PhD-FSTC-2017-58
The Faculty of Sciences, Technology and Communication

Dissertation

Defence held on 14/09/2017 in Luxembourg

to obtain the degree of

Docteur De L'Université Du Luxembourg En Informatique

by

Ameni Ben Fadhel
Born on 21 January 1989 in Megrine (Tunisia)

Comprehensive Specification and Efficient Enforcement of Role-based Access Control Policies using a Model-driven Approach

Dissertation Defense Committee

Prof. Dr.-Ing. Lionel Briand, Dissertation Supervisor
University of Luxembourg, Luxembourg

Dr. Jacques Klein, Chairman
University of Luxembourg, Luxembourg

Dr. Domenico Bianculli, Vice Chairman (Dissertation Co-supervisor)
University of Luxembourg, Luxembourg

Prof. Dr. Davide Balzarotti, Member
Eurecom, France

Prof. Dr. Friedrich Steimann, Member
University of Hagen, Germany

To the memory of my grandfather Mahmoud Ben Fadhel and my grand mother Nana Saida.
May you rest in peace.

To my beloved parents and sister for always supporting me and standing by me.

Abstract

Prohibiting unauthorized access to critical resources and data has become a major requirement for enterprises. Access control (AC) mechanisms manage requests from users to access system resources; the access is granted or denied based on the authorization policies defined within the enterprise. One of the most used AC paradigms is role-based access control (RBAC), in which access rights are determined based on the user's role.

In this dissertation, we focus on the problems of modeling, specifying and enforcing complex RBAC policies, by making the following contributions:

1. the **GemRBAC+CTX** conceptual model, a UML extension of the RBAC model that includes all the entities required to express the various types of RBAC policies found in the literature, with a specific emphasis on contextual policies. For each type of policy, we provided the corresponding formalization using the Object Constraint Language (OCL) to operationalize the access decision for a user's request using model-driven technologies.
2. the **GemRBAC-DSL** language, a domain-specific language for RBAC policies designed on top of the GemRBAC+CTX model. The language is characterized by a syntax close to natural language, which does not require any mathematical background for expressing RBAC policies. The language supports *all* the authorization policies captured by the GemRBAC+CTX model.
3. **MORRO**, a model-driven framework for the run-time enforcement of RBAC policies expressed in GemRBAC-DSL, built on top of the GemRBAC+CTX model. MORRO provides policy enforcement for both access and usage control.
4. three tools (an editor for GemRBAC-DSL, a model transformation tool for GemRBAC-DSL, a run-time enforcement framework) have been implemented and released as part of this work.

The GemRBAC+CTX model and the GemRBAC-DSL language have been adopted by our industrial partner for the specification of the access control policies of a Web application in the domain of disaster relief

intervention. We have extensively evaluated the applicability and the scalability of MORRO on this Web application. The experimental results show that an access decision can be made on average, in less than 107 ms and that the time for processing a notification of an AC-related event is less than 512 ms. Furthermore, both the access decision time and the execution time for processing a notification of an AC-related event scale—in the majority of the cases—linearly with respect to the parameters characterizing AC configurations; in the remaining cases, the access decision time is constant.

Acknowledgements

Four years ago, more precisely in October 2013, I left my family, my friends and my country for a new challenging experience which is getting a PhD degree. However, I realized that doing a PhD it is not only about research, experiments and writing papers but also about the way you approach things and interact with other people. During these years, I have been in contact with people, from all over the world, who I would like to acknowledge for letting my PhD experience an enjoyable period in my life.

I would like to express my greatest thanks to my supervisor Prof. Lionel Briand who supported me with valuable and constructive feedback and always kindly encouraged me to succeed my thesis. I still remember his words on my first day in the university: "doing a PhD is like preparing for olympic games". It was a great honor and privilege to be part of his research team Software Verification and Validation.

I would like to express my infinite gratitude to my co-supervisor Dr. Domenico Bianculli. His counsel, dedication, availability, comments and corrections are a relevant lead to the success of this work. I would like to thank him for his guide and support to get through all the difficult moments, and also for his precious friendship. Throughout these years, I have learned a lot from his rigorous and positive attitude on both levels, professional and personal.

I would like to also express my gratitude to all team members of our industrial partner HITEC Luxembourg. This industrial collaboration helped me in validating my work with a real-world application. Moreover, I believe that my working experience with Hitec added a significant value to my professional background and carrer.

I would like to thank the members of my defense committee: Dr. Jacques Klein, Prof. Dr. Davide Balzarotti, and Prof. Dr. Friedrich Steimann for taking the time and furnishing efforts to evaluate this work.

I would also like to thank my former co-advisors Dr. Marouane Kessentini and Prof. Khaled Ghedira. Dear Professors, thank you for introducing me into the world of research.

I would like to express my gratitude to colleagues for providing a such motivating and great work environment.

I would like to thank my friends who have always supported and encouraged me. The list is really long, I will just cite Soraya Mefteh who has been a second mother for me in Luxembourg. I thank her for her affection, encouragement and long discussions.

I would like to express my deep gratitude and thanks to my family for their support and encouragement.

Finally, I would like to express my greatest thanks to all those who contributed in one way or another, to make this work.

Contents

List of Figures

List of Tables

Acronyms

BoD Binding of Duty

DSoD Dynamic Separation of Duty

DSoDCR Dynamic Separation of Duty on conflicting roles

DSoDCU Dynamic Separation of Duty on conflicting users

OCL Object Constraint Language

RBAC Role-based Access Control

SoD Separation of Duty

SSoD Static Separation of Duty

SSoDCP Static Separation of Duty on conflicting permissions

SSoDCR Static Separation of Duty on conflicting roles

SSoDCU Static Separation of Duty on conflicting users

UCON Usage control

UML Unified Modeling Language

Part I

Overture

Chapter 1

Introduction

1.1 Context and Motivation

Prohibiting unauthorized access to critical resources and data has become a major requirement for enterprises. Access control (AC) mechanisms manage requests from users to access system resources; the access is granted or denied based on the authorization policies defined within the enterprise. Access control systems can be grouped into three categories [1]: *discretionary* (DAC), *mandatory* (MAC), and *role-based* (RBAC). In DAC, access rights are directly assigned to each user; moreover, a user is the only entity that can control the access to her own object(s), by assigning access rights to other users. In the second category, MAC, the access rights are determined according to mandated regulations stated by a central authority. In RBAC, access rights are determined based on the user's role, e.g., her job or function, as well as on the permissions assigned to each role. A permission is an abstraction of a set of objects, i.e., resources, and the operations that can be performed on them. By decoupling users from permissions, RBAC simplifies the administration and the deployment of access control policies in large enterprises. In this dissertation, we focus on RBAC, since it has become the de facto standard for access control in enterprise systems [2].

The concept of role-based access control was initially proposed by Sandhu et al. in 1996 [3]; later on, the various initial proposals of RBAC models were consolidated into a unified standard model for RBAC, proposed by the NIST [4]. The basic RBAC model is composed of:

1. entities, corresponding to users, roles, sessions, and permissions;

2. relations among these entities.

A user is allowed to execute a set of permissions that corresponds to the role(s) assigned to her; in other words, a role maps each user to a set of permissions. A session maps each user to the set of her active role(s).

RBAC supports three security *principles*: least privilege, data abstraction, and separation of duty. The least privilege principle requires a user to be authorized

to execute only the minimal set of permissions needed for a given task, as determined by her role. The data abstraction principle is satisfied by abstracting low-level operations (e.g., the *read* and *write* operations provided by the operating system) with high-level operations defined for each business object in the system (e.g., updating the list of employees). The separation of duty principle states that no user should be given sufficient permissions to misuse the system. Although these principles are supported by RBAC, they are not automatically enforced by a system implementing RBAC: additional authorization constraints, called also *policies*[1] have to be defined to restrict the user's access.

Various types of authorization policies have been proposed in the literature. For instance, cardinality policies represent a bound on the number of roles and sessions to which a user can be assigned. Prerequisite policies are a precondition on user-role assignment, stating that a user can be assigned to a role only if the user is already a member of another role. Separation of duty policies (SoD) define a mutual exclusion relation among roles, permissions, or users. Dually, binding of duty (BoD) policies define a correlation among a set of operations that must be performed by the same user. Delegation policies allow a user to temporarily transfer a set of permissions associated to her role to another user. Context policies restrict a user from performing an action depending on her current location and/or on the time at which the action should happen.

Various extensions of the original RBAC model (commonly referred to as RBAC96) have been proposed to support these different types of policies. For example, support for delegation policies have been added in the models proposed in [5, 6, 7, 8]; the models introduced in [9, 10, 11, 12, 13] have added support for contextual policies. However, there is no unified framework that can be used to define all these types of authorization policies in a coherent way, using a common model. The lack of a unified framework makes difficult for practitioners to understand, select among, and implement the different types of policies proposed in the literature. Moreover, only few of them provide algorithms to evaluate these policies in order to make an access decision.

On a par with the definition of complex and more expressive RBAC models, there is the problem of defining *policy specification languages* that are at least as expressive as the policies supported by the existing models. While RBAC models provide the fundamental concepts needed to formalize various types of RBAC policies, policy specification languages represent a means to express RBAC policies that can be used (for both policy definition and enforcement) in practice. One group of proposals to define such languages revolves around XACML [14], the OASIS standard for defining access control policy languages. Since XACML does not support RBAC models natively, it has been extended with profiles specific to RBAC [15, 16]. Other types of RBAC policy languages are ontology-based [17, 18]

[1] In this dissertation we will use the word "policies" to avoid the confusion with "(OCL) constraints".

or logic-based [19, 20, 21] languages. The main problem of existing RBAC specification languages is that they do not support all the types of RBAC policies defined in the literature. For example, a simple delegation transfer policy like "any user with role r_1 can transfer her role to any user assigned to role r_2" cannot be expressed in any of the existing languages. Moreover, the semantics of some of these languages is not executable for the purpose of enforcing the policies specified with them. Furthermore, many of them are not designed to be used by practitioners.

The policies defined according to the models and languages described above can be checked at run time, by an enforcement mechanism, to make an access decision (allow/deny). In the context of Web applications, this mechanism is represented by a security layer, which serves as an intermediate filter between the client web service and the database (storing the application resources). An enforcement architecture based on the XACML language has been standardized by the OASIS community [14]. This architecture is essentially composed of a policy enforcement point (PEP) and a policy decision point (PDP). While the PEP intercepts a user request, the PDP checks the received request based on the authorization policies and makes an access decision. Some proposals have been inspired by this XACML architecture. For instance, authors in [22] propose a model-driven implementation for the PDP using UML/OCL. Authors in [5] propose a rule-based engine to enforce delegation and revocation policies. Other proposals [23, 24, 25, 26, 27, 28] extend the XACML architecture by introducing a second decision point SDP. Access decisions are cached in the SDP and they are reused if the request matches the one of the cached authorizations. Other proposals [29] deal with the aspects generation from policy specifications; the generated aspects are inserted into the application to be executed at run time. The main problem is that each enforcement mechanism and its corresponding model, implement a limited set of policies supported by the model and built into the mechanism.

These limitations have practical implications, since the lack of expressive models and policy specification languages limits the adoption, among practitioners, of the more expressive RBAC models proposed in the literature. In turn, this situation makes practitioners use simple(r) RBAC models, resulting in systems underspecified from the point of view of access control. We identified practical needs from HITEC Luxembourg[1], the industrial partner with which this project has been carried out. HITEC Luxembourg is a provider of situational-aware information systems for emergency scenarios. The requirements gathered from our partner show the need for an expressive model, a language to specify RBAC policies, and a mechanism to enforce these policies.

[1] http://www.hitec.lu/

1.2 Research Contributions

In this dissertation, we aim to address the aforementioned limitations of both specification and enforcement of RBAC policies. More specifically, we focus on the following research goals:

1. a formalization of RBAC policies to enable and facilitate the operationalization of the access decision procedure,
2. a high-level policy specification language,
3. and an efficient run-time enforcement mechanism.

To achieve these research goals, we make the following contributions:

1. the **GemRBAC+CTX** conceptual model, which is a *Generalized Model for RBAC supporting ConTeXt*. The model includes all the entities required to define the various types of access control policies proposed in the literature. We also propose a template that can be used for the formalization of the RBAC policies supported on the model to enable their operationalization. The specification follows a model-driven approach, based on UML and the Object Constraint Language (OCL): the supported policies are formalized as constraints expressed with OCL on the GemRBAC+CTX model.

   This contribution has been published in [30] and in [31]. This contribution is discussed in chapter 3.

2. the **GemRBAC-DSL** language, a specification language for RBAC policies. The language has been designed to cover the various types of RBAC policies captured by the GemRBAC+CTX model. The language sports a syntax close to natural language, to encourage its adoption among practitioners. The language semantics has been defined using a model-driven approach by mapping the constructs of the language to the corresponding OCL constraints defined for the GemRBAC+CTX model. We also define semantic checks to detect conflicts and inconsistencies among the policies written in a the GemRBAC-DSL language.

   This contribution has been published in [32] and is discussed in chapter 4.

3. a **run-time enforcement approach of RBAC policies** which checks whether a user is allowed to perform a requested operation. We designed MORRO, which a is a MOdel-driven fRamework for Run-time of RBAC pOlicies, by modeling the system state, from an RBAC point of view, as an instance of the GemRBAC+CTX model. Checking whether an access request should be granted, based on RBAC policies, will be reduced to checking whether the GemRBAC+CTX instance satisfies the corresponding OCL constraints.

   This contribution is under submission and is discussed in chapter 6.

4. **GemRBAC-DSL-Editor**, and **GemRBAC-DSL-Transform** and **MORRO** are available tools implemented during the PhD research work. GemRBAC-DSL-Editor and GemRBAC-DSL-Transform are the editor and the model transformation tool for the GemRBAC-DSL language, respectively. These tools are presented in chapter 4. MORRO is a model-driven framework for run-time enforcement of policies defined on the GemRBAC+CTX model, and is discussed in chapter 6.

5. an **empirical evaluation** is performed on an industrial architecture to assess the scalability and the performance of the model-driven framework for run-time enforcement of RBAC policies. We integrated MORRO into a real system, provided by our industrial partner.

1.3 Dissemination

The research work we performed during the PhD program has led to the following publications:

Journal paper

- Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. A Comprehensive Modeling Framework for Role-based Access Control Policies. *JSS*, 107:110–126, September 2015

Conference papers

- Ameni Ben Fadhel, Domenico Bianculli, Lionel Briand, and Benjamin Hourte. A Model-driven Approach to Representing and Checking RBAC Contextual Policies. In *Proc. of CODASPY2016*, pages 243–253. ACM, 2016

- Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. GemRBAC-DSL: A High-level Specification Language for Role-based Access Control Policies. In *Proc. of SACMAT2016*, pages 179–190. ACM, 2016

Under submission

- Model-driven Run-time Enforcement of Role-based Access Control Policies

1.4 Organization of the Dissertation

The rest of the thesis is structured as follows. After a preliminary chapter in which we present some background concepts on the original RBAC conceptual model and the various types of RBAC policies proposed in the literature, there are two parts. Part II includes chapter 3 on modeling role-based access control policies, chapter 4 presenting the policy specification language, and chapter 5 provides a review of existing role-based access control models and specification languages.

Part III contains chapter 6 on run-time enforcement of the access control policies expressed in part II. Chapter 7 summarizes the thesis contributions and discusses future research directions.

Chapter 2

Background

This chapter provides background information on the basic concepts used in this thesis. Section 2.1. presents the first RBAC conceptual model. Section 2.2 surveys the various types of access control policies existing in the literature and classifies them in a taxonomy.

2.1 The Original RBAC Conceptual Model

The original RBAC conceptual model, proposed in 1996 by Sandhu et al. [3], is composed of *users*, *roles*, *sessions*, and *permissions*; figure 2.1 illustrates the different components of this model and the relations between them. According to Sandhu et al., a role can be seen, at the same time, both as a collection of permissions and as a collection of users. A role can be assigned to one or more users via a *user-role assignment* relation. A *role-permission assignment* relation maps each role to one or more permissions. 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. A permission allows a user to perform some operation(s)

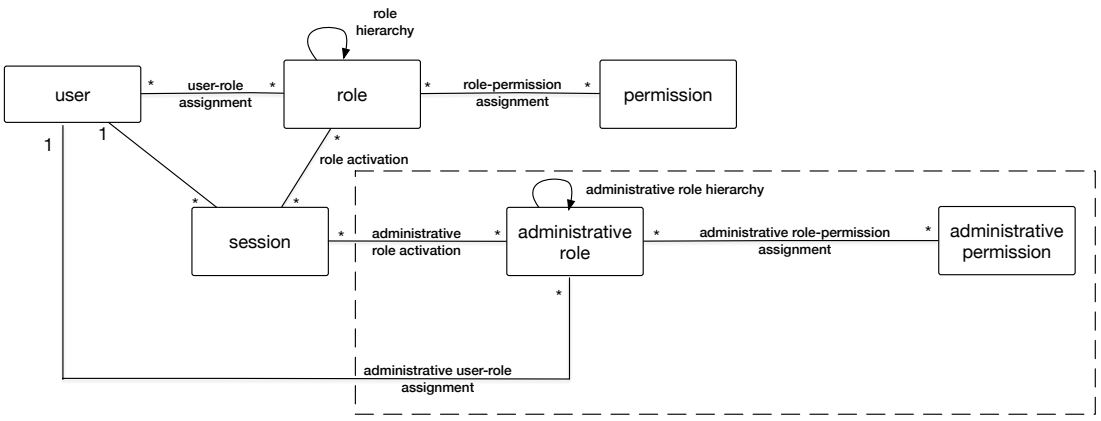


Figure 2.1: The original RBAC model [3]; the dashed line encloses the administrative model for RBAC

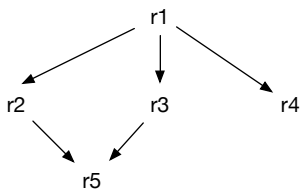


Figure 2.2: Role hierarchy example

on some resource(s) of the system.

A role can be inherited using a *role hierarchy* relation, as shown in figure 2.2. A role can have one or more juniors (sub-roles) denoted by an arrow. For instance, r_2, r_3 and r_4 are juniors of r_1. In addition to its assigned permissions, r_2 inherits all permissions from its ancestor r_1. Moreover, a junior role can have one or more ancestors (senior roles). As shown in figure 2.2, r_5 inherits not only the permissions of r_2 but also the ones of r_3.

RBAC administrative model

An instance of an RBAC model can include a large number of objects: the complexity and the size of such model instance represent a challenge to manage and maintain it. To ease its management, the RBAC model can be extended with an *administrative* part [3]; this part is shown in figure 2.1, enclosed with a dashed line. The administrative extension contains the concepts of *administrative role* and *administrative permission*. Examples of the latter are assigning a user to a role or adding a new permission or constraint. An *administrative role* can acquire only *administrative permissions* via an *administrative role-permission assignment*. In addition, *administrative user-role assignments* map administrative roles to users. An administrative role hierarchy defines inheritance relations between administrative roles.

2.2 RBAC Policies Taxonomy

In this section we present our classification of the existing types of RBAC policies found in the literature. The taxonomy shown in figure 2.3, contains at the top level eight types of access control policies; these are described in detail in the next sub-sections.

2.2.1 Prerequisite Policy

A prerequisite policy is a precondition on a role assignment; it can be evaluated either at the role level or at the permission level [3, 33]. A policy at the role level states that a user can be assigned to a role only if the user has been already assigned to another role. For instance, to acquire the role *developer* in a company,

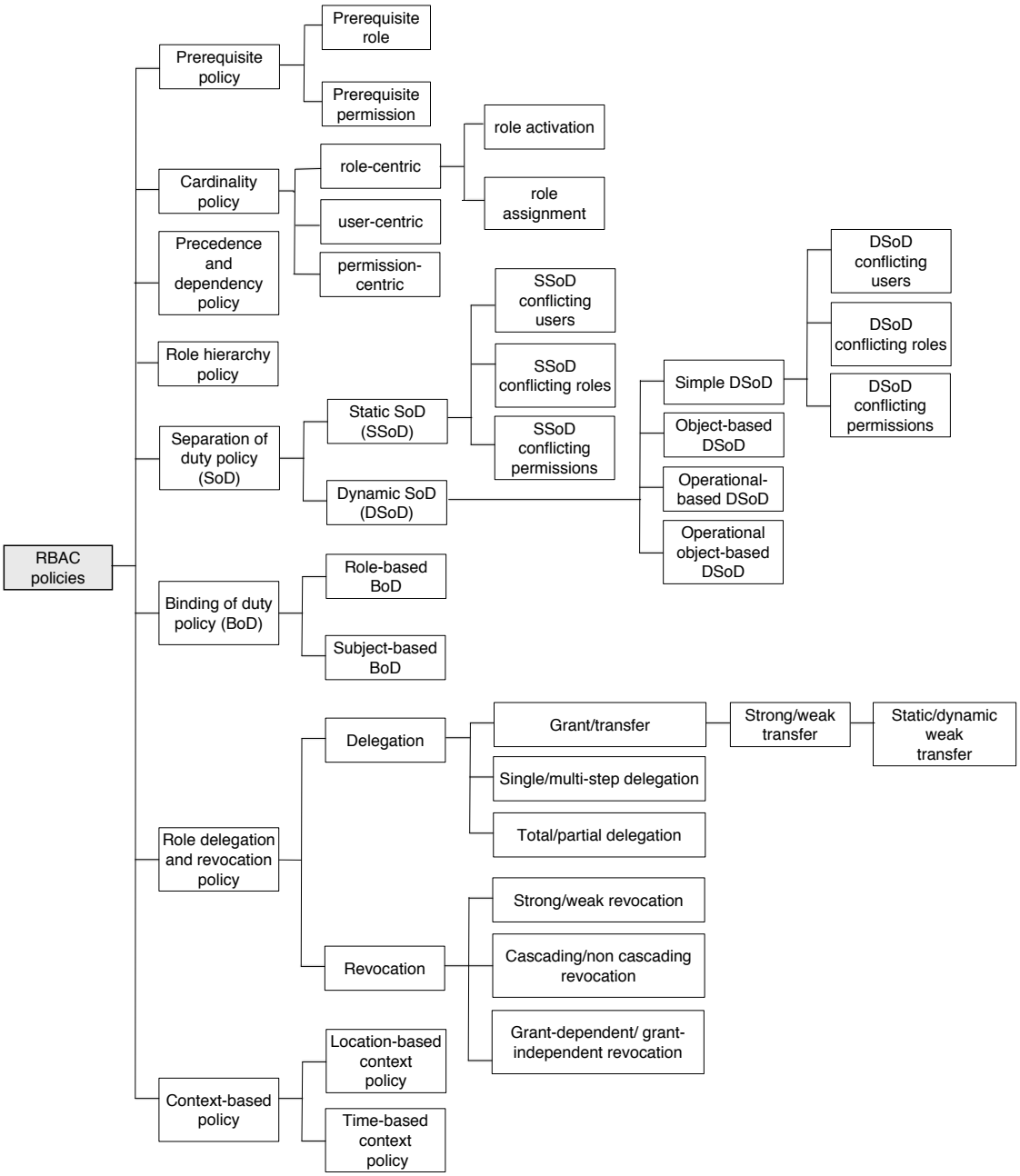


Figure 2.3: RBAC policies taxonomy

Bob must be already an *employee* in the company. A policy at the permission level indicates that a permission p_1 can be assigned to a role r only if this role already has permission p_2. For example, an employee cannot have the permission *write document* if she does not have the permission *read document*.

2.2.2 Cardinality Policy

We classify cardinality policies as role-centric, permission-centric, and user centric. A role-centric cardinality policy can represent a bound on the cardinality of either the *role activation* relation or the *user-role assignment* one [34]. An example of the former is a policy like: "a user cannot activate more than three roles in a session"; an example of the latter is "a role cannot be assigned to more than four users". A permission-centric cardinality policy can represent a bound on the number of roles assigned to a permission. An example of this policy is "a permission cannot be assigned to more than six roles". A user-centric cardinality policy can represent a bound on the number of roles assigned to a user. An example of this policy is "a user cannot be assigned to more than two roles".

2.2.3 Precedence and Dependency Policy

In some systems, assigning a role to a user does not entail that the user can activate it at any time. A role that can be activated is called *enabled*. Role enabling and role activation can be controlled by specific policies that determine precedence/dependency relationships [35] between two or more roles. For example, a policy like "the resident physician role can be enabled only if the attending physician role has been already activated" defines a precedence policy between the enabling of a role and the activation of another one. A precedence policy can be complemented with the corresponding dependency policy on the deactivation of a role; to continue the example above, the corresponding dependency policy would look like "the attending physician role cannot be deactivated if there is an activated resident physician role".

2.2.4 Role Hierarchy Policy

This type of policy specifies the assignments of roles through a hierarchy. As explained in section 2.1, assigning role r to user u implies assigning u all junior roles of r. A role hierarchy policy can also be applied at the permission level: if a role acquires a permission p, all its sub-roles will also acquire it [3]. For instance, considering figure 2.2, role r_5 will inherit the permissions of roles r_2 and r_3.

2.2.5 Separation of Duty (SoD) Policy

Separation of duty (SoD) policies [36] are used to define mutual exclusion relations among roles, permissions, or users. Mutually-exclusive entities are also called *conflicting*. In the literature, there are two types of SoD: static (SSoD) and dynamic (DSoD).

2.2.5.1 Static Separation of Duty (SSoD) Policy

This type of separation of duty is also known as strong exclusion [37]. It can refer to users, roles, and permissions [19, 34]. A user-centric static separation, also called *SSoD conflicting users*, states that two conflicting users cannot be assigned to the same role. A role-centric separation, also called *SSoD conflicting roles*, specifies that the same user cannot be assigned to mutually-exclusive roles. SSoD can also be permission-centric: this means that a user is not allowed to acquire two conflicting permissions and, symmetrically, that two conflicting permissions cannot be assigned to the same role.

2.2.5.2 Dynamic Separation of Duty (DSoD) Policy

Dynamic separation of duty deals with user-role activation through a session. In this case, a user is allowed to acquire conflicting roles; however, she cannot activate them at the same time. There are different types of DSoD [37]:

- **Simple DSoD** specifies that conflicting roles cannot be activated in the same session. As in SSoD, simple DSoD can also be user-, role- or permission-centric [33].
- **Object-based DSoD** allows a user to activate two conflicting roles at the same time, as long as she does not operate on the same object. For instance, a user can be an *author* or a *reviewer*; an *author* can submit a paper but cannot be a *reviewer* for it. This type of policy is also called Resource-based Dynamic Separation of Duty [38].
- **Operational-based DSoD** aims to prevent a user from performing all the operations in the same business task (e.g., a sequence of operations defined in a workflow). This means that a user can activate two conflicting roles at the same time, as long as the union of the operations allowed by the roles assignment does not correspond to the entire sequence of operations defined in the business task.
- **Operational Object-based DSoD** is a combination of the two previous types of policy. During the execution of a certain business task, a user can activate two conflicting roles at the same time, even if the union of the operations allowed by the roles assignment correspond to the entire sequence of operations defined in the business task. The only policy is that no user

can perform all the operations on the same object. This type of policy is also called History-based SoD [37] because the history of the operations performed by a user determines what she is allowed to do. Reference [37] also introduces the concepts of *order-dependent* and *order-independent* history-based SoD. The former requires that a role performs its operations in a particular order; the latter does not take into account the order of operations.

2.2.6 Binding of Duty Policy (BoD)

Unlike SoD policies, binding of duty policies define a correlation between a set of permissions; the permissions being correlated and the corresponding operations are also called *bounded*. BoD policies are usually defined in the context of workflow systems, whose activities can be performed by different subjects with different roles. Reference [39] classifies this type of policy as *role-based* and *subject-based*. In role-based BoD, the operations allowed by two or more permissions have to be performed by the same role. In subject-based[1] BoD, the same user must perform the operations allowed by the bounded permissions; moreover, the user has to maintain the same role while performing all these operations.

2.2.7 Role Delegation and Revocation Policy

A delegation allows a user (called the *delegator)* to transfer the permissions associated with her role (called the *delegated role)* to another user (called the *delegate*). A delegation takes place only if the delegate has not already been assigned to the delegated role or has already received it by means of another delegation. Furthermore, when a hierarchy has been defined for roles, the delegate receives not only the delegated role but also all its sub-roles. A delegation is put to an end through a *revocation* action. Below we present the different types of role delegation and revocation policies which can be set within an RBAC system.

2.2.7.1 Role Delegation Policy

A user can delegate her role or permission to another user. A delegation can be single- or multi-step, total or partial [5], and can be either of type "grant" or "transfer" [7].

A user can acquire a role through a standard user-role assignment (in which case the role is called *original*), or through a delegation (in which case the role is called *delegated*). A *single-step* delegation forbids a user to delegate a delegated role. On the other hand, a *multi-step* delegation allows a user to delegate a delegated (i.e., non-original) role; however, the number of delegation steps is bounded and should not exceed a maximum delegation depth, predefined for the system.

[1] The word *subject* refers to a user having activated a certain role.

With a *total* delegation, a user delegates all the permissions belonging to a certain role; with a *partial* delegation, a user delegates only a subset of the role permissions.

When a delegation is of type "grant", the delegator can continue to use the role that has been delegated. On the other hand, when the delegation is of type "transfer", right after the delegation the delegator is no longer assigned to the role that has been just delegated.

As mentioned above, when a hierarchy has been defined for roles, the *delegate* receives not only the delegated role but also all its sub-roles. Delegations of type "transfer" can be *strong* or *weak* depending on the assignment of the juniors of the delegated role to the delegator. With a *strong transfer*, the delegator is not assigned to the delegated role and to all its sub-roles anymore. A *weak transfer* can be classified as *static* or *dynamic*. With a *static weak transfer*, the delegator keeps using a subrole r of the delegated role only if she is a member of another senior of role r. In case of a *dynamic weak transfer*, the delegator keeps using a subrole r of the delegated role only if she activates a senior of role r. As an example, consider the role hierarchy in figure 2.2 and assume that user u_1 is assigned to r_2 and to r_3. If user u_1 delegates her role r_2 to user u_2, the latter will acquire role r_2 and its junior role r_5. If the delegation is a static weak transfer, after the delegation u_1 will still be a member of role r_5, since she is still a member of role r_3, which is a senior of role r_5. On the other hand, if the delegation by user u_1 of role r_2 to user u_2 is of type dynamic weak transfer, the delegator will be still a member of role r_5 after the delegation only if role r_3 is active.

2.2.7.2 Role Revocation Policy

A delegation is often followed by a revocation; in the following we refer to the revocation model proposed in [5].

A role can be revoked either by any user who acquired the role via a user-role assignment, or by the user who delegated the role. In the former case, the revocation is called *grant-independent*; in the latter it is called *grant-dependent*.

The *dominance* of a revocation refers to its effects on the user-role assignment relation, as determined by role hierarchy; it can be either *weak* or *strong*. Let us consider the case in which a user may be directly assigned to a role or may be assigned to a role by inheriting it through a role hierarchy. A *weak* role revocation only removes the user from the delegated role and does not impact on the other roles acquired through the role hierarchy. A *strong* revocation removes the user from the delegated role and also from the ones inherited through the role hierarchy. For instance, if user u_1 delegates her role r_1 to user u_2, u_2 will acquire r_1 and its juniors r_2, r_3, r_4 and, r_5 as shown in figure 2.2. With a strong revocation, u_2 will be revoked from r_1 and all its junior roles.

A revocation can affect not only the user who received the role being revoked, but also the other delegate users determined by a multi-step delegation. A *cascading* revocation removes the delegated role assignment and the assignment(s) resulting from a multi-step delegation. A *non-cascading* revocation removes only the requested delegation and does not affect the delegation(s) of the delegated role.

If a delegation has a certain duration, a revocation can be triggered automatically after the duration expires. If a duration is not specified, a delegation remains active until a user revokes it explicitly.

2.2.8 Context-based Policy

Contextual policies allow a user to perform an action depending on her current location (*location-based context policies*) [10] or on the time (*time-based context policies*) at which the action should happen [11].

Contextual information can be assigned to users, roles and/or permissions.[1] A user context refers to her current position and time. A role (respectively, a permission) context refers to the location from and the time at which the corresponding role (respectively, permission) can be activated.

A location can be *physical* or *logical*. A *physical* location corresponds to some specific geographic coordinates; a *logical* location corresponds to a bounded space like a specific room in a building. With a *location-based context policy*, roles can be enabled and activated when the user's position matches the location specified in the policy. A role is automatically disabled when the user leaves the geofence determined by the location specified with the policy.

Time-based context policies specify the periodicity and/or the duration of role activation [9]. For example, a policy can restrict a role to be activated only on working days within a predefined range of hours. Moreover, a policy can limit the cumulative time during which a role is active, e.g., "for three hours per day".

A context-based policy can be specified at the permission level to prohibit a user to perform a permission assigned to her *active* role, when her contextual information is not valid.

These policies can also be used to manage conflicting roles that a user can acquire through a role hierarchy. Each of the conflicting roles can have a time-based context policy that restricts its activation to a certain period of time. If the time windows of the activation of conflicting roles do not overlap with each other, the SoD policy will not be violated. More in general, context policies can be applied to a hierarchy policy to restrict role inheritance and activation depending on location and/or time [13].

[1] A wider notion of the context is covered in the attribute-based access control (ABAC) model [40] by considering the context as an attribute that can be assigned to users, roles, and permissions.

Part II

Specification of Role-based Access Control Policies

Chapter 3

Modeling Access Control Policies

As mentioned in the previous chapter, many types of RBAC policies have been proposed in the literature. However, the original RBAC model [3] supports a limited number of these authorization policies, which cannot fulfill the expressiveness requirements that have emerged in the recent years in modern organizations. Examples of these new requirements are supporting delegation and revocation of roles/permissions, and enabling roles based on the spatio-temporal context of users.

To fill this gap, researchers have proposed several extensions of the original RBAC model, to support the definition of new types of policies. Though this work opens new possibilities for applying RBAC in modern enterprise systems, it is not easy to exploit in its current form. Indeed, these types of policies and their corresponding models are scattered across multiple sources, are defined using different formalisms, and sometimes the concepts are expressed in an ambiguous manner.

This situation is very impractical for practitioners who want to select the relevant types of policies to be implemented in their systems. Moreover, they are faced with several models, often partially overlapping with each other, but with slightly different semantic variations. Furthermore, to the best of our knowledge, there is no model that can express *all* types of policies that we have identified in our taxonomy (see chapter 5). Last, scattered and heterogenous models make it difficult for researchers to understand the state of the art in a coherent manner.

We contend there is clearly a need for organizing the various types of RBAC authorization policies systematically, based on a unified framework. Our goal is to define a conceptual model significantly more expressive than the state of the art, on top of which we can operationalize the access decision procedure. A more expressive and operational model critically determines its applicability in real scenarios. To achieve this goal, we follow a model-driven approach, based on UML and the Object Constraint Language (OCL) [41].

This chapter makes the following contributions:

1. the GemRBAC+CTX model, which is a generalized model for RBAC that includes all the entities required to define the policies classified in the tax-

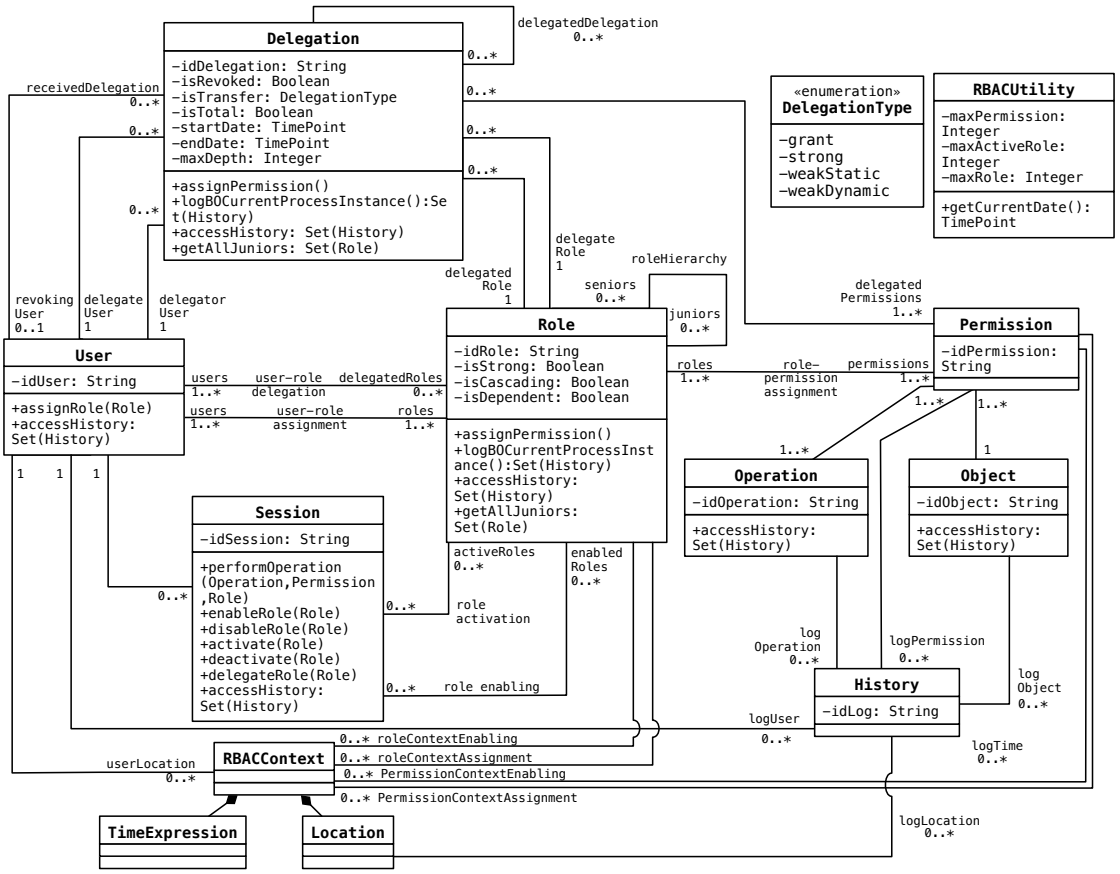


Figure 3.1: The GemRBAC+CTX conceptual model

onomy presented in chapter 2, with a specific emphasis on context-based policies;

2. a template for the formalization of RBAC policies included in the taxonomy as OCL constraints to operationalize the access decision for user's requests using model-driven technologies;
3. the application of the GemRBAC+CTX model for the specification of real RBAC policies in an industrial setting.

The rest of the chapter is organized as follows. Section 3.1 introduces the GemRBAC+CTX model. Section 3.2 presents the templates for the formalization of RBAC policies using OCL constraints defined on the GemRBAC+CTX model. Section 3.3 reports on an industrial application of the proposed model for the verification of RBAC policies.

3.1 The GemRBAC+CTX Model

In this section we present our GemRBAC+CTX model (shown in figure 3.1 as a UML class diagram), which extends the original RBAC96 model with additional concepts. The components of the original model are modeled as classes (`User`, `Session`, `Role`, `Permission`) and associations (between `User` and `Role`, `User` and

`Session`, `Role` and `Permission`). We define a permission as a set of operations that can be performed on an object: we model this by associating the class `Permission` with the classes `Object` and `Operation`. The class `Object` can be extended to include additional information as required by the application needs, e.g., state information for a stateful object.

Role activation is modeled as an association between the `Session` and `Role` classes. We also model role enabling with another association between these two classes.

In GemRBAC+CTX the concept of delegation is represented by the class `Delegation`. This class contains various attributes: `startDate` and `endDate` represent the bounds of a delegation period; the boolean attributes `isTotal` and `isRevoked` represent, respectively, whether the delegation is total and whether the delegation has been revoked or not. The attribute `isTransfer` indicates whether the delegation is of type "transfer" or "grant". The concepts of delegator, delegate and revoking users are represented as associations between the `User` and `Delegation` classes. Similarly, the concepts of the role of the delegator, the role of the delegate, and the delegated role are represented as an association between class `Role` and class `Delegation`. We also model the set of delegated roles as an association between classes `User` and `Role`. The specific type of revocation is represented by specific boolean attributes of the `Role` class: `isDependent`, `isStrong`, and `isCascading`.

