies such as the Unified Modelling Language (UML) and Object Constraint Language (OCL) [39]. More specifically, we *reduce the problem of enforcing RBAC policies to the evaluation of the corresponding OCL constraints on an instance of the GemRBAC+CTX model.* One advantage of such an approach is that the translation of GemRBAC-DSL policies into OCL constraints is already defined [7]: by adopting a model-driven approach, we can leverage the existing operationalization through OCL constraints of GemRBAC-DSL policies. Moreover, OCL constraint checking, given that OCL is a standard, is a consolidated technology implemented in mature tools, such as the constraint/query evaluator included in Eclipse OCL [19].

Our model-driven enforcement approach not only enforces policies upon receiving a user request, but also provides a continuous enforcement after making an access decision, by handling events corresponding to changes in the RBAC configuration, to guarantee that a new configuration still fulfills the AC policies. The relevant access decisions are re-evaluated whenever a new change, from an AC point of view, occurs at the system state level (e.g., a user changes location, or a new user is authenticated). For instance, whenever a user changes her location, our enforcement mechanism checks whether her active roles should be deactivated (because of her new location). This enforcement is known as usage control [40] in the area of AC.

Although there have been a few proposals for model-driven enforcement [27, 33, 47], they adopt an RBAC model much simpler than GemRBAC+CTX (and thus can deal with a limited set of AC policy types); moreover, they consider a subset of the AC requests/events supported by our approach. Furthermore, defining algorithms to precisely decide when and how to enforce OCL constraints corresponding to GemRBAC-DSL policies, as well as defining and engineering an architecture to integrate the constraint checker into a Web application, remain open questions that are addressed in this paper.

We implemented our model-driven enforcement mechanism in a prototype, called MORRO. We integrated MORRO into a Web application developed by our industrial partner; following the guidelines of the XACML standard architecture [37], our implementation includes a policy enforcement point (PEP) and a policy decision point (PDP). Although the proposed architecture has been designed based on our partner specifications, it can be generalized and integrated into other Web applications. We evaluated MORRO in terms of applicability and scalability. The evaluation results show that MORRO can be adopted without considerably impacting the overall performance (in terms of response time) of a Web application and that MORRO scales linearly with respect to the various parameters (e.g., the number of users and roles) characterizing an AC configuration. Overall, the results confirm the feasibility of using a model-driven approach to efficiently enforce complex RBAC policies.

To summarize, the main contributions of this paper are: 1) a model-driven approach for enforcing access control policies defined on top of the GemRBAC+CTX model, including algorithms specifying when and how the policy constraints are enforced; 2) an extensive empirical evaluation of our approach when integrated in an industrial system, to assess its performance and scalability.

The paper is organized as follows. Section 2 presents background material. Section 3 illustrates our model-driven approach for enforcing GemRBAC+CTX policies. Section 4 describes the integration of the proposed approach into the architecture of an industrial Web application. Section 5 presents the empirical evaluation results. Section 6 reviews the state of the art. Section 7 concludes the paper.

2 BACKGROUND: THE GEMRBAC+CTX MODEL

The GemRBAC+CTX model [6, 8] is an extension of the RBAC96 model [43] that has been designed after surveying the various types of the authorization policies proposed in the literature. The rest of this section gives an overview of the main entities of GemRBAC+CTX that are used in the subsequent sections.

The GemRBAC+CTX model, defined as a UML class diagram, contains all the entities (`User`, `Role`, `Session`, `Permission`) of the original RBAC model. These entities are modeled as UML classes. A permission is represented as a set of operations that can be performed on a set of objects. The relations among these RBAC entities are modeled as UML associations. Each role is assigned to a set of permissions and to a set of users. A role can be inherited using a role hierarchy relation. The inheritance of role assignment relationships can be defined using a *role hierarchy policy*; a user (or a role) assigned to a role (respectively permission) must also be assigned to *all* its sub-roles (respectively sub-permissions) [43].

A session is a mapping of one user to a subset of the roles that have been assigned to her; this mapping *activates* the role(s) for a certain user. However, in some systems only a subset of the assigned roles can be activated (e.g., because of the user's location), which are called *enabled*. Once a role is enabled, a user can request its activation within a session. Both role enabling and activation are modeled as UML associations between the `Role` and `Session` classes. Similarly, a permission is enabled if the user is allowed to perform its associated operations.

In addition to assignment relations, authorization policies are defined to restrict a user access. For instance, role and permission enabling/disabling can be regulated through precedence and contextual policies. *Precedence policies* define a precedence relationship between the enabling of a role and the activation of another role; for example, role *student* is enabled only if a *supervisor* role has been already activated. *Contextual policies* restrict a user to activate a role or perform an operation assigned to a permission of her role depending on her location [10] (location-based policy) and the current time [28] (time-based policy). The context (i.e., spatial and temporal information) is modeled with class `RBACContext`, which contains a `TimeExpression` and/or a `Location`. The GemRBAC+CTX model supports policies with fine-grained temporal and spatial expressions, such as *"the first Monday of each month, from April 9, 2018 to January 11, 2019"* and *"the first floor of building A"*. A role is disabled if its corresponding contextual or a precedence policy is violated; a permission is disabled if its corresponding contextual policy is violated.

The GemRBAC+CTX model supports other types of authorization policies. *Prerequisite policies* define a precondition on user-to-role assignment, allowing a user to acquire a role only if she is already assigned to another one [2, 43]. Prerequisite policies can also be defined at the permission level, allowing a role to acquire a permission only if this role is already assigned to another permission. *Cardinality policies* define a bound on the cardinality of role activation and assignment relations [1]; e.g., a policy of type cardinality on role activation restricts a user from activating a number of roles that exceeds a given threshold. *Dependency policies*, complementary to the precedence ones, restrict the deactivation of a role if another role is still active [44]. *Separation of duty policies (SoD)* define a mutual exclusion relation among roles, permissions, or users; the entities involved in such relations are called *conflicting*; SoD policies can be either static or dynamic. *Static SoD* policies deal with user-to-role and role-to-permission assignments; for example, static SoD on conflicting roles specifies that the same user cannot be assigned to mutually exclusive roles. *Dynamic SoD* policies deal with user-role activation through a session; in this case, a user is allowed to acquire conflicting roles but she cannot activate them at the same time. Examples of this type of policy are dynamic SoD on conflicting roles (DCR) or users (DCU) and history-based (His) DSoD [6, 45]. *Binding of Duty (BoD) policies* are the dual of the SoD ones and define a correlation between a set of permissions, which are called *bounded*; they are usually used in the context of workflow systems, whose activities can be performed by different users with different roles. For example, in role-based BoD, the operations allowed by two or more permissions have to be performed by the same role [48]. To support history-based policies such as dynamic SoD or BoD, operations performed by a user on a given object in a certain context, are recorded and modeled as instances of class `History`. *Delegation policies* allow a user to delegate or transfer her role to another user [18, 54]. A delegation is partial if only a strict subset of the permissions associated to a role has been delegated; total otherwise. *Revocation policies* allow a user to revoke a delegated role [54].