The context is modeled with the class `RBACContext`, which is composed of two classes, `TimeExpression` and `Location`. The `RBACContext` class represents spatial and temporal information, which are associated with each instance of the `User`, `Role` and `Permission` classes. The user context, representing her spatial information, is modeled with the `userLocation` association between the `User` and `RBACContext` classes. The context in which a role should be assigned (as prescribed by a contextual policy) is modeled with the `roleContextAssignment` association between the `RBACContext` and `Role` classes; similarly, the context in which a role should be enabled (as prescribed by a contextual policy) is modeled with the `roleContextEnabling` association between these two classes. The context for permission enabling and assignment is modeled in a similar way with the `permissionContextAssignment` and `permissionContextEnabling` associations between the `RBACContext` and `Permission` classes. The temporal context, modeled as class `TimeExpression` refers to the current time or the time on which a given role, respectively permission, can be enabled/assigned. The spatial context, modeled as class `Location`, refers to a specific bounded area or geographical location. It can be assigned to a user, role or permission. A role, respectively permission, is enabled/assigned if its location matches the user's position. The context specification in the GemRBAC+CTX model is detailed in the next section.

Since policies such as *History-based SoD* require a record of operations performed over time, we introduce the `History` class. An instance of this class records

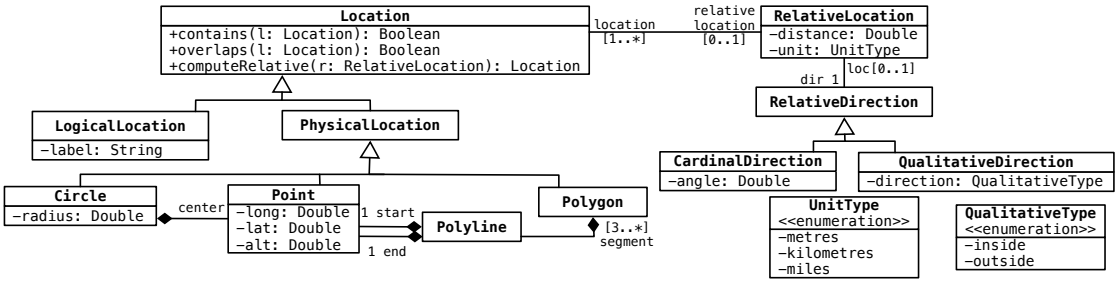


Figure 3.2: Spatial context in GemRBAC+CTX.

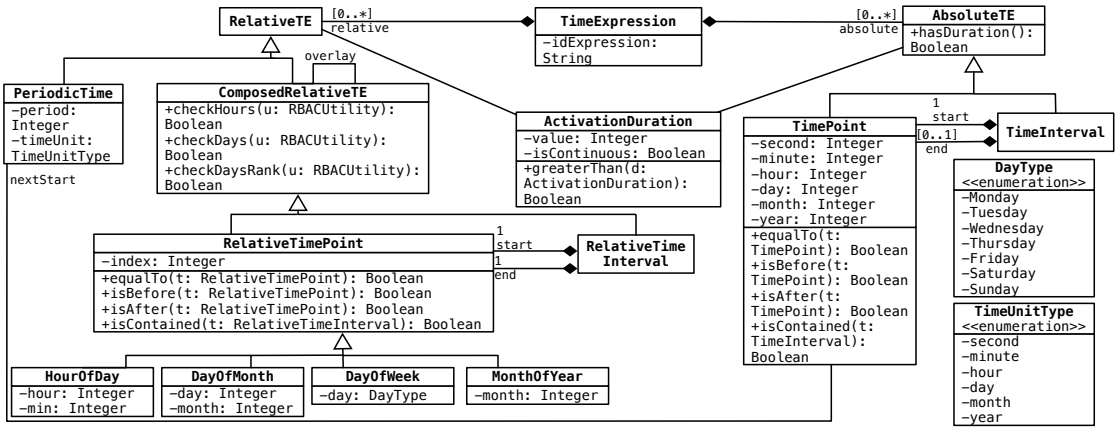


Figure 3.3: Temporal context in GemRBAC+CTX.

that a user performed a certain operation on a given object according to a given permission, in a certain location, while having a certain role; these data are gathered through the associations with (respectively) `User`, `Operation`, `Object`, `Permission`, `Location` and `Role`. Notice that we also record the time at which the user performed the operation (see section 3.3 for more details).

3.1.1 Modeling Context in GemRBAC+CTX

A temporal (respectively, spatial) context models the time (location) on which a given role, or permission, can be enabled/assigned; a role or permission can be enabled/assigned if its contextual information matches the user's one. In this section, we extend class `RBACContext` to support more detailed spatial and temporal concepts that can be specified by context-based policies. An example of these policies is "enable role *r every Monday, from 9.00 to 11.00*" or "enable role *r within a radius of 20 miles from the main building*". In the rest of this section, we provide a description of the temporal and spatial context over the GemRBAC+CTX model.

3.1.2 Modeling Temporal Context

To support temporal context specification in the GemRBAC+CTX model, we introduce a new class hierarchy under class `TimeExpression`. The new classes and

their associations are shown in the class diagram in Figure 3.3.

A time expression is composed of absolute and/or relative time expressions; these concepts are modeled as classes `AbsoluteTE` and `RelativeTE`. An absolute time expression refers to a concrete point or interval in the timeline. An absolute time point, modeled with the class `TimePoint`, corresponds to a given time instant, e.g., "*January 21, 2014 at 8:00:00*". Hereafter, to improve the readability we will omit the hours from a time point when we refer to midnight. A time interval, modeled with class `TimeInterval`, corresponds to a segment in the timeline; a time interval can be either of type bounded or unbounded. A bounded time interval corresponds, for example, to the expression "*from January 21, 2014 to April 25, 2015*". This interval has a start `TimePoint` (*January 21, 2014*) and an end `TimePoint` (*April 25, 2015*). An unbounded interval corresponds to the expression "*starting from October 15, 2013*"; it has only the start `TimePoint` (*October 15, 2013*) and is unbounded to the right.

A relative time expression is an expression that cannot be mapped *directly* to a point or to an interval in the timeline. For example, the common expression "*(at) 9 a.m.*" by itself cannot be directly mapped to a point in the timeline unless another expression, e.g., "*(on) May 2, 2015*" is specified. Class `RelativeTE` has two subclasses, `RelativeTime` and `PeriodicTime`. By analogy with the class `AbsoluteTE`, the class `RelativeTime` has two subclasses, `RelativeTimePoint` and `RelativeTimeInterval`. Class `RelativeTimePoint` has four subclasses: `HourOfDay` refers to a specific hour of the day, e.g., "*(at) 9 a.m.*"; `DayOfWeek` corresponds to a given day of the week, e.g.,"*(on) Monday*" refers to any Monday; `DayOfMonth` refers to a day in a month such as "*(on) April, 5*"; `MonthOfYear` refers to a given month, e.g., "*(in) April*". Unlike class `TimeInterval`, class `RelativeTimeInterval` always refers to a bounded time interval, whose start and end points have both the same type (a subclass of `RelativeTimePoint`). Class `ComposedRelativeTE` can be recursively composed with itself through the association `overlay`, to represent composite time expressions. These composite expressions are required to have composite elements of different granularity. We enforce this requirement by defining a structural constraint on the model. Informally, a `MonthOfYear` can overlay either a `DayOfWeek` or an `HourOfDay`; a `DayOfWeek` or a `DayOfMonth` can overlay only an `HourOfDay`. The same constraint applies if any subclass c of `RelativeTimePoint` mentioned in it is replaced with a `RelativeTimeInterval` with bounds of type c. An example of an expression that can be modeled by composing different instances of `ComposedRelativeTE` by means of the `overlay` association is "*in February, from the second Monday to third Friday, from 10:00:00 to 12:00:00*". This expression is modeled by an instance of `MonthOfYear` (*February*) overlaid with an instance of `RelativeTimeInterval`, with start- and end-point of type `DayOfWeek` (*from Monday to Friday*), overlaid with an instance of `RelativeTimeInterval`, with start- and end-point of type `HourOfDay` (*from 10:00:00 to 12:00:00*). The indexes that

refer to a specific occurrence of Monday and Friday are modeled with the `index` attribute, which is defined only for class `DayOfWeek`.

Class `RelativeTE` can also represent periodicity in temporal expressions such as "*every 3 months*". The periodicity is modeled with its subclass `PeriodicTime`. Its attribute `period` is a numeric value associated with a time unit (e.g., day, hour, month) modeled with the attribute `timeUnit`. A `PeriodicTime` is always part of a `TimeExpression` that has exactly one `AbsoluteTE`; the latter defines either the starting time of the period (as in "*every 3 months, starting from April 5, 2015*") or the time interval in which it applies (as in "*every 3 months, from April 5, 2015 to June 8, 2017*"). We assume that each `PeriodicTime` has a `nextStart` association with a `TimePoint` corresponding to the beginning of the next period.

In the context of RBAC, a temporal context can have a time-based policy that represents a bound for the sum of activation durations of a given role (or permission). For instance, a security engineer could enable a certain role from Monday to Friday but allow users to activate it only for two hours over the five days. We keep track of this duration with class `ActivationDuration`, which is associated with classes `RelativeTE` and `AbsoluteTE`. Moreover, this duration can be cumulative (i.e., related to multiple sessions) or noncumulative (i.e., related to the current session); this concept is represented by the boolean attribute `isContinuous` of the class `ActivationDuration`.

3.1.3 Modeling Spatial Context

Similarly to what we have done for temporal context, to support temporal context specification in the GemRBAC+CTX model, we introduce a new class hierarchy under class `Location`. The new classes and their associations are shown in the class diagram in Figure 3.2.

At a very high-level, a location represents a specific bounded area or point in space. A location can be either physical or logical; these concepts are modeled as classes `PhysicalLocation` and `LogicalLocation`.

A physical location identifies a precise position in a geometric space. We consider three possible ways to express a physical location and we model them as subclasses of `PhysicalLocation`. Class `Point` represents a geographic coordinate with latitude, longitude and altitude. Class `Circle` represents a circular area, characterized by a radius and a center. Class `Polygon` is an area enclosed by at least three segments, which are modeled with class `Polyline`; each `Polyline` is a segment composed of a start and an end `Point`. Notice that a `Polygon` (as a set of `Polylines`) can model areas with complex shapes, such as the border of a city.

A logical location is an abstraction of one or many physical locations. For instance, the logical location "*offices on the second floor*" refers to the set of physical locations corresponding to the actual office rooms in the second floor of a building. A logical location can also be a convenient shorthand to identify a geographical landmark without providing its coordinates. The concept of logical location is

modeled with class `LogicalLocation`. We assume that there is a geocoding function that maps each `LogicalLocation` to the corresponding `PhysicalLocation`(s).

A location can be defined *relatively* to another location by providing a direction and optionally a distance. We model these concepts with class `RelativeLocation`, which is associated with class `RelativeDirection`, and has a `distance` attribute. The latter has two subclasses, `CardinalDirection` and `QualitativeDirection`. Class `CardinalDirection` represents the degrees of rotation based on cardinal points on a compass. An example of a location using a relative location denoted with a cardinal direction is "*6 miles West from the Tour Eiffel*". This expression contains a distance (*6 miles*), a cardinal direction (*Southwest, i.e., 225°*) and a logical location (*Tour Eiffel*). Class `QualitativeDirection` represents a relative proximity to a location, such as "*inside*" or "*outside*". An example of a location using a relative location denoted with a qualitative direction is "*100 meters outside the White House*". This expression contains a distance (*100 meters*), a qualitative direction (*outside*) and a logical location (*White House*). Class `Location` provides some operations that check for topological relations between locations: operation `contains` checks if a location is a part of another one; operation `overlaps` checks if two locations share a common area.

Notice that the user context is composed only of a set of `Location`s including *at most one* `PhysicalLocation` and a set of `LogicalLocation`s.

3.2 OCL Specification of RBAC Policies

In this section we show how the RBAC policies described in section 2.2 of chapter 2 can be formalized as OCL constraints on the GemRBAC+CTX model. This formalization aims to precisely specify the semantics of such policies in such a way that they can be operationalized, for example through an OCL constraint checker. Thus, the problem of making an access decision for RBAC policies can be reduced to checking the corresponding OCL constraints on an instance of the GemRBAC+CTX model. While defining the OCL constraints, we made the following working assumption which states that "at any time during the execution of the system for which RBAC policies are defined, we are able to take a snapshot of the system state and represent it as an instance of the GemRBAC+CTX model". This assumption is based on previous work on model-driven run-time verification [42], which shows how a run-time system can be represented as a "live" instance of a conceptual model, on which to check OCL constraints. For the purpose of the formalization, we enrich the model with some helper classes and operations, also included in figures 3.1, 3.2 and 3.3.

The Ecore version of the GemRBAC+CTX model, the OCL constraints defined in the following subsections, and model instances that violate/satisfy them are available at https://github.com/AmeniBF/GemRBAC-model.git.

3.2.1 Prerequisite Policy

A prerequisite policy can specify a pre-condition either for user-role assignment or for role-permission assignment. In the first case, the policy states that to acquire role r_1, the user must have been already assigned to role r_2. This policy can be written in OCL as an invariant of the `Role` class:

```
1 context User inv PreqRole:
2 let roleSet: Set(Role) = self.roles -> union(self.delegatedRoles) -> asSet()
3 in roleSet-> select (r:Role | r.idRole ='r1' or r.idRole ='r2') -> size() =2
```

In this constraint we first select the set of roles assigned and delegated to the user (line 2); these roles are derived from navigating the `roles` and `delegatedRoles` associations. Then, we check whether roles r_1 and r_2 are included in this list (line 3).

A prerequisite policy on role-permission assignment has a similar structure:

```
context Role inv PreqPermisssion:
self.permissions -> select (p:Permission | p.idPermission = 'p1'
                    or  p.idPermission = 'p2') -> size() = 2
```

3.2.2 Cardinality Policy

A cardinality policy on the role activation relation is expressed in OCL as an invariant of the `Session` class:

```
context Session inv CardinalityActivation:
let u: RBACUtility = RBACUtility.allInstances() -> any(true)
in  self.activeRoles -> size() <= u.maxActiveRole
```

In this expression, the number of roles activated for the current session is determined by the cardinality of the `activeRoles` association between classes `Session` and `Role`; `maxActiveRole` is a constant defined in the class `RBACUtility`.

The policies for the number of roles assigned to a user and for the number of permissions assigned to a role are defined in a similar way for classes `User` and `Role`:

```
context User inv CardinalityRole:
let u: RBACUtility = RBACUtility.allInstances() -> any(true),
    roleSet : Set(Role) = self.roles -> union(self.delegatedRoles)-> asSet()
in  self.roleSet -> size() <= u.maxRole

context Role inv CardinalityPermission:
let u: RBACUtility = RBACUtility.allInstances() -> any(true)
in  self.permissions -> size() <= u.maxPermission
```

Notice that when expressing the policy on the number of roles assigned to a user, we consider both assigned and delegated roles.

3.2.3 Precedence and Dependency Policies

Precedence and dependency policies control the enabling and the deactivation of roles. A precedence policy on the enabling of a role with respect to the activation of another one can be expressed as an invariant of the `Session` class:

```
1 context Session inv RoleEnablingPrecedence:
2 let r1: Role = self.enabledRoles-> select(a: Role| a.idRole = 'r1')->any(true),
3     r2: Role = Role.allInstances()->select(a: Role| a.idRole = 'r2')->any(true)
4 in if not r1.oclIsInvalid() then
5         Session.allInstances()->exists(s: Session| s.activeRoles->includes(r2))
6     endif
```

In the OCL expression above, we first select, among the roles enabled in the session, the instance corresponding to role r_1 (line 2); these roles are derived from navigating the `enabledRoles` association between classes `Session` and `Role`. Then, we select the instance corresponding to role r_2 (line 3). We check if role r_1 is enabled by checking whether the selected instance of this role is not null (line 5); this check is done by calling the `oclIsInvalid` operation. If role r_1 is enabled then r_2 must be among the activated roles; the list of these roles is derived from the `activeRoles` association.

The corresponding dependency constraint can be expressed as an invariant of of the `Role` class:

```
1 context Role inv RoleActivationDependency:
2 if a.idRole = 'r2' and self.sessionsA -> isEmpty() then
3     let r1: Role = Role.allInstances()-> select(a: Role| a.idRole = 'r1')
4             ->any(true)
5     in Session.allInstances() -> forAll (s:Session
6           | s.activeRoles -> union(s.enabledRoles) -> excludes(r1))
7 endif
```

In this OCL constraint we first check (line 2) if the current role is r_2 and if this role is active. The list of sessions where the role is active is derived from the `sessionsA` association. If role r_2 is not active (corresponding to the case in which the condition `self.sessionsA -> isEmpty()` is satisfied), we select (line 3) the instance of role r_1 from all the instances of class `Role`. Then, we check if role r_1 is enabled or active in all instances of class `Session`.

3.2.4 Role Hierarchy Policy

A role hierarchy policy can be expressed on user-role and permission-role assignment relations. In the first case, it states that if a user acquires a role, she will also acquire all its juniors. This can be expressed in OCL as an invariant of the `User` class:

```
context User inv RoleHierarchy:
let r1:Role = self.roles -> select(a: Role | a.idRole = 'r1')->any(true)
```

```
in if not r1.oclIsInvalid() then
        self.roles -> includesAll(r1.juniors)
   endif
```

In this expression we first select role r_1 from the roles assigned to the user; this list is derived from the `roles` association. Then, we check if role r_1 is assigned to the user by checking whether the selected instance of this role is not null (line 3); this check is done by calling the `oclIsInvalid` operation. The condition in line4, states that the roles assigned to the user should include all junior roles of role r_1; the junior roles are derived from the `juniors` association.

A role hierarchy policy on the role-permission assignment states that if a role acquires a permission, all its sub-roles will acquire it. This policy is defined as an invariant of the class `Role`:

```
context Role inv PermissionHierarchy:
let p1: Permission = self.permissions
                    -> select(a: Permission| a.idPermission = 'p1')->any(true)
in if not p1.oclIsInvalid() then
        self.juniors -> forAll (r: Role | r.permissions -> includes(p))
    endif
```

In this expression, we check if each sub-role is assigned to the p_1 permission.

3.2.5 Separation of Duty Policy (SoD)

3.2.5.1 Static SoD

The static SoD (SSoD) can be user-, role- or permission-centric. The user-centric SoS specifies that role r_1 can be assigned either to user u_1 or to user u_2, but not to both. This policy can be expressed in OCL as an invariant of the `Role` class:

```
1 context Role inv SSoDCU:
2 if self.idRole ='r1' then
3   let conflictingUsers: Set (User) = self.users -> select (u:User
4                                     | u.idUser = 'u1' or u.idUser = 'u2')
5   in conflictingUsers -> Size() <2
6 endif
```

In the OCL expression above, we first select the list of conflicting users, u_1 and u_2, from the list of users assigned to role r_1; these users are derived from the `users` association. The condition states that the list of users assigned to the role should contain either user u_1 or user u_2. but not both. In other words, the list of conflicting users assigned to role r_1 should contain at most one user.

The role-centric and permission-centric SoD are defined in a similar way as invariants of the `User` and `Permission` classes, respectively:

```
context User inv SSoDCR:
let roleSet : Set(Role) = self.roles -> union(self.delegatedRoles) -> asSet(),
   conflictingRoles: Set (Role) = roleSet-> select (r:Role
```

```
                    | r.idRole='r1' or r.idRole='r2')
in conflictingRoles -> size()<2

context Role inv SSoDCP1:
let conflictingP: Set (Permission) = self.permissions -> select (p:Permission
                    | p.idPermission='p1' or p.idPermission='p2')
in conflictingP -> size()<2

context Permission inv SSoDCP2:
if self.idPermission  ='p1' then
   let conflictingRoles: Set (Role) =  self.roles  -> select (r:Role
                    | r.idRole='r1' or  r.idRole='r2')
   in conflictingRoles -> size()<2
endif
```

3.2.5.2 Dynamic SoD

Unlike static [1]SoD, dynamic SoD (DSoD) allows a user to acquire two conflicting roles but she cannot activate them at the same time. To express that roles r_1 and r_2 should not be active in the same session, we can write the following OCL constraint as an invariant of the `Session` class:

```
context Session inv DSoDCR:
let conflictingRoles: Set (Role) = self.activeRoles -> select (r:Role
                            | r.idRole='r1' or r.idRole='r2')
in conflictingRoles -> size () <2
```

Object-based SoD is another variation of DSoD which allows a user to activate two conflicting roles (r_1 and r_2) at the same time, as long as she does not operate on the same object. It can be expressed in OCL as an invariant the `Session` class:

```
1  context Session inv ObjectDSOD:
2  let conflictingRoles: Set (Role) = self.activeRoles -> select (r:Role
3                          | r.idRole='r1' or r.idRole='r2')
4  in  if conflictingRoles -> size()>1 then
5  let r1:Role = conflictingRoles -> select (r:Role | r.idRole='r2') -> any(true),
6      r2:Role = conflictingRoles -> select (r:Role | r.idRole='r1') -> any(true),
7      logr1: Set (History) = r1.accessHistory() -> select (a: History
8                          |  a.user= self.user),
9      logr2: Set (History) =  r2.accessHistory() -> select (a: History
10                         |  a.user= self.user),
11     objects1: Set (Object) = logr1 -> collect (object),
12     objects2: Set (Object) = logr2 -> collect (object)
13 in  objects1  -> intersection (objects2) -> isEmpty()
14 endif
```

In this constraint we refer to instances of the `History` class, which keeps track of each operation performed in the system, recording the user who performed it, the

[1]For all abbreviations see the glossary on page xiii.

role she had, the object on which the operation was performed, the time and the location. We use the operation `accessHistory` of the `Role` class to retrieve the instances of `History`. filtered on the current user (lines 8 and 10). Then, for each conflicting role (in this case r_1 or r_2), we collect the list of objects (`objects1` and `objects2`, respectively) from the list of logs performed by the current user while activating a conflicting role (lines 11 and 12). Checking whether the history of the conflicting role(s) does not contain any operation performed on the same object is equivalent to checking whether the intersection of the list of objects (`objects1` and `objects2`) in their history is empty (line 13). The generalized version of this policy containing more than two conflicting roles is expressed in a similar way:

```
context Session inv ObjectDSOD:
let conflictingRoles: Set (Role) = self.activeRoles -> select (r:Role
                                 | r.idRole='r1' or r.idRole='r2'
                                 or r.idRole='r3')
in if conflictingRoles -> size()>1 then
       conflictingRoles-> forAll(r:Role|
        let rlog:  Set (History) =  r.accessHistory()-> select (log:History
                                  | log.user = self.user)-> asSet(),
            roleslog: Set (History) = conflictingRoles -> excluding(r)
                                  -> collect(r.accessHistory()),
            userlog: Set (History) = log -> select (log:History
                                  |  log.user = self.user )-> asSet(),
            objects: Set(Object) = userlog -> collect (object)-> asSet(),
        in rlog -> collect  (object) -> intersection (objects) -> isEmpty())
    endif
```

With an Operational-based DSoD policy, a user can activate two conflicting roles (r_1 and r_2) at the same time, as long as the union of the operations allowed by the roles assignment does not correspond to the entire sequence of operations (op_1 and op_2) defined in a business task. This can be expressed in OCL as an invariant of class `Session`:

```
1  context Session inv OperationalDSoD:
2  let conflictingRoles: Set (Role) = self.activeRoles -> select (r:Role
3                                   | r.idRole='r1' or r.idRole='r2')
4  in if conflictingRoles -> size()>1 then
5        let opBT : Set(Operation) = Operation.allInstances()
6                                   -> select(o: Operation | o.idOperation = 'op1'
7                                   or o.idOperation ='op2')->asSet()
8           op: Set(Operation) = conflictingRoles
9                                   -> collect (permissions.operations) -> asSet()
10       in (opBT - op) -> notEmpty()
```

On line 5 we select the instances of operations op_1 and op_2, operations defined in the business task. We have to check that the union of the operations allowed by the conflicting roles is a proper subset of the business task operations. This is

equivalent to stating that the difference between the two sets is not empty (line 10). This check is done if both roles are active (line 4).

The History-based DSoD policy combines both the Object-based one and the Operational-based one. Differently from these two, History-based DSoD allows a user to activate two conflicting roles at the same time, as long as the user does not perform all the operations on the same object. This can be specified as an invariant of the `Session` class:

```
1  context Session inv DSoDHis:
2  let conflictingRoles : Set(Role) = self.activeRoles ->select (r: Role
3                                   | r.idRole = 'r1' or r.idRole ='r2')->asSet()
4  in if conflictingRoles -> size()>1 then
5       let roleslog: Set (History) = conflictingRoles -> collect(logRole)
6                                   -> select (log:History| log.user = self.user)
7                                   -> asSet(),
8          objects: Set(Object) = roleslog -> collect (object)-> asSet(),
9          opBT : Set(Operation) = Operation.allInstances()->select(o: Operation
10                                  | o.idOperation='op1' or o.idOperation='op2')
11                                  -> asSet()
12      in objects -> forAll (o: Object
13                    | let logObject: Set (History) = roleslog -> select(l:History
14                                  | l.object=o and opBT->includes(l.operation)),
15                        opObjBT: Set(Operation) = logObject-> collect(operation)
16                                 -> asSet()
17                    in (opBT-opObjBT) -> notEmpty())
18    endif
```

In the OCL expression above, we first select the list of conflicting roles r_1 and r_2 active in the current session (lines 3 and 4). Then, we check if the user is activating more than one conflicting role (line 4). We retrieve the list of logs performed by the current user while activating a conflicting role (lines 5–7). We compute the list `objects` of objects from the logs performed by the current user. We also select the list of operations (op_1 and op_2) defined in the business task (lines 9–11). For each object `o` in the list `objects`, we compute the list `opObjBT` of operations performed by the user (either under role r_1 or r_2) on the object `o` (line 11). The condition checks whether the list `opLog` of operations defined in a business task is a proper subset of the list `opObjBT`.

3.2.6 Binding of Duty Policy (BoD)

Binding of duty policies define a correlation between a set of permissions. A role-based BoD policy requires that bounded operations must be executed by the same role. This policy can be specified in OCL as an invariant of the `Role` class:

```
1  context Role inv RoleBoD:
2  let boundedPermissions: Set (Permission)= self.permissions
3                                       ->select(p:Permission|p.idPermission='p1'
4                                       or p.idPermission = 'p2')
```

```
5 in if boundedPermissions-> size() = 2 then
6     let boundedroles: Set (Role)= Role.allInstances()-> excluding (self)
7                                     -> select (r:Role | r.permissions
8                                     -> includesAll(boundedPermissions)),
9          roleLog: Set (History)= self.logBOCurrentProcessInstance()
10     in if not(roleLog -> isEmpty()) then
11             boundedroles->forAll(r:Role
12                           |r.logBOCurrentProcessInstance()->isEmpty())
13        endif
14   endif
```

In the OCL expression above, we first retrieve the set of bounded permissions (lines 2–4). Then, the condition checks whether the current role is a bounded role by checking its assignment to all bounded permissions (line 5). We retrieve the list of roles assigned to the bounded permissions (lines 6–8). Then, we determine which bounded operations have been performed by the current role in the current process instance[1] by calling the operation `logBOCurrentProcessInstance()` of the `Role` class (line 9). If the current role has performed some bounded operations (line 10), then the condition is satisfied if none of the bounded operations have been previously performed by *any* other role different from the current one (line 12). In other words, if a user with the current role has already performed a bounded operation, any other user with the *same role* is allowed to perform another bounded operation in the current process instance.

The subject-based BoD policy requires that bounded operations must be executed by the same subject (user and role). This policy can be expressed in OCL as:

```
1  context Role inv SubjectBoD:
2  let boundedPermissions: Set (Permission)= self.permissions
3                                     ->select(p:Permission|p.idPermission='p1'
4                                     or p.idPermission = 'p2')
5  in if boundedPermissions-> size() = 2 then
6      let boundedroles: Set (Role)= Role.allInstances()-> excluding (self)
7                                     -> select (r:Role | r.permissions
8                                     -> includesAll(boundedPermissions)),
9          subjectLog: Set (History)= self.logBOCurrentProcessInstance()
10                                   select (a: History | a.user= self.user)
11     in if not(subjectLog -> isEmpty()) then
12             boundedroles -> forAll(r:Role
13                           |r.logBOCurrentProcessInstance()->isEmpty())
14       endif
15   endif
```

This OCL constraint is similar to the one defined for role-based BoD. However, the log `subjectLog` corresponds to the bounded operations that have been

[1]We recall that BoD policies are usually defined in the context of process-based workflow systems.

performed by a given user `u` with a given role `r` in the current process instance.

3.2.7 Role Delegation and Revocation policies

3.2.7.1 Role Delegation Policy

A delegation is characterized by a delegated role, a delegator, a delegate and their corresponding roles. In a multi-step delegation, a user is allowed to delegate a delegated role according to a maximum delegation depth (hereafter called *maxDepth*). This type of delegation can be specified in OCL as an invariant of the `Delegation` class:

```
1 context Delegation inv MultiStepDelegation:
2 self.getAbsoluteDelegationPath() -> size() <= self.maxDepth
```

In the OCL expression shown above, the operation `getAbsoluteDelegationPath` returns the list of delegation steps starting from the original (non-delegated role). The size of this list is then compared with the attribute `maxDepth`. The single-step delegation policy can be defined as a multi-step delegation with a maximum delegation depth equal to 1.

A delegation can be total or partial depending on the number of permissions being delegated. A total delegation delegates all the permissions belonging to a certain role; it can be specified in OCL as an invariant of the `Delegation` class:

```
1 context Delegation inv TotalDelegation:
2 self.isTotal implies
3 self.delegatedPermissions = self.delegatedRole.permissions
```

This expression states that if the delegation is total (represented by the attribute `isTotal`) then the list of delegated permissions (derived from the association `delegatedPermissions`) should be equal to the list of permissions associated to the delegated role.

A partial delegation (characterized by the attribute `isTotal` being false) is defined in a similar way:

```
1 context delegation inv PartialDelegation:
2 not (self.isTotal) implies
3 (self.delegatedRole.permissions - self.delegatedPermissions) -> notEmpty()
```

A delegation can be either of type "grant" or "transfer". While a "grant" type delegation does not affect the permissions of the delegator, in case of a delegation of type "transfer", the delegator cannot use the delegated role after the delegation. A delegation of type "transfer" can be either *strong* or *weak*. In case of *strong transfer*, in addition to the delegated role, the delegator is no longer assigned to any of its juniors. This policy can be expressed in OCL as an invariant of the `Delegation` class:

```
1 context Delegation inv StrongTransfer:
2 let roles: Set(Role) = self.delegatedRole.getAlljuniors() -> including (self.
      delegatedRole)
3 in self.isTransfer = delegationType::strong implies
4 self.delegatorUser.roles -> excludesAll(roles)
5                             and self.delegateUser.delegatedRoles ->
6                             includes(self.delegatedRole)
```

In the OCL expression shown above, we first retrieve the list of roles from the `Delegation` object, including the delegated role and its juniors (line 2). The operation `getAlljuniors` returns the juniors of the delegated role, walking through the transitive closure of the hierarchy relation. For instance, the operation `getAlljuniors` applied to role r_1 in figure 2.2 returns the list of roles: r_2, r_3, r_4 and r_5. The OCL expression at line 3 states that if the delegation is of type *strong transfer* (represented by the expression `self.isTransfer = delegationType::strong`) then the list of roles assigned to the delegator (derived from the `delegatorUser.roles` association) should include neither the delegated role nor any of its juniors. Besides, the list of roles delegated to the delegate should include the delegated role; these roles are derived by navigating the `delegateUser.delegatedRoles` association.

A delegation of type *weak transfer* can be either *static* or *dynamic*. In case of *static weak transfer*, the delegator keeps using a subrole r of the delegated role only if she is a member of another senior of role r. This policy can be expressed in OCL as an invariant of the `Delegation` class:

```
1  context Delegation inv StaticWeakTransfer:
2  let acquiredRoles: Set(Role) = self.delegatorUser.roles
3                  -> union(self.delegatorUser.delegatedRoles),
4      allowedRoles: Set(Role) = self.delegatedRole.getAllJuniors()
5                  -> select (r : Role | (r.seniors -> excluding(delegatedRole))
6                  -> exists (r1 : Role | acquiredRoles ->includes(r1))),
7      roles: Set(Role) = (self.delegatedRole.getAllJuniors()
8                  -> including(self.delegatedRole)) - allowedRoles
9  in self.isTransfer = delegationType:: weakStatic implies
10         self.delegatorUser.roles -> excludesAll(roles)
11         and self.delegatorUser.roles -> includesAll(allowedRoles)
12         and self.delegateUser.delegatedRoles -> includes(self.delegatedRole)
```

In the OCL expression shown above, we first retrieve the list (`acquiredRoles`) of roles available to the delegator, defined as the union of the roles assigned to and delegated to the delegator (lines 2 and 3). Then, we select among the subroles of the delegated role, the roles (`allowedRoles`) that the delegator is allowed to acquire. We check if one of the juniors of the delegated role has another senior in the list `acquiredRoles` (lines 4–6). We compute the list of roles that the delegator cannot use after the transfer, keeping into account the `allowedRoles` (lines 7 and 8). If the transfer is of type *static weak* (condition checked at line 9) the list of roles assigned to the delegator should include neither the delegated role nor any of

its juniors (except for those allowed by the hierarchy relation). Finally, the list of roles delegated to the delegate should include the delegated role (line 12).

The policy for the delegation of type *dynamic weak transfer* has a similar structure:

```
1  context Delegation inv DynamicWeakTransfer:
2  let  acquiredRoles : Set(Role) = self.delegatorUser.roles
3                                -> union(self.delegatorUser.delegatedRoles),
4       allowedRoles : Set(Role) = self.delegatedRole.getAllJuniors()
5                                -> select (r : Role | (r.seniors
6                                -> excluding(delegatedRole)) -> exists(r1 : Role
7                                | self.delegatorUser.sessions -> exists(s:Session
8                                | s.activeRoles -> includes(r1)))),
9       roles : Set(Role) = (self.delegatedRole.getAllJuniors()
10                               -> including(self.delegatedRole)) - allowedRoles
11 in  self.isTransfer = delegationType:: weakDynamic implies
12         self.delegatorUser.roles -> excludesAll(roles)
13         and self.delegatorUser.roles -> includesAll(allowedRoles)
14         and self.delegateUser.delegatedRoles -> includes(self.delegatedRole)
```

Notice that in this case the list `allowedRoles` (lines 4–8) includes the subroles having an active senior in the delegator session.

3.2.7.2 Role Revocation Policy

A delegation is revoked by setting the attribute `isRevoked` to true. The delegation is revoked automatically when its duration expires; this policy can be specified as an invariant of the `Delegation` class as follows:

```
context Delegation inv AutomaticRevocation:
let u: RBACUtility = RBACUtility.allInstances() -> any(true),
in  u.currentDate >= self.endDate implies self.isRevoked
```

A revocation is called grant-dependent if only the delegator is allowed to revoke the delegation. On the other hand, a grant-independent revocation allows not only the delegator but also any *original* user to revoke the delegation. This policy can be expressed in OCL as an invariant of the `Delegation` class:

```
1  context Delegation inv RevacationDependency:
2  if self.isRevoked then
3    if self.delegatedRole.isDependent then
4      self.revokingUser = self.delegatorUser
5    else
6      self.revokingUser = self.delegatorUser or self.delegatedRole.users
7                          ->includes(self.revokingUser)
8    endif
9  endif
```

In the OCL expression above, we first check if the revocation is grant-dependent, by checking the attribute `isDependent` of the association `delegatedRole` (line 3).

If this is the case, only the delegator is allowed to revoke the delegation (line 4). Otherwise, the delegation can be revoked either by the delegator or by any user *assigned* to the delegated role (lines 6 and 7).

A revocation can be classified as strong or weak according to its dominance. A strong revocation removes from a user not only the delegated role but also its junior roles. A strong revocation can be expressed in OCL as an invariant of the `Role` class:

```
1 context Delegation inv StrongRevocation:
2 if self.isRevoked then
3    self.delegatedRole.isStrong implies
4    self.delegateUser.delegatedRoles
5    -> excludesAll(self.delegatedRole.getAllJuniors())
6 endif
```

In the constraint above, the implication states that if the revocation is strong (as determined by the attribute `isStrong`), the set of roles received by the delegation (`delegateUser.delegatedRoles`) should not include juniors of the delegated role.

A revocation can be classified as cascading or non-cascading according to its propagation. A cascading revocation removes all delegations resulting from a multi-step delegation. It can be specified in OCL as an invariant of the `Delegation` class:

```
1 context Delegation inv CascadingRevocation:
2 if self.isRevoked then
3 self.delegatedRole.isCascading implies
4 self.delegatedDelegation -> forAll (d: Delegation | d.isRevoked = true)
5 endif
```

In the constraint above, the implication states that in case of a cascading revocation (as determined by the attribute `isCascading`), all the delegated delegations should be revoked; this list of delegations is derived from the association `delegatedDelegation`.

3.2.8 Context-based Policies

While the non-contextual policies described above are specified as OCL constraints, contextual policies are expressed on the model level by adding additional instances of one or more UML classes (see section 3.1.1). As explained in Section 3.1.1, the context in which a `Role` (or a `Permission`) can be enabled or assigned (as prescribed by a contextual policy), is captured on the UML model. RBAC contextual policies can then be checked by verifying OCL constraints on the GemRBAC+CTX model. In this way, an access decision (e.g., allowing a user to activate a role) can be performed by checking whether an instance of the GemRBAC+CTX satisfies the OCL constraints associated with it. In the rest of this section we provide several templates that can be used to formalize contextual RBAC policies for role enabling

or assignment as OCL constraints on the GemRBAC+CTX model. These RBAC policies are based on real policies defined in our industrial case study.

In the definition of the OCL constraints, we make some working assumptions. We assume that each snapshot contains the time at which it was taken (modeled as an association between classes `RBACUtility` and `TimePoint`) and the current day of week (modeled as an association between classes `RBACUtility` and `DayOfWeek`). This assumption can be guaranteed by applying a timestamp to each snapshot. We also assume that the position of the user is always known, by means of a GPS; this is very reasonable nowadays. Lastly, we assume that policies are not conflicting with each other; e.g., we avoid the case of having two policies, one enabling (assigning) and another one disabling (unassigning) the same role/permision in the same context.

The Ecore version of the GemRBAC+CTX model, the OCL constraints defined in this section, and model instances that violate/satisfy them are available at https://github.com/AmeniBF/GemRBAC-CTX-model.git.

3.2.8.1 Time-based Policy

A policy on role enabling with an absolute time expression restricts the time interval at which a role can be enabled, as in "role r_1 is enabled from January 21, 2014 to April 25, 2015". This policy can be checked by verifying the following OCL invariant of class `Session`:

```
1  context Session inv AbsoluteBTIRoleEnab:
2  let u: RBACUtility = RBACUtility.allInstances() -> any(true),
3      r: Role = Role.allInstances()->select(r:Role |r.idRole = 'r1') ->any(true),
4      temporalContext: Set(TimeExpression) = r.roleContextEnabling.timeexpression
5                                           -> flatten() -> asSet(),
6      timeE: Set(AbsoluteTE) = temporalContext.absolute -> flatten() -> asSet(),
7      timeI: Set(AbsoluteTE) = timeE -> select(e | e.oclIsTypeOf(TimeInterval)
8                               and e.oclAsType(TimeInterval).end->notEmpty())
9  in if timeI.oclAsType(TimeInterval)
10       -> exists(i| u.getCurrentTime().isContained(i)) then
11            self.enabledRoles -> includes(r) or self.activeRoles -> includes(r)
12    endif
```

In this OCL expression, we first select the instance corresponding to role r_1 (line 3). Then, we retrieve the list `temporalContext` of time expressions in which the role should be enabled (lines 4–5) and compute, over the elements of this list, the list `timeE` of absolute expressions assigned to them (line 6). In this example, since there is only one `TemporalExpression` object containing one `AbsoluteTE` object, the `timeE` list will include only one instance of `TimeInterval` whose start and end `TimePoints` corresponds to "January 21, 2014" and "April 25, 2015". Since the enabling temporal context in the policy is expressed as a bounded time interval, we have to select, among the elements of `timeE`, the list `timeI` of expressions in the form of a time interval (lines 7–8) with a bounded end point; this last condition

is checked with the expression at line 8. Afterwards, we check if the time when the snapshot was taken—obtained by calling the operation `getCurrentTime` of class `RBACUtiliy`—is contained in one of the time intervals in list `timeI` (lines 9–10). If this is the case, we check whether role r_1 is in the list of enabled or active role of the current session (line 11).

A policy on permission assignment with a relative time expression restricts the time at which a permission can be assigned to a role. As explained in Section 3.1.1, we support different forms of relative time expression. For the purpose of illustration, we consider a relative time expression structured as a `DayOfWeek` (or a `RelativeTimeInterval` with bounds of type `DayOfWeek`), which, subsequently, can overlay an `Hour` (or a `RelativeTimeInterval` with bounds of type `Hour`). An example of a policy with a relative time expression of this form is "assign role r_1 to user u_1 *only* from Wednesday to Friday, from 10:00 to 14:00". Such a policy can be checked by verifying the following OCL invariant of class `Permission`:

```
1  context Permission inv DayOfWeekHourPermAssign:
2  if self.idPermission = 'p1' then
3   let u: RBACUtility = RBACUtility.allInstances()-> any(true),
4       day: RelativeTimePoint = u.getDayOfWeek(),
5       r: Role = Role.allInstances()->select(r: Role| r.idRole= 'r1')->any(true),
6       temporalContext: Set(TimeExpression) = self.permissionContextAssignment.
7                                       timeexpression -> flatten() -> asSet(),
8       timeE: Set (ComposedRelativeTE) = temporalContext.relative.
9                                       oclAsType(ComposedRelativeTE)
10                                      -> flatten() -> asSet(),
11      days: Set (ComposedRelativeTE) = timeE ->select(t|
12                                    (t.oclIsTypeOf(RelativeTimeInterval) and
13                                    t.oclAsType(RelativeTimeInterval).start.
14                                    oclIsTypeOf(DayOfWeek) and day.isContained
15                                    (t.oclAsType(RelativeTimeInterval)))
16                                    or (t.oclIsTypeOf(DayOfWeek) and
17                                    day.equalTo(t.oclAsType(DayOfWeek))))
18   in if days -> exists (t| t.checkHours(u)) then
19               self.roles -> includes (r)
20        else
21              self.roles -> excludes (r)
22     endif
23 endif
```

In this OCL expression if the current permission is p_1, we select the day corresponding to the day of week at which the snapshot was taken, by calling the `getDayOfWeek` operation of the class `RBACUtility` (lines 3–4). Then, we select the instance corresponding to role r_1 (line 5). We retrieve the list of time expressions `temporalContext` in which the permission should be assigned to role r_1 (lines 6–7) and compute, over the elements of this list, the list `timeE` of relative time expressions assigned to them (lines 8–10). Based on the type of policy described above, we have to select, among the elements of `timeE`, the list `days` of rel-

ative time expressions having a `ComposedRelativeTE` of type `DayOfWeek` or of type `RelativeTimeInterval` with bounds of type `DayOfWeek` (lines 11–17). While selecting the time expressions in this list, we check whether the day at which the snapshot was taken is contained in the selected `TimeExpression`. To do so, we check separately for the `DayOfWeek`, by calling operation `equalTo` of class `RelativeTimePoint` (line 17), and for the `RelativeTimeInterval` by calling operation `isContained` of class `RelativeTimePoint` (line 14). In this specific example, list `days` will include a `TimeExpression` that contains two `ComposedRelativeTE`. These objects are: a `RelativeTimeInterval` (whose start and end `RelativeTimePoint`s correspond to "Wednesday" and "Friday"); and a `RelativeTimeInterval` (whose start and end `RelativeTimePoint`s correspond to "10:00" and "14:00"). We remark that the first object `overlay`s the second. We check whether the time at which the snapshot was taken is contained in one of the `TimeExpression`s in `days`. To do so, we check the hours overlaid by the day(s) of the week by calling operation `checkHours` of class `ComposedRelativeTE` (line 18). If the check succeeds, we require role r_1 to belong to the list of roles of permission p_1 (line 19). Otherwise, we require the role not to be in this list (line 21). For the sake of simplicity, in the remaining of this section we focus only on the specification of policies at the role level.

A policy on role assignment with a relative time expression containing an index of a specific `DayOfWeek` restricts the day in which a given user can acquire a given role, as in "*assign role r_1 to user u_1 on the 2nd Monday of June*". This policy can be checked by verifying an OCL invariant of class `Role`:

```
1  context Role inv indexRoleAssign:
2  let u: RBACUtility = RBACUtility.allInstances()-> any(true),
3      month: ecore::EInt = u.getCurrentTime().month,
4      day: RelativeTimePoint =  u.getDayOfWeek(),
5      temporalContext: Set(TimeExpression) =self.roleContextAssignment
6                                       .timeexpression -> flatten() -> asSet(),
7      timeE: Set(ComposedRelativeTE) = temporalContext. relative
8                                       .oclAsType(ComposedRelativeTE) -> flatten()
9                                       -> asSet()
10 in self.idRole = 'r1' and self.users -> select(u| u.idUser ='u1') -> notEmpty()
11              implies timeE -> exists(t | t.oclIsTypeOf(MonthOfYear)
12              and t.oclAsType(MonthOfYear).month  = month
13              and t.checkDayIndex(u))
```

In this invariant we first select the month and day of week at which the snapshot was taken by calling the `getCurrentTime` and `getDayOfWeek` operations of class `RBACUtility` (lines 3–4). Then, we retrieve the list `temporalContext` of time expressions in which the role should be assigned (lines 5–6) and compute, over the elements of this list, the list `timeE` of relative time expressions assigned to them (lines 7–9). The implication at lines 10–13 states that if the current role is r_1 and user u_1 is a member of this role, the temporal context for role assignment

should match the current DayOfWeek; this condition is verified by calling operation checkDayIndex of class ComposedRelativeTE.

A policy on role assignment with time expression containing a periodic expression restricts the time at which a role can be assigned to a user as in "user u_1 acquires role r_1 every 5 days starting from July 10, 2014 at 16:00". This policy can be checked in OCL as an invariant of class Role:

```
1  context Role inv periodicUnboundTIRoleAssign:
2  let u: RBACUtility = RBACUtility.allInstances()-> any(true),
3      u1: User = User.allInstances()->select(u: User| u.idUser= 'u1')->any(true),
4      temporalContext: Set(TimeExpression) = self.roleContextAssignment
5                                  .timeexpression -> flatten() -> asSet(),
6      timeE: Set(AbsoluteTE) = temporalContext.absolute -> flatten()-> asSet(),
7      absoluteE: Set(AbsoluteTE) = timeE -> select (t | t.absolute
8                                  .oclAsType(TimeInterval) -> exists(a
9                                  | a.start.equalTo(u.getCurrentTime())
10                                 or a.start.isBefore(u.getCurrentTime()))),
11     periodicE: Set(PeriodicTime)= absoluteE.relative.oclAsType(PeriodicTime)
12                                 -> flatten() -> asSet()
13 in self.idRole= 'r1' and self.users->includes(u1) implies periodicE.nextStart
14    ->select( a | a.equalTo(u.getCurrentTime()))->notEmpty()
```

In this invariant, we first select the instance corresponding to user u_1 (line 3). We retrieve the list temporalContext of time expressions in which the role should be assigned to user u_1 (lines 4–5) and compute, over the elements of this list, the list timeE of absolute time expressions assigned to them (line 6). In this example, the list timeE will include a TimeExpression with an unbounded TimeInterval whose start TimePoint corresponds to "July 10, 2014 at 16:00". Then we select among the element of list timeE, the list (absoluteE) of expressions having an absolute TimeInterval that contains the TimePoint at which the snapshot was taken (lines 7–10). We check this containment by comparing the time at which the snapshot was taken with the start TimePoint of the unbounded TimeInterval. Afterwards, we retrieve the list of PeriodicTime objects in each expression in list absoluteE (lines 11–12). The implication at lines 13–14 states that if the current role is r_1, and user u_1 is member of this role, the time at which the snapshot was taken should match the starting time (derived from the nextStart association) of the next period.

A policy on role enabling with a duration associated with an absolute time expression restricts the activation of a role up to a specific duration, as in "enable all roles on April 23, 2015 from 8:00 to 18:00; each role can be active for 3 hours cumulatively". This policy can be checked in OCL as an invariant of class Session:

```
1  context Session inv DurationAbsoluteBTIRoleEnab:
2  let u : RBACUtility = RBACUtility.allInstances()-> any(true),
3     rolesA: Set(Role) = self.enabledRoles-> select(r| r.getCurrentAbsoluteTE(u)
4          -> notEmpty() and r.getCurrentAbsoluteTE(u).hasDuration())
5  in rolesA -> forAll(r | r.getCurrentAbsoluteTE(u).duration.greaterThan
```

```
6            (u.getCumulativeActiveDuration(r,self.user, r.getCurrentAbsoluteT(u)
7            .duration.timeUnit)))
```

In this OCL constraint, we select a subset (list `rolesA`) of the roles enabled in the current session (lines 3–4). This subset includes the roles whose temporal context for enabling contains an absolute time expression that matches the time at which the snapshot was taken (checked by calling the operation `getCurrentAbsoluteTE` of the class `Role`). For each role in `rolesA`, this absolute time expression should be associated with a duration (checked by calling the operation `hasDuration` of the class `AbsoluteTE`). Then, we check whether the duration of each role in the list is less than the duration specified in its temporal context for enabling (lines 5–7). We assume that the duration of the activation of each role for each user is recorded in a database and made available through the operation `getCumulativeActiveDuration` of class `RBACUtility`.

3.2.8.2 Location-based Policy

A policy on role assignment with a physical location forbids the role assignment when the user is not located in a physical location belonging to the role spatial context for assignment, as in "role r_1 is assigned to user u_1 *only* if the latter is in location loc_1". We assume that loc_1 is of type `PhysicalLocation`. This policy can be checked in OCL as an invariant of class `Role`:

```
1  context Role inv physicalLocationRoleAssign:
2  let u1: User= User.allInstances()->select(u: User| u.idUser = 'u1')->any(true),
3      u1Loc: Set(PhysicalLocation) = u1.userLocation.location -> select(l
4                                    |l.oclIsTypeOf(PhysicalLocation))->any(true),
5      spatialContext: Set(Location) = self.roleContextAssignment.location
6                                    -> flatten() -> asSet(),
7      locPh: Set(Location) = spatialContext -> select (loc
8                                    | loc.oclIsTypeOfe(PhysicalLocation))
9  in if self.idRole = 'r1' and loc -> exists(l| l.contains(u1Loc)) then
10           self.users -> includes(u1)
11      else
12           self.users -> excludes(u1)
13    endif
```

In this OCL expression, we first select the instance corresponding to user u_1 (line 2) and the physical location associated to user u_1 (lines 3–4). As mentioned in section 3.1.1, we assume the user to have only one physical location. Then, we retrieve the list `spatialContext` of locations at which the role should be assigned to user u_1 (lines 5–6) and compute, over the elements of this list, the list `locPh` of physical locations (line 8). We check if the current role is r_1 and if a physical location in list `locPh` matches the user's location, by calling the operation `contains` of class `Location`. If this is the case, the list of roles assigned to user u_1 should contain role r_1 (lines 9–10). If it is not the case, the role should not be included in this list (line 12).

A policy on role enabling with a logical location is checked in a similar way by replacing the instances of `PhysicalLocation` with instances of `LogicalLocation`:

```
1  context Session inv logicalLocationRoleEnabling:
2  let u1Loc: Set(LogicalLocation) = self.userLocation.location -> select(l
3                                    |l.oclIsTypeOf(LogicalLocation))->any(true),
4      r1: Role = Role.allInstances() -> select(r |r.idRole = 'r1')->any(true),
5      spatialContext: Set(Location) = r1.roleContextAssignment.location
6                                    -> flatten() -> asSet(),
7      loc: Set(Location) = spatialContext -> select (loc
8                                    | loc.oclIsTypeOfe(LogicalLocation))
9  in if  loc -> exists(l| l.contains(u1Loc)) then
10           self.enabledRoles -> union(self.activeRoles) -> includes(r1)
11      else
12            self.enabledRoles -> union(self.activeRoles) -> excludes(r1)
```

A policy on role assignment with a relative location forbids the role assignment when the user is not located in a relative location belonging to the role spatial context for assignment, as in "enable role r_1 *only* within 3 meters outside location loc_1". Location loc_1 can be either of type `PhysicalLocation` or `LogicalLocation`. This policy is checked in OCL as an invariant of class `Session`:

```
1  context Session inv relativeLocationRoleEnabling:
2  let r1: Role = Role.allInstances()->select(r: Role| r.idRole= 'r1')->any(true),
3      spatialContext: Set(RBACContext) = self.roleContextEnabling.location
4                                       -> flatten() -> asSet(),
5      loc: Set(Location) = spatialContext -> select(l |l.relativelocation
6                                       -> notEmpty()) -> flatten() -> asSet(),
7      relativeLoc: Set(Location) = loc -> collect(l| l.computeRelative
8                                       (l.relativelocation)) ->flatten()->asSet()
9  in if relativeLoc -> exists(l| self.user.userLocation.location
10      -> exists(pos| l.contains(pos))) then
11        self.enabledRoles -> includes(r1) or self.activeRoles -> includes(r1)
12      else
13        self.enabledRoles -> excludes(r1)  and  self.activeRoles -> excludes(r1)
14    endif
```

In this OCL invariant, we first select the instance corresponding to role r_1 (line 2). We retrieve list `spatialContext` of locations at which the role should be enabled (lines 3–4) and compute the list `loc` of all locations associated with a relative one (lines 5–6). For each location in list `loc`, we compute in `relativeLoc` the location resulting from the call to operation `computeRelative` of class `Location` (lines 7–8). This operation takes in input `RelativeLocation` and is applied to a `PhysicalLocation` or `LogicalLocation`, hereafter called *base location*. It returns the location resulting from the application to the base location of the parameters (distance and direction) of the relative location. The resulting location is always of type `PhysicalLocation`. We check if any of locations in `relativeLoc` matches the user's position (lines 9–10). If it is the case, the role r_1 should be enabled or active

(line 11). Otherwise, the role should be disabled and deactivated in the current session (line 13).

In this section we have shown how the access decision for spatial and temporal RBAC policies defined according to the GemRBAC+CTX model can be reduced to the verification of OCL constraints of an instance of the GemRBAC+CTX model. For simplicity reasons, we have considered temporal and spatial policies in isolation. Nevertheless, we support also *composite context-based policies*, i.e., policies that contain both a temporal and a spatial context. These policies can be checked in OCL by a logical conjunction of the individual OCL constraints corresponding to the composite spatial and temporal policies.

3.3 Application to an Industrial Case Study

In the rest of this section we report on the application of our approach based on GemRBAC+CTX for the modeling of a real application and of its RBAC policies. More specifically, we show how the GemRBAC+CTX model can be instantiated to represent some states corresponding to run-time changes of the system, and how we can define RBAC policies using the OCL templates proposed in the previous section.

Application Scenario

This application scenario has been developed by our partner HITEC Luxembourg. The application is an integrated communication solution, to be used in case of an emergency scenario (e.g., a natural large-scale disaster or a civil war situation), to ease the process of assisting refugees and/or casualties in such situations.

The application allows different (humanitarian) organizations to participate to various missions. During a mission, a user of the application can send alerts to request treatment services for injured people. Each user belongs to at least one organization and can be assigned to one or many missions. Each mission is characterized by a name, a geofence, and a set of policies. The geofence is a geographic boundary that defines where users assigned to a certain role should be situated during the mission period.

The membership of a user to an organization or to a mission does not automatically grant the access to the corresponding resources. Following the principles of RBAC, the access is allowed (or denied) according to the user's role. In the rest of this section, we present a small excerpt of the application and consider only a subset of the actual RBAC entities. Moreover, the description has been sanitized for confidentiality reasons. We consider the mission *Philippine*, which is situated within the geofence labeled *AbayZone1*. We refer to the following RBAC entities:

- **users** = {Joe, Kim, James, Alice, Mallory};

- **roles** = {missionAdmin, missionAgency, missionMember, securityOfficer, trainee};
- **permissions** = {manageRefugee, manageDevice, sendAlert, noBandWithLimit};
- **operations** = {create, read, update, delete};
- **objects** = {refugee, device, alert};
- **spatial context** = {AbayZone1};
- **temporal context** = {freeTime}.

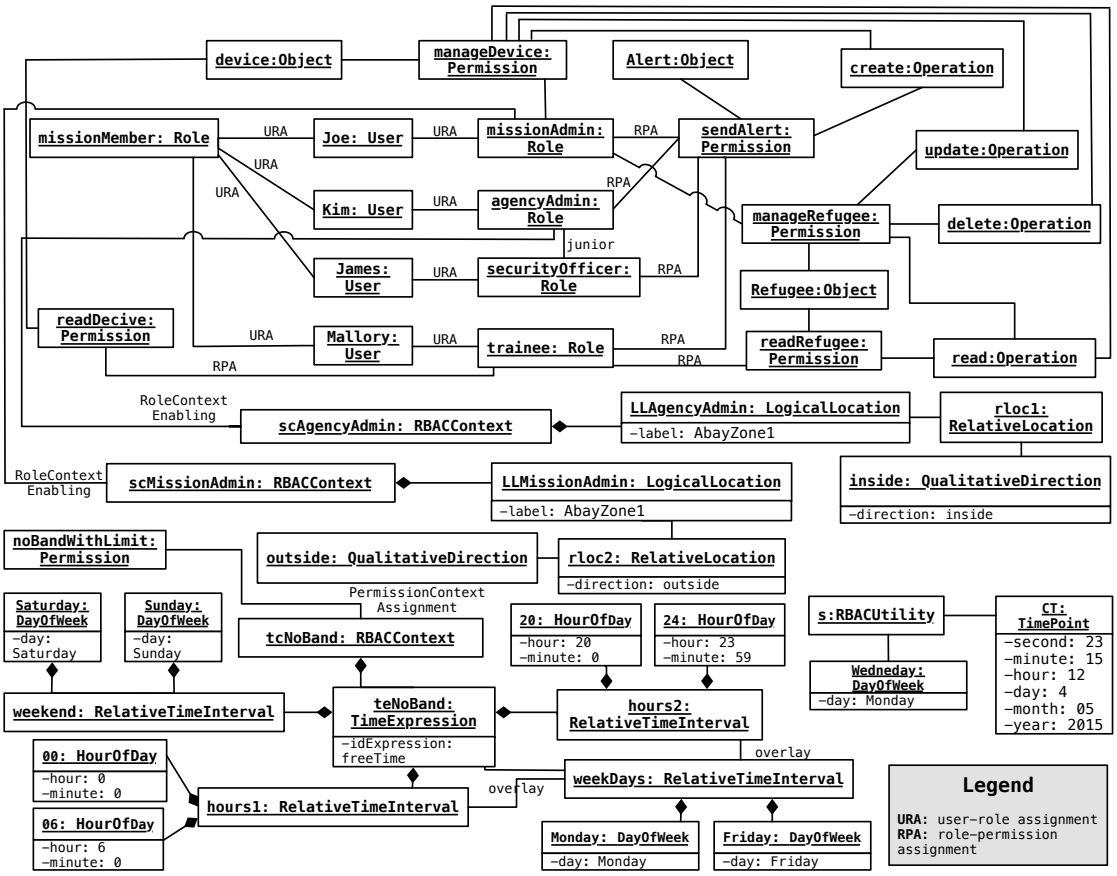


Figure 3.4: Initial system state

Figure 3.4 depicts an instance of the GemRBAC+CTX model representing the initial system state for the mission *Philippine*. Each role is assigned to a set of permissions. A permission is an abstraction of a set objects and a set of operations. Notice that we only consider one object assigned to each permission for the sake of readability. For instance, the permission *manageDevice* corresponds to the execution of the the operations *create*, *read*, *update*, and *delete* on the object *device*. All roles, except role *missionMember* are given the permission *sendAlert*,

to send an alert. The role *missionAdmin* can manage all the resources; permissions of the form *manage** include all the operations defined in the system. Role *trainee* is given the permissions of the form *read** for the objects of type *device* and *refugee*. Role *securityOfficer* is a subrole of role *agencyAdmin*; hence, it inherits all the permission of its senior. Roles are assigned to users as follows: all users are assigned to role *missionMember*, *Joe* is assigned to role *missionAdmin*, *Kim* is assigned to role *agencyAdmin*, *James* is assigned to role *securityOfficer*, and *Mallory* is assigned to role *trainee*. In addition to the user-role and role-permission assignments shown in figure 3.4, some additional policies can further restrict the user access. The following policies are defined for the system:

PL1: *permission* noBandwidthLimit *is assigned to role* missionMember *only during* freeTime *that ranges from 00.00 to 06.00 and from 20.00 to 23.59 during weekdays and all-day during the weekend.* This policy is typically used to ensure a fair use of the available bandwidth.

PL2: *role* agencyAdmin *is enabled only outside Zone1.* This policy is typically used to ensure that administrative tasks are performed, for security reasons, outside the area of the mission.

PL3: *role* missionAdmin *is enabled only inside Zone1.* This policy is typically used for guaranteeing that mission management is done locally.

PL4: *role* trainee *is enabled only if role* securityOfficer *is active.* This policy is typically used to deal with the supervision of the *trainee* activities.

PL5: *role* securityOfficer *can be delegated as a strong transfer to any user assigned to role* missionAdmin. We recall that right after a delegation of type strong transfer, the delegator is no longer assigned to the delegated role and all its juniors.

Since no user is connected to the system, no session is created for the initial system state (figure 3.4). According to policy **PL1**, the temporal context for assignment of permission `noBandwidthLimit` is `freeTime`. It is modeled as a `TimeExpression` composed of four `RelativeTimeIntervals`. Interval `weekend` has a start (`Saturday`) and end (`Sunday`) `RelativeTimePoint` of type `DayOfWeek`. Interval `weekDays` has a start (`Monday`) and end (`Friday`) `RelativeTimePoint` of type `DayOfWeek`. Interval `weekDays` overlays `hours1` and `hours2`: these intervals are of type `HourOfDay`. Let us consider the case in which one wants to check policy **PL1** on this instance. This policy can be checked using the OCL invariant `DayOfWeekHourPermAssign` introduced on page 36. The `if` condition at line 18 is `false` because the time at which the snapshot was taken is not included in the temporal context for enabling permission *noBandwidthLimit*. Hence, we follow the `else` branch, calling operation `excludes` at line 21. Since role *missionMember* is not assigned to permission *noBandwidthLimit*, policy **PL1** is not violated. According to policy **PL2**, the spatial context for enabling role `AgencyAdmin` is modeled as a `LogicalLocation` (*LLAgencyAdmin*) associated with a `RelativeLocation` (*rloc1*) that contains a `QualitativeDirection` (`inside`). The spatial context for

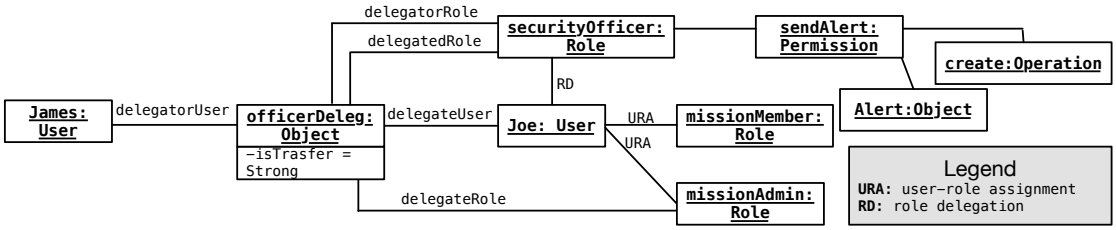


Figure 3.5: A portion of the system state after the delegation of role *securityOfficer*

enabling role `MissionAdmin`, indicated in policy **PL3**, is modeled in a similar way (see *LLMissionAdmin*, *rloc2*). The snapshot in Figure 3.4 includes an instance of `RBACUtility` that captures the `TimePoint` and the `DayWeek` at which it was taken (*Monday, May 4, 2015 at 12:15:23*).

In the rest of this section, we consider three run-time changes of the system. Each change is represented by a snapshot that captures the system state. On each snapshot we check whether the policies defined above are satisfied or violated.

First, let us consider the case in which user *James* gets a new job. Since he cannot participate to the mission anymore, he has to delegate, as a transfer, his role *securityOfficer* to *Joe*. To perform the delegation, he has to first activate the role to delegate. Figure 3.5 depicts, as an object diagram, an instance of the GEM-RBAC+CTX model that corresponds to a portion of the system state right after this delegation. Since the delegation of *James* to *Joe* is of type transfer, *James* is no longer member of role *securityOfficer*; the latter is assigned to *Joe* via a `role delegation` association. Notice that the object `officerDeleg` of type `Delegation`, having the attribute `isTransfer` set to `strong`, has been created. It keeps track of the delegated role (*securityOfficer*), the delegator user (*James*), the role of the delegator at the time of the delegation (*securityOfficer*), the delegated role (*securityOfficer*), the delegate user (*Joe*), and the role of the delegate (*missionAdmin*). Policy **PL5** states that right after a delegation of type strong transfer, the delegator is no longer assigned to the delegated role and all its juniors. This policy can be expressed using the OCL invariant `StrongTransfer` provided on page 32. As *James* is no longer a member of role *securityOfficer*, one can check that constraint policy **PL5** is satisfied.

When users *Joe* and *Kim* connect to the system, a new session is created for each of them, as shown in figure 3.6, with objects *sesJoe* and *sesKim*. In session *sesJoe*, role `missionAdmin` is active and roles `missionMember` and *securityOfficier* are enabled for user *Joe*. In session `sesKim`, roles `missionMember` and `agencyAdmin` are enabled for user *Kim*. This model instance also captures the location of the two users at the time of their connection. Each of these locations is represented with an association between each `User` and his `RBACContext`, which contains an object of type `Point`. Objects `pK` and `pJ` refers to the position of users *Kim* and *Joe*. We assume that only *Joe* is located in the defined zone *Zone1*. We now consider the case in which one wants to check policy **PL2** on this model instance. This policy

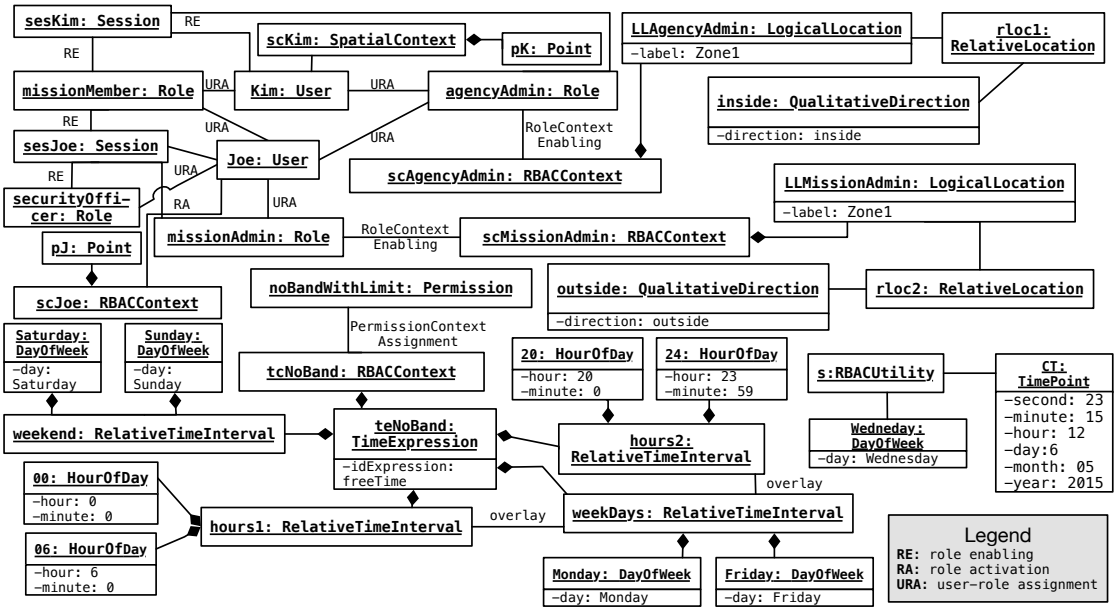


Figure 3.6: A portion of the system state after *Joe* and *Kim*'s connections

can be checked on both `Session`s, `sesKim` and `sesJoe`, using the OCL invariant `relativeLocationRoleEnabling` (introduced on page 40) parametrized with role *agencyAdmin*. For session `sesKim`, the `if` condition at lines 9–10 is true because *Kim*, according to the assumption made above, is outside *Zone1*, meaning that her position (object `pK`) is contained in the location `LLAgencyAdmin` associated with the spatial context for enabling role *agencyAdmin* (object `scAgencyAdmin`). Hence, we follow the `then` branch, calling the operation `includes` at line 11. Since role `agencyAdmin` is enabled in the session, this operation returns true, meaning that policy **PL2** is not violated for Kim. Policy **PL3** is checked in a similar way on sessions `sesKim` and `sesJoe`, using the same OCL invariant parametrized with role *missionAdmin*. This policy is not violated since role *missionAdmin* is not enabled for `Kim` (i.e., there is no association between objects `sesKim` and `missionAdmin`) and is active for `Joe` (i.e., there is an association between `sesJoe` and `missionAdmin`).

We now consider yet another change of the system corresponding to the instant when user *Mallory* is connected. A new session `sesMallory` is created for him. Only role `missionMember` is enabled in `sesMallory`. The dependency policy **PL4** states that role *trainee* is enabled if role *securityOfficer* is active. Since role *securityOfficer* is not active, *Mallory* cannot activate his role: as shown in figure 3.7, role *trainee* is not enabled in session *sesMallory*. This constraint can be expressed using the OCL invariant `RoleActivationDependency` provided on page 33, by replacing the parameter r with role *trainee*, and role r_1 with role *securityOfficer*.

In this case, let us assume that *Mallory*, while activating his role *missionMember*, finds an injured person who needs medical treatment services. She sends an alert (permission *sendAlert*) to request help for the casualty. As shown in figure 3.8, a new object of the class `History` has been created to record the details of the

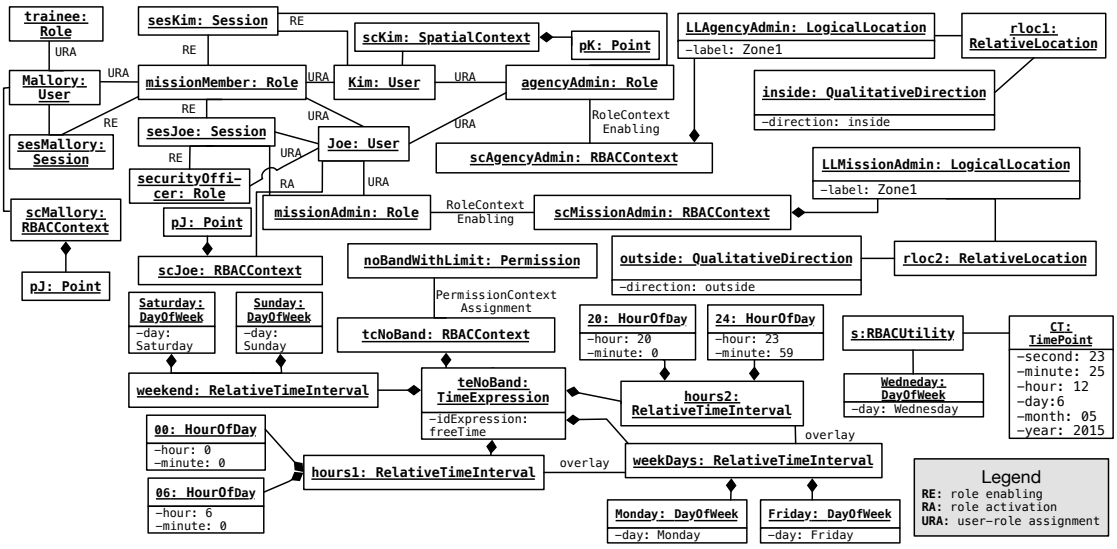


Figure 3.7: A portion of the system state after *Mallory* connection

operation: the user (`Mallory`), the user's position (`pM`), the current time `CT`, the operation (`create`) performed on the object `alert`, the corresponding permission `sendAlert`, and the activated role `missionMember`.

In the application of the proposed approach to a real scenario, we mainly focused on assessing whether all policies required by the application could be expressed with our approach. The new model has allowed the security engineers of our partner to define *19 new types* of RBAC contextual policies. With these new policies, engineers can now tune the definition of fine-grained (from the point of view of context) RBAC policies. This is a major improvement over the previous solution, which granted permissions under any context, for the lack of better specification methods. Moreover, since the GemRBAC+CTX defines structural constraints among entities, it will prevent end-users to define RBAC policies that are not well-formed.

Overall, the application of the modeling approach supported by the GemRBAC+CTX model has been warmly welcome by the security engineers of HITEC. Nevertheless, the engineers also reported some drawbacks of our current approach. In particular, they remarked that defining RBAC policies as OCL constraints on the GemRBAC+CTX class model was sometimes cumbersome, especially for complex policies (with large corresponding models). To address this limitation, we have developed a domain-specific language (DSL), defined on top of the GemRBAC+CTX model and which will be presented in chapter 4, to allow the definition of policies at a higher-level of abstraction, using a syntax close to natural language.

3.4 Summary

In this chapter we have proposed an extension of the original RBAC conceptual model, called GemRBAC+CTX, which includes all the entities required to ex-

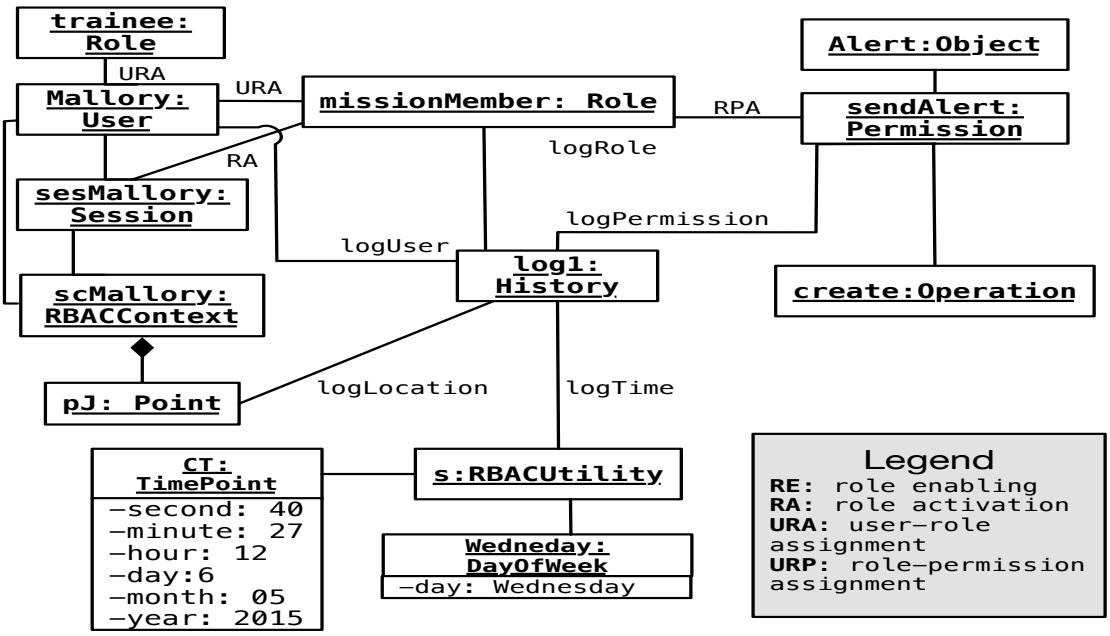


Figure 3.8: A portion of the System state after sending an alert

press the various types of RBAC policies with a specific emphasis on context-based policies. These RBAC policies have been selected based on an analysis and classification of the various RBAC extensions proposed in the literature. The various RBAC policies have been formalized as OCL constraints on the UML representation of the GemRBAC+CTX model. Our goal is to enable the specification of policies that make use of the concepts previously proposed in the literature, using a unified model (GemRBAC+CTX) and a standardized language (OCL) for their definition. With respect to the state of the art (presented in section 5.1 of chapter 5), not only we consider all types of RBAC policies proposed in the literature, but we also use a common model and notation to define them, improving their understanding.

In terms of application, from the point of view of the definition of RBAC policies, we believe that this work can represent a one-stop source for security engineers, who can access the taxonomy of the various types of RBAC constraints, determine their exact meaning by referring to their formalization as OCL constraints on the GemRBAC+CTX model, select the constraints that suit the needs of their organization, and operationalize them based on readily-available OCL checkers.

We also maintain that our framework can be a basis on which to further develop a model-driven approach for the verification of RBAC policies. Verification should be intended here in its broadest scope, including design-time verification (e.g., consistency checking of policies [43, 44]) and run-time enforcement (e.g., allowing access to resources only if an instance of the GemRBAC+CTX model satisfies the constraints associated with it as will be illustrated in chapter 6). In particular, for the latter, we assume that the run-time infrastructure collects snapshots representing the state of the system (from the point of view of RBAC) as instances of the

GemRBAC+CTX model. RBAC policies can be defined as OCL constraints that the instances of the GemRBAC+CTX model should satisfy; these constraints are based on the OCL templates proposed in section 3.2. An OCL checker (such as Eclipse OCL [45]) can be used to check if a model instance satisfies the OCL constraints associated with it, resulting in the enforcement of the corresponding access control policies.

Chapter 4

Policy Specification Language

As mentioned in chapter 1, several RBAC models have been proposed to represent different types of RBAC policies. However, the expressiveness of these models has not been matched by specification languages for RBAC policies. Indeed, existing policy specification languages do not support all the types of RBAC policies defined in the literature. As part of this thesis, we aim to bridge the gap between highly-expressive RBAC models and policy specification languages, by presenting GemRBAC-DSL, a new specification language designed on top of the GemRBAC+CTX model presented in chapter 3. The language sports a syntax close to natural language, to encourage its adoption among practitioners. We also define semantic checks to detect conflicts and inconsistencies among the policies written in a GemRBAC-DSL specification. We show how the semantics of GemRBAC-DSL can be expressed in terms of the formalization of RBAC policies (presented in chapter 3) as OCL (Object Constraint Language) constraints on the corresponding RBAC conceptual model. This formalization paves the way to define a model-driven approach for the enforcement of policies written in GemRBAC-DSL.

This chapter makes the following contributions:

1. the GemRBAC-DSL specification language for RBAC policies;
2. the definition of the semantic checks for a GemRBAC-DSL policy specification;
3. a publicly-available implementation of an editor to write policies in GemRBAC-DSL and check for potential conflicts and inconsistencies among them.

The rest of the chapter is organized as follows. Section 4.1 illustrates a motivating example. Section 4.2 presents the language, illustrating the syntax and providing examples for each type of policy. Section 4.3 defines the semantic checks for policies expressed in GemRBAC-DSL. Section 4.4 provides a brief overview of the semantics of the language. Section 4.5 discusses the design trade-offs and the limitations of GemRBAC-DSL, as well as its adoption by our industrial partner.

4.1 Motivating example

In this section we illustrate an example of RBAC policy specifications that motivates our work. The example represents a subset of a real-world case study, defined in collaboration with our industrial partner, HITEC Luxembourg. The case study deals with the specification of the RBAC policies for a Web application that provides information related to humanitarian missions, ranging from satellite images to highly-confidential data about refugees and casualties. For confidentiality reasons we consider a small, sanitized subset of the system, but provide a representative list of policies that covers exhaustively all the types of RBAC policies used in the policy specifications of the case study.