An instance of the GemRBAC+CTX model corresponds to a snapshot of the system state from an RBAC point of view, at a given time point. For example, the object diagram in figure 1 depicts an instance of the GemRBAC+CTX model that represents the following RBAC entities: two `Users`, u_1 and u_2; two `Roles` r_1 and r_2; two `Permissions`, p_1 and p_2; two `Operations` op_1 and op_2; four `Objects`, o_1, o_2, o_3 and o_4. Permission p_1 is assigned to role r_1 and permission p_2 is assigned to role r_2 through role-permission assignment associations (RPA). Permission p_1 maps operations op_1 and op_2 to objects o_1 and o_2; similarly, permission p_2 maps operation

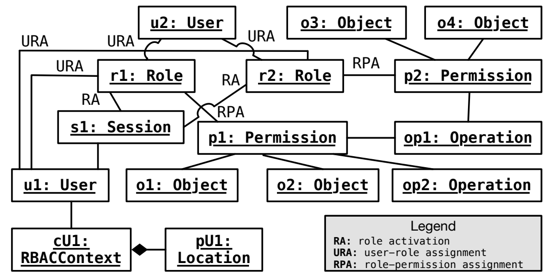

Figure 1: An instance of the GemRBAC+CTX model representing a system state.

op_1 to objects o_3 and o_4. Moreover, both roles r_1 and r_2 are assigned both to user u_1 and to user u_2 through user-role assignment associations (URA). At the time when the snapshot has been taken, only user u_1 is connected through her session s_1 and has activated her assigned roles (r_1 and r_2) as shown by the role-activation associations (RA). The location of user u_1 is modeled with object cU_1, an instance of the `RBACContext` class with a `Location` object pU_1.

The policies supported by the GemRBAC+CTX model have been formalized as OCL constraints [6, 8], to enable their operationalization. For instance, a *Dynamic Separation of duty policy (DSoD)* on conflicting roles, such as "a user can activate either role r_1 or r_2", is checked by verifying the following invariant (taken from [6]) of the class `Session`, defined as an OCL constraint: `Session`, defined as an OCL constraint:

```
1  context Session inv DSoD:
2  let r1:Role = Role.allInstances()
3             -> select(r:Role | r.idRole='r1'),
4      r2:Role = Role.allInstances()
5             -> select(r:Role | r.idRole='r2')
6  in if self.activeRoles -> includes(r2)
7      or self.activeRoles -> includes(r1) then
8          self.activeRoles -> includes(r2)
9          xor self.activeRoles -> includes(r1)
10     endif
```

The DSoD policy above can be checked on the model instance shown in figure 1 by evaluating the invariant `DSoD` on the `Session` object s_1. In this case, the condition at lines 6–7 is `true` because both roles r_1 and r_2 are active. Therefore, we follow the `then` branch and evaluate the boolean expression at lines 8–9. This expression states that the list of active roles associated with session `s1` should contain either r_1 or r_2, but not both. Since both roles are active, the expression evaluates to `false`, meaning that the policy is violated.

The OCL formalization of the RBAC policies has been used to define the semantics of GemRBAC-DSL [7], a high-level policy specification language built on top of GemRBAC+CTX. Each policy expressed in GemRBAC-DSL is mapped to an OCL constraint to operationalize its checking. For instance, the DSoD policy above can be expressed in GemRBAC-DSL as:

```
PL1-DSoD: conflicting-roles-activation r1, r2;
```

We remark that GemRBAC-DSL does not express basic AC policies, i.e., those encoded as a) role-to-user and role-to-permission assignments, and as b) role-to-session activation or enabling relations; GemRBAC-DSL assumes that such policies are already defined at the model level as UML associations.

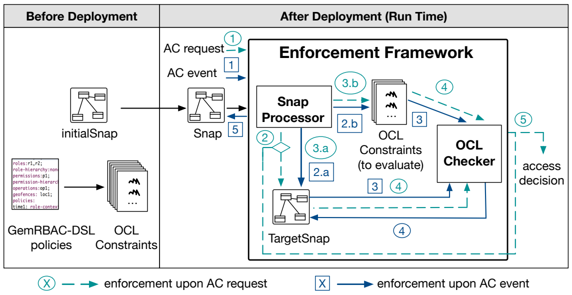

Figure 2: Overview of the proposed model-driven framework for policy enforcement.

3 MODEL-DRIVEN ENFORCEMENT OF COMPLEX POLICIES

The goal of this paper is to propose an automated mechanism to enforce access control policies defined on top of the GemRBAC+CTX model and expressed using the GemRBAC-DSL language. One basic idea to achieve this goal is to leverage the operationalization through OCL constraints of GemRBAC-DSL policies proposed in [7], to define a model-driven enforcement approach. At the base of this approach there is *the reduction of the problem of enforcing GemRBAC-DSL policies to the evaluation of the corresponding OCL constraints on an instance of GemRBAC+CTX* (which captures the system state from an AC point of view). Adopting a model-driven approach for enforcing GemRBAC-DSL policies has two main advantages: 1) the possibility of building upon the existing translation of GemRBAC-DSL policies into OCL constraints, which is already optimized for efficient checking; 2) the reliance on OCL, which is a standard and for which there exists mature constraint checking technology.

Nevertheless, putting such an approach in operation, requires to define algorithms to precisely decide when and how to enforce such constraints, as well as to outline an architecture that describes how to integrate the constraint checker into a Web application architecture: these are open questions that will be addressed by the coming sections.

Figure 2 illustrates how our approach can be realized in an enforcement framework. Before deploying the application, we assume that a security admin has defined the initial system state from the point of view of AC. This means that the main RBAC entities (i.e., users, roles, permissions, operations and objects) of the system have been defined, together with their assignment relations (e.g., assignment of permissions to the various roles). These entities, representing the static RBAC view of the system, are captured in an instance of the GemRBAC+CTX model called *initialSnap*. We also assume that the security admin has defined the AC policies in GemRBAC-DSL. These policies are then translated into a set of OCL constraints (with respect to the GemRBAC+CTX model) using the translation defined in [7].

After deployment, when the system is executing, the enforcement framework works as follows. Its inputs are:

- The set of OCL constraints corresponding to the GemRBAC-DSL policies defined for the system.
- A snapshot *Snap* of the system state from the point of view of AC, represented as a GemRBAC+CTX instance. This snapshot captures both the static RBAC view of the system and the dynamic RBAC view (e.g., active sessions, users' contexts); it is updated at run time as the AC configuration evolves. Notice that right after the start of the system execution, *Snap* corresponds to *initialSnap*.
- The actual AC request that has to be enforced. It contains (as instances of the corresponding classes of the GemRBAC+CTX model) the `Object` to access and the `Operation` to perform on it, such as "`GET /url/to/resource`".

Table 1: Policies checked for each type of AC request/event

R \ P	Prq	RH	AC	AS	Prec	Dep	S	DCR	DCU	Obj	Op	His	BoD	CT	CL	Deleg	Rev
RA			✓					✓	✓								
AR										✓	✓	✓	✓				
RD	✓	✓		✓			✓									✓	
RR	✓	✓		✓													✓
AO	✓	✓		✓			✓							✓	✓		
UA					✓									✓	✓		
ULC															✓		
UD						✓											

Legend. R: AC request/event. P: policy. *Type of AC request/event:* RA: role activation; AR: access to a resource; RD: role delegation, RR: role revocation; AO: administrative operation; UA: user authentication; ULC: user's location change; UD: user disconnection. *Type of policy:* Prq: prerequisite; RH: role hierarchy; AC: cardinality on role activation; AS: cardinality on assignment relations; Prec: precedence; Dep: dependency; S: static SoD; DCR: dynamic SoD on conflicting roles; DCU: dynamic SoD on conflicting users; Obj: object-based DSoD; Op: operational-based DSoD, His: history-based DSoD; BoD: binding of duty; CT: time-based context; CL: location-based context; Deleg: delegation; Rev: revocation.

The various steps of the enforcement process are shown in figure 2 and are marked with green dashed lines and circles. The main component of the enforcement framework is the *SnapProcessor*. Once the framework receives an AC request (step 1), the *SnapProcessor* first analyzes the request by checking whether the request is valid (step 2). For instance, a user cannot request to activate a role that is not assigned to her. In case the request is not valid, the access is denied and the access decision is returned (step 5).