We consider a humanitarian mission taking place from February 12, 2016 to June 8, 2016 in a geographical area symbolically known as "*Zone1*", delimited by four segments with coordinates (longitude and latitude in decimal degrees, elevation in meters): (15:24:200)–(20:27:200), (20:27:200)–(17:27:200), (17:27:200)–(15:27:200), (15:27:200)–(15:24:200). The mission defines five roles (*admin*, *assistant*, *trainee*, *participant*, *analyst*), five permissions (*addCasualty*, *modifyCasualty*, *deleteCasualty*, *analyseSatellitePhoto*, *saveSatellitePhoto*), four operations (*create*, *read*, *update*, *delete*). The access control policies for this mission are:

PL1: To acquire role *trainee*, a user must be assigned to role *participant*.

PL2: Role *assistant* cannot be assigned to more than three users.

PL3: Role *trainee* is enabled only if role *admin* is active. The latter cannot be deactivated if the role *trainee* is still active.

PL4: If a user acquires role *assistant*, she will also acquire all its junior roles.

PL5: A user can acquire either role *assistant* or *trainee*.

PL6: A user can activate roles *assistant* and *admin* at the same time, as long as she does not perform all the operations (*create*, *read*, *update*, *delete*) on the same object (of type "casualty record").

PL7: The operations allowed by permissions *addCasualty*, *modifyCasualty*, and *deleteCasualty* should be performed by users having the same role.

PL8: In case a user assigned to role *admin* is on leave, she has to delegate all the permissions associated with her role to another user who is assigned to role *assistant*. The delegation lasts for two weeks; during this period the delegator is still allowed to execute the permissions associated with the role she has delegated. Moreover, the delegated role can be further delegated (by a delegate), with a maximum delegation depth of 2.

PL9: The delegation regulated by policy PL8 can be revoked by any user assigned to role *admin*. The revocation will not affect the (further) delegations of role *admin* possibly performed by delegated users. Moreover, the revocation will only remove the affected users from the delegated role *admin*, and will not impact the other roles possibly acquired through a role hierarchy (of the delegated role).

PL10: Role *analyst* is a part-time job; it can be active for a maximum duration of 4 hours per day.
PL11: Role *participant* is enabled for the entire duration of the mission.
PL12: Permission *addCasualty* is assigned to role *trainee* only during weekdays from 8:00 to 17:00.
PL13: Role *admin* is enabled only in zone *Zone1*.
PL14: Role *trainee* is enabled at 100 meters from the boundary inside *Zone1*.

The policies above show that defining the access control requirements of our example requires to deal with several types of policies (see taxonomy in chapter 2): prerequisite (PL1), cardinality (PL2), precedence (PL3), role hierarchy (PL4), SoD (PL5, PL6), BoD (PL7), delegation (PL8), revocation (PL9), contextual (PL10–PL12). To express these policies security engineers need a policy specification language *expressive enough* to support all of them.

4.2 The GemRBAC-DSL language

The GemRBAC-DSL policy specification language has been designed as a domain-specific language built on top of the GemRBAC+CTX model. The choice of the underlying model for the language has been dictated by the need to support a large variety of RBAC policies, like the ones used for the specification of our industrial case study (see Section 4.1). Hence, the language inherits the expressiveness of the GemRBAC+CTX model (see chapter 3).

The main goal during the design of the language has been to encourage its use among practitioners. Indeed, the language captures the main RBAC concepts that security analysts are familiar with and allows for their specification using a syntax close to natural language. Furthermore, the language design process has incorporated the feedback provided by the security analysts of our industrial partner, who have commented on the expressiveness and the clarity of the language. At the time of writing, the language is being introduced into the security development lifecycle of our partner, to support the top-down definition of access control policies and enforcement mechanisms.

4.2.1 Syntax

The syntax of GemRBAC-DSL is shown in figure 4.1, using the Extended Backus-Naur Form (EBNF) notation: non-terminal symbols are enclosed in angle brackets; terminal symbols are enclosed in single quotes; (derivation) rules are denoted with the ::= symbol; alternatives within a rule are indicated using a vertical bar; a star stands for zero or more occurrences of an element; a plus stands for one or more occurrences of an element; square brackets denote optional elements.

⟨*RBAC-definition*⟩ ::= ⟨*preamble*⟩ ⟨*policies*⟩

⟨*preamble*⟩ ::= ⟨*users*⟩ ⟨*roles*⟩ ⟨*permissions*⟩ ⟨*operations*⟩ ⟨*role-hierarchy*⟩ ⟨*permission-hierarchy*⟩ ⟨*geofences*⟩

⟨*users*⟩ ::= '`users:`' ⟨*user*⟩ ('`,`' ⟨*user*⟩)* '`;`'

⟨*roles*⟩ ::= '`roles:`' ⟨*role*⟩ ('`,`' ⟨*role*⟩)* '`;`'

⟨*permissions*⟩ ::= '`permissions:`' ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)* '`;`'

⟨*operations*⟩ ::= '`operations:`' ⟨*operation*⟩ ('`,`' ⟨*operation*⟩)* '`;`'

⟨*id*⟩ ::= ('`a`'-'`z`' | '`A`'-'`Z`' | '`0`'-'`9`')+

⟨*user*⟩ ::= ⟨*id*⟩

⟨*role*⟩ ::= ⟨*id*⟩

⟨*permission*⟩ ::= ⟨*id*⟩

⟨*operation*⟩ ::= ⟨*id*⟩

⟨*role-hierarchy*⟩ ::= '`role-hierarchy:`' (⟨*rHierarchy*⟩ ('`,`' ⟨*rHierarchy*⟩)* | '`none`') '`;`'

⟨*permission-hierarchy*⟩ ::= '`permission-hierarchy:`' (⟨*pHierarchy*⟩ ('`,`' ⟨*pHierarchy*⟩)* | '`none`') '`;`'

⟨*rHierarchy*⟩ ::= ⟨*role*⟩ '`: {`' ⟨*role*⟩ ('`,`' ⟨*role*⟩)* '`}`'

⟨*pHierarchy*⟩ ::= ⟨*permission*⟩ '`: {`' ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)* '`}`'

⟨*geofence*⟩ ::= '`geofences:`' (⟨*geofence*⟩ ('`,`' ⟨*geofence*⟩)* | '`none`') '`;`'

⟨*geofence*⟩ ::= ⟨*id*⟩

⟨*policies*⟩ ::= '`policies:`' (⟨*policy*⟩'`;`')+

⟨*policy*⟩ ::= ⟨*id*⟩ '`:`' (⟨*Prerequisite*⟩ | ⟨*Cardinality*⟩ | ⟨*PrecEnabling*⟩ | ⟨*Hierarchy*⟩ | ⟨*SSoD*⟩ | ⟨*DSoD*⟩ | ⟨*BoD*⟩ | ⟨*Delegation*⟩ | ⟨*Revocation*⟩ | ⟨*ContextPolicy*⟩)

Figure 4.1: Grammar of GemRBAC-DSL

A GemRBAC-DSL policy specification (captured by the start symbol ⟨*RBAC-definition*⟩) contains a ⟨*preamble*⟩ and a list of ⟨*policies*⟩. The ⟨*preamble*⟩ contains the declaration of the main entities that will be used in the rest of the specification[1]: the list of users ⟨*users*⟩, the list of roles ⟨*roles*⟩, the list of permissions ⟨*permissions*⟩, and the list of operations ⟨*operations*⟩. The ⟨*preamble*⟩ contains also the list ⟨*role-hierarchy*⟩ of role hierarchy relations, and the list ⟨*permission-hierarchy*⟩

[1]Notice that the assignments of users to roles, of permissions to roles, and of operations to permissions *are not* specified with GemRBAC-DSL. We assume that these assignments are defined in the RBAC system on which the policies are going to be enforced.

of permission hierarchy relations. Within these lists, each hierarchy relation (⟨*rHierarchy*⟩ for role hierarchy and ⟨*pHierarchy*⟩ for permission hierarchy) declares the parent (role or permission) followed by the list of its junior (roles or permissions, respectively). The absence of role (or permission) hierarchies is explicitly denoted with the keyword '`none`'. The ⟨*preamble*⟩ ends with the list ⟨*geofences*⟩ of logical locations, i.e., symbolic abstractions that refer to real physical locations(see section 3.1.3 of chapter 3). All the lists used in the ⟨*preamble*⟩ are comma-separated and contain alphanumeric identifiers. Finally, the list of policies ⟨*policies*⟩ contains the actual policy specifications, where each policy is composed by an identifier and by its body. The following subsections illustrate each type of policy supported by GemRBAC-DSL; for each policy, we include the syntax, its explanation, and an example of specification based on the policies defined in section 4.1.

4.2.2 Prerequisite policy

The syntax of a prerequisite policy is defined as:

⟨*Prerequisite*⟩ ::= ⟨*PrereqRole*⟩ | ⟨*PrereqPermission*⟩ (1)

⟨*PrereqRole*⟩ ::= '`assign-role`' ⟨*role1*⟩ '`prerequisite`' ⟨*role2*⟩ (2)

⟨*PrereqPermission*⟩ ::= '`assign-permission`' ⟨*permission1*⟩ '`prerequisite`' ⟨*permission2*⟩ (3)

The syntax uses keywords for defining a prerequisite policy either at the role (keyword '`assign-role`' in rule 2) or at permission level (keyword '`assign-permission`' in rule 3). In rule 2, ⟨*role2*⟩ corresponds to the precondition for the assignment of ⟨*role1*⟩. Similarly, in rule 3, ⟨*permission2*⟩ corresponds to the precondition for the assignment of ⟨*permission1*⟩. For example, the prerequisite policy on role assignment PL1 is expressed in GemRBAC-DSL as:

```
PL1: assign-role trainee prerequisite participant;
```

4.2.3 Cardinality policy

The syntax of a cardinality policy is defined as:

⟨*Cardinality*⟩ ::= ⟨*CardActivation*⟩ | ⟨*CardUser*⟩ | ⟨*CardPermission*⟩ | ⟨*CardRoleToUser*⟩ | ⟨*CardRoleToPermission*⟩ (1)

⟨*CardActivation*⟩ ::= '`maxActiveRoles =`' ⟨*integer*⟩ (2)

⟨*CardUser*⟩ ::= '`maxUsers =`' ⟨*integer*⟩ ['`only-for-role`' ⟨*role*⟩] (3)

⟨*CardPermission*⟩ ::= '`maxPermissions =`' ⟨*integer*⟩ ['`only-for-role`' ⟨*role*⟩] (4)

⟨*CardRoleToUser*⟩ ::= '`maxRoles-User =`' ⟨*integer*⟩ ['`only-for-user`' ⟨*user*⟩] (5)

⟨*CardRoleToPermission*⟩ ::= '`maxRoles-Permission =`' ⟨*integer*⟩ ['`only-for-permission`' ⟨*Permission*⟩] (6)

GemRBAC-DSL supports five types of cardinality policies: maximum number of active roles within a session (rule 2), maximum number of users assigned to a role (rule 3), maximum number of permissions assigned to a role (rule 4), maximum number of roles assigned to a user (rule 5), maximum number of roles assigned to a permission (rule 6). In rules 2–6, ⟨*integer*⟩ represents the cardinality bound. In rules 3–6, if the optional element is omitted, it means that the bound will apply, respectively, to all roles (rules 3–4), all users (rule 5), all permissions (rule 6). For example, the cardinality policy on user-to-role assignment PL2 is expressed in GemRBAC-DSL as:

```
PL2: maxUsers = 3 only-for-role assistant;
```

4.2.4 Precedence and dependency policies

The syntax of a precedence policy is defined as:

⟨*PrecEnabling*⟩ ::= '`enable`' ⟨*role1*⟩ ' `if active`' ⟨*role2*⟩ (1)
['`,`' ⟨*timeShift*⟩] ['`deactivation-dependency`']

⟨*timeShift*⟩ := '`after`' ⟨*integer*⟩ ⟨*timeUnit*⟩ (2)

⟨*timeUnit*⟩ ::= '`second`' | '`minute`' | '`hour`' | '`day`' | '`week`' | '`month`' | '`year`' (3)

In rule 1, ⟨*role2*⟩ denotes the role whose activation has to precede the enabling of the role denoted by ⟨*role1*⟩. An optional ⟨*timeShift*⟩ can be specified to define the amount of time that has to pass between the role enabling and the role activation events (rules 2–3). The optional keyword '`deactivation-dependency`' is used to express a dependency policy. For example, the precedence and dependency policy PL3 is expressed in GemRBAC-DSL as:

```
PL3: enable trainee if active admin deactivation-dependency;
```

4.2.5 Role hierarchy policy

The syntax of an hierarchy policy is defined as:

⟨*Hierarchy*⟩ ::= '`trigger-`' (⟨*RoleHierarchy*⟩ | ⟨*PermissionHierarchy*⟩) (1)

⟨*RoleHierarchy*⟩ ::= '`role-hierarchy`' ⟨*role*⟩ (2)

⟨*PermissionHierarchy*⟩ ::= '`permission-hierarchy`' ⟨*permission*⟩ (3)

The syntax uses two different keywords for distinguishing between role hierarchy (rule 2) and permission hierarchy (rule 3). Notice that while the preamble of a GemRBAC-DSL specification declares the role and permission hierarchy relations for the system, a security analyst has to explicitly define a role hierarchy policy (for a role or permission) to put the hierarchy relation(s) into effect. For example, the role hierarchy policy PL4 can be expressed as:

```
PL4: trigger-role-hierarchy assistant;
```

4.2.6 Separation of duty policy

The language supports the various types of separation of duty policies introduced in section 2.2.5 of chapter 2.

4.2.6.1 Static Separation of duty (SSoD)

The syntax of an SSoD policy syntax is defined as:

⟨*SSoD*⟩ ::= ⟨*SSoDCR*⟩ | ⟨*SSoDCU*⟩ | ⟨*SSoDCP*⟩ (1)

⟨*SSoDCR*⟩ ::= '`conflicting-roles-assignment`' ⟨*role*⟩ ('`,`' ⟨*role*⟩)+ ['`on permission`' ⟨*permission*⟩] (2)

⟨*SSoDCU*⟩ ::= '`conflicting-users-assignment`' ⟨*user*⟩ ('`,`' ⟨*user*⟩)+ ['`on role`' ⟨*role*⟩] (3)

⟨*SSoDCP*⟩ ::= '`conflicting-roles-assignment`' ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)+ ['`on role`' ⟨*role*⟩] (4)

SSoD policies can define conflicting roles (rule 2), conflicting users (rule 3), and conflicting permissions (rule 4). Rules 2–4 have an optional block that indicates that the SSoD policy is applied only when the roles are assigned to a specific permission (rule 2) and when the users (rule 3) or the permissions (rule 4) are assigned to a specific role. For example, the SSoD policy on conflicting roles PL5 is expressed in GemRBAC-DSL as:

```
PL5: conflicting-roles-assignment assistant, trainee;
```

4.2.6.2 Dynamic Separation of duty (DSoD)

GemRBAC-DSL supports the specification of four types of DSoD: simple, object-based, operational-based, and history-based DSoD. The syntax for DSoD policies is similar to the one for SSoD policies but uses different keywords:

⟨*DSoD*⟩ ::= ⟨*DSoDCU*⟩ | ⟨*DSoDCP*⟩ | ⟨*DSoDCR*⟩ (1)

⟨*DSoDCU*⟩ ::= '`conflicting-users-activation`' ⟨*user*⟩ ('`,`' ⟨*user*⟩)+ ['`on role`' ⟨*role*⟩] (2)

⟨*DSoDCP*⟩ ::= '`conflicting-permissions-activation`' ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)+ ['`on role`' ⟨*role*⟩] (3)

⟨*DSoDCR*⟩ ::= '`conflicting-roles-activation`' ⟨*role*⟩ ('`,`' ⟨*role*⟩)+ ['`depending-on-business-task-list`' ⟨*operation*⟩ ('`,`' ⟨*operation*⟩)+] ['`on-same-object`'] (4)

The optional keyword '`on-same-object`' in rule 4 is used to express an object-based DSoD policy. Similarly, the keyword '`depending-on-business-task-list`' followed by a list of ⟨*operation*⟩s is used to specify an operational-based DSoD. A history-based DSoD is defined by combining these two keywords. For example, the history-based DSoD policy PL6 is expressed in GemRBAC-DSL as:

```
PL6: conflicting-roles-activation assistant, admin
    depending-on-business-task-list create,read,update,delete on-same-object;
```

4.2.7 Binding of duty policy

The syntax of a BoD policy is defined as:

⟨*BoD*⟩ ::= '`bounded-permissions`' ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)+
('`role-BoD`' | '`subject-BoD`')

The syntax distinguishes between a role- or a subject-based policy with the two keywords '`role-BoD`' and '`subject-BoD`'. The bounded permissions are specified as a list of ⟨*permission*⟩s. For instance, the role-based BoD policy PL7 is expressed in GemRBAC-DSL as:

```
PL7: bounded-permissions addCasualty, modifyCasualty, deleteCasualty  role-BoD;
```

4.2.8 Delegation policy

The language supports the various types of delegation introduced in section 2.2.7.1 of chapter 2. The syntax of a delegation policy is defined below:

⟨*Delegation*⟩ ::= ('`user` '⟨*user*⟩ | '`role` '⟨*role*⟩) '`can-delegate`' ⟨*role*⟩ (1)
('`to users`' ⟨*user*⟩ ('`,`' ⟨*user*⟩)* | '`to roles`'⟨*role*⟩ ('`,`' ⟨*role*⟩)*) '`as`'
('`total`' | '`partial with permissions (`'⟨*delegated-permissions*⟩'`)`') '`,`'
('`grant`' [⟨*duration*⟩] ('`single`' | '`multi-step`' ⟨*integer*⟩)
|'`transfer`' ('`strong`'|'`weak-static`'|'`weak-dynamic`'))

⟨*delegated-permissions*⟩ ::= ⟨*permission*⟩ ('`,`' ⟨*permission*⟩)* (2)

⟨*duration*⟩ ::= '`for`' ⟨*integer*⟩ ⟨*timeUnit*⟩ (3)

In the syntax, keywords '`user`' and '`role`' are used to denote the delegator. The keyword '`can-delegate`' denotes the ⟨*role*⟩ being delegated. The list of delegate ⟨*user*⟩s is denoted by the keyword '`to users`'; similarly, the keyword '`to roles`' denotes the list of delegate ⟨*role*⟩s. If the delegation is partial, the keyword '`partial-with-permissions`' denotes the list of ⟨*permission*⟩s being delegated. In the case of a multi-step delegation, the syntax requires to indicate the ⟨*integer*⟩ corresponding to the maximum number of delegation steps allowed. If the delegation is of type grant, a duration (denoted with the keyword '`for`', rule 3) can be optionally specified to indicate the amount of time after which the delegation is automatically revoked. For example, the delegation policy PL8 defines a delegation that is *multi-step* (with a maximum delegation depth of 2), *total* (because all the permissions of the delegated role have to be delegated), of type *grant* (because the delegator is still allowed to execute the permissions associated with the delegated role), with a *duration* of at most two weeks. This policy is expressed in GemRBAC-DSL as:

```
PL8: role admin can-delegate admin to roles assistant as total, grant for 2
    week, multistep 2;
```

4.2.9 Revocation policy

The language supports the various types of revocation introduced in section 2.2.7.2 of chapter 2. The syntax of a revocation policy is defined as:

⟨*Revocation*⟩ ::= (‘`user`’ ⟨*user*⟩ | ‘`role`’ ⟨*role*⟩ | ‘`delegator`’)
‘`can-revoke-delegation`’ ⟨*id*⟩
(‘`from users`’ ⟨*user*⟩ (‘`,`’ ⟨*user*⟩)* | ‘`from roles`’ ⟨*role*⟩ (‘`,`’ ⟨*role*⟩)*)) ‘`as`’
(‘`strong`’ | ‘`weak`’) ‘`,`’ (‘`nonCascading`’ | ‘`cascading`’)

The syntax allows for specifying who can revoke a certain delegation; the keywords ‘`user`’ and ‘`role`’ denote, respectively, an explicit user or role, while the keyword ‘`delegator`’ implicitly refers to the user or role that originally performed the delegation. The delegation that is being revoked is referenced through its identifier, preceded by the keyword ‘`can-revoke-delegation`’. The keyword ‘`from users`’ denotes the list of ⟨*users*⟩ from which the delegation is revoked; similarly, the keyword ‘`from roles`’ denotes the list of ⟨*roles*⟩ from which the delegation will be revoked. The additional keywords that come after the keyword ‘`as`’ indicate the type of revocation. For example, the revocation policy PL9 is defined as *weak* (because it will not impact the other roles possibly acquired through a role hierarchy) and as *non-cascading* (because it will not affect the further delegations performed along a delegation chain). This policy is expressed in GemRBAC-DSL as:

```
PL9: role admin can-revoke-delegation PL8 from roles assistant as weak,
    nonCascading;
```

4.2.10 Contextual policy

The syntax for a contextual policy is defined as follows:

⟨*ContextPolicy*⟩ ::= ⟨*RoleContextPolicy*⟩ | ⟨*PermContextPolicy*⟩ (1)

⟨*RoleContextPolicy*⟩ ::= ‘`role-context`’ ⟨*role*⟩ (2)
(⟨*activeDuration*⟩
| ((‘`assign`’ | ‘`unassign`’) [‘`to user`’ ⟨*user*⟩] ⟨*context*⟩
| (‘`enable`’ | ‘`disable`’) ⟨*context*⟩
[‘`,`’ (‘`assign`’ | ‘`unassign`’) [‘`to user`’ ⟨*user*⟩] ⟨*context*⟩])
[‘`,`’ ⟨*activeDuration*⟩]))

⟨*activeDuration*⟩ :: = ‘`activation`’ (3)
(‘`duration`’ ⟨*integer*⟩ ⟨*timeUnit*⟩
| ‘ `cumulative duration =` ’ ⟨*integer*⟩ ⟨*timeUnit*⟩ ‘`,`’

‘`reset =`’ (‘`none`’ | ⟨*periodicTime*⟩) ‘`,`’
‘`duration-per-session =`’ (‘`unlimited`’ | ⟨*integer*⟩ ⟨*timeUnit*⟩)))

⟨*periodicTime*⟩ ::= ‘`every`’ [⟨*integer*⟩] ⟨*timeUnit*⟩ (4)

⟨*PermContextPolicy*⟩ ::= ‘`permission-context`’ ⟨*permission*⟩ (5)
((‘`assign`’ | ‘`unassign`’) [‘`to role`’ ⟨*role*⟩] ⟨*context*⟩
| (‘`enable`’ | ‘`disable`’) ⟨*context*⟩
[‘`,`’ (‘`assign`’ | ‘`unassign`’) [‘`to role`’ ⟨*role*⟩] ⟨*context*⟩])

⟨*context*⟩ ::= ‘`@`’ (⟨*temporalContext*⟩ | ⟨*spatialContext*⟩ (6)
| ⟨*SpatioTemporalContext*⟩ (‘`&&`’ ⟨*SpatioTemporalContext*⟩)*)

⟨*SpatioTemporalContext*⟩ ::= ⟨*spatialContext*⟩ ⟨*temporalContext*⟩ (7)

A contextual policy can be specified either at the role (rule 2) or at the permission level (rule 5). At the role level, a contextual policy can define a) a bound for the sum of activation durations of a given role and/or, b) the context of role assignment and/or role enabling. An *activation duration* represents the amount of time during which a role is active. As shown in rule 3, an activation duration can be specified for a single session (denoted with keyword ‘`duration`’) or for multiple sessions (denoted with keyword ‘`cumulative duration =`’). In the second case, a security analyst should specify a reset period (line 3 of rule 3) and a bound for the maximum duration per single session (line 4 of rule 3). The reset period corresponds to a specific period of time after which the cumulative duration is reinitialized to zero. This period is represented by a periodicity expression as indicated in rule 4. The keyword ‘`none`’ is used to indicate the absence of a reset period (rule 3). Similarly, the keyword ‘`unlimited`’ is used to indicate the absence of a bound for the activation duration per session (rule 3). In addition to the activation duration, a security analyst can specify if a role should be assigned/unassigned (possibly to a specific user, as denoted by the optional keyword ‘`to user`’), or if a role should be enabled/disabled in a specific ⟨*context*⟩(rule 2). Notice that the same policy can restrict both role enabling/disabling and assignment/unassignment as indicated by the optional part in line 4 of rule 2. Rule 5 is structured similarly to rule 2 (lines 1–4) but it is used for specifying the enabling/disabling and/or assignment/unassignment of permissions. As shown in rule 6, GemRBAC-DSL supports temporal, spatial and spatio-temporal context specifications preceded by the ‘`@`’ symbol. Temporal and spatial policies will be illustrated in the next subsections, using the concepts of the GemRBAC+CTX model. Since spatio-temporal specifications can be seen as the conjunction of a temporal policy and a spatial one, we will omit their description.

An example of a contextual policy on role activation with a reset period is PL10, which can be expressed in GemRBAC-DSL as:

```
PL10: role-context analyst activation cumulative duration = 4 hour, reset =
    every day, duration-per-session = unlimited;
```

4.2.10.1 Policies with temporal context

The syntax for defining a temporal context is:

⟨*temporal*⟩ ::= '`time`' (⟨*absoluteTime*⟩ | ⟨*relativeTime*⟩
| (⟨*compositeTime*⟩ ('`&`' ⟨*compositeTime*⟩)*)
⟨*compositeTime*⟩ ::= ⟨*absoluteTime*⟩ ⟨*relativeTime*⟩

We recall (chapter 3) that an absolute time expression refers to a concrete point or interval in the timeline; conversely, a relative time expression cannot be mapped directly to a concrete point or interval in the timeline. Furthermore, absolute time and relative expressions can also be composed. The syntax of an absolute time expression is:

⟨*absoluteTime*⟩ ::= (1)
((⟨*date*⟩ ['`at`' ⟨*hour*⟩] | '`(`' ⟨*date*⟩ ('`,`'⟨*date*⟩)+'`)`')
|('`starting from`' ⟨*date*⟩ ['`at`' ⟨*hour*⟩]
| '`[`'⟨*date*⟩'`,`'⟨*date*⟩'`]`'
| '`(`' '`[`'⟨*date*⟩'`,`'⟨*date*⟩'`]`' ('`, [`'⟨*date*⟩'`,`'⟨*date*⟩'`]`')+'`)`')
[⟨*periodicTime*⟩])

⟨*periodicTime*⟩ ::= '`every`' [⟨*integer*⟩] ⟨*timeUnit*⟩ (2)

⟨*date*⟩ ::= ⟨*sDayOfMonth*⟩('`1`'-'`9`')('`0`'-'`9`')('`0`'-'`9`')('`0`'-'`9`') (3)

⟨*sDayOfMonth*⟩ ::= ⟨*integer*⟩ ⟨*sMonth*⟩ (4)

⟨*sMonth*⟩ ::= '`Jan`' | '`Feb`' | '`Mar`' | '`Apr`' | '`May`' (5)
| '`June`' | '`July`' | '`Aug`' | '`Sept`' | '`Oct`' | '`Nov`' | '`Dec`'

⟨*hour*⟩ ::= (('`0`'-'`1`')('`0`'-'`9`') | ('`2`')('`0`'-'`3`')) '`:`' (6)
('`0`'-'`5`') ('`0`'-'`9`') '`:`' ('`0`'-'`5`') ('`0`'-'`9`')

An absolute time expression can have different forms. The simplest form is captured by ⟨*date*⟩, which is composed of a day of the month ⟨*sDayOfMonth*⟩ and a year (rule 4). An ⟨*sDayOfMonth*⟩ denotes a day, represented as an ⟨*integer*⟩, and a month, represented as an ⟨*sMonth*⟩. The latter corresponds to the abbreviation for a specific month (rule 6). A ⟨*date*⟩ can be optionally followed by the '`at`' keyword and an ⟨*hour*⟩, to represent a specific hour during a day[1]. An absolute time expression can also correspond to a list of ⟨*date*⟩s enclosed in round brackets. Another type of absolute time expression is represented by intervals. An unbounded time interval is specified with a ⟨*date*⟩ prefixed by the keyword '`starting from`'. A bounded time interval is represented as two ⟨*date*⟩s enclosed in square brackets. Lists of bounded intervals are enclosed in round brackets. Unbounded and bounded time intervals as well as lists of bounded time intervals can be followed by a periodicity expression (denoted with the keyword '`every`', see rule 2), which

[1]The current version of GemRBAC-DSL does not support the concept of time zone.

specifies how often, during the selected interval(s), the action determined by the policy (e.g., enabling a role) should be in effect. For example, the role enabling policy PL11 can be expressed as:

```
PL11: role-context  participant enable @time  [12 Feb 2016, 8 Jun 2016];
```

A relative time expression is a time expression that cannot be mapped directly to a concrete point or interval in the timeline. The syntax of a relative time expression is:

⟨*relativeTime*⟩ ::= ((⟨*iHour*⟩ ('`,`' ⟨*iHour*⟩)*)
| (⟨*dayOfMonthH*⟩ ('`and @ time`' ⟨*dayOfMonthH*⟩)*)
| (⟨*dayOfWeekH*⟩ ('`and @ time`' ⟨*dayOfWeekH*⟩)*)
| (⟨*monthDayOfWeekH*⟩
('`and @ time`' ⟨*monthDayOfWeekH*⟩)*))

A relative time expression can have different forms. The first form is as a list of hour intervals, which are intervals whose start and end points are hours. The syntax of an hour interval is:

⟨*iHour*⟩ ::= '`from`' ⟨*hour*⟩'`to`' ⟨*hour*⟩ (1)
['`excluding (`' ⟨*exHour*⟩ ('`,`' ⟨*exHour*⟩)* '`)`']

⟨*exHour*⟩ ::= '`from`' ⟨*hour*⟩'`to`' ⟨*hour*⟩ (2)

Within the definition of an ⟨*iHour*⟩, one can also specify a list of hour intervals to be excluded, denoted with the keyword '`excluding`' (rule 2).

A relative time expression can be also defined as a list of expressions starting with a day of month (⟨*dayOfMonthH*⟩s). This expression corresponds to a day of month (⟨*dayOfMonth*⟩) that optionally overlays an hour interval. The syntax of a relative expression with a day of month is:

⟨*dayOfMonthH*⟩ ::= ⟨*dayOfMonth*⟩('`,`'⟨*dayOfMonth*⟩)* (1)
[(⟨*iHour*⟩ ('`,`' ⟨*iHour*⟩)*)]

⟨*dayOfMonth*⟩ ::= ⟨*sDayOfMonth*⟩ | ⟨*iDayOfMonth*⟩ (2)

⟨*iDayOfMonth*⟩ ::= '`from`' ⟨*sDayOfMonth*⟩ '`to`' (3)
⟨*sDayOfMonth*⟩ ['`excluding (`' ⟨*exDayOfMonth*⟩
('`,`' ⟨*exDayOfMonth*⟩)* '`)`']

⟨*exDayOfMonth*⟩ ::= ⟨*sDayOfMonth*⟩|⟨*exIDayOfMonth*⟩ (4)

⟨*exIDayOfMonth*⟩ ::= '`from`' ⟨*sDayOfMonth*⟩ '`to`' (5)
⟨*sDayOfMonth*⟩

A day of month can correspond to a single day (⟨*sDayOfMonth*⟩, see page 15) or an interval of days of month (⟨*iDayOfMonth*⟩) (rule 2). The latter can also be defined to exclude a single day of month or an interval of days of month ⟨*exIDayOfMonth*⟩; notice that exclusion is not recursive.

A relative time expression can also have the form of a list of ⟨*dayOfWeekH*⟩s. The latter is a day of week that optionally overlays GemRBAC-DSLGemRBAC-DSLan hour interval. The syntax of a relative expression with a day of week is:

⟨*dayOfWeekH*⟩ ::= ⟨*dayOfWeek*⟩ (‘`,`’ ⟨*dayOfWeek*⟩)* (1)
[⟨*iHour*⟩ (‘`,`’ ⟨*iHour*⟩)*]

⟨*dayOfWeek*⟩ ::= ⟨*sDayOfWeek*⟩ | ⟨*iDayOfWeek*⟩ (2)

⟨*sDayOfWeek*⟩ ::= [[‘`on`’] ‘`the`’ ⟨*integer*⟩] (‘`Monday`’ (3)
| ‘`Tuesday`’ | ‘`Wednesday`’ | ‘`Thursday`’ | ‘`Friday`’ | ‘`Saturday`’ | ‘`Sunday`’)

⟨*iDayOfWeek*⟩ ::= ‘`from`’ ⟨*sDayOfWeek*⟩ ‘`to`’ (4)
⟨*sDayOfWeek*⟩ [‘`excluding (`’ ⟨*exDayOfWeek*⟩
(‘`,`’ ⟨*exDayOfWeek*⟩)* ‘`)`’]

⟨*exDayOfWeek*⟩ ::= ⟨*sDayOfWeek*⟩ | ⟨*exIDayOfWeek*⟩ (5)

⟨*exIDayOfWeek*⟩ ::= ‘`from`’ ⟨*sDayOfWeek*⟩ ‘`to`’ (6)
⟨*sDayOfWeek*⟩

This syntax follows a pattern similar to the ones seen above. For example, the time-based policy on permission assignment PL12 is expressed in GemRBAC-DSL as:

```
PL12: permission-context addCasualty assign to role trainee @time from Monday
    to Friday from 08:00:00 to 17:00:00;
```

A relative time expression can be also defined as a set of ⟨*monthDayOfWeekH*⟩s. The latter is a list of ⟨*month*⟩s that optionally overlays a ⟨*dayOfMonthH*⟩ or an ⟨*iHour*⟩. The syntax of ⟨*monthDayOfWeekH*⟩ is:

⟨*monthDayOfWeekH*⟩ ::= ⟨*month*⟩ (‘`,`’ ⟨*month*⟩)* (1)
[(‘`#`’ ⟨*dayOfWeekH*⟩)+
|(⟨*iHour*⟩ (‘`,`’ ⟨*iHour*⟩)*)]

⟨*month*⟩ ::= ⟨*sMonth*⟩ | ⟨*iMonth*⟩ (2)

⟨*iMonth*⟩ ::= ‘`from`’ ⟨*sMonth*⟩ ‘`to`’ ⟨*sMonth*⟩ (3)
[‘`excluding (`’ ⟨*exMonth*⟩ (‘`,`’ ⟨*exMonth*⟩)* ‘`)`’]

⟨*exMonth*⟩ ::= ⟨*sMonth*⟩ | ⟨*exIMonth*⟩ (4)

⟨*exIMonth*⟩ ::= ‘`from`’ ⟨*sMonth*⟩ ‘`to`’ ⟨*sMonth*⟩ (5)

Also this syntax follows the same structure of the previous definitions. Notice that in this case, the list of ⟨*month*⟩s can overlay either a list of ⟨*iHour*⟩s or a list of ⟨*dayOfWeekH*⟩s.

An ⟨*sDayOfWeek*⟩ can contain an index (represented as an ⟨*integer*⟩), which refers to a specific occurrence of a day, as in “on the first Monday” (of a month).

4.2.10.2 Policies with spatial context

The syntax for defining a spatial context is:

⟨*spatial*⟩ ::= '`location`' ⟨*location*⟩ ('`,`' ⟨*location*⟩)* (1)

⟨*location*⟩ ::= [relativeLocation] ('`physical`' ⟨*physicalLocation*⟩ | '`geofence`' ⟨*geofence*⟩) (2)

⟨*physicalLocation*⟩ ::= ⟨*point*⟩ | ⟨*polygon*⟩ | ⟨*circle*⟩ | ⟨*userPos*⟩ (3)

⟨*point*⟩ ::= '`(lat`' ⟨*float*⟩'`: long`' ⟨*float*⟩'`: alt`' ⟨*float*⟩'`)`' (4)

⟨*userPos*⟩ ::= '`position`' ⟨*user*⟩ (5)

⟨*circle*⟩ ::= '`center`' ⟨*point*⟩ '`radius`' ⟨*float*⟩ ⟨*locUnit*⟩ (6)

⟨*polygon*⟩ ::= ⟨*polyline*⟩ ⟨*polyline*⟩ ('`,`' ⟨*polyline*⟩)+ (7)

⟨*polyline*⟩ ::= '`line {`' ⟨*point*⟩ '`,`' ⟨*point*⟩ '`}`' (8)

⟨*relativeLocation*⟩ ::= [⟨*integer*⟩ ⟨*locUnit*⟩] ⟨*direction*⟩ (9)

⟨*locUnit*⟩ ::= '`miles`' | '`meters`' | '`kilometers`' (10)

⟨*direction*⟩ ::= ⟨*cardinalDir*⟩ | ⟨*qualitativeDir*⟩ (11)

⟨*cardinalDirection*⟩ ::= ('`N`' | '`E`' | '`S`' | '`W`' | '`NE`' | '`SE`' | '`SW`' |'`NW`') | '`degree`' ⟨*integer*⟩ (12)

⟨*qualitativeDirection*⟩ ::= '`inside`' |'`outside`' |'`around`' (13)

We recall (chapter 3) that the spatial context refers to a set of locations; these location can be of type physical (a precise position in a geometric space) or logical (a symbolic abstraction of one or many physical locations). Physical locations are denoted in GemRBAC-DSL with the keyword '`physical`', while the keyword '`geofence`' denotes logical locations. Notice that the identifiers that can be used as logical locations are those declared in the preamble under the rule ⟨*geofences*⟩.

The simplest type of physical location is a ⟨*point*⟩, i.e., a set of geographic coordinates denoted with the keywords '`lat`', '`long`', and '`alt`', corresponding to latitude, longitude, and altitude (rule 4). Each coordinate is expressed as a floating-point number. The keyword '`position`' followed by a user id (rule 5) is used to define a location in terms of the coordinates of a user. Bounded physical locations can have the shape of a circle or of a polygon. A ⟨*circle*⟩ is denoted with a '`center`' and a '`radius`'; the latter is specified using units of length (see rules 6 and 10). A polygon is defined in terms of polylines, which are denoted with the keyword '`line`' and a start and an end ⟨*point*⟩ (rules 7–8). For example, the location-based policy on role enabling PL13 is expressed in GemRBAC-DSL as:

```
PL13: role-context enable admin @location physical
line {(lat 15 : long 24 : alt 200), (lat 20 : long 27 : alt 200)},
line {(lat 20 : long 27 : alt 200), (lat 17 : long 27 : alt 200)},
line {(lat 17 : long 27 : alt 200), (lat 15 : long 27 : alt 200)},
line {(lat 15 : long 27 : alt 200), (lat 15 : long 24 : alt 200)};
```

As shown in rule 2, both physical and logical locations can be optionally prefixed by ⟨*relativeLocation*⟩, which represents a location defined with respect to another one. A ⟨*relativeLocation*⟩ is expressed with a ⟨*direction*⟩ and an optional distance expressed with a unit of length (rule 9). A direction of type ⟨*cardinalDirection*⟩ is denoted with symbols corresponding to cardinal and ordinal directions or with the degrees of rotation (denoted with the '`degree`' keyword followed by an integer) on a compass (rule 12). A direction of type ⟨*qualitativeDirection*⟩ represents a relative proximity to a location and is defined using the keywords '`inside`', '`outside`', or '`around`' (rule 13). For example, the contextual policy PL14, which contains a relative location, is expressed in GemRBAC-DSL as:

```
PL14: role-context trainee enable @location 100 meters inside geofence Zone1;
```

4.3 Semantic Checks

A security analyst can erroneously write policies that are inconsistent or conflicting. In the following paragraphs we describe all the possible conflicts that can be found in a GemRBAC-DSL specification. We mainly focus on inter-policy conflicts, i.e., global conflicts between different policies. The Eclipse-based editor for GemRBAC-DSL includes semantic checks for these conflicts, which are then reported to the user as errors or warnings.

Prerequisite role and SSoD on conflicting roles policies. Let PR be the set of roles involved in a prerequisite role policy, and SCR be the set of conflicting roles in a SSoDCR policy. If $PR \subseteq SCR$, the two policies are in conflict. The reason is that, while the prerequisite role policy requires the assignment of two roles to the same user (in a certain order), the SSoDCR policy prohibits this assignment. This situation can be avoided by not specifying prerequisite role policies and SSoDCR policies for the same subset of roles. This conflict is reported as an error. The conflict between the prerequisite permission policy and the SSoDCP one is defined in a similar way.

Prerequisite role and Role hierarchy policies. Let PR be the set of roles in a prerequisite role policy, and RH be the set $\{r\} \cup juniors(r)$ in a role hierarchy policy, where $junior()$ is a function that returns the junior roles of its argument. If $PR \subseteq RH$, the prerequisite role and the role hierarchy policies will require the assignment of the same subset of roles. Hence there is no need to define a prerequisite policy between a role and its parent role. This conflict is reported as a warning. The conflict between the prerequisite permission policy and the permission hierarchy one is defined similarly.

Cardinality (role-to-user assignment) and Role hierarchy policies. Let n be the number of juniors of role r in a role hierarchy policy, and $maxRoles$ be the maximum number of roles that can be assigned to a user, as specified by a cardinality policy. If $n \geq maxRoles$, the cardinality policy will be violated. This situation can be avoided

by having *maxRoles* greater than the number of juniors of any role. This conflict is reported as an error. The conflict between the cardinality (role-to-permission assignment) policy and the permission hierarchy one is defined similarly.

Cardinality (permission-to-role assignment) and Binding of duty policies. Let n be the number of bounded permissions in BoD policy, and *maxPerm* be the maximum number of permissions that can be assigned to a role, as specified by a cardinality policy. If $n > maxPerm$, the cardinality policy will be violated, because the BoD policy will require a role to be assigned to more than *maxPerm* permissions. This situation can be avoided by having *maxPerm* be equal or greater than the number of bounded permissions in a BoD policy. This conflict is reported as an error.

Role hierarchy and SSoD on conflicting roles policies. Let RH be the set $\{r\} \cup juniors(r)$ in a role hierarchy policy, where $junior()$ is a function that returns the junior roles of its argument; let SCR be the set of conflicting roles in an SSoDCR policy. If $|RH \cap SCR| > 1$ the two policies are in conflict. Indeed, while the role hierarchy policy requires the assignment of a set of roles, the SSoDCR policy prohibits this assignment. To avoid this situation an SSoDCR policy should not contain a role and its junior(s) or, similarly, two juniors of the same role. This conflict is reported as an error. The conflict between the permission hierarchy policy and the SSoDCP one is defined similarly.

Role hierarchy and Context (role unassignment) policies. Let JRH be the set containing the juniors of role r. If a context policy on role un-assignment is specified for any role $s \in JRH$, the role hierarchy policy will be violated. Indeed, while the role hierarchy requires the assignment of a junior of role r, the role context policy can prohibit this assignment. This conflict is reported as an error. The conflict between the permission hierarchy and context-based (permission assignment) policies is defined similarly.

SSoD and DSoD on conflicting roles policies. Let SCR and DCR be the sets of, respectively, conflicting roles in an SSoDCR policy and a DSoDCR one. If $|SCR \cap DCR| > 1$, the assignment of at least two conflicting roles will be allowed by the DSoDCR policy but forbidden by the SSoDCR policy, generating an inconsistency in the system. This conflict is reported as a warning. The conflict between the SSoD and DSoD on conflicting users (or permission) policies is defined similarly. Notice that an SSoDCU policy and a DSoDCU one with the same list of users on different roles are not conflicting.

SSoD on conflicting permissions and Binding of duty policies. Let SCP be the set of conflicting permissions in an SSoDCP policy and let $PBoD$ be the set of bounded permissions in a BoD policy. If $|SCP \cap PBoD| > 1$, the two policies are in conflict. Indeed, while the SSoDCP restricts the assignment of at least two conflicting permissions, the BoD policy requires this assignment. To avoid this situation, an SSoDCP policy should not contain permissions that are used in a BoD policy. This conflict is reported as an error.

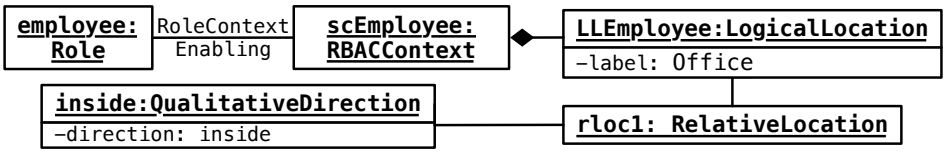


Figure 4.2: A fragment of an instance of the GemRBAC+CTX model

Delegation and SSoD on conflicting roles policies. Let *SCR* be the set of conflicting roles in an SSoDCR policy, r be the role being delegated, and *RECR* be the set of roles that will receive the delegation in a delegation policy. If $(\{r\} \cup RECR) \subseteq SCR$, the two policies are in conflict. The reason is that, while the delegation policy allows the assignment of a set of roles to the same user, the SSoDCR policy prohibits this assignment. This conflict is reported as an error.

Additional checks. The editor also detects overlapping intervals in policies with temporal context, and circular dependencies for role hierarchy and precedence policies.

4.4 Semantics

As shown in chapter 3, the GemRBAC+CTX model, which is the conceptual RBAC model on top of which GemRBAC-DSL has been designed, comes with an operationalization of the semantics of the policies it supports. Since the GemRBAC+CTX model and GemRBAC-DSL have the same expressiveness, we can define the semantics of GemRBAC-DSL by mapping its constructs to the corresponding OCL constraints defined for the GemRBAC+CTX model.

Each entity in the ⟨*preamble*⟩ of a GemRBAC-DSL specification corresponds to an instance of a UML class in the GemRBAC+CTX model: users, roles, permissions, operations, and logical locations (⟨*geofences*⟩) are mapped to instances of the homonymous classes in GemRBAC+CTX. Similarly, role and permission hierarchies correspond to the homonymous associations in the GemRBAC+CTX model.

Each type of RBAC policy is mapped to the corresponding OCL constraint template defined in the GemRBAC+CTX model; in each template the symbolic parameters are replaced with the actual entities used in the specification. For instance, the semantics of the object-based DSoD policy

```
objDSoD: conflicting-roles-activation author, reviewer on-same-object;
```

can be defined by the OCL invariant `DSoDCR` of the class `Session` (introduced on page 27), by replacing the parameters `r1` and `r2` with roles `author` and `reviewer`.

Regarding contextual policies, the context to be assigned/enabled (as prescribed by the policy) is represented in the GemRBAC+CTX model, as an association with the corresponding role/permission. For example, consider the policy

```
loc: role-context enable employee only @location inside office;
```

which enables role *employee* only inside the logical location denoted by the label "office". Figure 4.2 depicts an excerpt of an instance of the GemRBAC+CTX model in which role `employee` is associated to a `SpatialContext` object that contains the object *LLEmployee* of type `LogicalLocation`, which denotes the location "office". This object is associated with object *rloc1* of type `RelativeLocation`, which contains a `QualitativeDirection`. The policy `loc` can be mapped to the OCL invariant `relativeLocationRoleEnabling` of class `Session` (introduced on page 40), parametrized with role `employee`.

Expressing the semantics of GemRBAC-DSL policies as OCL constraints on the GemRBAC+CTX model enables the users of the language to benefit from the model-driven policy enforcement mechanisms as we will show in chapter 6. Briefly, making an access decision for a policy can be reduced to checking the corresponding OCL constraint on a instance of the GemRBAC+CTX model, which represents a snapshot of the system at a certain time.

Table 4.1: Mapping of GemRBAC-DSL constructs to OCL constraints on the GemRBAC+CTX model

Type of policy	**OCL constraint**	**ref**
⟨*PrereqRole*⟩	`context User inv PreqRole`	page 24
⟨*PrereqPermission*⟩	`context Role inv PreqPermisssion`	page 24
⟨*CardActivation*⟩	`context Session inv CardinalityActivation`	page 24
⟨*CardUser*⟩	`context User inv CardinalityRole`	page 24
⟨*CardPermission*⟩	This policy is expressed in a similar way as the previous one by replacing the context of `User` with the context of `Permission`.	page 24
⟨*CardRoletoPermission*⟩	`context Role inv CardinalityPermission`	page 24
⟨*CardRoletoUser*⟩	This policy is expressed in a similar way as the previous one by replacing the instances of `permissions` with instances of `users`.	page 24
⟨*PrecEnabling*⟩	`context inv RoleEnablingPrecedence`	page 25
Dependency ⟨*PrecEnabling*⟩	`context Session inv RoleActivationDependency`	page 25
⟨*RoleHierarchy*⟩	`context User inv RoleHierarchy`	page 26
⟨*PermissionHierarchy*⟩	`context Role inv RoleHierarchy`	page 26
⟨*SSoDCU*⟩	`context Role inv SSoDCU`	page 26
⟨*SSoDCR*⟩	`context User inv SSoDCR` `context Role inv SSoDCP2`	page 26
⟨*SSoDCR*⟩	`context User inv SSoDCR` `context Role inv SSoDCP2`	page 26
⟨*SSoDCP*⟩	`context Role inv SSoDCP1`	page 26
⟨*DSoDCR*⟩	`context Session inv DSoDCR`	page 27
⟨*DSoDCU*⟩	`context Role inv DSoDCU`	web1

continued on next page

⟨*DSoDCP*⟩	`context Role inv DSoDCP`	web1
⟨*DSoDCR*⟩	`context Session inv ObjectDSOD`	page 27
⟨*DSoDCR*⟩	`context Session inv OperationalDSoD`	page 28
⟨*DSoDCR*⟩	`context Session inv HistoryDSOD`	page 29
Role-based ⟨*BoD*⟩	`context Session inv RoleBoD`	page 29
Subject-based ⟨*BoD*⟩	`context Session inv SubjectBoD`	page 30

⟨*Delegation*⟩	`context Delegation inv TotalDelegation`
	`context Delegation inv MultiStepDelegation`
	`context delegation inv PartialDelegation`
	`context Delegation inv StrongTransfer`
	`context Delegation inv StaticWeakTransfer`
	`context Delegation inv DynamicWeakTransfer`
	`context Delegation inv AutomaticRevocation`
		page 32

⟨*Revocation*⟩	`context Delegation inv RevacationDependency`
	`context Delegation inv StrongRevocation`
	`context Delegation inv CascadingRevocation`
		page 33

TPA with ⟨*absoluteTime*⟩	`context Session inv AbsoluteBTIRoleEnab`
	`context Permission inv AbsoluteBTIPermAssign`
	`context Role inv AbsoluteTPRoleAssign`
	`context Role inv AbsoluteUBIRoleAssign`
		page 35
		web2

TPA with ⟨*periodicTime*⟩	`context Role inv periodicUnboundTIRoleAssign`	page 38
TPA with ⟨*activeDuration*⟩	`context Session inv DurationAbsoluteBTIRoleEnab`	page 38
TPRInd ⟨*sDayOfWeek*⟩	`context Role inv indexRoleAssign`	page 37
TPRH ⟨*iHour*⟩	`context Role inv RelativeHoursRoleAssign`	web2
TPRDM ⟨*dayOfMonthH*⟩	`context Role inv DayOfMonthHoursRoleAssign` `context Permission inv DayOfMonthHoursPermAssign`	web2
TPRDW ⟨*dayOfWeekH*⟩	`context Permission inv DayOfWeekHourPermAssign`	page 36
TPRMD ⟨*monthDayOfWeekH*⟩	`context Role inv MonthDayOfWeekHourRoleAssign`	web2

continued on next page

TPCT ⟨*compositeTime*⟩	This policy can be checked by a logical conjunction of two temporal policies: one with absolute time and one with relative time.	
SPP ⟨*physicalLocation*⟩	`context Role inv physicalLocationRoleAssign`	page 39
SPL ⟨*geofence*⟩	`context Session inv logicalLocationRoleEnabling`	page 40
SPR ⟨*relativeLocation*⟩	`context Session inv relativeLocationRoleEnabling`	page 40
SPT⟨*SpatioTemporal*⟩	This policy can be checked by a logical conjunction of the spatial and temporal policies.	

Legend. TP: temporal policy; TPA: TP with absolute time; TPR: TP with relative time; TPRInd: TPR containing an index; TPRH: TPR of type hour interval; TPRDM: temporal policy with a relative time of type day of month that optionally overlays hours; TPRDW: TPR of type day of week that optionally overlays hours; TPRMD: TPR of type day of month that optionally overlays days of week (the days of week may optionally overlay hours); TPCT: TP with composite time; SP: spatial policy; SPP: SP with a physical location; SPL: SP with a logical location; SPR: SP with a relative location; SPT: spatio-temporal policy.

Table 4.1 describes the mapping of each RBAC policy supported by GemRBAC-DSL to its corresponding OCL constraint(s) defined on the GemRBAC+CTX model. The first column indicates the type of policy and the corresponding grammar rule. The second column denotes the corresponding OCL constraints, whose full definition can be found in the reference indicated in the third column. The reference "web" and "web2" are the websites `https://github.com/AmeniBF/GemRBAC-model` and `https://github.com/AmeniBF/GemRBAC-CTX-model.git`, respectively.

4.5 Discussion

Adoption. GemRBAC-DSL has been used by our industrial partner for the specification of the RBAC policies of a production-grade Web application. The adoption of GemRBAC-DSL has allowed its engineers to easily specify all the policies for their system, including 19 new types of contextual policies. Despite the fact that some constructs of the language are non-trivial, the engineers were able to use GemRBAC-DSL confidently after three half-day training sessions.

Limitations and Design Trade-offs. GemRBAC-DSL can express *all and only* the types of policies supported by its underlying model, GemRBAC+CTX. Since GemRBAC+CTX is quite an expressive model, GemRBAC-DSL includes many constructs that could have increased its level of complexity, hindering its adoption. Designing a simpler language would have implied providing limited support in terms of policy types, leading to partial fulfillment of our expressiveness requirements and a limited improvement over the state of the art. Hence, at the language

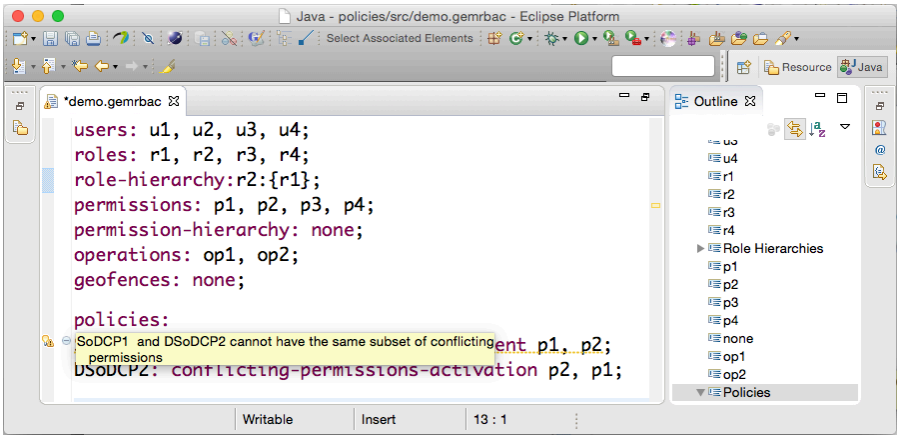


Figure 4.3: The GemRBAC-DSL editor

design stage, we decided to pursue our expressiveness requirements, and to provide a syntax close to natural language to favor the adoption among practitioners and compensate (also by means of a feature-rich editor) for the complexity of the language.

4.6 Tool Support

The GemRBAC-DSL editor (**GemRBAC-DSL-Editor**) has been implemented as an Eclipse plugin. We used Xtext 2.8 to define the textual syntax and the semantic checks (illustrated in section 4.3) for the language. As can be seen in figure 4.1, the editor supports syntax highlighting and conflict detection. Furthermore it performs also syntactic checks, such as detecting duplicated items in lists, or verifying that the identifiers of the entities (e.g., roles, users) used in the policies have been declared in the preamble. The editor is publicly available at https://github.com/AmeniBF/GemRBAC-DSL.git. Moreover, the language semantics has been implemented as a model transformation tool **GemRBAC-DSL-Transform** using Eclipse Epsilon [46]. The **GemRBAC-DSL-Transform** takes as input RBAC policies expressed in GemRBAC-DSL and produces as output the corresponding OCL constraints, and a set of operations (e.g., add hierarchy, add context ...). These operations will be applied to the GemRBAC+CTX instance at deployment time. The model transformation tool is publicly available at https://github.com/AmeniBF/GemRBAC-DSL-Transform.

4.7 Summary

In this chapter we presented GemRBAC-DSL, a domain-specific language that facilitates the specification and consistency checking of policies based on highly-expressive RBAC models. GemRBAC-DSL supports all types of policies captured by the GemRBAC+CTX model, a comprehensive model encompassing all proposed types of policies. We have shown how the language can be used to specify the RBAC policies of an industrial application with complex, context-aware policies. The semantics of GemRBAC-DSL has been defined with a mapping to an existing OCL formalization of the RBAC policies supported by GemRBAC+CTX. This mapping paves the way for automating the enforcement of policy specifications written in GemRBAC-DSL, using a model-driven approach.

Chapter 5

Modeling Access Control Policies: Related Work

This chapter provides an overview of the state of the art related to modeling RBAC policies. Section 5.1 illustrates existing extensions to the original RBAC model. Section 5.2 provides existing work dealing with the formalization of RBAC policies as OCL constraints. Finally, section 5.3 reviews existing work related to policy specification languages.

5.1 RBAC Model Extensions

The original RBAC model, introduced in chapter 2 (section 2.1), was proposed by Sandhu et al. in [3] and is commonly referred to as RBAC96. It is actually defined as a family of reference models: RBAC0, RBAC1, and RBAC2. The basic model, RBAC0, is composed of users, permissions, sessions, and assignment and activation relations. The other two models are defined incrementally over RBAC0 by adding concepts to it: RBAC1 adds the concept of role hierarchy while RBAC2 adds authorization constraints.

Several researchers extended RBAC96 to support additional type of policies such as delegation and context. In the rest of this section, we review some of these works.

RDM2000 [5] extends RBAC96 to support role delegation. It includes two types of user-role assignment: in addition to the original user assignment defined in RBAC96, RDM2000 proposes the *delegated user assignment*, which maps a user to a delegated role. A limitation of this work is that it only supports *total* delegation; this means that a user is not allowed to delegate a subset of her assigned permissions. Another extension related to the concept of delegation is PDM, permission-based delegation authorization model, proposed in [6]. In this model, if a user wants to delegate a set of permissions, she has to create a new role with the required permissions; this role can inherit from any original role. A limitation shared both by RBDM2000 and by PDM is that they only support grant

delegation; a complete delegation model supporting both grant and transfer operations has been proposed by Crampton et al. [7]. However, the models provided by RBDM2000, by PDM, and by Crampton et al. do not support the various types of revocation policies presented in our taxonomy (see section 2.2 of chapter 5). Sohr et al. [8] extends their previous model introduced in [38] to support the role delegation properties of RDM2000; this extension supports the various types of revocation.

Regarding temporal policies, the first temporal RBAC model, TRBAC, was proposed by Bertino et al. in [9]. TRBAC introduces periodic policies on role enabling and disabling, which define the period of time (delimited by two time points) during which a role can be enabled/disabled. TRBAC also supports the definition of dependencies among the enable/disable actions of roles; for instance one can require that role r_1 must be enabled whenever role r_2 is disabled. While TRBAC supports temporal policies only for role enabling/disabling, there is a more general version called GTRBAC [11, 47] that supports temporal policies also on role activation and assignment. Moreover, GTRBAC adds support for temporal aspects in role hierarchy, cardinality, dependency, and SoD policies. The GTRBAC was extended in [48] to support delegation by adding the delegation properties supported in PDM. A limitation of these models is the lack of support for temporal policies at the permission level.

The first model to support location-based constraint has been GEO-RBAC [10]. It introduces the concept of *spatial role*, which consists of a role and its corresponding region, i.e., the area or the place in which the role can be enabled. In this model, each user is associated to a real position, obtained by a positioning device, and a logical position, which is the mapping of the user's real position into a region of the system. Role enabling is conditioned by the logical position of a user, which should match the one specified in the spatial role definition. This means that the list of enabled roles evolves according to the user's position. An administrative model for GEO-RBAC, called GEO-RBAC Admin was proposed in [49]; in this model an administrative role is defined as a spatial role. While GEO-RBAC supports location-based policies only for role enabling (and consequently for role activation), the LRBAC [50] and SRBAC [51] models extend these policies also to user-role and role-permission assignments. However, these models do not support spatial policies for permission enabling and have limited support for role disabling.

Other works support full context-based policies by combining spatial and temporal information. The LoTRBAC model [52] extends GTRBAC by assigning a location to each user, role and permission. In this model, location and temporal information control role enabling and activation. LoTRBAC does not support permission enabling and the specification of the perimeter of physical locations. STARBAC [12], with support for contextual policies for role enabling. ESTARBAC [53] extends STARBAC to support contextual policies for role enabling and both user-role and role-permission assignment.

Table 5.1: Support of non-contextual policies in the various RBAC models

	Prq	RH	Card	SoD				
				S	D	Obj	Op	His
RBAC96 [3]	+	+	+	+	+	-	-	-
RDM2000 [5]	+	+	+	+	+	-	-	-
PDM [6]	N/A	+	+	+	+	-	-	-
Crampton et al. [7]	N/A	+	+	+	+	-	-	-
Sohr et al. [8, 38]	+	+	+	+	+	+	+	+
TRBAC [9]	N/A	-	+	-	-	-	-	-
GTRBAC [11]	N/A	time	+	time	time	-	-	-
GEO-RBAC [10]	N/A	+	N/A	+	+	-	-	-
LRBAC [50]	N/A	-	+	-	-	-	-	-
LoTRBAC [52]	N/A	+	+	+	+	-	-	-
STRBAC [13]	N/A	context	N/A	context	context	-	-	-
GSTRBAC [54]	+	context	+	context	context	-	-	
SRBAC [51]	N/A	-	N/A	location	location	-	-	-
STARBAC [12]	-	-	-	-	-	-	-	-
ESTARBAC [53]	-	+	-	+	+	-	-	-
OrBAC [55]	+	+	+	+	+	-	-	-
GemRBAC+CTX	+	+	+	+	+	+	+	+

	Delegation			Precedence	Revocation			BoD
	G/Tr	S/M	P/T		Dom	Pr	Dep	
RBAC96 [3]	-	-	-	-	-	-	-	-
RDM2000 [5]	G	+	T	-	-	-	-	-
PDM [6]	G	+	+	-	-	-	-	-
Crampton et al. [7]	+	+	+	-	-	-	-	-
Sohr et al. [8, 38]	G	+	T	-	+	+	+	-
TRBAC [9]	-	-	-	+	-	-	-	-
GTRBAC [11]	G	+	+	+	-	-	-	-
GEO-RBAC [10]	-	-	-	-	-	-	-	-
LRBAC [50]	-	-	-	-	-	-	-	-
LoTRBAC [52]	-	-	-	-	-	-	-	-
STRBAC [13]	G	-	+	-	-	-	-	-
GSTRBAC [54]	-	-	-	-	-	-	-	-
SRBAC [51]	-	-	-	-	-	-	-	-
STARBAC [12]	-	-	-	-	-	-	-	-
ESTARBAC [53]	-	-	-	-	-	-	-	-
OrBAC [55]	-	-	-	-	-	-	-	-
GemRBAC+CTX	+	+	+	+	+	+	+	+

Legend. Prq: Prerequisite; RH: Role Hierarchy; Card: Cardinality; S: Static SoD; D: Dynamic SoD; Obj: Object-based DSoD; Op: Operational-based DSoD, His: History-based DSoD; G/Tr: Grant/Transfer delegation; S/M: Single/Multi-step delegation; P/T: partial/total delegation; Dom: dominance; Pr: propagation; Dep: Dependency; BoD: Binding of duty;

Another model, STRBAC [13], allows for defining location- and time-based

Table 5.2: Support of contextual policies in the various RBAC models

	Contextual policies								Scope				Decision
	ART	PTE	I	AD	PL	LL	RL	CP	PA	PE	RA	RE	algorithm
TRBAC [9]	+	+	+	-	-	-	-	N/A	-	-	-	+	+
GTRBAC [11]	+	+	+	+	-	-	-	N/A	-	-	+	+	+
GeoRBAC [10]	-	-	-	-	+	+	-	N/A	-	-	-	+	+
LRBAC [50]	-	-	-	-	+	+	-	N/A	+	-	+	+	+
SRBAC [51]	-	-	-	-	+	+	-	N/A	+	-	+	+	-
STARBAC [12]	+	+	-	-	+	+	-	+	-	+	-	+	-
ESTRBAC [53]	+	+	+	-	+	+	-	+	+	-	+	+	+
STRBAC [13]	+	+	-	-	+	+	-	-	+	-	+	+	+
GSTRBAC [54]	+	+	-	-	+	+	-	+	+	-	+	+	-
OrBAC [55]	+	+	-	-	+	+	-	+	-	+	-	+	+
LotRBAC [52]	+	+	+	+	+	+	+	+	+	-	+	+	-
GemRBAC+CTX	+	+	+	+	+	+	+	+	+	+	+	+	+

Legend. ART: Absolute and Relative TE; PTE: Periodic TE; I: Index; AD: Activation Duration; PL: Physical Location; LL: Logical Location; RL: Relative Location; CP: Composite context-based policies, RA: User-role Assignment, RE: Role Enabling, PA: Role-permission Assignment, PE: Permission Enabling.

policies related to role-to-user and role-to-permission assignments. This may result in a limited subset of permissions allowed for a certain role, at a given time in a given place. Furthermore, STRBAC supports location- and/or time-based policies also for role hierarchy and separation of duty. STRBAC was extended in [56] to support delegation policies based on contextual information. In STRBAC, a change in a spatio-temporal constraint may lead to the addition of a new role. Moreover, STRBAC does not support the definition of composite context-based policies. This limitation is overcome by the GSTRBAC model [54, 57] which lifts this requirement by introducing the concept of spatio-temporal zone (*st-zone*). An *st-zone* is an RBAC entity abstracting a location and a time, and is assigned to other entities like users, roles, permissions, and resources. In this mode, a role is enabled if its *st-zone* matches the one of the user; similarly, a permission is enabled if its *st-zone* matches the one of the corresponding resource.

Table 5.1 shows to which extent the various models discussed in this section support the RBAC non-contextual policies included in the taxonomy presented in section 2.2. As one can notice, the GemRBAC+CTX is the only one that provides a complete support for non-contextual policies. Table 5.2 summarizes to which extent the RBAC models discussed above support the various concepts related to contextual policies. It also indicates the scope (assignment/enabling of permissions and/or roles) in which such policies can be used, the availability of an access decision algorithm. As one can see, the GemRBAC+CTX model is the only one that supports 1) all the various spatio-temporal concepts for contextual policies; 2) the use of such policies for the assignment and enabling of both roles and permissions; 3) a checking procedure for these policies.

We also remark that none of the existing RBAC models would be able to express the application presented in chapter 3 in section 3.3. More specifically, models RBAC96, RDM2000, PDM, Crampton et al., Sohr et al., TRBAC, GTRBAC, GEO-RBAC and LRBAC lack full support of context-based policies. Consequently, policies **PL1**, **PL2** and **PL3** could not be defined in these models. Although models LoTRBAC, STRBAC, ESTARBAC and GSTRBAC cover time-based and location-based policies, they do not support dependency and transfer policies. Consequently, policies **PL4** and **PL5** could not be defined in these models.

5.2 Using OCL for Modeling RBAC Policies

There have been several proposals for using OCL for the formalization of RBAC policies [8, 22, 33, 58, 59, 60, 61]; however, the types of policies considered in each of these formalizations are a subset of the ones presented in this work. In terms of model-based approaches for RBAC, SecureUML [62] is a modeling language for the model-driven development of secure systems, based on RBAC; it extends the original RBAC model to support authorization constraints, which are preconditions expressed in OCL, associated with operations that access system resources. In reference [63], authors present an RBAC model which combines SecureUML and ComponentUML. The latter is a UML-based language for modeling system entities and relationships between them. Authorization constraints are defined as OCL queries such as '*are there actions on concrete resources that every user can perform in the given scenario?*'. Similarly to our work, OCL queries are evaluated on the model instance which is a snapshot of the system state. However, we analyze the OCL constraints from a user's request point of view in order to make the access decision. The model-driven security approach proposed in [64] builds a security-aware graphical user interface model from the security model presented in [63] and a graphical user interface model. The goal is to automatically generate the graphical user interface application. Reference [65] shows how to incorporate RBAC policies into UML design models using UML diagram templates, but only supports role hierarchy and static and dynamic separation of duty constraints.

Our approach fills the gap between the existing OCL-based formalizations of the RBAC policies and the various types of policies proposed in the literature, by formalizing *all* the types of policies classified in our taxonomy as OCL constraints on the GemRBAC+CTX model.

5.3 RBAC Policy Specification Languages

One of the first policy languages proposed for RBAC is RCL2000 [19], which is a formal language based on first-order predicate logic and defined on top of the RBAC96 model. The language supports only role hierarchy and separation of duty policies. FORBAC [21] is also an extension of RBAC based on first-order logic.

Table 5.3: Support of policies in RBAC languages

	Prq	RH	Card	Prec	SoD					BoD	Context		Deleg	Rev
					S	D	Obj	Op	His		T	L		
RCL2000 [19]	-	+	+	-	+	+	-	-	-	-	-	-	-	-
FORBAC [21]	+	-	-	-	-	-	-	-	-	-	+	+	-	-
Tower [20]	+	+	+	+	+	+	+	+	+	-	-	-	+/-	+/-
XACML [15, 16]	+	+	+	-	+	+	-	-	-	-	+	+	GT	-
X-RBAC [66]	+	+	+	-	+	+	+	-	-	-	+	+	-	-
X-GTRBAC [67]	+	+	+	-	+	+	+	-	-	-	-	+	-	-
ROWLBAC [17]	+	+	+	-	+	+	+	-	-	-	-	-	GT	-
XACML+OWL [18]	+	+	+	+	+	+	+	+	+	-	-	-	-	-
RBAC-DSL [8]	+	+	+	+	+	+	+	+	+	-	-	-	GT	+

Legend. Prq: Prerequisite; RH: Role Hierarchy; Card: Cardinality; Prec: Precedence and Dependency; S: Static SoD; D: Dynamic SoD; Obj: Object-based DSoD; Op: Operational-based DSoD, His: History-based DSoD, Deleg: Delegation.

It adds support for attributes in policies and numeric constraints; both features enable the definition of more complex policies, like those containing contextual constraints. However, FORBAC does not support role hierarchy, delegation, cardinality, and separation of duty. Furthermore, a limitation shared both by RCL2000 and FORBAC is the difficulty of use by practitioners, since both languages require a strong mathematical background. Tower [20] is a high-level specification language for access control policies; it supports delegation and history-based SoD policies. However, delegation and revocation policies are defined only as administrative operations for role-to-user assignment, i.e., in terms of adding/removing a role to/from a user.

Another research stream considers XML-based languages, starting from the definition of XACML (eXtensible Access Control Markup Language) [14]. XACML is a language for access control, standardized by the OASIS community. The XACML standard provides not only the specification language for access control policies but also a reference enforcement architecture. XACML is a general-purpose language for expressing various types of access control models and policies; being general-purpose, it does not support RBAC natively (e.g., sessions are not supported). RBAC support can be added to XACML by means of profiles. The OASIS RBAC profile for XACML [16] supports only role hierarchy and static separation of duty policies. Another RBAC profile of XACML [15] supports separation of duty, delegation, and context-based policies. X-RBAC [66] is an XML-based specification language for RBAC policies in multi-domain environments where authorization policies are distributed over several domains. X-RBAC supports context-based, role hierarchy, cardinality and separation of duty policies. X-GTRBAC [67] is a language defined on top of the GTRBAC model [11] for specifying RBAC policies for heterogeneous and distributed enterprise resources. X-GTRBAC adds the concept of user's credentials to the GTRBAC model: users are grouped according to their credentials. X-GTRBAC supports cardinality, separation of duty, role

hierarchy, and temporal policies.

Another language, conceptually similar to XACML, is xfACL (eXtensible Functional Language for Access Control) [68]. xFACL is a general-purpose access control language, which tries to combine the benefits of XACML and RBAC. It is based on the specification of attributes for entities involved in decisions (e.g., users, operations) and supports auxiliary policies to extend its expressiveness. The latter is also its main drawback, since support for each type of policy has to be manually added by means of an auxiliary function.

Other languages deal with the integration of ontologies to provide a semantic interpretation of access control policies across different, heterogenous organizations, and to support advanced access control policies. For instance, ROWLBAC [17] is an ontology-based language that combines OWL (Web Ontology Language) and RBAC properties. The language supports the specification of prerequisite, role hierarchy, SoD, and delegation policies. The XACML+OWL framework [18] combines OWL and XACML. Role hierarchy and separation of duty policies are specified using OWL, while the XACML engine is used to make decisions for user access requests. The interactions between the XACML engine and the OWL ontology are defined through semantic functions.

RBAC DSL [8] is a domain-specific language for RBAC based on UML diagrams and OCL constraints. The corresponding meta-model includes two levels: the *policy* level and the *user Access* Level. The first level defines the basic RBAC concepts: roles, resources, permissions and operations. At this level, SoD, cardinality, and role hierarchy policies are represented as UML attributes and associations. The second level defines the concepts of user, session, resource access, and snapshot (i.e., an instance of an RBAC model at a specific time point). A predecessor/ successor relation is defined for the concept of user, session and access to identify the individual users, sessions and accesses over time. At this level, authorization policies are defined as OCL constraints based on the information available in the policy level. RBAC DSL supports also delegation and revocation policies. However, as acknowledged also in chapter 3, defining RBAC policies as OCL constraints can be difficult, since it requires a high level of knowledge and expertise with OCL, especially in our case in which OCL constraints tend to be rather complex to express RBAC policies.

Table 5.3 summarizes the support for the various types of RBAC policies in the policy specification languages discussed above. The types of policies used for the comparison have been taken from the taxonomy in chapter 2 and reflect the ones we have observed in our industrial case study. We remark that the specification of some types of policies, such as context-based and delegation, depends not only on the language but also on the underlying model.

One can see that none of these languages is expressive enough to express all the policies presented in Section 4.1, related to our industrial case study. Moreover, the analysis has also shown that the majority of existing policy specification languages

is based on some formalism (either first-order logic fragments, including OCL, or ontology languages based on description logic) that require a strong theoretical and mathematical background, which is rarely found among practitioners. Hence, we contend that there is a need for an expressive specification language for RBAC policies like GemRBAC-DSL that can also be used by practitioners.

Part III

Enforcement of Role-based Access Control Policies

Chapter 6

Model-driven Enforcement of RBAC policies

In the previous part, we presented the GemRBAC+CTX model and its corresponding language GemRBAC-DSL to facilitate the specification of RBAC policies. In this chapter, we show how to enforce these policies to prevent unauthorized user access. Our goal is to propose a model-driven approach for the enforcement of policies written in GemRBAC-DSL. More specifically, we leverage the operationalization through OCL constraints of GemRBAC-DSL policies to reduce the problem of enforcing RBAC policies to the evaluation of the corresponding OCL constraints on an instance of the GemRBAC+CTX model; this instance captures the state of the system (from the point of view of access control) at the time an access request is made.

This chapter makes the following contributions:

1. a model-driven framework for runtime enforcement of GemRBAC-DSL policies;
2. an extensive empirical evaluation on a real industrial system to assess the scalability and the performance of the proposed approach.

The rest of the chapter is organized as follows. Section 6.1 presents an overview of the model-driven approach for enforcing RBAC policies. Section 6.2 describes the integration of the proposed approach into the architecture of a Web application. Section 6.3 presents the results of the empirical evaluation. Finally, section 6.4 provides a review of existing enforcement approaches.

6.1 Model-driven Run-time Enforcement

As mentioned above, the idea at the basis of our model-driven enforcement framework is to map the GemRBAC-DSL policies to a set of OCL constraints (and a list of operations); this transformation is detailed in chapter 4. We assume to have access to a snapshot of the system state, represented as an instance of the GemRBAC+CTX model. At the beginning of the operations of the system, the initial instance contains the RBAC components (i.e., users, roles and permissions)

and their assignment relations as defined by the access control configuration of the system; the instance is then updated as the system configuration evolves from the point of view of access control. We mention that static checking of RBAC policies is out of the scope of this thesis, i.e., we assume that both the model instance and its corresponding OCL constraints are valid. In what follows, we will use the term *Snap* to refer to the model instance representing the current system state. Both the GemRBAC+CTX instance and the OCL constraints will serve as input for the enforcement framework. Figure 6.1 depicts an overview of the proposed model-driven approach. The enforcement framework includes two main components: *SnapProcessor* and *OCLChecker*. Once a user sends an access request, the processor *SnapProcessor* takes as input *Snap* and generates *TargetSnap* as output. The snapshot *Snap* captures the RBAC system state at the time when the access request has been made. The generated snapshot captures the next system state, as if the request had been allowed. Both snapshots, *Snap* and *TargetSnap*, are modeled as instances of the GemRBAC+CTX model expressed in UML. After creating *TargetSnap*, *SnapProcessor* selects based on the type and the parameters of the access request (see section 6.1.1), the list of policies to be checked. In this way, making a decision about an AC request is equivalent to verifying whether the *TargetSnap* instance satisfies the OCL constraints corresponding to the access control policies; this check is performed by the *OCLChecker*. If the checked 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 checked constraints evaluate to *false*, it means that allowing the request would violate one or more of the policies in place.

Our approach adopts also the usage control (UCON) [69] concept; the access decision should be reevaluated when a new update, from an access control point of view, occurs at the system state level. Once an AC-related event, such as a user authentication or a user change location, is triggered, the *SnapProcessor* updates the current system state (*Snap*) and enforces the access control policies, accordingly. For instance, upon successful login, the *SnapProcessor* creates a new session for the authenticated user and enables the set of roles assigned to the user within this session. We assume the enforcement framework to receive a notification from an external component whenever an AC-related event is triggered.

In the next subsections we show how we enforce the access control policies when making an access decision for an AC request or updating the system state upon receiving a notification for an AC-related event.

6.1.1 Policies Enforcement Upon Receiving an Access Request

A user can send an AC request to activate a role, to access a resource, to delegate a role, to revoke a delegated role, or to perform an administrative operation such

Algorithm 1: Check AC Request

input : a = AC Request; Snap = current system state; P = list of authorization policies;
output: d = access decision;

```
1  PC ⟵ ∅ ;                                                    // list of policies to check
2  targetSnap ⟵ null ;                                         // the next snap
3  switch a do
4      case role activation do
5          if role r1 is enabled for user u1 then
6              targetSnap ⟵ Snap.activateRole(r1, u1);
7              PC ⟵ selectPL(P, role activation, r1);
8          else
9              d ⟵ false;
10         end
11     end
12     case access to a resource do
13         if role r1 is active in session s1 then
14             p1 ⟵ selectPermission(r1, op1, obj1);
15             if p1 not null and p1 is enabled for role r1 then
16                 assignedPerm ⟵ true;
17                 if role r1 has been delegated to user u1 then
18                     deleg ⟵ selectDelegation(r1,u1 );
19                     if deleg is partial then
20                         assignedPerm ⟵ p1 ∈ delegatedPermissions
21                     end
22                 end
23                 if assignedPerm then
24                     targetSnap ⟵ Snap.addHistory(u1, r1, p1, op1, obj1);
25                     PLC ⟵ selectPolicies(P, resource access, p1);
26                 end
27             else
28                 d ⟵ false;
29             end
30         else
31             d ⟵ false;
32         end
33     end
34     case role delegation do
35         if user u2 is not assigned to role r1 then
36             targetSnap ⟵ Snap.delegate(u1, r1, u2, delegType);
37             PLC ⟵ selectPolicies(P, role delegation, r1);
38         else
39             d ⟵ false;
40         end
41     end
42     case role revocation do
43         targetSnap ⟵ Snap.revoke(u1,r1, d1);
44         PLC ⟵ selectPolicies(P, revocation, r1);
45     end
46     case user-to-role assignment do
47         targetSnap ⟵ Snap.assignUser(u1, r1);
48         PLC ⟵ selectPolicies(P, role assignment, r1);
49     end
50     case role-to-permission assignment do
51         targetSnap ⟵ Snap.assignPermission(p1, r1);
52         PLC ⟵ selectPolicies(P, permission assignment, p1);
53     end
54 end
55 if d is null then d ⟵ targetSnap.check(PLC);
56 if d then  Snap ⟵ targetSnap;
57 return d;
```

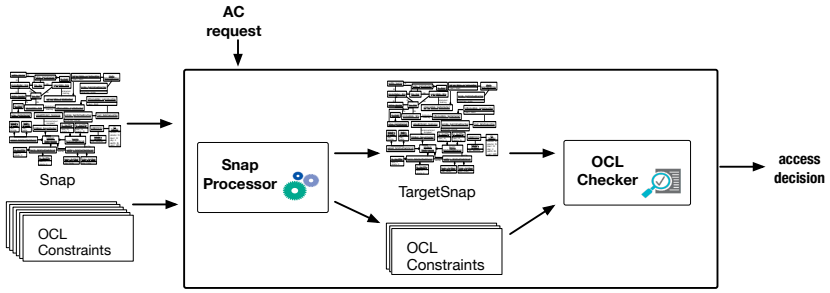


Figure 6.1: An Overview of the model-driven enforcement process in case of AC-request

as assigning a role to a user or to a permission. The procedure for checking an AC request is shown in algorithm 1. This algorithm takes as input an access request `a`, the current system state `Snap` at the time of the request and the list of policies `P` and returns an access decision `d` (allowed/denied).