Otherwise, if the request is valid, the *SnapProcessor* builds a new snapshot of the system state (step 3.a), starting from the current state captured by the *Snap*. This new snapshot, called *TargetSnap* and also represented as an instance of the GemRBAC+CTX model, captures the next system state (from the point of view of AC) *as if the AC request had been allowed.* After creating the *TargetSnap*, the *SnapProcessor* selects—based on the type and the parameters of the AC request—the OCL constraints to evaluate (step 3.b), corresponding to the policies to enforce; the policies selection follows the rules encoded in the top part of table 1. The selected OCL constraints are evaluated by an OCL checker (step 4). In this way, making an access decision for an AC request (step 5) is equivalent to verifying whether the *TargetSnap* satisfies the OCL constraints corresponding to the policies to enforce. If the constraints evaluate to *true*, it means that the AC request can be allowed, since it will not violate any policy. On the contrary, when the constraints evaluate to *false*, it means that allowing the request would violate one or more of the policies defined for the system.

Our approach adopts also the usage control [40] concept for AC, which aims to revise AC decisions (by re-enforcing AC policies) when a new update, from the point of view of AC, occurs at the system level. For instance, whenever a user changes her location, our enforcement mechanism re-evaluates the (OCL constraints corresponding to the) location-based policies for this user, to check

Algorithm 1 Enforcement upon receiving an AC request

Input: a = AC request,
$Snap$ = current system state,
P = list of system policies
Output: a tuple $\langle d, TargetSnap \rangle$, where:
d = access decision,
$TargetSnap$ = new system state

1: $PLC \leftarrow \emptyset$, $TargetSnap \leftarrow$ null, $d \leftarrow$ null
2: **if** `validateRequest`(a) **then**
3: $TargetSnap \leftarrow$ `buildTargetSnap`(a, $Snap$)
4: $PLC \leftarrow$ `selectPolicies`(P, a)
5: **else** $d \leftarrow$ false
6: **if** d is null **then** $d \leftarrow$ `check`($TargetSnap$, PLC)
7: **return** $\langle d, TargetSnap \rangle$

Algorithm 2 Enforcement after an AC event occurs

Input: e = AC event,
$Snap$ = current system state,
P = list of system policies
Output: $USnap$ = updated system state

1: $PLC \leftarrow \emptyset$, $USnap \leftarrow$ null
2: $USnap \leftarrow$ `updateState`(e, $Snap$)
3: $PLC \leftarrow$ `selectPolicies`(P, e)
4: **for each** policy p in PLC **do**
5: `check`($USnap$, p)
6: **if** p is not satisfied **then** `rectify`($USnap$, p, e)
7: **return** $USnap$

whether the new system state satisfies them. The various steps of the enforcement process upon an AC event are shown in figure 2 and are marked with blue solid lines and squared boxes. In our case, we assume that the enforcement framework will receive, from an external component, a notification when an AC event occurs (step $\boxed{1}$). Reacting to this notification, the *SnapProcessor* creates the *TargetSnap*, obtained by updating (according to the received event) the current system state captured by the *Snap* (step $\boxed{2.a}$). The *SnapProcessor* then selects—based on the type of the AC event—the OCL constraints to evaluate (step $\boxed{2.b}$), corresponding to the policies to enforce; the policies selection follows the rules encoded in the bottom part of table 1. Finally, it checks, by means of the OCL checker, whether the selected policies are still satisfied (step $\boxed{3}$). If a policy violation is detected, the *SnapProcessor* updates the *TargetSnap* by disabling/deactivating the corresponding role (step $\boxed{4}$); the updated *TargetSnap* then becomes the new *Snap* (step $\boxed{5}$).

The next subsections explain how our framework enforces AC policies when making an access decision for an AC request (section 3.1) and when handling notifications for AC events (section 3.2).

3.1 Making Access Decisions for AC Requests

The procedure for enforcing policies upon receiving an AC request is shown in algorithm 1. It takes as input an AC request a, a snapshot *Snap* corresponding to the system state (from the point of view of AC) at the time of the request, and the list P of policies defined for the system; it returns a tuple, containing the access decision d (a boolean value, with *true* corresponding to "allow" and *false* to "deny") and a snapshot *TargetSnap* (an instance of GemRBAC+CTX corresponding to the new system state as if the request had been authorized). Besides variables d and *TargetSnap*, the procedure uses an auxiliary variable *PLC*, representing the list of policies to check for a specific type of the AC request and initialized to an empty list. Both variables *TargetSnap* and d are initialized to *null*. Our approach considers AC requests of type: *role activation*, *access to a resource*, *role delegation*, *role revocation*, and *administrative operation* (i.e., assigning a role to a user or to a permission).

First, the *SnapProcessor* checks the validity of the request by calling operation `validateRequest` (line 2, corresponding to step 2 in figure 2). The validity is determined based on the type of the AC request as follows. In case of a *role activation*, the request is valid if the role to activate is already enabled for the user who made the request. In case of an *access to a resource*, the requested permission (e.g., p) should be assigned to an active role r in the current session of the user who made the request and should be enabled; if the user who made the request acquired role r through a delegation, this delegation should include permission p. In case of a *role delegation*, the role being delegated should be assigned to the user who made the request and not assigned to the user who will receive the delegation. In case of an *administrative operation*, the requested user (respectively, permission) should not belong to the list of users (respectively, permissions) assigned to the role indicated in the request.

If the AC request a is not valid, the *SnapProcessor* sets the access decision d to *false* (line 5). Otherwise, the *SnapProcessor* builds the *TargetSnap* by calling operation `buildTargetSnap` (line 3, corresponding to step 3.a in figure 2). This operation takes as input the AC request a and the snapshot *Snap*; its behavior depends on the type of the AC request:

- *Role activation.* We consider a request of the form "user u_1 requesting to activate role r_1". First, we remove role r_1 from the list of enabled roles. Then, we add it to the list of active roles for user u_1; if a precedence policy is specified for role r_1, the *SnapProcessor* enables the list of roles which should be enabled for other users, according to the precedence relation.
- *Access to a resource.* We consider a request of the form user "u_1 with role r_1 requesting to perform operation op_1 on object o_1". To build *TargetSnap*, we add a new instance of type `History` to the current *Snap*. This instance records that user u_1, while having role r_1, performed operation op_1 on object o_1 through permission p_1.
- *Role delegation.* We consider a request of the form "user u_1 requesting to delegate her role r_1 to user u_2". The *TargetSnap* is obtained by adding role r_1 to the list of delegated roles for user u_2 and creating a new instance of class `Delegation`.
- *Role revocation.* We consider a request of the form "user u_1 requesting to revoke delegation d_1"; we also assume that user u_2 acquired role r_2 through delegation d_1 (originated from u_1). We build the *TargetSnap* by removing role r_2 from the list of delegated roles assigned to user u_2, marking delegation d_1 as *revoked*, and recording the revoking user (u_1).
- *Administrative operation.* We consider a request of the form "admin requesting to assign role r_1 to user u_1" or "'admin requesting to assign role r_1 to permission p_1". The *SnapProcessor* builds the *TargetSnap* by adding the appropriate assignment relation, i.e a role-to-user assignment or a role-to-permission assignment relation.

After building the *TargetSnap*, the *SnapProcessor* extracts the list *PLC* of policies to check from the system policies list P by calling operation `selectPolicies` (line 4, corresponding to step 3.b in figure 2). This operation determines the list of policies to check based on the type (i.e., according to table 1) and the parameters of the request. For instance, in case of an AC request of type *role activation*, the list *PLC* will contain all the policies in P whose type is indicated in row *RA* (i.e., AC, DCR, DCU), and whose parameters match at least one of the request parameters (i.e., the user who made the request and the role to activate).