After initializing variable `PLC`, representing the list of policies to check, to an empty list, and `targetSnap` to `null`, the algorithm goes through a `switch` statement, depending on the type of the access request. In each alternative branch, the algorithm creates `targetSnap` as follows.

Role activation: in this case the request consists of user u_1 sending a request to activate role r_1 (lines 4–11). First, the *SnapProcessor* checks if role r_1 is enabled for user u_1. If it is the case, the *SnapProcessor* builds the *TargetSnap* by calling the `activateRole` operation of the `Session` class. This operation removes role r_1 from the list of enabled roles and adds this role to the list of active roles for user u_1. If a precedence policy is specified for role r_1, the operation `activateRole` enables the list of roles which should be enabled for other users, accordingly. Then, the *SnapProcessor* retrieves list `PLC` from the list of policies `P` by calling operation `selectPolicies` (line 7). The list of policies `PLC` is selected based on the type (*role activation*) and on the parameters (u_1, r_1) of the AC request. Table 6.1 shows the list of policies to check for each type of AC request. In this case, list `PLC` will contain policies of type "cardinality on role activation" and "dynamic separation of duty for conflicting users and roles". To be checked, the parameters of these policies should match at least one of the AC request parameters. For instance, a dynamic separation of duty for conflicting users policy should have user u_1 among the list of conflicting users. If the requested role is disabled, the *SnapProcessor* follows the `else` branch in lines 8–9 and the decision `d` is set to *false*.

Access to a resource: in this case the request consists of user u_1 with role r_1 sending a request to perform operation op_1 on object o_1 (lines 12–33). First, *SnapProcessor* checks if role r_1 is active in the current session of user u_1 (line 13). If role r_1 is not active, the *SnapProcessor* follows the *else* branch in line 30 and the decision `d` is set to *false*. However, if the role is active, *SnapProcessor* selects, among the list of permissions assigned to role r_1, permission p_1 which abstracts operation op_1 and object o_1 (line 14). Then, if the selected permission p_1 is assigned to role

Table 6.1: Checked policies for each type of AC request/event

	Prq	RH	Card		Prec	Dep	SoD						BoD	Context		DelegRev	
			AC	AS			S	DCR	DCU	Obj	Op	His		T	L		
Role activation			✓					✓	✓								
Access to a resource										✓	✓	✓	✓				
Role delegation	✓	✓		✓			✓									✓	
Role revocation	✓	✓		✓													✓
Administrative operation	✓	✓		✓			✓							✓	✓		
User authentication					✓									✓	✓		
User change location															✓		
User disconnection						✓											

Legend. Prq: Prerequisite; RH: Role Hierarchy; Card AC: Cardinality on role activation; Card 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, Deleg: Delegation, Rev: Revocation.

r_1 and is enabled, the *SnapProcessor* checks whether role r_1 has been delegated to user u_1 (line 17). However, if permission p_1 is not assigned to role r_1, the *SnapProcessor* follows the *if* branch at line 27. If the user acquired the role (r_1) through a delegation, the *SnapProcessor* verifies if the delegation is of type *partial* (see section 2.2.7.1 in chapter 2) and if permission p_1 belongs to the set of delegated permissions. If it is the case or if the delegation is of type total, the *SnapProcessor* builds the *TargetSnap* by adding a new instance of type `History` to the current *Snap* (line 24). This instance is obtained by calling the operation `addHistory` which records user u_1 who performed operation op_1 on object o_1 according to permission p_1, while having role r_1. Then, the *SnapProcessor* retrieves list `PLC` (line 25); this is selected based on the type (*access to a resource*) and on the parameters (r_1, p_1, op_1, and o_1) of the AC request. As shown in table 6.1, list `PLC` will contain four types of policies: object-based, operational-based, history-based separation of duty, and binding of duty (BoD). To be checked, the parameters of these policies should match at least one of the AC request parameters. For instance, a BoD policy should have permission p_1 among the list of bounded permissions.

Role delegation: in this case the request consists of user u_1 sending a request to delegate her role r_1 to user u_2 (lines 34–41). First, the *SnapProcessor* checks if user u_2 is already assigned to role r_1. If it is the case, the *SnapProcessor* follows the `else` branch in line 38 and the decision `d` is set to *false*. However, if user u_2 is not assigned to role r_1, the *SnapProcessor* builds the *TargetSnap* by adding role r_1 to the list of delegated roles for user u_2 and creating a new instance of type `Delegation`. The *TargetSnap* is built by calling the `delegate` operation of the `Session` class. Then, the *SnapProcessor* retrieves list `PLC` from the list of policies `P` by calling operation `selectPolicies`. The list of policies `PLC` is selected

based on the type (`role delegation`) and on the parameters (u_1, r_1, r_2) of the AC request. As shown in table 6.1, list `PLC` will contain the prerequisite, role hierarchy, cardinality on role assignment, static separation of duty on conflicting roles and delegation policies. To be checked, the parameters of these policies should match at least one of the AC request parameters. For instance, a prerequisite policy should have role r_1 among the list of roles in its parameters.

Role revocation: in this case the request consists of user u_1 sending a request to revoke delegation d_1 (lines 42–45). We assume user u_2 acquired role r_2 through the delegation d_2. The *SnapProcessor* builds the *TargetSnap* by calling the `revoke` operation in line 43. This delegation removes role r_2 from the list of delegated roles assigned to user u_2, sets the attribute `isRevoked` of delegation d_1 to *true*, and records the revoking user (u_1). Then, the *SnapProcessor* retrieves list `PLC` from the list of policies `P` by calling operation `selectPolicies`. The list of policies `PLC` is selected based on the type (`role revocation`) and on the parameters (u_2, r_2 and d_1) of the AC request. As shown in table 6.1, list `PLC` will contain the prerequisite, role hierarchy, cardinality on role assignment and revocation policies. To be checked, the parameters of these policies should match at least one of the AC request parameters. For instance, a role hierarchy policy should have role r_2 among the list of roles in its parameter.

Administrative operation: in this case the request consists of an admin sending a request to assign role r_1 to user u_1 (lines 46–49), or role r_1 to permission p_1 (lines 50–53). The *SnapProcessor* builds the *TargetSnap* by adding the assignment relation. The role-to-user assignment is obtained by calling the operation `assignUser` in line 47. The role-to-permission assignment is obtained by calling the operation `assignPermission` in line 51. Then, the *SnapProcessor* retrieves list `PLC` from the list of policies `P` by calling operation `selectPolicies`. The list of policies `PLC` is selected based on the type (`role assignment` or `permission assignment`) and on the parameter (r_1 or p_1 respectively) of the AC request. Based on the type of administrative operation, i.e., user-to-role or role-to-permission assignment, list `PLC` will contain policies specified at the role or at the permission level, respectively. As shown in table 6.1, list `PLC` will contain the prerequisite, role hierarchy, cardinality on role assignment, static separation of duty and context-based policies. To be checked, the parameters of these policies should match at least one of the AC request parameters. For instance, a static separation of duty on conflicting roles policy should have role r_1 among the list of conflicting roles.

At the end of the `switch` case, if the decision variable `d` has no value, the algorithms invokes the *OCLChecker*, to check the policies in `PLC` on the *TargetSnap*. The result of invocation of method `check` will contain the access decision. If the access has been granted (line 56), the *SnapProcessor* updates the current *Snap* with the value of *TargetSnap*, to update the system state (from the point of view of access control) with the operation that is about to be performed. At the end, the access decision is returned.

6.1.2 Policy Enforcement upon receiving AC-related Event Enforcement

Whenever an AC-related event is triggered, the model-driven enforcement framework has to update the current system state (*Snap*), and check whether the AC policies are still satisfied. An AC-related event can be of type *user authentication*, *user change location* or *user disconnection*: the *user authentication* event corresponds to the case when a new user is connected; the *user change location* event corresponds to the case when a connected user changes her location; the *user disconnection* event corresponds to the case when a user is disconnected due to network issues. The procedure for checking an access request is shown in algorithm 2. This algorithm takes as input an AC-related event `e`, the current system state `Snap` at the time of the notification and the list of policies `P` and returns an updated snap `USnap`. In what follows we explain the enforcement process for each type of AC-related event.

User authentication: when a user logins correctly, we assume the enforcement framework to receive a notification from an authentication server, which checks the user credentials and allows her to authenticate. This notification has the form $\{u_1,\ s_1,\ loc\}$ where u_1 corresponds to the identifier of the user being authenticated; s_1 corresponds to the token identifier; and *loc* corresponds to the current position of the authenticated user[1]. After receiving the notification, the *SnapProcessor* updates the *Snap* by adding a new session for the authenticated user and updates the user location (line 5). Then, the *SnapProcessor* enables all the roles assigned to user u_1 and retrieves list `PLC` of policies to check from the list of policies `P` (line 6). The `PLC` is selected based on the type of the AC-related event (in this case `user authentication`) and on the set of enabled roles for user u_1 within the created session. Table 6.1 shows the list of policies to be checked for each type of AC-related event. In this case, list `PLC` will contain the precedence and context-based policies.

Then, for each selected policy, the *OCLchecker* verifies whether the policy is satisfied. In case a policy has been violated, the *SnapProcessor* disables its corresponding role (lines 8–12).

User change location: we assume the existence of a *geo-localization* server which keeps track of the user position. Whenever a user changes her location, the *geo-localization* server sends a notification to the enforcement framework. This notification has the form $\{u_1,\ loc_1,\ loc_2\}$ where loc_1 and loc_2 correspond to the previous and to the new position of user u_1, respectively. After receiving the notification for a user change location event, the *SnapProcessor* updates *Snap* by calling operation `updateUserLoc` in line 17. Then, the *SnapProcessor* retrieves list `PLC` of policies to check from the list of policies `P` by calling operation `selectPolicies`.

[1]We assume that the position of the user is obtained by means of a GPS. In case the user's position is not known, the notification will have the form $\{u_1,\ s_1\}$.

The list `PLC` is selected based on the type of the AC-related event (in this case `user change location`). As shown in table 6.1, list `PLC` will contain location-based policies[1]. If a violation has been reported, the *SnapProcessor* updates *Snap* according to the state of the role parameter of the policy (lines 22–26). If the role is enabled (respectively active), the *SnapProcessor* will disable (respectively deactivate) it from all the sessions of user u_1 by calling operation `disable` (respectively `deactivate`) of the `Session` class.

Algorithm 2: Check AC-related Event

```
   input  : e = AC-related event;
            Snap = current system state;
            P = list of policies;
   output: USnap = updated system state;
 1 PLC ⟵ ∅; ;                                          // list of policies to check
 2 USnap ⟵ Snap; ;                                     // updated Sanp
 3 switch e do
 4     case user authentication do
 5         USnap.addSession (u1, s1, loc);
 6         PLC ⟵ selectPolicies(P, user authentication);
 7         USnap.enableAllRoles (u1,s1);
 8         foreach role r in enabledRoles do
 9             foreach policy p in PLC do
10                 if r ∈ p.roles and p is not satisfied then
11                     USnap.disable(r,s1);
12                 end
13             end
14         end
15     end
16     case user change location do
17         USnap.updateUserLoc(u1, loc1, loc2);
18         PLC ⟵ selectPolicies(P, user change location);
19         foreach role r in u1.roles do
20             foreach policy p in PLC do
21                 if r ∈ p.roles and p is not satisfied then
22                     if r is enabled in s1 then
23                         USnap.disable(r, u1)
24                     else
25                         USnap.deactivate(r, u1)
26                     end
27                 end
28             end
29         end
30     end
31     case user disconnection do
32         USnap.removeSession(u1, s1);
33         PLC ⟵ selectPolicies(P, user disconnection);
34         foreach policy p in PLC do
35             if p is not satisfied then
36                 USnap.disable(p.role) from all sessions;
37                 USnap.deactivate(p.role) from all sessions;
38             end
39         end
40     end
41 end
```

User disconnection event: to logout, a user sends a request to authentication server which forwards it to the enforcement mechanism. This request is checked

[1]For simplicity, we only consider location-based policies on role enabling; this type of policy defines the location where a given role should be enabled.

following the same process explained in section 6.1.1. However, a user can be disconnected due to network issues. In this case, we assume the authentication server to send a notification to the enforcement framework to update the current system state. This notification has the form $\{u_1, s_1\}$. After receiving a notification for a user disconnection, the *SnapProcessor* removes session s_1 of the involved user (line 32) and retrieves list `PLC` of policies to check from the list of policies `P` by calling operation `selectPolicies` in line 33. The list `PLC` is selected based on the type of the AC-related event (in this case `user disconnection`). As shown in table 6.1, list `PLC` will contain dependency policies (see chapter 2).

In this case, the *SnapProcessor* updates the *Snap* by disabling and deactivating role r_1 from all sessions in the system (lines 36 and 37).

6.2 Integrating the Enforcement Framework into a Web Application Architecture

We have integrated the enforcement framework into the architecture of a Web application developed by our industrial partner. The system architecture comprises four components: a Web application, a set of micro-services, a geo-localization server, and an authentication server. The Web application allows a user to request access to resources exposed by micro-service(s). The geo-localization server records the user's position. The authentication server allows a user to connect to the Web application based on her identity. Once a user is authenticated, she can send a request to access resources or performs operations, which can be seen as AC request. As mentioned previously, a user request is allowed or denied based on a set of authorization policies. To enable the enforcement of these policies, we implemented the model-driven enforcement approach introduced in section 6.1 as an *authorization server* composed of the *SnapProcessor* and the *OCLChecker*. We also enriched the architecture with a *proxy* which will act as intermediate between the user, the *authorization server* and the set of micro-services. The complete architecture is illustrated in figure 6.2. For simplicity, we consider a system configuration with two micro-services *M1* and *M2*. The enforcement components are:

- an *authorization server* that checks whether an authenticated user can access the resources exposed by the set of micro-services;
- *AC_data* corresponds to local access control data and it includes the *Snap* and the RBAC policies expressed as OCL constraints;
- *proxy* is a gateway service which intercepts user requests, and forwards them to the involved micro-service(s) if the request has been granted.

Although the proposed enforcement architecture has been designed based on the architectural specifications provided by our industrial partner, it can be generalized

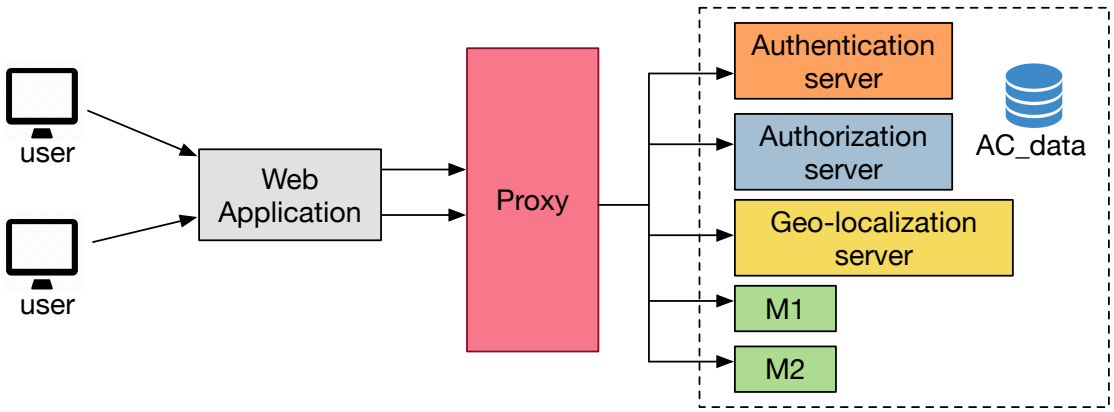


Figure 6.2: An overview of the proposed enforcement architecture

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 [70]; the *authorization server* and the *AC_data* are additional components that can be deployed on any Web application server.

Implementation

The core of the enforcement framework is a component, called MORRO (MOdel-driven fRamework for Run-time of RBAC pOlicies), which includes the *authorization server* and the *proxy*. MORRO has been implemented in Java with a micro-service based architecture using the SpringBoot [71] framework and the ZuuL [72] proxy. The implementation of the *authorization server* makes use of the Eclipse Modeling Framework (EMF) and Eclipse OCL [45]. The *Snap* is expressed as an Ecore [73] model.

6.3 Evaluation

We evaluated the performance of our model-driven run-time enforcement framework MORRO for enforcing RBAC policies. Our goal is to evaluate the time needed to make an access decision, to assess the applicability of the MORRO framework in a real application when considering real AC configurations: we want to show that MORRO can be adopted without considerably impacting on the overall performance (in terms of response time) of a Web application. We also aim to assess how the access decision time in MORRO is affected by the size of the system (from the point of view of access control); i.e., we want to evaluate its scalability under various configurations, as defined by the system parameters.

More specifically, we answer the following research questions:

RQ1: how does MORRO perform on a real industrial system under different scenarios with respect to each type of policy?

RQ2: how does the *authorization server* scale with respect to each type of policy?

RQ3: what is the communication overhead between the *authorization server* and the *proxy* in case of an AC request, 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.

6.3.1 Evaluation Settings

To evaluate our approach we apply it to an industrial application; we deployed MORRO onto a micro-service-based architecture provided by our industrial partner HITEC Luxembourg. This architecture was running on a machine equipped with a Quad-core 8 GHz CPU and 25 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.

Based on the AC configuration of the test application defined by our partner, we consider two types of AC requests, *role activation* and *access to a resource*, and two types of AC-related event, *user authentication* and *user change location*.

To answer **RQ1** (section 6.3.2), we considered instances of the GemRBAC+CTX model based on real configurations provided by our industrial partner. To answer **RQ2** (section 6.3.3) we used synthesized model instances, to evaluate the scalability in terms of various system configuration parameters; this synthesized instances were produced using an internally-developed generator.

6.3.2 Performance On a Real Industrial System

To address **RQ1**, we decompose it into subquestions which take into account different scenarios considering the type of AC requests/events. For each scenario, we assess the performance with respect to 1) *a basic system configuration*, i.e., an AC configuration that is only determined by role assignment and activation relations; 2) configurations that add to the basic system configuration additional policies to be checked (i.e., selected by the *SnapProcessor*), as determined by the type and the parameters of the AC request/event (as described in section 6.1) . The research subquestions are listed below:

How does MORRO perform on a real industrial system:

(RQ1.1) in the context of an AC request of type *access to a resource*:

- with respect to a basic system configuration,
- with respect to a DSoD policy,
- with respect to a BoD policy?

(RQ1.2) in the context of an AC request of type *role activation*:

- with respect to a basic system configuration,

- with respect to a cardinality policy,
- with respect to a DSoD policy?

(RQ1.3) in the context of an AC-related event of type *user authentication*:

- with respect to a basic system configuration,
- with respect to a precedence-based policy,
- with respect to a time-based policy,
- with respect to a location-based policy?

(RQ1.4) in the context of an AC-related event of type *user change location*:

- with respect to a basic system configuration,
- with respect to a location-based policy?

In case of an AC request, we measure the *access decision time* within the *authorization server*, i.e., the time difference from the time the *authorization server* receives an AC request to the time it yields an access decision, for various system configurations. We also measure the *access decision time* within the *proxy* (which will be used further on to answer **RQ3**), which is the time difference from the time the *proxy* receives an AC request from the user until the time the *proxy* receives an access decision from the *authorization server*. In measuring the access decision time, we took into account the fact that it can be affected by various factors related to the Java-based environment on which it runs, such as garbage collection and optimization in the JVM [74]. We also considered the noise introduced by the network-based communication between the *proxy* and the *authorization server*. For these reasons, for each system configuration, we sent ten AC-requests. However, we discarded the value for the first one, since it is affected by the loading time of the run-time libraries. Hence, we report the average value over checking the nine subsequent requests. To keep the same instance over the different runs, we designed the (initial) AC configuration of the system such that the access decision for each request evaluates to *deny*.

In case of an AC-related event, we measure the *execution time* needed to update the current system state (*Snap*), for various system configurations; the execution time is the time difference from the time the *authorization server* receives a notification for an AC-related event until the time it updates the current system state. Differently from the case of an AC request, processing the notifications of AC-related events is only affected by the intrinsic noise due to the Java-based environment, since there is no interaction with the proxy. 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 report the average value over processing the four subsequent notifications.

6.3.2.1 System Configuration

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 partners; the corresponding sanitized, natural-language versions of these policies are:

PL1: A user is not allowed to activate more than three roles at the same time. This policy can be checked using the OCL invariant `CardinalityActivation` of class `Session` introduced on page 24.

PL2: A user can activate either role r_1 or role r_2. This policy can be checked using the OCL invariant `DSoDCR` of class `Session` introduced on page 27.

PL3: A user is allowed to activate roles r_1 and r_2 as long as she does not perform all operations (op_1, op_2) of a business task at the same resource. This policy can be checked using the OCL invariant `DSoDHis` of class `Session` introduced on page 29.

PL4: Role r_1 is enabled only if role r_2 is active. This policy can be checked using the OCL invariant `RoleEnablingPrecedence` of class `Session` introduced on page 25.

PL5: Role r_1 is enabled if the user is located inside *ZoneA*. This policy can be checked using the OCL invariant `logicalLocationRoleAssign` of class `Session` introduced on page 40.

PL6: Permissions p_1 and p_2 should be performed by the same subject[1]. This policy can be checked using the OCL invariant `SubjectBoD` of class `Role` introduced on page 30.

PL7: Role p_1 can be activated only during the time interval [d1, d2]. This policy can be checked using the OCL invariant `AbsoluteBTIRoleEnab` of class `Session` introduced on page 35.

6.3.2.2 AC Request: Access to a Resource

To answer **RQ1.1**, we consider the case when user u_1 sends a request to perform operation op_1 on object o_1; u_1 activates role r_1 within session s_1 at the time of the request. We assume that both object o_1 and operation op_1 are assigned to permission p_1. Moreover, we consider two main scenarios: 1) role r_1 is assigned to user u_1, 2) role r_2 has been delegated to user u_1, through a partial delegation.

First, we consider a basic system configuration in which only AC assignment and activation relations are in place. For scenario 1, we focus on three parameters: the number of sessions in the system, the number of active roles in session s_1, and the number of permissions assigned to role r_1.

As for scenario 2, we replace the last parameter with the number of permissions delegated to user u_1. We measure the access decision time for different

[1]The word subject refers to a user having activated a certain role.

configurations of the system state; these configurations are obtained by varying one parameter and fixing the other two.

All system configurations we have considered, correspond to the worst case scenario when: i) all roles are assigned to all users, and ii) all users are connected and activate all their assigned roles. Figure 6.3 reports[1] the access decision time within the *authorization server* (indicated with the mark •) with respect to a basic system configuration in case of both role assignment and delegation scenarios.

By fixing the number of active roles in session s_1 to 396 and the number of sessions in the system to 1648, we vary the number of permissions assigned to role r_1 from 5 to 53, considering the scenarios when 10%, 30%, 50%, 70% and 100% of the total number of permissions are assigned to role r_1 or delegated to user u_1. We fixed the number of permissions assigned to role r_1 to 53 (maximum number of permissions) for the delegation scenario. Figures 6.3a and 6.3b show the access decision time, within the *authorization server*, with respect to the number of permissions assigned to role r_1 and with respect to the number of permissions delegated to user u_1, respectively. For each figure, the y-axis refers to the access decision time while the x-axis shows the parameter being varied (in this case number of permissions) through various configurations. The time needed to make an access decision with respect to the number of permissions ranges from 37 ms to 50 ms for the assignment scenario and from 48 ms to 50 ms for the delegation scenario.

By fixing the number of sessions in the system to 1648 and the number of permissions assigned to role r_1 to 42 (value provided by our partner), we vary the number of active roles in session s_1 from 39 to 396. We follow the same percentages used for the number of permissions assigned to role r_1 when varying the number of active roles in session s_1. As for the delegation scenario, we fixed the number of delegated permissions to user u_1 to 41; we consider the worst case scenario for a partial[2] delegation where the set of delegated permissions corresponds to the biggest subset of the permissions assigned to the delegated role.
More specifically, the number of the permissions being delegated is equal to the total number of permissions assigned to the role minus one (42-1 in this case).

Figures 6.3c and 6.3d show the access decision time, within the authorization server, with respect to the number of active roles in session s_1 for the role assignment and delegation scenarios, respectively. The time needed to make an access decision with respect to the number of active roles in session s_1 ranges from 36 ms to 41 ms for the assignment scenario and from 43 ms to 54 ms for the delegation scenario.

By fixing the number of active roles in session s_1 to 396 and the number of permissions assigned to role r_1 to 42, we vary the number of sessions in the system. We consider the scenarios when 10%, 30%, 50%, 70% and 100% of users are connected

[1] The mark • shown in this figure and in all the subsequent ones indicates the access decision time within the *proxy* and will be used to answer **RQ3** in section 6.3.4.

[2] We recall that in partial delegation, a user delegates only a subset of the role permissions.

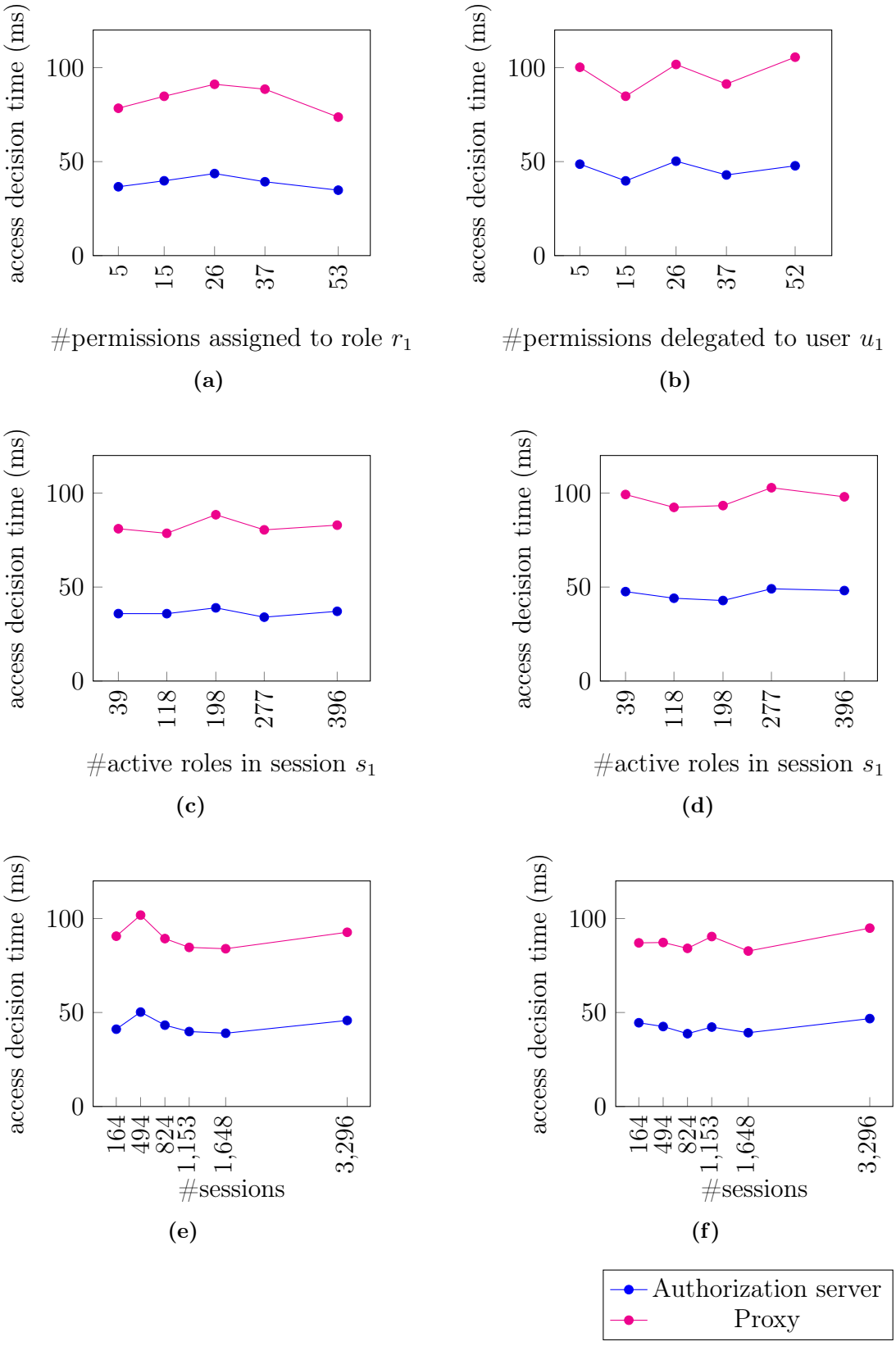


Figure 6.3: Access decision time for an AC-request of type *access to a resource* in case of role assignment ((a), (c), (e)) and delegation ((b), (d), (f)) scenarios with respect to a basic system configuration

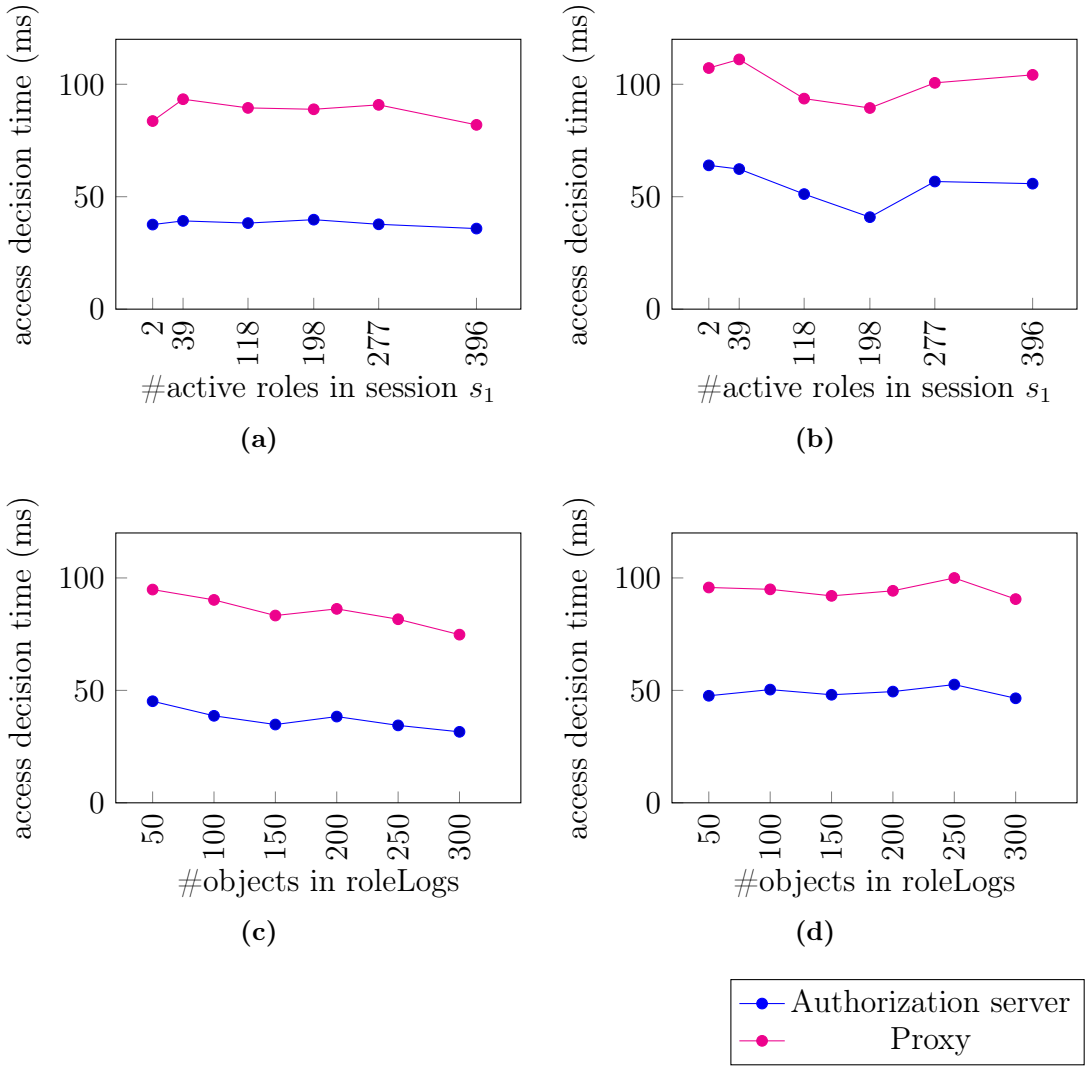


Figure 6.4: Access decision time for an AC-request of type *access to a resource* with respect to a History-based DSoD policy in case of role assignment ((a) and (c)) and delegation ((b) and (d)) scenarios

using one device, and the scenario when all users all connected using a second device. As for the delegation scenario, we fixed the number of delegated permissions to user u_1 to 41. Figures 6.3e and 6.3f show the access decision time, within the *authorization server*, with respect to the number of sessions in the system for the role assignment and delegation scenarios, respectively. The time needed to make an access decision with respect to the number of session ranges from 38 ms to 50 ms for the assignment scenario and from 39 ms to 46 ms for the delegation scenario.

We also consider a system configuration with a DSoD policy and we observe the decision time while varying the system state. As shown in table 6.1, three types of DSoD policies: object-based, operational-based and history-based, are checked in case of an AC request of type *access to a resource*. For a set of conflicting roles only one of these policies should be specified. In our case, we consider the history-

based DSoD policy (**PL3** in section 6.3.2.1) as it is a combination of the object and operational-based. For this case, we focus on four parameters: 1) the number of active roles in the session of the user who made the request, 2) the number of logs, referred to here after as *roleLogs*, associated with the conflicting roles and with the user who made the request, 3) the number of objects within the set of logs, and 4) the number of operations in the system. We fixed the total number of logs in *roleLogs* to 1k (500 logs per conflicting role). As for the number of operations, we consider the number of operations in the system (4 operations in this case). As mentioned previously, we also consider the worst case scenario when: i) all roles are assigned to all users, and ii) all users are connected and activate all their assigned roles.

By fixing the number of objects in *roleLogs* to 300 (maximum number of objects), we vary the number of active roles in session s_1 from 2 to 396. As a DSoD policy is checked *only* if the user is activating *at least* two conflicting roles, both roles r_1 and r_2 are active in various system configurations used when checking policy **PL3**. We also consider the case when user u_1 has already performed operation op_2 on object o_1. Therefore, by performing the requested operation, policy **PL3** will be violated and the access will be denied. Figures 6.4a and 6.4b show the access decision time related to a system configuration with policy **PL3**, within the *authorization server*, with respect to the number of active roles in session s_1, for the role assignment and delegation scenarios, respectively. The time needed to make an access decision with respect to the number of active roles in session s_1 ranges from 35 ms to 39 ms for the assignment scenario and from 40 ms to 63 ms for the delegation scenario.

By fixing the number of active roles in session s_1 to 396, we vary the number of objects in *roleLogs* from 50 to 300 (maximum number of objects). Figures 6.4c and 6.4d show the access decision time related to a system configuration with policy **PL3**, within the *authorization server*, with respect to the number of objects in *roleLogs*, for the role assignment and delegation scenarios, respectively. The time needed to make an access decision with respect to the number of objects in *roleLogs* ranges from 31 ms to 45 ms for the assignment scenario and from 46 ms to 53 ms for the delegation scenario.

We also consider a system configuration with BoD policy. As shown in table 6.1, two types of BoD policies (role-based and subject-based) are checked in case of an AC request of type *access to a resource*. For a set of bounded permissions only one type of these policies should be specified. In our case, we consider the subject-based BoD policy (**PL6** in section 6.3.2.1). To evaluate this policy, we consider a system configuration where each user is assigned to *all* roles, each role is assigned to all permissions, and all users are connected and activate all their roles. We fix to 1k the number of logs, related to a given process[1], and assigned to user u_1 while

[1]We recall that BoD constraints are usually defined in the context of process-based workflow systems.

activating role r_1, and we measure the access decision time for the assignment and delegation scenarios. The time needed to make an access decision with respect to BoD policy is equal to 38 ms for the role assignment scenario, and to 44 ms for the role delegation scenario.

The answer to **RQ1.1** is that the access decision time is almost *constant* for the evaluation of an AC-request of type *access to a resource* applied to various system configurations: 1) basic, 2) with respect to a history-based DSoD policy, 3) with respect to a subject-based BoD policy. The small variations shown in the previous plots can be considered random and correspond to noise introduce by the underlying Java run-time system (which can also affect the actual time measurements). Moreover, the access decision time is slightly higher for the role delegation scenario.

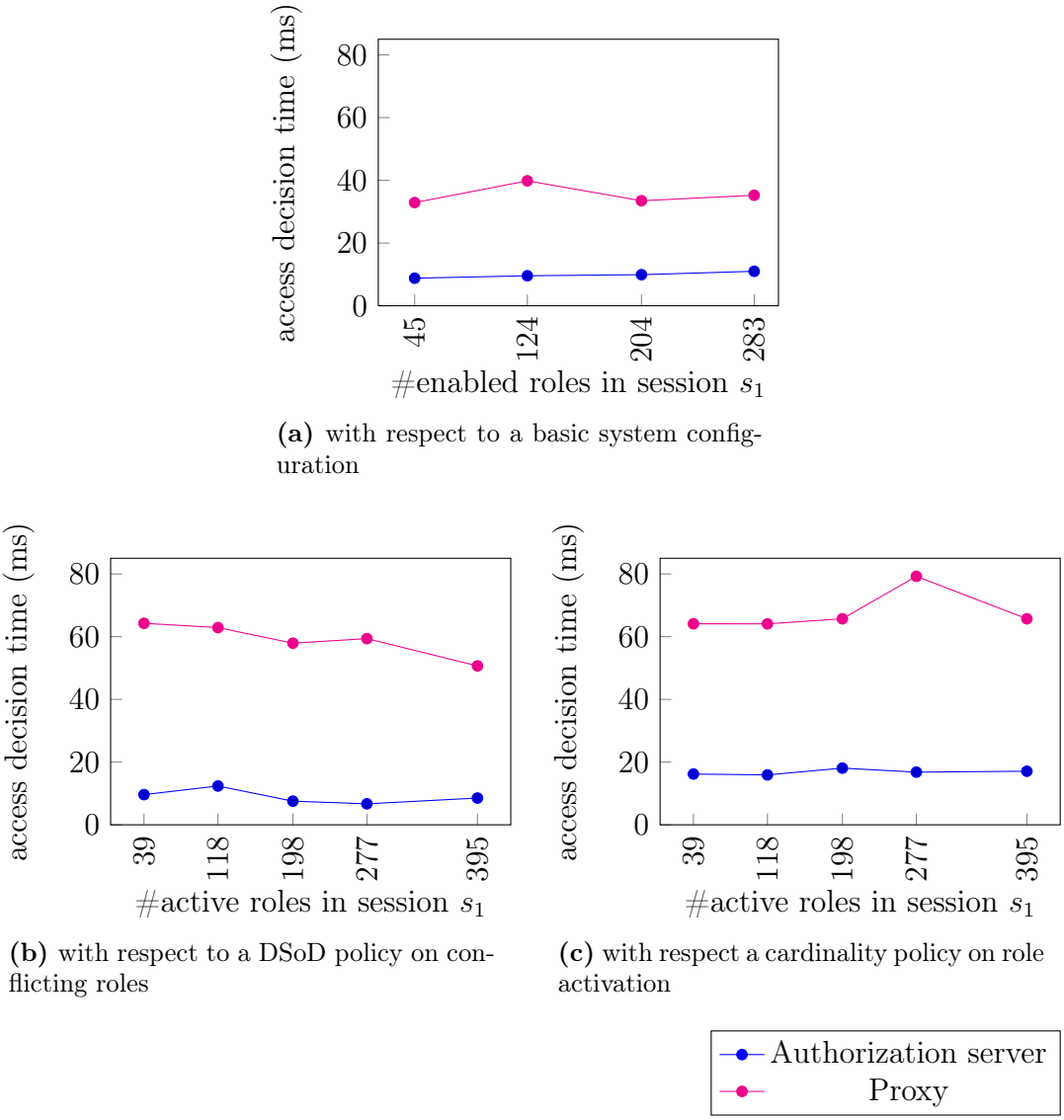
Overall, the access decision time within the *authorization server* is less than 64 ms for the various system configurations described above, while considering both scenarios.

6.3.2.3 AC Request: Role Activation

To answer **RQ1.2**, we consider the case when user u_1 sends a request to activate role r_1 within session s_1.

First, we consider a basic system configuration in which only AC assignment and activation relations are in place. For this case, we focus on one parameter, the number of enabled roles in session s_1. Notice that the list of roles enabled in a session is retrieved from the list of roles assigned to its corresponding user. We consider a system configuration where all roles are assigned to all users. Then, we vary the number of enabled roles within session s_1 from 39 to 396 (maximum number of roles); we consider the scenarios when 10%, 30%, 50%, 70% and 100% of the roles assigned to user u_1 are enabled in session s_1. We recall that to evaluate an AC-request of type *role activation*, the *authorization server* builds the *targetSnap* and checks if the built system state satisfies the RBAC policies expressed as OCL constraints. When considering a system configuration with only assignment and activation relations, the access is always granted if the requested role is enabled. To avoid the noise in the measurements due to the loading of various Java libraries in the system, for each scenario, we send seven AC requests of type *role activation* for various roles and we report the time needed to evaluate the last request. Figure 6.5a shows the access decision time, within the *authorization server*, with respect to the number of enabled roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* ranges from 9 ms to 11 ms.

We also consider a system configuration with a cardinality on role activation policy. We use the same basic system configuration where all roles are enabled and we consider the cardinality policy (**PL1**). For this type of policy, we only control



(a) with respect to a basic system configuration

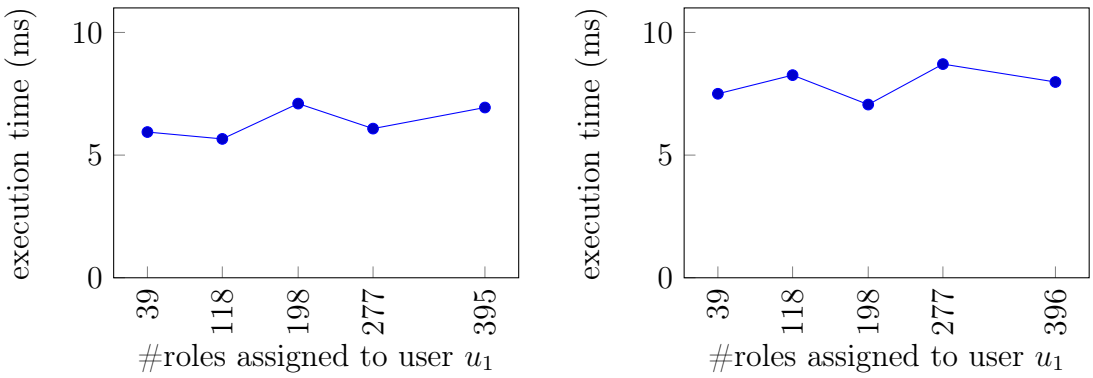
(b) with respect to a DSoD policy on conflicting roles

(c) with respect a cardinality policy on role activation

Figure 6.5: Access decision time for an AC-request of type *role activation*

the number of active roles in session s_1; we vary this parameter from 39 to 395 (maximum number of roles minus one). Notice that role r_1 is not active and is enabled in all system instances. Figure 6.5c shows the access decision time, within the *authorization server*, with respect to the number of active roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* with respect to a cardinality on role activation policy (**PL1**) ranges from 16 ms to 18 ms.

We also consider a system configuration with a DSoD on conflicting roles (DSoDCR). We keep the same instances used for the evaluation of an access resource of type *role activation* described above and we replace the cardinality policy (**PL1**) with the DSoDCR policy (**PL2**). Figure 6.5b shows the access decision time while varying the number of active roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* with respect to a DSoDCR



(a) the user's position is not known

(b) the user's position is known

Figure 6.6: Execution time for an AC-related event of type *user authentication* with respect to a basic system configuration

(**PL2**) ranges from 7 ms to 13 ms.

The answer to **RQ1.2** is that the access decision time is almost *constant* for the evaluation of an AC-request of type *role activation* applied to various system configurations: 1) basic, 2) with respect to a cardinality policy, 3) with respect to a DSoDCR policy. Moreover, the time needed to evaluate an AC request on *role activation* with respect to a cardinality policy is slightly higher than the time needed to evaluate the same request with respect to a DSoDCR policy. Overall, the access decision time is less than 19 ms.

6.3.2.4 AC-related Event: User Authentication

To answer **RQ1.3**, we consider the case when the *authentication server* sends a notification to the *authorization server* to create a new session and to enable the roles assigned to the authenticated user within the created session. We consider two main scenarios: 1) the user position is not known (the notification has the form $\{u_1, s_1\}$ where u_1 refers to the user id and s_1 refers to the id of the session to be created for u_1), 2) the user position is known (the notification has the form $\{u_1, s_1, loc_1\}$ where loc_1 corresponds to the authenticated user's position . For the sake of simplicity, we only consider locations of type *logical*. In what follows we consider the worst case configuration scenario for role assignment and activation relations; all users, excluding user u_1, are connected and activate all their assigned roles.

First, we consider a basic system configuration in which only AC assignment and activation relations are in place. We vary the number of roles assigned to user u_1 from 39 to 396 (maximum number of roles). Figures 6.6a and 6.6b report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 6 ms to 7 ms when the

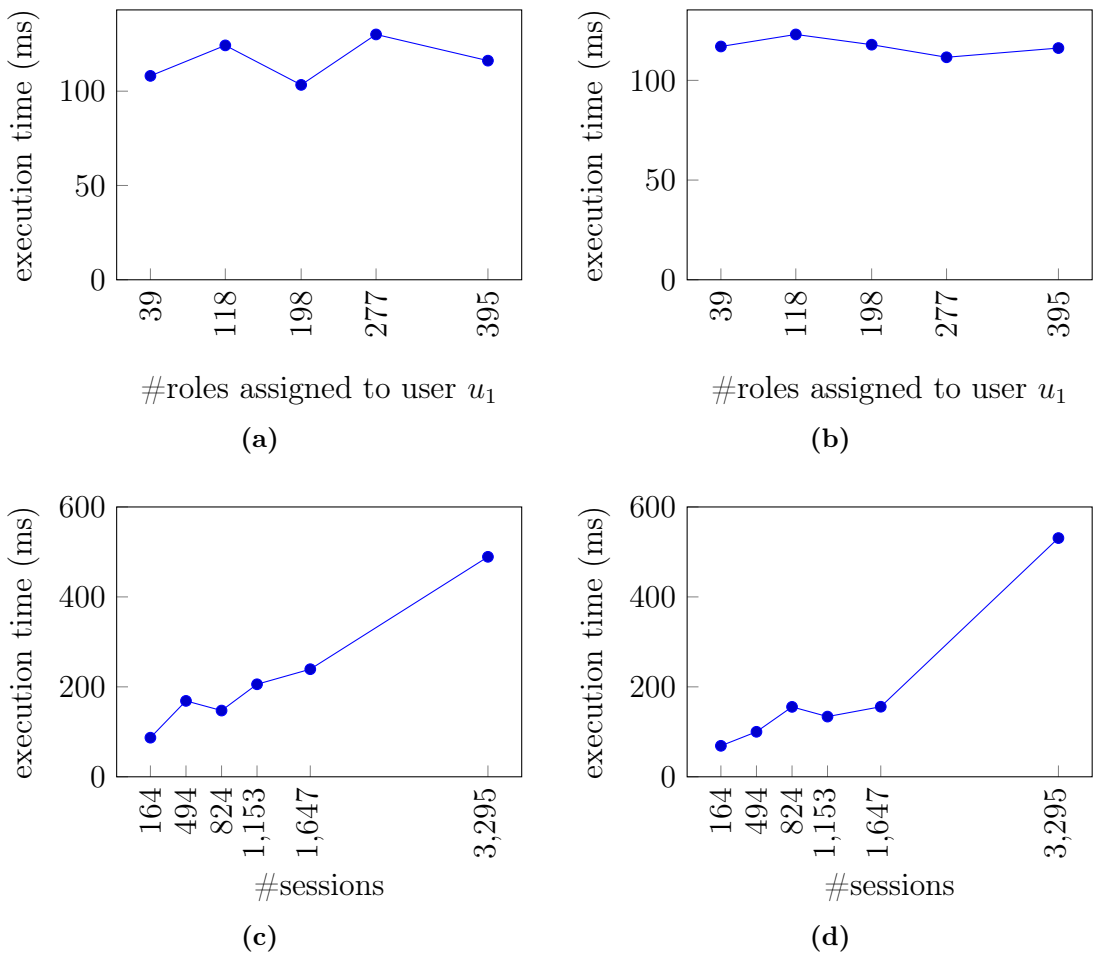


Figure 6.7: Execution time for an AC-related event of type *user authentication* with respect to a precedence policy depending on the user's position availability (the user position is known (a) and (c) and not known in (b) and (d))

user's position is not known, and from 7 ms to 9 ms when the user's position is known.

We also consider a system configuration with a precedence policy. In this case, we consider the precedence policy **PL4** defined in section 6.3.2.1. To achieve a valid system state with respect to policy **PL4**, user u_1 should not be member of role r_2. Moreover, we consider the worst case scenario when role r_2 is not active in any session. To evaluate this policy, we focus on two parameters: the number of roles assigned to user u_1 and the number of sessions in the system.

By fixing the number of sessions in the system to 1647 (this number corresponds to the case when all users excluding user u_1 are connected), we vary the number of roles assigned to user u_1 from 39 to 395. Figures 6.7a and 6.7b report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 103 ms to 130 ms when the user's position is not known, and from 112 ms to 124 ms when the user's position is known.

By fixing the number of roles assigned to user u_1 to 395 (role r_2 is excluded), we vary the number of sessions in the system. We consider the scenarios when 10%, 30%, 50%, 70% and 100% of users, excluding user u_1 are connected using one device, and the scenario when all users all connected using a second device. Figures 6.7c and 6.7d report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 87 ms to 490 ms when the user's position is not known, and from 69 ms to 512 ms when the user's position is known.

We also consider a system configuration with a time-based policy on role enabling. We recall that a temporal context in the GemRBAC+CTX model is composed of time expressions and each time expression is composed of absolute and/or relative expressions. For the sake of simplicity, we only consider time expressions composed of absolute time intervals. In this case, we consider a system configuration with a time-based policy (**PL7** in section 6.3.2.1). We consider the worst case scenario for checking **PL7** in which the current time (i.e., the time when user u_1 made the request) is not contained in any interval contained in the role enabling context. We focus on three parameters: the number of contexts where role r_1 can be enabled, the number of time expressions contained in these contexts, and the number of time intervals contained in these time expressions. For each parameter, we measure the execution time for different configurations of the system state; these configurations are obtained by varying one parameter and fixing the other two.

We consider a system configuration with one enabling context assigned to role r_1; this context is composed of one time expression. We vary the number of time intervals within this time expression from 1 to 100, with a 10 step increment. Figures 6.8a and 6.8b report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 26 ms to 39 ms when the user's position is not known, and from 26 ms to 35 ms when the user's position is known.

By fixing the number of time intervals contained in each time expressions to 10 and the number of role enabling contexts assigned to role r_1 to 1, we vary the number of time expressions within this context from 1 to 100, with a 10 step increment. Figures 6.8c and 6.8d report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 26 ms to 194 ms when the user's position is not known, and from 29 ms to 208 ms when the user's position is known.

By fixing the number of time expressions contained in each role enabling contexts assigned to role r_1 to 10, and time intervals contained in each time expression to 10, we vary the number of role enabling contexts from 10 to 100, with a 10 step increment. Figures 6.8e and 6.8f report the execution time needed to create a new

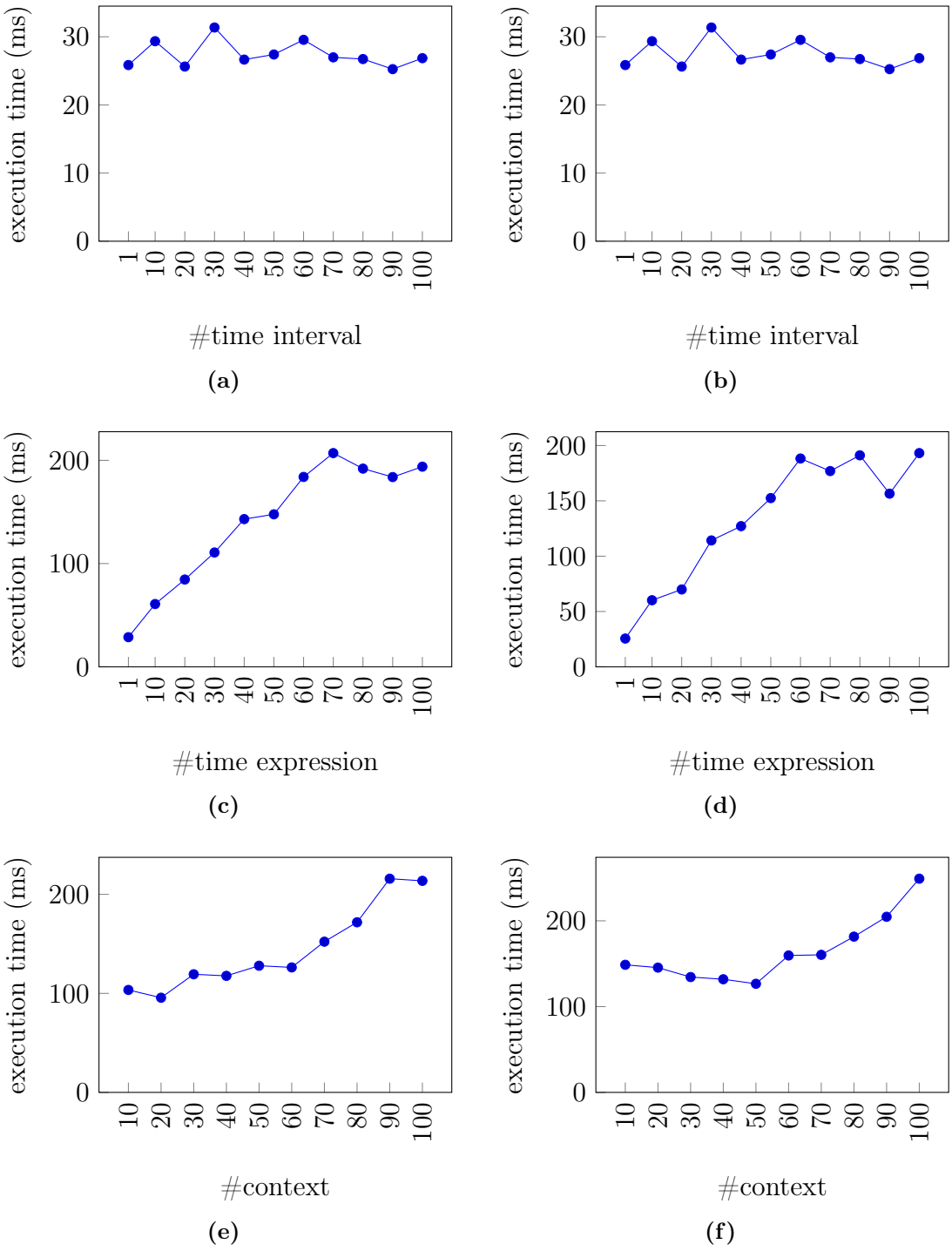


Figure 6.8: Execution time for an AC-related event of type *user authentication* with respect to a time-based policy depending on the user's position availability (the user position is known in (a), (c) and (e) and not known in (b), (d) and (e))

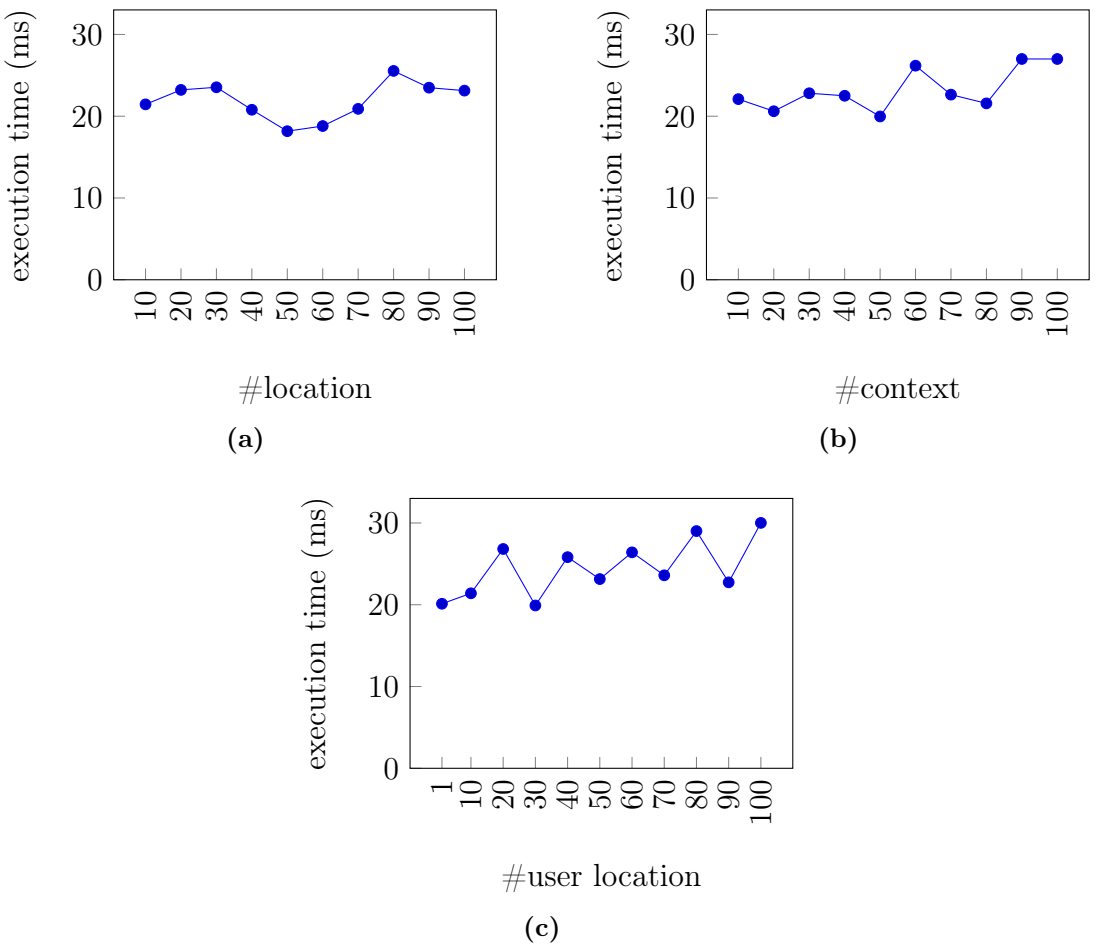


Figure 6.9: Execution time for an AC-related event of type *user authentication* with respect to a location-based policy

session for both scenarios depending on the user's position availability. The execution time ranges from 97 ms to 250 ms when the user's position is not known, and from 127 ms to 216 ms when the user's position is known.

We also consider a system configuration with a location-based policy. We recall that a spatial context in the GemRBAC+CTX model is composed of a set of locations which can be of type physical, logical or relative. For the sake of simplicity, we only consider locations of type *logical*. In this case, we consider a system configuration with the location-based policy (**PL5** in section 6.3.2.1). We consider the worst case scenario for checking **PL5** in which the user position (at the time when user u_1 made the request) is not contained in any of the logical locations of the role enabling context assigned to role r_1. We focus on three parameters: the number of contexts where role r_1 can be enabled, the number of logical locations contained in these contexts, and the number of locations associated with the user. For each parameter, we measure the execution time for different configurations of the system state; these configurations are obtained by varying one parameter and fixing the other two.

We consider a system configuration with one user's location and one enabling context assigned to role r_1. We vary the number of locations within this role enabling context 10 to 100, with a 10 step increment. Figure 6.9a shows the execution time needed to create a new session. The execution time ranges from 19 ms to 26 ms.

By fixing the number of locations assigned to user u_1 to 1 and the number of locations contained in each role enabling context assigned to role r_1 to 10, we vary the number of role enabling contexts assigned to role r_1 from 10 to 100, with a 10 step increment. Figure 6.9b shows the execution time needed to create a new session. The execution time ranges from 20 ms to 30 ms.

By fixing the number of role enabling contexts assigned to role r_1 to 1, and the number of locations contained in this context to 20, we vary the number of locations assigned to user u_1 from 1 to 100, with a 10 step increment. Figure 6.9c shows the execution time needed to create a new session. The execution time ranges from 20 ms to 31 ms.

The answer to **RQ1.3** is that the execution time for an AC-related event of type *user authentication* is

- *constant* for a basic system configuration with respect to the number of roles assigned to user u_1;
- *constant* with respect to the number of roles assigned to user u_1 and *linear* with respect to the number of sessions, for a system configuration with a precedence policy;
- *constant* with respect to the number of time intervals in a time expression and *linear* with respect to the number of role context enabling and with respect to the number of time expressions contained in a role context enabling, for a system configuration with a time-based policy;
- *constant* with respect to the number of role enabling contexts, the number of locations contained in the role enabling context, and the number of locations associated with a the user who made the request, for a system configuration with a location-based policy.

We also report that the execution time is not affected by the user's position availability for the basic configuration system and for a system configuration with a precedence or time-based policy; however, the execution time *slightly* increases when the user position is known.

Overall, the maximum execution time we measured was 512 ms, obtained for a system configuration with a precedence policy and when all users are connected using two devices.

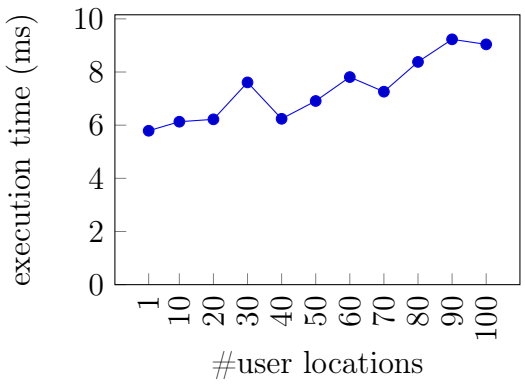


Figure 6.10: Execution time for an AC-related event of type *user change location* with respect to a system basic configuration

6.3.2.5 AC-related Event: User Change Location

To answer **RQ1.4**, we consider the case when the *geo-localization server* sends a notification to the *authorization server* to update the user's position in *Snap*. This notification has the form (u_1, loc_1,loc_2), where loc_1 and loc_2 refer to the previous and the new position of user u_1.

First, we consider a basic system configuration in which only AC assignment and activation relations are in place. We vary the number of locations assigned to user u_1 from 1 to 100, with a 10 step increment. Figure 6.10 reports the execution time needed to update the user's position in case of an AC-related-event of type *user change location*. The execution time ranges from 6 ms to 10 ms.

We also consider a system configuration with a location-based policy. In this case, we consider various system configurations used for the AC-related event *user authentication* with respect to policy **PL5** (section 6.3.2.4).

By fixing the number of locations assigned to user u_1 to 10 and the number of role enabling contexts assigned to role r_1 to 1, we vary the number of locations contained in this role enabling context from 10 to 100, with a 10 step increment. Figure 6.11a shows the execution time needed to update the user position with respect to the number of locations in the role enabling context. The execution time ranges from 7 ms to 12 ms.

By fixing the number of locations assigned to user u_1 to 10 and the number of locations contained in each role enabling contexts to 10, we vary the number of role enabling context assigned to role r_1 from 10 to 100, with a 10 step increment. Figure 6.11b shows the execution time needed to update the user position with respect to the number of role enabling context assigned to role r_1. The execution time ranges from 7 ms to 9 ms.

Figure 6.11c shows the execution time needed to update the user position with respect to the number of locations assigned to user u_1. The execution time ranges from 8 ms to 12 ms.

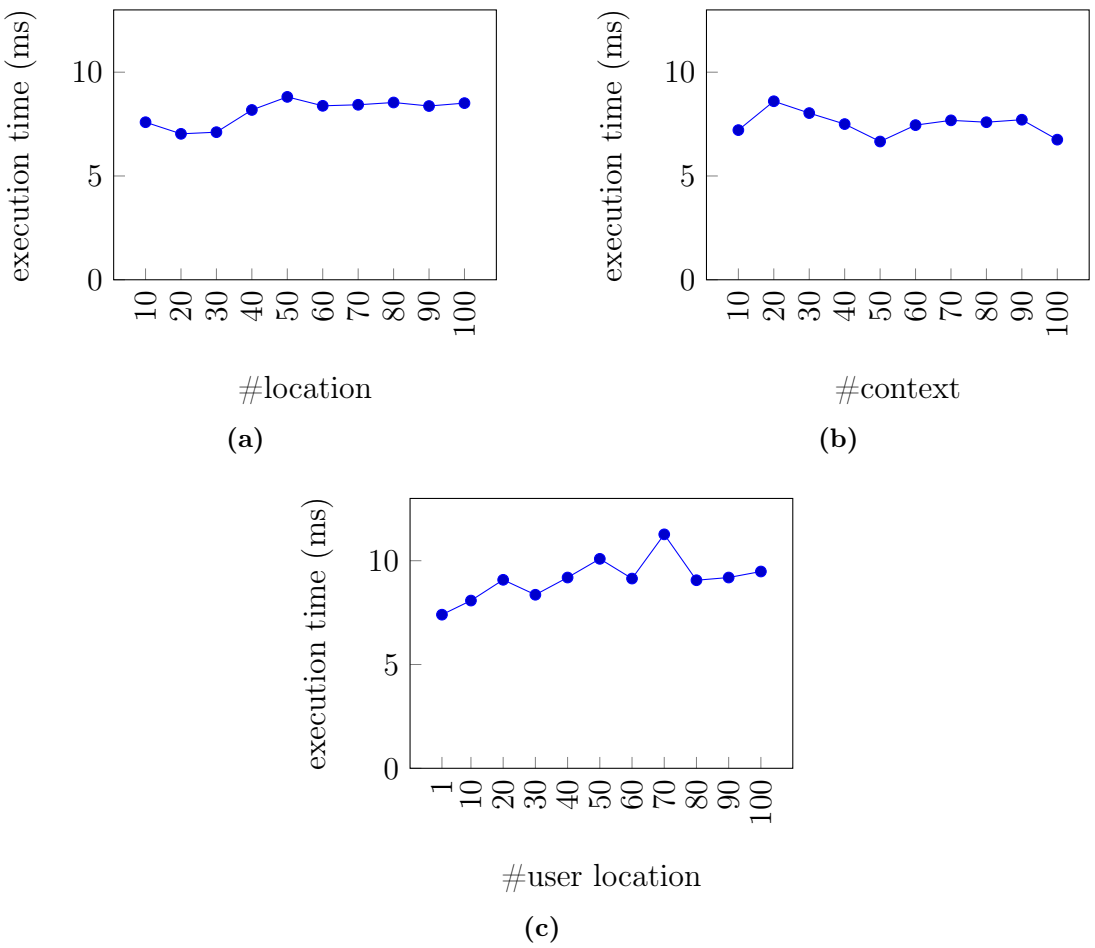


Figure 6.11: Execution time for an AC-related event of type *user change location* with respect to a location-based policy

The answer to **RQ1.4** is that the execution time is almost *constant* for the evaluation of for an AC-related event of type *user change location* applied to both system configurations: basic, and with respect to a location-based policy.

To conclude, the answer to **RQ.1** is that, when enforcing policies in a real industrial system under various configurations:

- The access decision time within the *authorization server* is less than 64 ms. 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 [75]

on average 1880 ms of networking time. 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-related event is less than 512 ms. Also this value has 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-related 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 [76], a common benchmark for Web applications, considers an average think time of 7 s; the maximum value for the execution time measured in our system (512 ms) is well below this threshold.

6.3.3 Scalability of the Proposed Architecture

To address **RQ2**, which focuses on the scalability of the architecture, we decompose it into the following subquestions which take into account different types of AC requests/events:

How does the authorization server scale:

(**RQ2.1**) in the context of an AC request of type *access to a resource* with respect to the main parameters characterizing:

- a basic system configuration,
- a DSoD policy,
- a BoD policy?

(**RQ2.2**) in the context of an AC request of type *role activation* with respect to the main parameters characterizing:

- a basic system configuration,
- a cardinality policy,
- a DSoD policy?

(**RQ2.3**) in the context of an AC-related event of type *user authentication* with respect to the main parameters characterizing:

- a basic system configuration,
- a precedence-based policy,
- a time-based policy,

- a location-based policy?

(**RQ2.4)** in the context of an AC-related event of type *user change location*, with respect to the main parameters characterizing:

- a basic system configuration,
- a location-based policy?

We recall that we use synthesized model instances to evaluate MORRO scalability in terms of the various system configuration parameters.

6.3.3.1 AC Request: Access to a Resource

We consider the case when user u_1 sends a request to perform operation op_1 on object o_1; u_1 activates role r_1 within session s_1 at the time of the request. We assume that both object o_1 and operation op_1 are assigned to permission p_1. To evaluate the scalability, we use instances with the same settings as the ones used while evaluating the performance in case of AC-request of type *access to a resource* (introduced in section 6.3.2.2, page 92). To answer **RQ2.1**, we increase one parameter and measure the access decision time.

We followed the same general evaluation methodology described in section 6.3.2.2 but we varied the values of the main relevant parameters to assess the scalability of the system. More specifically, for a basic system configuration, we considered three parameters: the number of sessions in the system, the number of active roles in session s_1, and the number of permissions assigned to role r_1.

By varying the number of permissions assigned to the parameter corresponding to role r_1 from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.12a and 6.12b: the time needed to make an access decision with respect to the number of permissions ranges from 53 ms to 248 ms for the assignment scenario and from 78 ms to 272 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of permissions assigned to role r_1. This is due to the operation `selectPermission` executed at line 14 in algorithm 1, which checks through the set of permissions assigned to role r_1, whether there is any permission p_1 assigned to both operation op_1 and object o_1; this check is indeed linear with respect to the number of permissions.

We recall that the list of active roles in a session is retrieved from the list of roles assigned to its corresponding user. To increase the number of active roles within session s_1, we increased the number of roles assigned to user u_1 from 396 to 10k. By varying the number of active roles in session s_1 from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.12c and 6.12d: the time needed to make an access decision with respect to the number active roles within session s_1 ranges from 26 ms to 37 ms for the assignment scenario and from 33 ms to 46 ms for the delegation scenario. As shown in both figures, the access decision time is *almost constant* with respect to the number of roles active in session s_1.

This is due to the fact that the check at line13 of algorithm 1 is performed in a constant time.

To evaluate the access decision with respect to the number sessions in the system, we increased the number of users from 1648 to 10k. By varying the number of sessions parameter from 10k to 25k[1], with a step increment of 5k, we obtained the results shown in figures 6.12e and 6.12f: the time needed to make an access decision with respect to the number of sessions ranges from 49 ms to 72 ms for the assignment scenario and from 67 ms to 132 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of sessions in the system. This can be explained in terms of the OCL implementation of the authorization server, in which the *SnapProcessor* searches through the set of all sessions in the system to select the instance of the session matching the corresponding parameter in the AC request.

As for the system configuration with a history-based DSoD policy, we focus on four parameters: 1) the number of active roles in the session of the user who made the request, 2) the number of logs, referred to here after as *roleLogs*, associated with the conflicting roles and with the user who made the request, 3) the number of objects within the set of logs, and 4) the number of operations in the system.

Number of active roles in the session. We increased the number of users in the system from 396 to 10k. By varying the number of active roles in session s_1 from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.13a and 6.13b: the time needed to make an access decision with respect to the number of active roles in session s_1 ranges from 73 ms to 99 ms for the assignment scenario and from 96 ms to 108 ms for the delegation scenario. As shown in both figures, the access decision time is *almost constant* with respect to the number of active roles in session s_1. This is due to the definition of the OCL constraint corresponding to the History-based DSoD (OCL invariant `DSoDHis` of class `Session` introduced on page 29), in which we first check if both conflicting roles are active in session s_1; this check is performed in a constant time.

Number of logs in *roleLogs*. By varying the number of logs in *roleLogs* from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.13c and 6.13d: the time needed to make an access decision with respect to the number of logs in *roleLogs* ranges from 37 ms to 46 ms for the assignment scenario and from 123 ms to 148 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of logs associated with the conflicting roles. This is due to the definition of the OCL constraint corresponding to the History-based DSoD policy (OCL invariant `DSoDHis` of class `Session` introduced on page 29), in which we select the subset of logs associated with the current user from the set of logs associated with the conflicting roles; this selection requires to navigate the set of logs associated with each conflicting role.

[1]This number is defined with respect to the available memory of the machine where the authorization server is running.

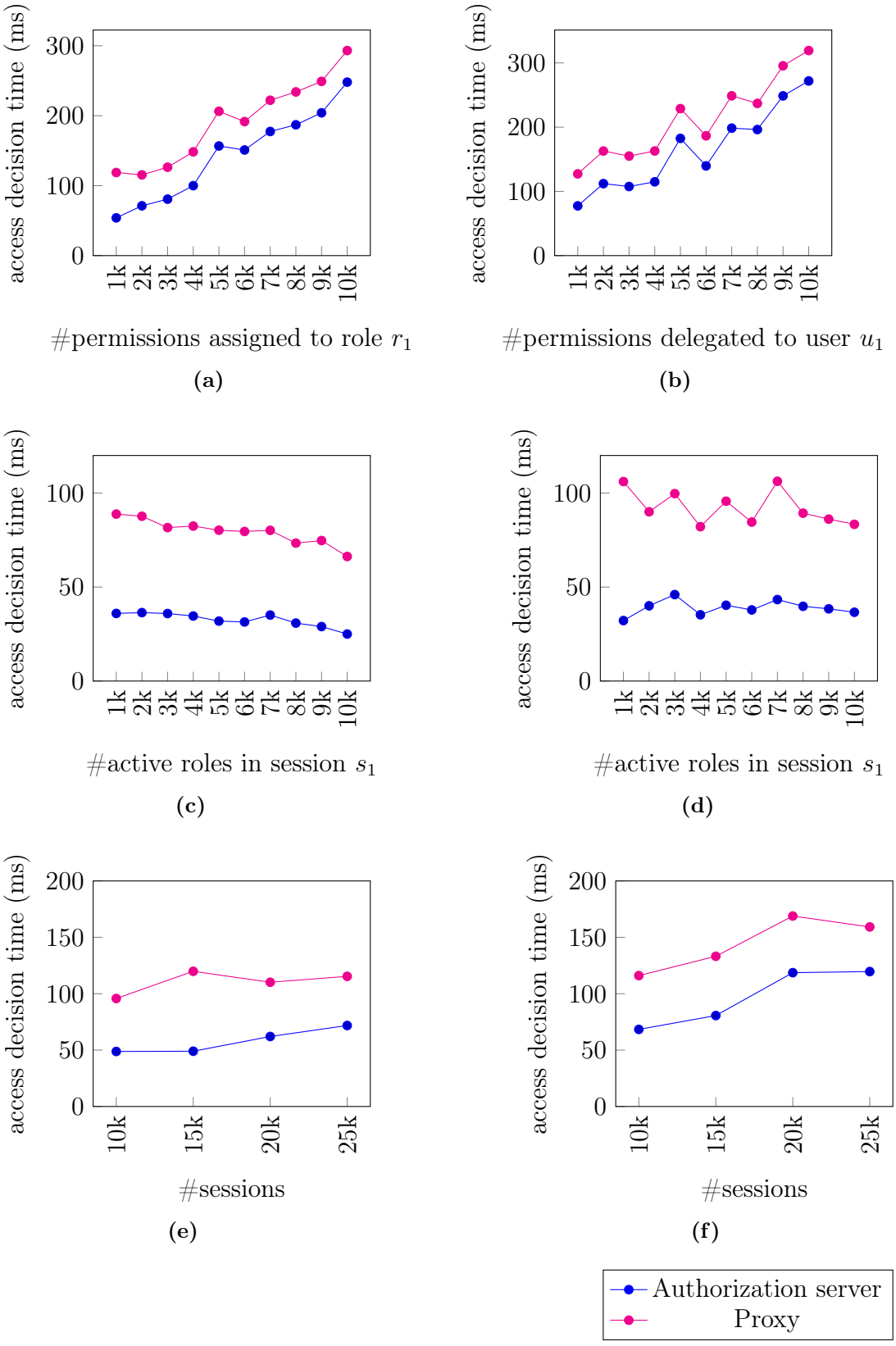


Figure 6.12: Scalability for of an AC-request of type *access to a resource* in case of role assignment ((a), (c), (e)) and delegation ((b), (d), (f)) scenarios with respect to a basic system configuration

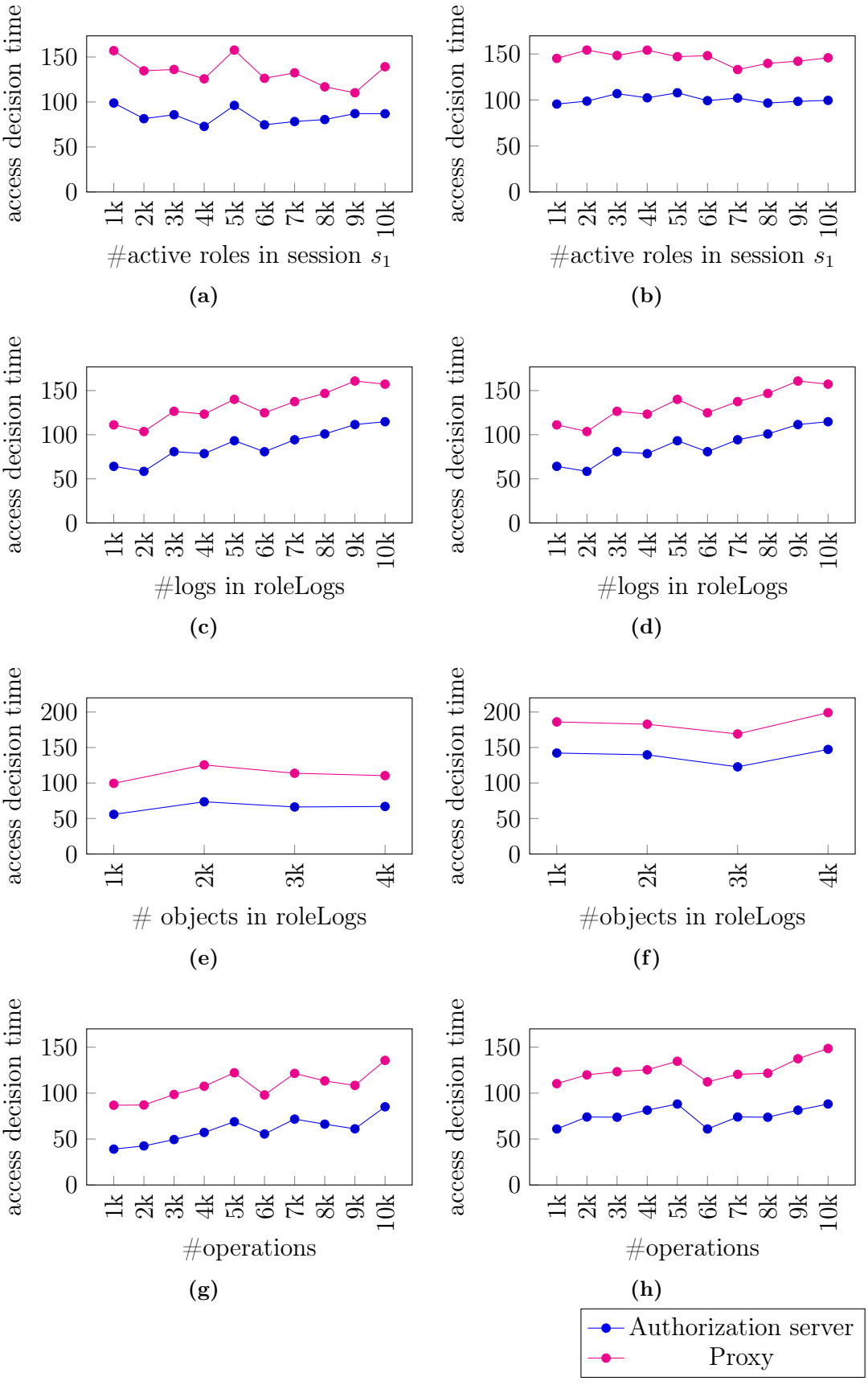


Figure 6.13: Scalability of an AC-request of type *access to a resource* with respect to a History-based DSoD policy in case of role assignment ((a), (c), (e)) and delegation ((b), (d), (f)) scenarios

Number of objects within the set of logs. We increase the number of objects in the system to 6k and the number of logs in *roleLogs* to 8k. By varying the number of objects in *roleLogs* from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.13e and 6.13f: the time needed to make an access decision with respect to the number of objects in *roleLogs* ranges from 56 ms to 67 ms for the assignment scenario and from 123 ms to 147 ms for the delegation scenario. As shown in both figures, the access decision time is *almost constant* with respect to the number of objects in *roleLogs*. This is due to the definition of the OCL constraint corresponding to the History-based DSoD policy (OCL invariant `DSoDHis` of class `Session` introduced on page 29), in which we collect all the objects associated with the logs in the *roleLogs* set; the time taken by this operation is not affected by the number of objects associated with the logs.