Then, if the access decision d has not been set yet (i.e., it is *null*), the algorithm invokes the OCL checker (operation `check` at line 6, corresponding to step 4 in figure 2). This operation evaluates, on the *TargetSnap*, the OCL constraints corresponding to the policies in *PLC*; the result of the evaluation will contain the access decision. The algorithm ends by returning the tuple with both the access decision d and the new system state *TargetSnap* (step 5 in figure 2). We recall that when the access is denied *TargetSnap* will be *null*.

3.2 Handling Notifications for AC Events

The procedure for updating the system state captured by the *Snap* and enforcing policies upon receiving a notification for an AC event is shown in algorithm 2. It takes as input an AC event e, a snapshot *Snap* corresponding to the system state (from the point of view of AC) at the time of the notification, and the list of policies P defined for the system; it returns a snapshot *USnap*, which is an instance of the GemRBAC+CTX model corresponding to the updated system state. Besides variable *USnap* (initialized to *null*), the procedure uses an auxiliary variable *PLC*, representing the list of policies to check for a specific type of event and initialized to an empty list. Our approach considers events of type *user authentication*, *user's location change*, and *user disconnection*.

Upon receiving an event notification, the *SnapProcessor* first updates the system state according to the received event by calling operation `updateState` (line 2, corresponding to step $\boxed{2.a}$ in figure 2). This operation takes as input the received AC event e and the current system state captured by the *Snap*; it returns the updated state in *USnap*. The behavior of operation `update` depends on the type of the event e:

- The *user authentication* event corresponds to the case of a user logging in the system. We assume that the enforcement framework receives the notification from an authentication server, which checks the user credentials and allows her login. In this case, we update the state by adding a new instance of class `Session` for the authenticated user, updating the user's location, and enabling, within the newly added session, all the roles assigned to the user.
- The *user's location change* event corresponds to the case of a connected user changing her location. We assume that a geo-localization server keeps track of the user position; this server sends a notification to the enforcement framework whenever a connected user changes her location. In this case, the state is updated by updating the user's location.
- The *user disconnection* event corresponds to the case when a user is experiencing network issues. We assume that the authentication server periodically checks for the online status of a user and sends a notification to the enforcement framework when it detects that the user is offline[1]. In this case, we update the state by removing the session of the disconnected user.

Afterwards, the *SnapProcessor* extracts the list *PLC* of policies to check from the system policies list P, by calling operation `selectPolicies` (line 3, corresponding to step $\boxed{2.b}$ in figure 2). This operation determines the list of policies to check based on the type (according to table 1) and the parameters of the received event notification. For instance, in case of a *user authentication* event, the list *PLC* will contain all the policies in P whose type is indicated in row *UA* (i.e., CT, CL), and whose parameters match at least one of the notification parameters. Then, for each policy p in *PLC*, the *SnapProcessor* invokes the OCL checker (operation `check` at line 5, corresponding to step $\boxed{3}$ in figure 2), to evaluate, on the *USnap*, the OCL constraint(s) corresponding to p. If the result of the evaluation is false, it means that the new system state (as determined in response to the event e) violates policy p. Applying the usage control concept, the *SnapProcessor* amends the *USnap*, by calling operation `rectify` (line 6, corresponding to step $\boxed{4}$ in figure 2); the behavior of this operation depends on the type of the event e:

- *User authentication.* We consider a notification of the form $\{u_1, s_1, loc\}$, where u_1 is the user being authenticated, s_1 is the token representing the user's session, *loc* is the current position of the user. For each role r enabled for user u_1 in session s_1, the *SnapProcessor* amends *USnap* by disabling role r in session s_1.
- *User's location change.* We consider a notification of the form $\{u_1, loc_1, loc_2\}$, where u_1 denotes the user and loc_1 and loc_2 correspond, respectively, to the previous and the new position of user u_1. For each role r assigned to user u_1, the *SnapProcessor* amends *USnap* according to the state of role r: if it is enabled (respectively, active), the *SnapProcessor* will disable (respectively, deactivate) it from all the sessions of user u_1.
- *User disconnection.* We consider a notification of the form $\{u_1, s_1\}$, where u_1 is the user being authenticated and s_1 is the token representing the user's session. For each role r assigned to user u_1, the *SnapProcessor* amends *USnap* by disabling and deactivating role r_1 from all sessions in the system.

The algorithm ends by returning *USnap* (step $\boxed{5}$ in figure 2).

4 RUN-TIME ARCHITECTURE AND IMPLEMENTATION

We have integrated the enforcement framework presented in section 3 into the architecture of a Web application developed by our partner. This architecture includes: a) a Web application and a set of micro-services, which expose resources accessible through the web interface; b) a geo-localization server, which records the users' position; and c) an authentication server, for authenticating users based on their credentials.

To integrate our enforcement framework, we have added two new components to this architecture: an *authorization server* and a *proxy*. These components follows the guidelines of the XACML standard architecture [37], which prescribes to use two components: a *policy enforcement point* (PEP) and a *policy decision point* (PDP). In this standard architecture, a user AC request is intercepted by the PEP, which will transform it into an XACML request and forward it to the PDP; the latter evaluates the request based on the authorization policies. In our case, the PEP is the *proxy*, while the PDP is the *authorization server*. The *authorization server* integrates the *SnapProcessor* and the OCL Checker shown in figure 2; it receives AC requests and notification of AC events and enforces the policies as described in section 3, making sure that only authorized users can access the resources exposed by the set of micro-services. The *proxy* is a gateway that intercepts user AC requests: it first forwards them to the *authorization server*, which makes an access decision that is returned to the *proxy*; if the access decision is "allow" the *proxy* forwards the original user request to the corresponding micro-service. In addition, we have included a storage component for the access

[1]The case of a user sending a log out request to the authentication server is treated by forwarding the request to the enforcement mechanism, which is then processed as explained in section 3.1.

control data, which contains the snapshot *Snap* (given in input to and updated by the *authorization server*) and the GemRBAC+CTX policies to enforce.

Although the resulting enforcement architecture has been designed based on the architectural specifications provided by our industrial partner, it can be generalized and integrated into other Web applications. More specifically, the *proxy* can be integrated seamlessly within existing load balancers, which are very common in Web applications [16]; the *authorization server* and the storage are additional components that can be deployed on any Web application server.

Implementation. The core of our framework is a component, called MORRO (MOdel-driven fRamework for Run-time enforcement of RBAC pOlicies), and includes the *authorization server* and the *proxy*. MORRO has been implemented in Java with a micro-service based architecture using the SpringBoot [49] framework and the ZuuL proxy v.1.2.7 [36]. The implementation of the *authorization server* uses the Eclipse Modeling Framework (EMF) and Eclipse OCL v.5.2 [19]. The *Snap* is expressed as an Ecore [20] model.

5 EVALUATION

In this section we report on the evaluation of MORRO when deployed in a real Web application, with a complex AC configuration. We assess the efficiency and applicability of MORRO by answering the following research questions:

RQ1: How long does the authorization server in MORRO take to process AC requests/events, when deployed on a real industrial system, under various AC configurations?

RQ2: How do the access decision time and the AC event processing time of the authorization server in MORRO scale with respect to changes of the various parameters potentially affecting the performance of an AC configuration?

RQ3: What is the communication overhead between the authorization server *and the* proxy *in case of an AC request?*

5.1 Evaluation Settings

We considered a real AC configuration used by our industrial partner, consisting of 1648 users, 396 roles, 53 permissions, 300 objects and 4 operations (*create*, *read*, *update*, and *delete*). We defined a set of GemRBAC-DSL policies in collaboration with the security engineers of our partner. We then determined the types of policies used in the specification and, for each type, we considered a representative example to answer the research questions mentioned above. To enable MORRO to enforce them, we used the mapping of these policies to OCL constraints we previously proposed in [6, 8].

We deployed MORRO onto a micro-service-based architecture provided by our industrial partner. This architecture was running on a development machine equipped with a dual CPU Intel Xeon E5-2640 v2 2 GHz and 24 GB of memory; we used this machine to run all the experiments. All time measurement were performed by invoking the `System.nanoTime()` method of the standard Java library, version 1.8.

Due to space reasons, in the following we present only a summary of the evaluation results. We refer to the first author's PhD thesis [5, chapter III] for a complete description of the policies used in the evaluation and for the detailed experimental results.

5.2 Performance on an Industrial System

Methodology. To address *RQ1*, we measured the time taken by the authorization server in MORRO to process different types of AC requests and events. More specifically, in case of an AC request we measured the *access decision time*, i.e., the time difference from the time the authorization server receives the request to the time it yields an access decision. In case of an AC event, we measured the *execution time* needed to update the current system state (*Snap*), i.e., the time difference from the time the authorization server receives a notification for an AC event until the time it updates the current system state. Based on the AC configuration of the test application defined by our industrial partner, we generated two types of AC requests and two types of AC events. For each type of request (respectively, event) the access decision time (respectively, execution time) was assessed both on a *basic configuration*—i.e., an AC configuration that is only determined by role assignment and activation relations—and on configurations that add to the basic configuration other policies to be checked. The types of requests and events generated are:

- *Access to a resource.* We consider two scenarios: 1) when the role of the user making the request has been assigned and 2) when a subset of the permissions assigned to this role has been delegated. As for the configurations, in addition to the basic one, we use one with a history-based DSoD policy and another with a BoD policy.
- *Role activation.* The additional configurations use the cardinality on role activation and the DCR policies.
- *User authentication.* We consider two scenarios, in which we distinguish whether the user's position is known or not. The additional configurations use precedence, location-based, and time-based policies.
- *User's location change.* We consider one additional configuration with a location-based policy.

For all configurations we considered the worst-case scenario, with the maximum allowed value for each system parameter (e.g., maximum number of roles assigned to a user).

Since MORRO runs on a Java-based environment, the measurements of the running time are affected by various factors [23]. Furthermore, the network-based communication between the *proxy* and the *authorization server* introduces some noise. For these reasons, when measuring the access decision time for AC requests, we sent ten AC requests, discarded the first one (since it is affected by the loading time of the run-time libraries), and measured the average value over checking the nine subsequent requests. As for measuring the execution time for processing AC events, we were able to achieve stable results by sending only five notifications. As above, since the first value is affected by the loading of the run-time libraries, we discarded it and measured the average value over processing the four subsequent notifications. In both cases, to keep the same instance over the different runs, we designed the (initial) AC configuration of the system such that the OCL checker yields false (denying the access request).

Results. We answer *RQ1* by summarizing the main results. The *access decision time* within the authorization server is less than 64 ms; the highest value is obtained while evaluating an AC request of type *access to a resource* for a configuration with a history-based DSoD policy in a role delegation scenario. This value has to be analyzed in the context of Web applications which are accessed by users from a browser. In modern Web applications, the complexity of each single Web page requires a relatively high network time (i.e., the time needed by a browser to fetch all resources to be displayed on a page); for example, a web page from Wikipedia requires on average 1880 ms of networking time [17]. Under this scenario, a maximum overhead of 64 ms due to the AC enforcement framework would correspond to less than 4% increase over the total networking time, which is quite affordable in practice.

The *execution time for processing a notification of an AC event* is less than 512 ms; the highest value is obtained while evaluating an AC request of type *user authentication* for a configuration with a precedence policy considering the case when the user position is known. This value has also to be interpreted in the context of Web applications. In such a context, an AC event is triggered by a user action and its processing should be completed before the next user request, so that the latter can be evaluated on the updated system data (as modified by the AC event). Hence, the execution time for processing the notification of an AC event should be less than the time between the completion of a user request and the start of a new one (i.e., the *think time*). TCP-W [51], a common benchmark for Web applications, considers an average think time of 7 s; the maximum value for the execution time in our system (512 ms) is well below this threshold.

5.3 Scalability

Methodology. To answer *RQ2*, we evaluated the scalability of the authorization server. Scalability is concerned with analyzing the change in access decision time (and AC event execution time) as parameters increase in value, with respect to different scenarios and configurations. Our goal is to use such information to draw conclusions on how our solution is likely to tackle even more complex AC situations.

We considered the same AC requests and AC events used to answer *RQ1*, and the corresponding scenarios and configurations. To assess the effect of a parameter, we varied it while keeping all the other relevant system parameters constant. The parameter varied either between a range of values with a certain step increment, or through a set of predefined values; the latter case occurred for parameters that affect the evaluation of spatial and temporal policies. The snapshots of the system state corresponding to all these configurations were generated using an internally developed, parametrized generator. In each experimental run, we sent an AC request or an AC event notification. In the case of AC requests, we measured the access decision time; in the case of AC events, we measured the execution time for processing the event. In both cases, we measured these values following the same procedure described in the answer to *RQ1*.

Results. For space reasons, here we only present the results corresponding to one type of AC request and to one type of AC event, on a specific configuration.

Table 2 reports the evaluation results for an AC request of type *access to a resource* (*AR*) and for an AC event of type *user authentication* (*UA*). We consider 1) an AC request of the form "u_1 with role r_1 in session s_1 requesting to perform operation op_1 on object o_1", on a configuration with a history-based DSoD policy (*His*), for both

Table 2: Scalability of MORRO under various system configurations.

AC Conf	Param	Values Range		Step Inc	Time (ms)		
		lower	upper		min	max	Data trend
AR-His -RLA	b	1K	10K	1K	73	99	alc M: 84.30, SD: 8.53
	d	1K	10K	1K	37	49	linear
	e	1K	4K	1K	56	74	alc M: 65.64, SD: 7.33
	f	1K	10K	1K	39	85	linear
AR-His -RLD	b	1K	10K	1K	96	108	alc M: 100.75, SD: 4.05
	d	1K	10K	1K	58	115	linear
	e	1K	4K	1K	123	148	alc M:138.02, SD:10.63
	f	1K	10K	1K	61	123	linear
UA-Prec -UL	g	1K	10K	1K	199	339	linear
	c	1K	5K	1K	845	1444	linear
	a	10K	25K	5K	891	1607	linear
UA-Prec -KL	g	1K	10K	1K	233	452	linear
	c	1K	5K	1K	1497	1628	linear
	a	10K	25K	5K	901	2017	linear

Parameters labels. a: #sessions in the system; b: #active roles in s_1 (current session of the user who made the request); c: #active roles in all sessions. d: #logs associated with conflicting roles; e: #objects within the set of logs; f: #operations in the system; g: #roles assigned to user u1.
Configuration scenarios labels. RLA: role assignment scenario; RLD: role delegation scenario; UL: the user location is unknown; KL: the user location is known.

scenarios of role assignment (*RLA*) and role delegation (*RLD*); and 2) an AC event of type *user authentication* (*UA*), with a notification of the form $\{u_1, s_1, loc\}$—where u_1 is the user being authenticated, s_1 is the token representing the user's session, *loc* is the current position of the user—on a configuration with a precedence policy (*Prec*), for two scenarios, in which we distinguish whether the user's position is known (*KL*) or not (*UL*). Column *AC Conf* indicates for each request/event the considered configuration and scenario. For instance, the configuration on the first row (*AR-His-RLA*) corresponds to the case of an AC request of type *access to a resource* on a configuration with a "*His*" policy while considering a role assignment scenario. Column *Param* indicates (with a label, see legend at the bottom of the table) the parameter being assessed during the run; column *Values Range* denotes the lower and upper bounds of the range of values through which the parameter is varied; the step increment is shown in column *Step Inc*; column *Time* indicates the access decision time; columns *min* and *max* denote the lowest and the highest time values observed across runs; column *Data trend* indicates the trend observed for the data points: in case of an "almost constant" (referred to as *alc*) trend, we include the average (column *M*) and the standard deviation (*SD*).

The answer to *RQ2* is that the access decision time and the execution time for processing a notification of an AC event are either *linear* with respect to the parameters of the various configurations or *almost constant* (i.e., there is no much variation across runs, with low SD). These trends can be explained in terms of the operations called in the OCL constraints (see [6, 8]) evaluated for each policy. For example, for the first configuration ("AC-His-RLA") with parameter "b" (the number of active roles in session s_1), the access decision time is *almost constant*, i.e., it does not depend on parameter "b": this is due to the definition (in [6]) of the OCL constraint corresponding to the history-based DSoD policy), in which checking whether both conflicting roles are active in session s_1 is performed in a constant time. Similar trends can be observed for the other configurations, for each type of AC request/event. Overall, these results imply that our solution is applicable for even more challenging AC situations with larger numbers of sessions, roles, and permissions.

The highest value for the access decision time we measured was 148 ms, in the case of a request of type *access to a resource*, in a configuration with a *His* policy, with a role delegation scenario, with 10K objects within the set of logs. Along the lines of the discussion for *RQ1*, such value would represent an 8% overhead with respect to the average networking time (see [17]) for fetching a complex Web page. Such an overhead is still acceptable when enforcing AC policies in large systems.

The highest value for the execution time we measured was 2017 ms, in the case of an AC event of type *user authentication*, in a configuration with a precedence policy, with 25K active sessions, with a known user location. As also discussed for *RQ1*, this value would still be below the think time threshold (7 s) provided by the TCP-W [51] benchmark.

5.4 Overhead of the Communication between the Authorization Service and the Proxy

Methodology. To address *RQ3*, we measured the communication overhead between the authorization service and the proxy, i.e., the time taken to dispatch an authorization request from the proxy to the authorization server, plus the time to propagate the access decision from the authorization server back to the proxy. We computed this overhead as the difference between the access decision time measured within the proxy and the access decision time measured within the authorization server; the access decision time within the proxy is the difference between the time instant at which the proxy receives an AC request from the user and the time instant at which the proxy receives an access decision for that request from the authorization server.

We measured this difference for all the requests/events, scenarios and configurations mentioned in section 5.2.

Results. The answer to *RQ3* is that the overall overhead of the communication between the authorization service and the proxy is less than 60 ms. When considering both the access decision time within the authorization server and the communication overhead between the authorization server and the proxy, the most taxing AC request is one of type *access to a resource*, for a configuration with a "*His*" policy, with an access decision time within the proxy of 107 ms. Along the lines of the discussion for *RQ1*, such value would represent less than 6% of the average networking time (see [17]) for fetching a complex Web page. Furthermore, this value is far below the threshold (200 ms) indicated in the requirements specifications of the Web application developed by our industrial partner.

6 RELATED WORK

This work leverages our previous work on modeling and specifying complex RBAC policies using the GemRBAC+CTX model [6, 8] and the GemRBAC-DSL language [7], including the operationalization of RBAC policies as OCL constraints. This paper complements and advances our previous work by providing an approach for the *enforcement* of RBAC policies, which includes the description of algorithms specifying when and how the policy constraints are enforced, the definition and engineering of an architecture to integrate the constraint checker into a Web application, and an empirical evaluation of the performance and scalability of the approach when integrated into an industrial system.

Table 3: Support of AC requests/events in existing policy enforcement approaches (abbreviations are defined in Table 1)

	AC Request					AC Event		
	RA	AR	RD	RR	AO	UA	ULC	UD
Sohr et al. [47]	+	+	-	-	+	+	-	-
Hummer et al. [27]	-	+	-	-	-	-	-	-
Martinez et al. [33]	+	+	-	-	-	-	-	-
Zhang et al. [54]	-	-	+	+	-	-	-	-
Margheri et al. [31]	-	+	-	-	-	-	-	-
Mourad et al. [34]	-	+	-	-	-	-	-	-
Kallel et al. [29]	+	+	-	-	+	-	+	-
Mariscal et al. [41]	-	+	-	-	-	-	-	-
Mustafa et al. [35]	+	+	-	-	+	-	+	-
Kirkpatrick et al. [30]	-	+	-	-	-	+	-	-
Bhatti et al. [12]	+	+	-	-	+	-	+	-
Ben David et al. [4]	+	+	-	-	-	+	-	+
MORRO (this work)	+	+	+	+	+	+	+	+

A work very close to our contribution has been proposed by Sohr et al. [47], which implements the PDP as a model-driven authorization engine, in the context of Web services. RBAC policies are expressed as OCL contraints using the USE tool, a validation tool for UML models and OCL constraints. Similar to MORRO, to make an access decision, the authorization engine checks whether the system state, represented as an UML object diagram, satisfies the RBAC policies expressed as OCL constraints. The main difference is that this work is based on the standard RBAC96 model, which supports a limited subset of the policies that can be specified in GemRBAC+CTX (and enforced by our approach). More precisely, the proposed enforcement mechanism supports only cardinality, prerequisite, and history-based SoD. Moreover, contextual policies assume that the context is represented symbolically: i.e., fine-grained spatial (e.g., with relative locations) and temporal (e.g., with intervals) policies are not supported. Because of these intrinsic limitations in the underlying model, the enforcement mechanism can deal with only a subset of the AC requests (*access to a resource*, *role activation*, and *administrative operation*) and events (*user authentication*) supported by MORRO. Furthermore, the approach presented in reference [47] does not support usage control.

Other model-driven approaches for policy enforcement have been presented in [27, 33]. In the approach by Hummer et al. [27], RBAC policies are written in a domain-specific language based on UML activity diagrams and mapped to Business Process Execution Language for Web services (WS-BPEL) specifications to be enforced at run time; however, this work supports only separation of duty and binding of duty policies. The work by Martinez et al. [33] deals with the generation of a PDP infrastructure from a specification written in a policy language, using ATL model transformations. A limitation shared by these model-driven approaches is that RBAC policies are only enforced as a response to a user AC request of type *role activation* or *access to a resource*. Similarly, the work by Zhang et al. [54] propose an enforcement framework that supports only delegation and revocation policies.

Recent work by Margheri et al. [31] proposes a framework for the specification, analysis, and enforcement of ABAC [25] (attribute-based access control) policies; RBAC can be seen as a specific case of ABAC, where role is one of the attributes. In this work, both AC requests and policies are expressed in a high-level language called FACPL; they are then translated into constraints to be solved using an SMT solver. Although this approach is at a high level conceptually similar to ours (both approaches formalize the semantics of AC policies as constraints, either in OCL or in SMT-LIB), the underlying AC models are different. As a consequence, the types of requests and events upon which the policies are enforced are also different: the FACPL-based framework only supports AC requests of type *access to a resource* and does not support the concept of usage control. Furthermore, its empirical evaluation considered only one small case study and randomly generated policies, assessing scalability only in terms of the number of attributes.

Other proposals deal with the generation of aspects from policy specifications; the generated aspects are inserted into the application to be executed at run time. Mourad et al. [34] propose the use of BPEL aspects to enforce AC policies in the context of web service composition. Kallel et al. [29] generate enforcement aspects in AspectJ from an RBAC specification written in TemporalZ. Mariscal et al. [41] introduce a new UML artifact, called *role-slice*, which is used to generated aspects. Mustafa et al. [35] propose an authorization engine in which policies written in a Z specification are translated into a Java Modeling Language (JML) specification to be checked by a JML runtime assertion checker.

A limitation shared by all approaches mentioned above is that they do not adopt the usage control concept, meaning that the proposed enforcement mechanisms cannot react to changes in the RBAC configuration.

Other proposals deal with context-based usage control in RBAC. Kirkpatrick et al. [30] propose a proximity-based enforcement mechanism for the GEO-RBAC [10] model using the XACML architecture. However, this work does not consider role activation as a separate request; when submitting a request to access a resource, the user has to specify the role to activate. Although the proposed mechanism incorporates usage control, only policies supported by the GEO-RBAC model, i.e., location-based and dynamic SoD on conflicting roles, are enforced. An authorization framework for enforcing time-based policies, based on the X-GTRBAC language and its model GTRBAC [28] has been proposed by Bhatti et al. [12]. Policies written in the X-GTRBAC language are enforced using a Java-based GUI application. Ben David et al. [4] propose a run-time enforcement mechanism composed of a monitor and a change analyzer. Both the running system and the RBAC policies are expressed using the *models@runtime* paradigm [13] as a running architecture model. By observing the system behavior, the monitor sends a notification to the change analyzer whenever a change is detected. Upon this notification, the change analyzer builds a target architecture model that will be used to evaluate the RBAC policies. This work is similar to our enforcement approach as they build a target model to enforce the RBAC policies. However, this approach was not implemented and only assignment and activation relations are supported.

Table 3 summarizes to which extent the policy enforcement approaches discussed above support the various AC requests/events presented in section 3. As one can see, the MORRO framework proposed in this paper is the only one that supports *all of them*.

In addition, none of the approaches discussed above provides a full support for the comprehensive set of authorization policies captured by GemRBAC+CTX. Although some approches [12, 27, 29, 30, 33–35, 41, 47, 54] provide a prototype implementation of their enforcement mechanisms, none of these implementations are available for a performance comparison; the only exception is the FACPL framework [31] that, however, supports a different AC model. Furthermore, only few of the aforementioned approaches [27, 31, 33, 47] provide an empirical evaluation assessing the access decision time; however, we could not compare these approaches with ours, since the underlying RBAC models and the application contexts are different.

While in this paper we have addressed the problem of enforcing AC policies, there is a series of work, orthogonal to ours, that focuses on testing and static verification of AC policies (and, in some cases, their implementation), using various techniques such as mutation testing [11, 32], model-based testing [53], model checking [21, 55], SAT solving [26], theorem proving [14], and static analysis [50].

7 CONCLUSION AND FUTURE WORK

In this paper we presented a model-driven enforcement framework for policies defined on top of a comprehensive role-based access control model (GemRBAC+CTX), which leverages the operationalization of the access control policies as OCL constraints. We reduce the problem of making an access decision to checking whether a system state (from an RBAC point of view) expressed as an instance of the GemRBAC+CTX model satisfies the OCL constraints corresponding to the RBAC policies to be enforced. Policies are enforced both when an AC request is made and when an AC event is triggered; we provide the checking algorithms for both cases. We implemented the core of our enforcement framework in a tool called MORRO and provided an integration strategy for a typical industrial Web application, following the guidelines of the XACML standard architecture. The evaluation results show that MORRO can be adopted without considerably impacting the response time of a Web application and that MORRO scales linearly with respect to the various parameters characterizing an AC configuration. Overall, the results confirm the feasibility of using a model-driven approach to efficiently enforce complex RBAC policies.

Although we considered the GemRBAC+CTX model in the application of our approach, the latter is generic and does not depend on GemRBAC+CTX: it can be applied to any other AC model that can be expressed in UML and whose policies can be expressed in OCL. As part of future work, we plan to assess the end-to-end performance of a system integrating MORRO under different evaluation settings, such as a production configuration deployed on an elastic cloud infrastructure. We also plan to optimize MORRO in terms of time efficiency by adopting cache-based enforcement [24, 52], and in terms of space efficiency, by adopting the Kevoree Modeling Framework (KMF) [22], which is optimized for manipulating models at run time on large distributed systems.

REFERENCES

[1] Gail-Joon Ahn. 2003. Specification and Classification of Role-based Authorization Policies. In *Proc. of WETICE 2003*. IEEE, Los Alamitos, CA, USA, 202–207.

[2] Gail-Joon Ahn and M.E. Shin. 2001. Role-based authorization constraints specification using Object Constraint Language. In *Proc. of WETICE 2001*. IEEE, Los Alamitos, CA, USA, 157–162.

[3] Subhendu Aich, Shamik Sural, and A.K. Majumdar. 2007. STARBAC: Spatiotemporal Role Based Access Control. In *Proc. of the OTM Conferences 2007 (LNCS)*, Vol. 4804. Springer-Verlag, Heidelberg, Germany, 1567–1582.

[4] Olivier-Nathanaël Ben David and Benoit Baudry. 2012. Toward a Model-driven Access-control Enforcement Mechanism for Pervasive Systems. In *Proc. of MDSEC 2012*. ACM, New York, NY, USA, 6:1–6:6.

[5] Ameni Ben Fadhel. 2017. *Comprehensive Specification and Efficient Enforcement of Role-based Access Control Policies using a Model-driven Approach*. Ph.D. Dissertation. University of Luxembourg.

[6] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. 2015. A Comprehensive Modeling Framework for Role-based Access Control Policies. *J. Syst. Softw.* 107 (September 2015), 110–126.

[7] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. 2016. GemRBAC-DSL: A High-level Specification Language for Role-based Access Control Policies. In *Proc. of SACMAT 2016*. ACM, New York, NY, USA, 179–190.

[8] Ameni Ben Fadhel, Domenico Bianculli, Lionel Briand, and Benjamin Hourte. 2016. A Model-driven Approach to Representing and Checking RBAC Contextual Policies. In *Proc. of CODASPY 2016*. ACM, New York, NY, USA, 243–253.

[9] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. 2001. TRBAC: A Temporal Role-based Access Control Model. *ACM Trans. Inf. Syst. Secur.* 4, 3 (Aug. 2001), 191–233.

[10] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. 2005. GEO-RBAC: A Spatially Aware RBAC. In *Proc. of SACMAT 2005*. ACM, New York, NY, USA, 29–37.

[11] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. 2016. Testing Access Control Policies Against Intended Access Rights. In *Proc. of SAC 2016*. ACM, New York, NY, USA, 1641–1647.

[12] Rafae Bhatti, Arif Ghafoor, Elisa Bertino, and James B. D. Joshi. 2005. X-GTRBAC: An XML-based Policy Specification Framework and Architecture for Enterprise-wide Access Control. *ACM Trans. Inf. Syst. Secur.* 8, 2 (May 2005), 187–227.

[13] Gordon Blair, Nelly Bencomo, and Robert B France. 2009. Models@run.time. *Computer* 42, 10 (2009), 22–27.

[14] Ivan Bocić and Tevfik Bultan. 2016. Finding Access Control Bugs in Web Applications with CanCheck. In *Proc. of ASE 2016*. IEEE, Los Alamitos, CA, USA, 155–166.

[15] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers, California, USA.

[16] V. Cardellini, M. Colajanni, and P. S. Yu. 1999. Dynamic load balancing on Web-server systems. *IEEE Internet Computing* 3, 3 (May 1999), 28–39.

[17] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. 2011. BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications. In *Proc. of WebApps 2011*. USENIX Association, Berkeley, CA, USA, 12.

[18] Jason Crampton and Hemanth Khambhammettu. 2008. Delegation in Role-based Access Control. *Int. J. Inf. Secur.* 7, 2 (2008), 123–136.

[19] Eclipse. 2011. Eclipse OCL Tools. http://www.eclipse.org/modeling/mdt/?project=ocl.

[20] Eclipse. 2013. Ecore. https://www.eclipse.org/ecoretools/.

[21] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. 2005. Verification and Change-impact Analysis of Access-control Policies. In *Proc. of ICSE 2005*. ACM, New York, NY, USA, 196–205.

[22] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. An Eclipse Modelling Framework Alternative to Meet the Models@ Runtime Requirements. In *Proc. of MODELS 2012 (LNCS)*, Vol. 7590. Springer-Verlag, Heidelberg, Germany, 87–101.

[23] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proc. of OOPSLA 2007*. ACM, New York, NY, USA, 57–76.

[24] Jeffrey Hieb, Jacob Schreiver, and James Graham. 2012. Using Bloom Filters to Ensure Access Control and Authentication Requirements for SCADA Field Devices. In *Proc. of ICCIP 2012 (LNCS)*, Vol. 390. Springer, Heidelberg, Germany, 85–97.

[25] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. 2015. Attribute-Based Access Control. *Computer* 48, 2 (Feb 2015), 85–88. https://doi.org/10.1109/MC.2015.33

[26] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *Int. J. Softw. Tools Technol. Transf.* 10, 6 (01 Dec 2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9

[27] Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. 2013. Enforcement of Entailment Constraints in Distributed Device-based Business Processes. *Inf. Softw. Technol.* 55, 11 (2013), 1884–1903.

[28] James B D Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. 2005. A Generalized Temporal Role-based Access Control Model. *IEEE Trans. Knowl. Data Eng.* 17, 1 (January 2005), 4–23.

[29] Slim Kallel, Anis Charfi, Mira Mezini, Mohamed Jmaiel, and Karl Klose. 2009. From Formal Access Control Policies to Runtime Enforcement Aspects. In *Proc. of ESSoS 2009 (LNCS)*, Vol. 5429. Springer, Heidelberg, Germany, 16–31.

[30] Michael S. Kirkpatrick and Elisa Bertino. 2010. Enforcing Spatial Constraints for Mobile RBAC Systems. In *Proc. of SACMAT 2010*. ACM, New York, NY, USA, 99–108.

[31] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. in press. A Rigorous Framework for Specification, Analysis and Enforcement of Access Control Policies. *IEEE Trans. Softw. Eng.* (in press), 40.

[32] Evan Martin and Tao Xie. 2007. A Fault Model and Mutation Testing of Access Control Policies. In *Proc. of WWW 2007*. ACM, New York, NY, USA, 667–676.

[33] Salvador Martínez, Jokin García, and Jordi Cabot. 2016. Runtime Support for Rule-based Access-control Evaluation Through Model-transformation. In *Proc. of SLE 2016*. ACM, New York, NY, USA, 57–69.

[34] Azzam Mourad, Sara Ayoubi, Hamdi Yahyaoui, and Hadi Otrok. 2010. New approach for the dynamic enforcement of Web services security. In *Proc. PST 2010*. IEEE, Los Alamitos, CA, USA, 189–196.

[35] Tanveer Mustafa, Michael Drouineaud, and Karsten Sohr. 2010. Towards Formal Specification and Verification of a Role-based Authorization Engine Using JML. In *Proc. of SESS 2010*. ACM, New York, NY, USA, 50–57.

[36] Netflix. 2014. Zuul. https://github.com/Netflix/zuul/.

[37] OASIS. 2005. eXtensible Access Control Markup Language (XACML) Version 2.0.

[38] Alan C. O'Connor and Ross J. Loomis. 2010. *Economic Analysis of Role-Based Access Control*. Technical Report. RTI International for National Institute of Standards and Technology.

[39] OMG. 2012. Object Constraint Language. http://www.omg.org/spec/OCL/.

[40] Jaehong Park and Ravi Sandhu. 2004. The UCONABC Usage Control Model. *ACM Trans. Inf. Syst. Secur.* 7, 1 (Feb. 2004), 128–174.

[41] J.A. Pavlich-Mariscal, T. Doan, L. Michel, S.A. Demurjian, and T.C. Ting. 2005. Role Slices: A Notation for RBAC Permission Assignment and Enforcement. In *Proc. of DBSec 2005*. LNCS, Vol. 3654. Springer, Heidelberg, Germany, 40–53.

[42] Indrakshi Ray and Manachai Toahchoodee. 2007. A Spatio-temporal Role-Based Access Control Model. In *Proc. of DBSec 2007 (LNCS)*, Vol. 4602. Springer, Heidelberg, Germany, 211–226.

[43] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-based Access Control Models. *Computer* 29, 2 (1996), 38–47.

[44] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. 2005. A Role-based Access Control Policy Verification Framework for Real-time Systems. In *Proc. of WORDS 2005*. IEEE, Los Alamitos, CA, USA, 13–20.

[45] Richard T. Simon and Mary Ellen Zurko. 1997. Separation of Duty in Role-based Environments. In *Proc. of CSFW 1997*. IEEE, Los Alamitos, CA, USA, 183–194.

[46] Karsten Sohr, Mirco Kuhlmann, Martin Gogolla, Hongxin Hu, and Gail-Joon Ahn. 2012. Comprehensive two-level analysis of role-based delegation and revocation policies with UML and OCL. *Inf. Softw. Technol.* 54, 12 (2012), 1396–1417.

[47] Karsten Sohr, Tanveer Mustafa, Xinyu Bao, and Gail-Joon Ahn. 2008. Enforcing role-based access control policies in web services with UML and OCL. In *Proc. of ACSAC 2008*. IEEE, Los Alamitos, CA, USA, 257–266.

[48] Mark Strembeck and Jan Mendling. 2011. Modeling Process-related RBAC Models with Extended UML Activity Models. *Inf. Softw. Technol.* 53, 5 (May 2011), 456–483.

[49] Spring Tool Suite. 2014. Spring Boot. https://projects.spring.io/spring-boot/.

[50] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proc. of SEC 2011*. USENIX Association, Berkeley, CA, USA, 11–11.

[51] Transaction Processing Performance Council. 2001. TPC Benchmark W (Web Commerce) Specification v.1.7. www.tpc.org/tpcw/spec/tpcw_V17.pdf.

[52] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. 2011. Authorization Recycling in Hierarchical RBAC Systems. *ACM Trans. Inf. Syst. Secur.* 14, 1, Article 3 (June 2011), 29 pages. https://doi.org/10.1145/1952982.1952985

[53] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. 2012. A Model-based Approach to Automated Testing of Access Control Policies. In *Proc. of SACMAT 2012*. ACM, New York, NY, USA, 209–218.

[54] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. 2003. A Rule-based Framework for Role-based Delegation and Revocation. *ACM Trans. Inf. Syst. Secur.* 6, 3 (2003), 404–441.

[55] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. 2005. Evaluating Access Control Policies Through Model Checking. In *Proc. of ISC 2005 (LNCS)*, Vol. 3650. Springer, Heidelberg, Germany, 446–460.

[56] Zhikun Zhang, Jianguo Xiao, Hanyi Li, and Youping Geng. 2008. An Extended Permission-based Delegation Authorization Model. In *Proc. of CSSE 2008*, Vol. 3. IEEE, Los Alamitos, CA, USA, 696–699.