Number of operations in the system. By varying the number of operations from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.13g and 6.13h: the time needed to make an access decision with respect to the number of operations ranges from 31 ms to 85 ms for the assignment scenario and from 61 ms to 89 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of operations in the system. This is due to the definition of the OCL constraint corresponding to the History-based DSoD policy (OCL invariant `DSoDHis` of class `Session` introduced on page 29), in which we compute the difference between two sets (line 17); set difference is linear in the size of the elements in the two sets.

As for the system configuration with a BoD policy, we focus on two parameters: the number of roles in the system and the number of logs of the subject who made the request. By varying the number of roles from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.14a and 6.14b: the time needed to make an access decision with respect to the number of roles ranges from 55 ms to 124 ms for the assignment scenario and from 54 ms to 106 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of roles in the system. This is due to the definition of the OCL constraint corresponding to the BoD policy (OCL invariant `SubjectBoD` of class `Role` introduced on page 30), in which we manipulate the variable `boundedroles` by iterating through the list of roles in the system; this operation is linear in the number of roles.

By varying the the number of logs assigned to the current subject (in the current process instance) from 1k to 10k, with 1k step increment, we obtained the results shown in figures 6.14c and 6.14d: the time needed to make an access decision with respect to the number of logs ranges from 59 ms to 67 ms for the assignment scenario and from 59 ms to 68 ms for the delegation scenario. As shown in both figures, the access decision time is *linear* with respect to the number of logs assigned to the current subject. This is due to the definition of the OCL constraint corresponding to the BoD policy (OCL invariant `SubjectBoD` of class `Role` introduced on page 30),

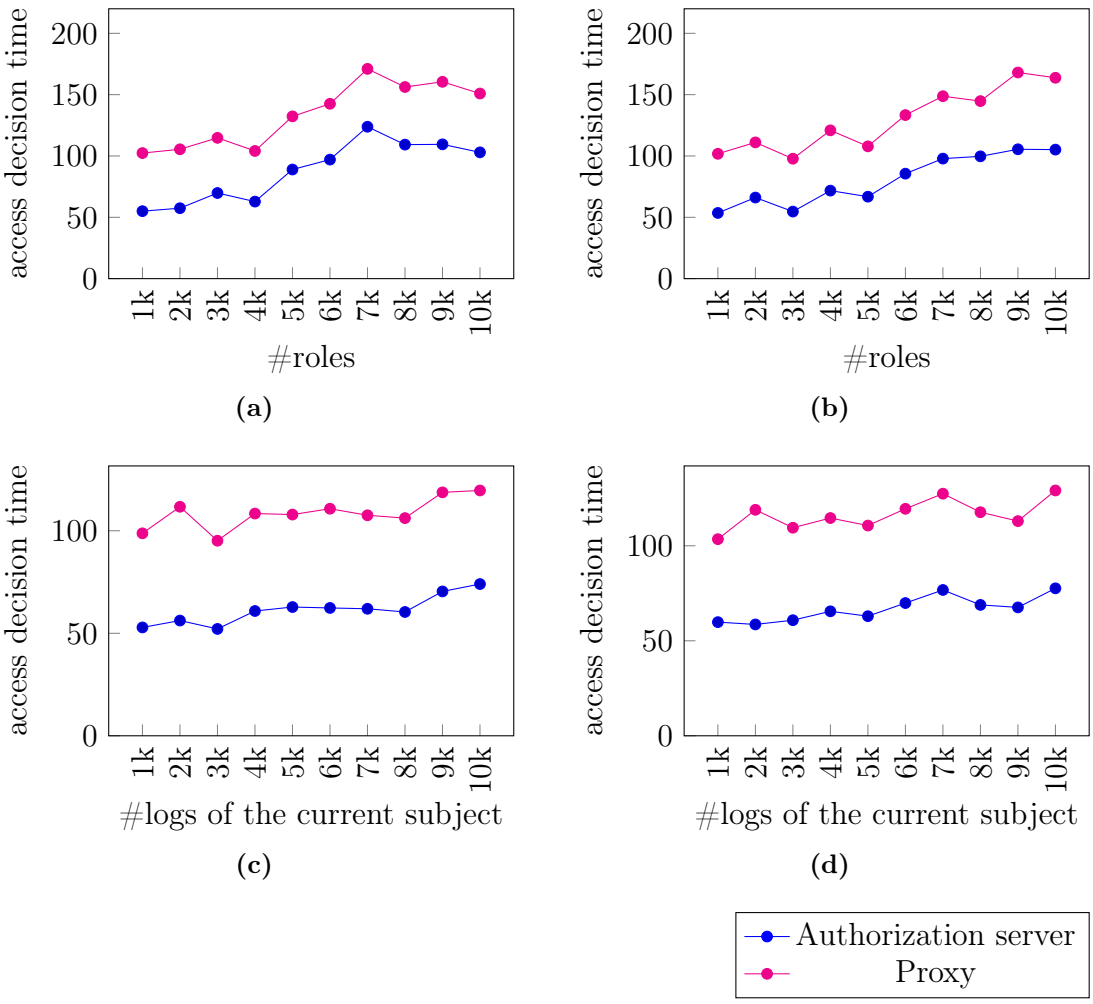


Figure 6.14: Scalability for an AC-request of type *access to a resource* in terms of scalability with respect to BoD in case of role assignment ((a), (c)) and delegation ((b), (d)) scenarios

in which we call operation `logBOCurrentProcessInstance`, which selects the set of logs associated with the current subject related to a business process; this operation is linear in the number of logs.

The answer to **RQ2.1** is that, in case of an AC request of type *access to a resource* considering role assignment and delegation scenarios, the access decision time within the *authorization server*, is:

- almost *constant* with respect to the number of active roles in session s_1, and *linear* with respect to the number of permissions assigned to role r_1 and number of sessions in the system, for a basic system configuration. Moreover, the access decision with respect to the number of permissions is *consistently higher* than the access decision time with respect to the number of sessions in the system;

- *almost constant* with respect to the active roles in session s_1 and with respect to the number of objects in *roleLogs*; *linear* with respect to the number of roles in *roleLogs* and with respect to the number of operations in the system, for a system configuration with a history-based DSoD policy;
- *linear* with respect to the number of roles and with respect to the number of logs assigned to the current subject, for a system configuration with a BoD policy.

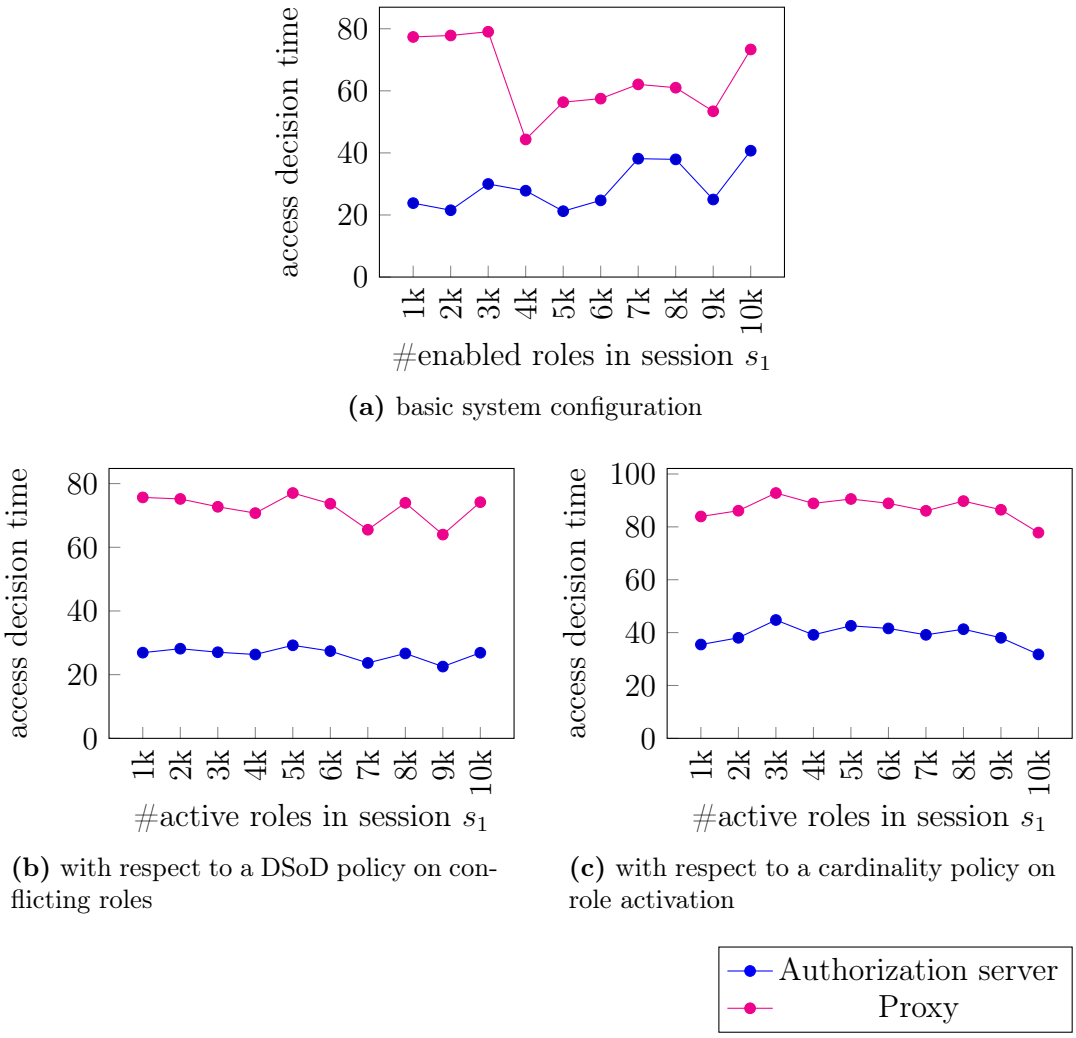
6.3.3.2 AC Request: Role Activation

We consider the case when user u_1 sends a request to activate role r_1 within session s_1. To evaluate the scalability, we use instances with the same settings as the ones used while evaluating the performance in case of AC-request of type *role activation* (introduced in section 6.3.2.3 page 97). To answer **RQ2.2**, we increase one parameter and measure the access decision time.

We followed the same general evaluation methodology described in section 6.3.2.3 but we varied the values of the main relevant parameters to assess the scalability of the system. More specifically, for a basic system configuration, we focus on the parameter corresponding to the number of enabled roles in session s_1. We increase the number of roles assigned to user u_1 from 396 to 10k and vary the number of enabled roles in session s_1 from 1k to 10k, with 1k step increment. Figure 6.15a shows the access decision time, with respect to the number of enabled roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* ranges from 22 ms to 41 ms. As hinted by the figure, the access decision time is constant with respect to the number of roles enabled in session s_1. This is due to the check at line 5 of algorithm 1; this check takes a constant time.

As for the system configuration with a DSoD policy, we focus on the parameter corresponding to the number of active roles in session s_1. We consider the same instances used for evaluation of the basic system configuration. Figure 6.15b shows the access decision time while varying the number of active roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* with respect to a DSoDCR (**PL2**) ranges from 23 ms to 29 ms. As shown in the figure, the access decision time is constant with respect to the number of active roles in session s_1. This is due to the definition of the OCL invariant `DSoDCR` of class `Session` introduced on page 27, in which we check whether the conflicting roles are active; this check takes a constant time.

As for the system configuration with a cardinality policy, we focus on the parameter corresponding to the number of active roles in session s_1 and consider



(a) basic system configuration

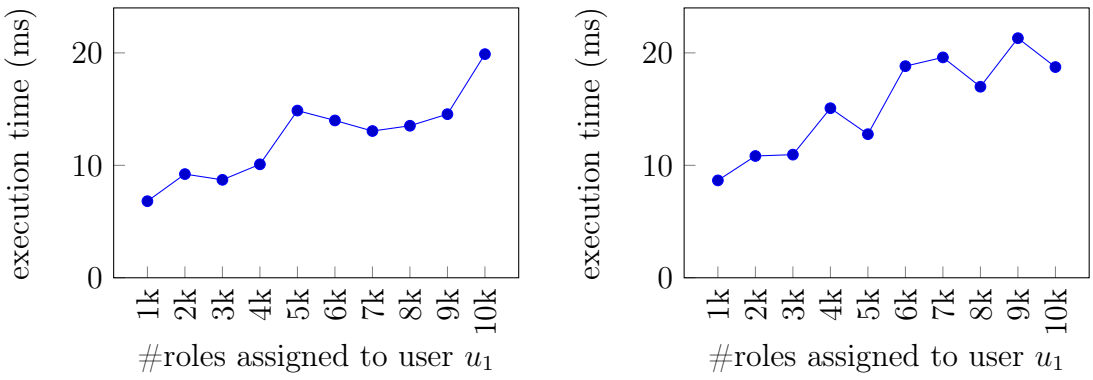
(b) with respect to a DSoD policy on conflicting roles

(c) with respect to a cardinality policy on role activation

Figure 6.15: Scalability of an AC-request of type *role activation*

the same instances used for the evaluation of the basic system configuration. Figure 6.15c shows the access decision time while varying the number of active roles in session s_1. The time needed to make an access decision for an AC-request of type *role activation* with respect to a cardinality policy (**PL1**) ranges from 32 ms to 45 ms. As shown in the figure, the access decision time is constant with respect to the number of active roles in session s_1. This is due to the definition of the OCL invariant `CardinalityActivation` of class `Session` introduced on page 24, in which we check the number of active roles in s_1; this check takes a constant time.

The answer to **RQ2.2** is that in case of an AC request of type *role activation*, the access decision time is *almost constant* for the various system configurations: 1) basic system configuration, 2) DSoD policy, and 3) cardinality policy. Overall, the access decision time within the authorization server is less than 45 ms; this value is achieved when considering a system configuration with a



(a) the user's position is not known

(b) the user's position is known

Figure 6.16: Scalability for an AC-related event of type *user authentication* with respect to basic system configuration

cardinality policy.

6.3.3.3 AC-related Event: User Authentication

We consider the case when the *authorization server* receives a notification about the authentication of user u_1. To evaluate the scalability, we use instances with the same settings as the ones used while evaluating the performance in case of an AC-related event of type *user authentication* (introduced in section 6.3.2.4, page 99). To answer **RQ2.3**, we increase one parameter and we measure the *execution time*.

We followed the same general evaluation methodology described in section 6.3.2.4 but we varied the values of the main relevant parameters to assess the scalability of the system. More specifically, for a basic system configuration, we vary the number of roles assigned to user u_1 from 1k to 10k, with a step increment of 1k. Figures 6.16a and 6.16b report the execution time needed to create a new session for both scenarios depending on the user's position availability. The execution time ranges from 7 ms to 20 ms when the user's position is not known, and from 9 ms to 23 ms when the user's position is known. As shown in both figures, the access decision time is *linear* with respect to the number of roles assigned to user u_1. This is due to the operation `enableAllRoles` in algorithm 2, which enables the set of roles assigned to the authenticated user; the role enabling operation is linear in the number of roles.

As for the system configuration with a precedence policy, we focus on two parameters: the number of roles assigned to user u_1 and the number of sessions in the system.

By varying the number of roles assigned to user u_1 from 1k to 10k, with a step increment of 1k, we obtained the results shown in figures 6.17a and 6.17b: the execution time ranges from 199 ms to 339 ms when the user's position is not

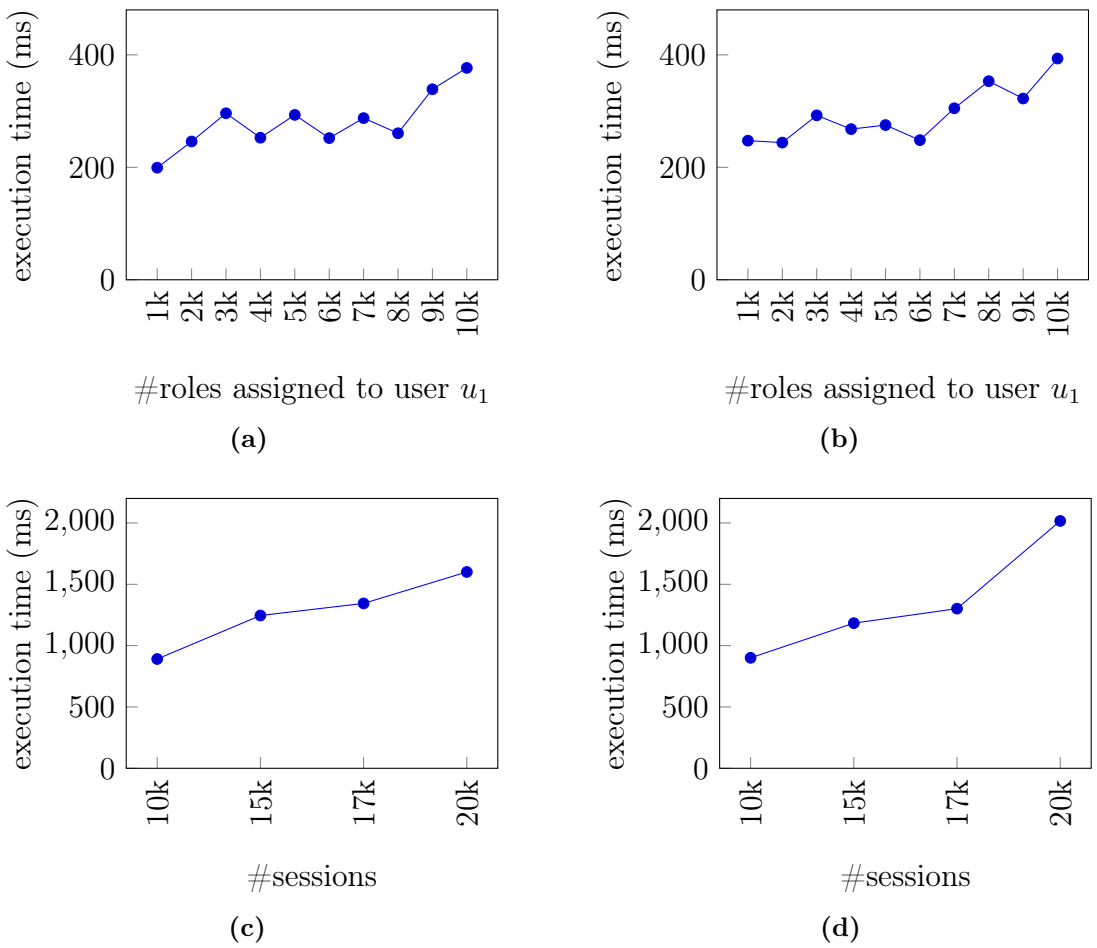


Figure 6.17: Scalability for an AC-related event of type *user authentication* with respect to a precedence policy depending on the user's position availability (the user position is known (a) and (c) and not known in (b) and (d))

known, and from 245 ms to 494 ms when the user's position is known. As shown in both figures, the access decision time is *linear* with respect to the number of roles assigned to the authenticated user. Also in this case, this is due to the complexity of the operation `enableAllRoles` in algorithm 2.

By varying the number of sessions in the system from 10k to 25k, with a step increment of 5k, we obtained the results shown in figures 6.17c and 6.17d: the execution time ranges from 885 ms to 1689 ms when the user's position is not known, and from 900 ms to 2775 ms when the user's position is known. As shown in both figures, the access decision time is *linear* with respect to the number of sessions. This is due to the definition of the OCL invariant `RoleEnablingPrecedence` of class `Session` introduced on page 25, in which we iterate through the list of sessions in the system; this operation is linear in the number of sessions.

As for the system configuration with a time-based policy on role enabling, we focus on three parameters: the number of contexts where role r_1 can be enabled, the number of time expressions contained in these contexts, and the number of

time intervals contained in these time expressions.

By varying the number of time intervals within a time expression considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figures 6.18a and 6.18b: the execution time ranges from 212 ms to 322 ms when the user's position is not known, and from 183 ms to 326 ms when the user's position is known. As shown in both figures, the access decision time is linear with respect to the number of time intervals contained in the time expression associated with a role enabling context. This is due to the definition of the OCL invariant `AbsoluteBTIRoleEnab` of class `Session` introduced on page 35, in which we iterate through all the time intervals (lines 9–10).

By varying the number of the number of time expressions considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figures 6.18c and 6.18d: the execution time ranges from 204 ms to 2086 ms when the user's position is not known, and from 232 ms to 2185 ms when the user's position is known. As shown in both figures, the access decision time is linear with respect to the number of time expressions contained in the role enabling context. This is due to the definition of the OCL invariant `AbsoluteBTIRoleEnab` of class `Session` introduced on page 35, in which we iterate through all the time expressions (lines 6–8).

By varying the number of role enabling contexts considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figures 6.18e and 6.18f: the execution time ranges from 1225 ms to 18 747 ms when the user's position is not known, and from 1243 ms to 18 674 ms when the user's position is known. As shown in both figures, the access decision time is linear with respect to the number of role enabling contexts associated with the role assigned to the authenticated user. This is due to the definition of the OCL invariant `AbsoluteBTIRoleEnab` of class `Session` introduced on page 35, in which we iterate through all the role enabling contexts (lines 4–5).

As for the system configuration with a location-based policy, we focus on three parameters: the number of contexts where role r_1 can be enabled, the number of logical locations contained in these contexts, and the number of locations associated with user u_1.

By varying the number of locations within the role enabling contexts considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figure 6.19a: the execution time ranges from 36 ms to 132 ms. As shown in both figures, the access decision time is linear with respect to the number of locations associated with the role enabling context. This is due to the definition of the OCL invariant `logicalLocationRoleAssign` of class `Session` introduced on page 40, in which we iterate through all the locations associated with the role enabling context (lines 7–8).

By varying the number of role enabling contexts considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figure 6.19b: the execution time ranges from 57 ms to 520 ms. As shown in both figures, the access decision time

is linear with respect to the number of role enabling contexts. This is due to the definition of the OCL invariant `logicalLocationRoleAssign` of class `Session` introduced on page 40, in which we iterate through all the role enabling contexts associated with the role assigned to the authenticated user (lines 5–6).

By varying number of locations assigned to the user who made the request considering the values 1k, 2k, 10k and 20k, we obtained the results shown in figure 6.19c: the execution time ranges from 22 ms to 36 ms.

As shown in both figures, the access decision time is linear with respect to the number of locations assigned to the authenticated user. This is due to the definition of the OCL invariant `logicalLocationRoleAssign` of class `Session` introduced on page 40, in which we iterate through all locations assigned to the context of the authenticated user (lines 2–3).

The answer to **RQ2.3** is that in case of an AC-related event of type *user authentication*, the execution time, within the *authorization server*, is *linear*:

- with respect to the number of roles assigned to user u_1, considering both scenarios depending on the user's position availability, for a basic system configuration;
- with respect to the number of roles assigned to user u_1 and with respect to the number of sessions in the system, considering both scenarios depending on the user's position availability, for a system configuration with a precedence policy;
- with respect to parameters: 1) number of role enabling contexts, 2) number of time expressions in each role enabling context, and 3) number of time intervals in each time expression, when considering both scenarios depending on the user's position availability, for a system configuration with a time-based policy;
- with respect to parameters: number of role enabling contexts, number of locations in each role enabling context, and number of locations assigned to the user for a system configuration with a location-based policy.

Moreover, the execution time with respect to the number of role enabling contexts is *consistently higher* than the execution time with respect the number of time expressions and with respect the number of time intervals; the execution time with respect to the number of role enabling contexts is *consistently higher* than the execution time with respect the number of locations in the role enabling context and with respect the number of locations assigned to the user.

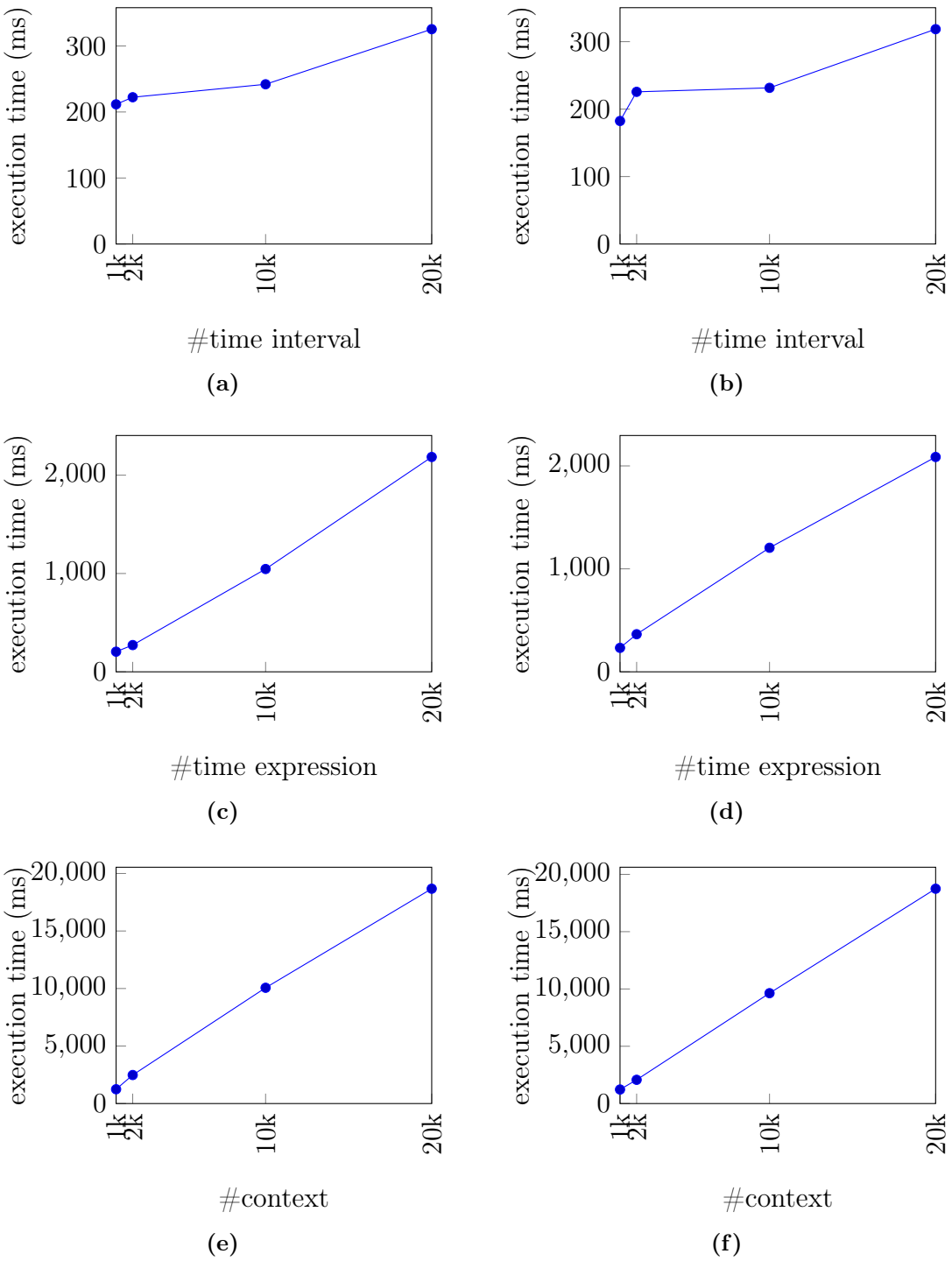


Figure 6.18: Scalability of an AC-related event of type *user authentication* with respect to a time-based policy depending on the user's position availability (the user position is known (a), (c) and (e), and not known in (b), (d) and (f))

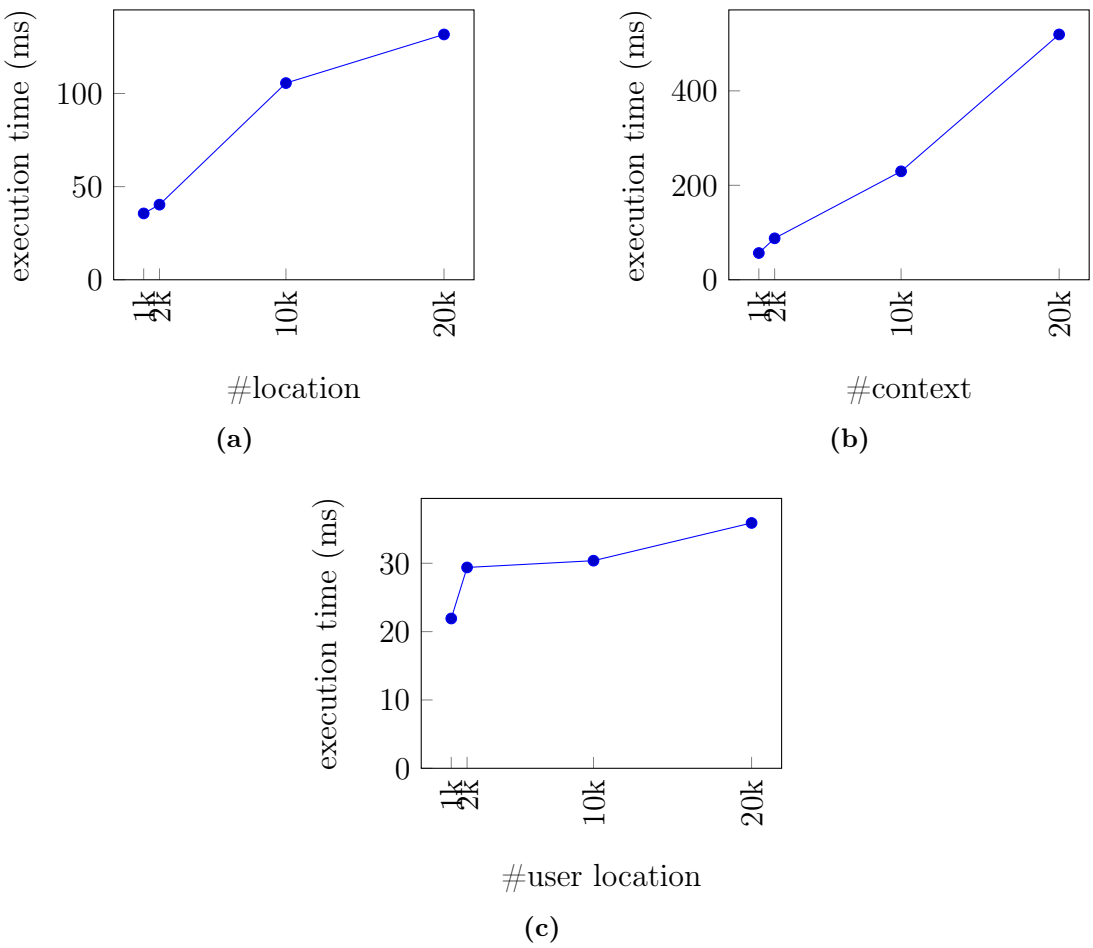


Figure 6.19: Scalability of an AC-related event of type *user authentication* with respect to a location-based policy

6.3.3.4 AC-related Event: User Change Location

We consider the case when the *authorization server* receives a notification about user u_1 changing location. To evaluate the scalability, we use instances with the same settings as the ones used while evaluating the performance in case of AC-related event of type *user change location* (introduced in section 6.3.2.5, page 105). To answer **RQ2.4**, we increase one parameter and we measure the *execution time*.

We followed the same general evaluation methodology described in section 6.3.2.5 but we varied the values of the main relevant parameters to assess the scalability of the system. More specifically, for a basic system configuration, we vary the number of locations assigned to user u_1 considering the values 1k, 2k, 10k and 20k. Figure 6.20 shows the execution time needed to update the user's position in case of an AC-related-event of type *user change location*. The execution time ranges from 20 ms to 88 ms.

As shown in the figure, the execution time is *linear* with respect to the number of location assigned to the user. This is due to the `updateUserLoc` operation in

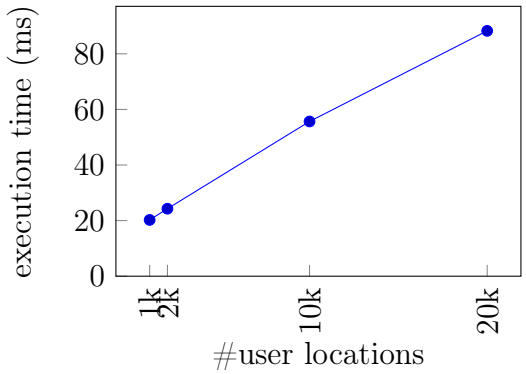


Figure 6.20: Scalability for an AC-related event of type *user change location* with respect to a basic system configuration

algorithm 2, which updates the user location; the complexity of this operation is linear with respect to the number of locations associated with the user's context.

We also consider a system configuration with a location-based policy. In this case, we consider various system configurations used for the AC-related event *user authentication* with respect to policy **PL5** (section 6.3.2.4).

By fixing the number of locations assigned to the user to 10 and the number of role enabling contexts assigned to role r_1 to 1, we vary the number of locations contained in this role enabling context considering the values 1k, 2k, 10k and 20k. Figure 6.21a shows the execution time needed to update the user position with respect to the number of locations in the role enabling context. The execution time ranges from 7 ms to 9 ms.

By fixing the number of locations assigned to the user to 10 and the number of locations contained in each role enabling contexts to 10, we vary the number of role enabling context assigned to role r_1 considering the values 1k, 2k, 10k and 20k. Figure 6.21b shows the execution time needed to update the user position with respect to the number of role enabling contexts assigned to role r_1. The execution time ranges from 7 ms to 9 ms.

Figure 6.21c shows the execution time needed to update the user position with respect to the number of locations assigned to user u_1. The execution time ranges from 14 ms to 81 ms.

All the plots shown in figure 6.21 have a linear trend. The explanations for this trend are the same provided for the corresponding plots in the case of an AC-related event of type *user authentication*, since we use the same system configuration with a location-based policy (and thus the same OCL invariant is involved in the checking).

The answer to **RQ2.4** is that, in case of an AC-related event of type *user change location*, the execution time, *within the authorization server*, is:

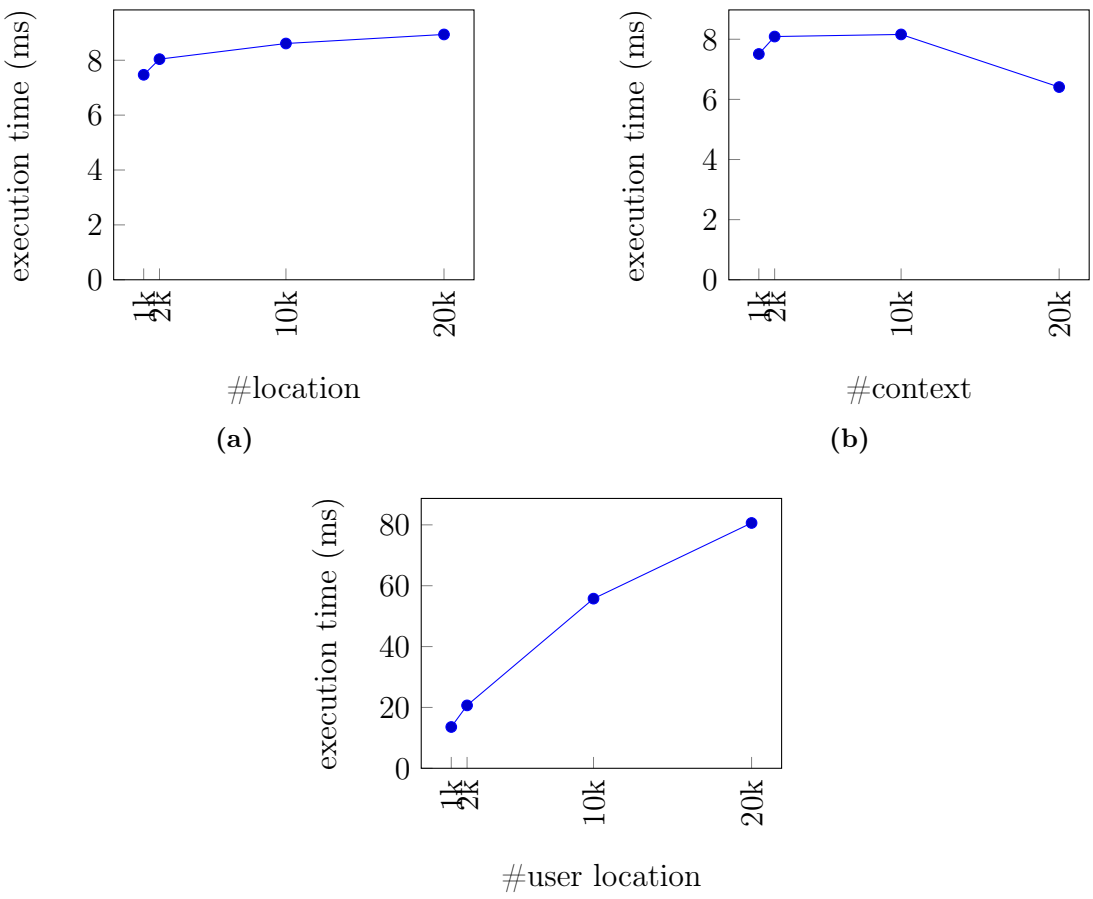


Figure 6.21: Scalability for an AC-related event of type *user change location* with respect to a location-based policy

- *linear* with respect to the number of locations assigned to the user, for a basic system configuration.
- *linear* with respect to the number of locations assigned to the user and *constant* with respect to the number of role enabling contexts and the number of locations in these contexts, in case of an AC-related event of type *user change location* for all system configurations with a location-based policy.

To conclude, the answer to **RQ2** is that, when enforcing policies in a real industrial system:

both the access decision time within the *authorization server* and the execution time for processing a notification of an AC-related event are—in the majority

of the cases—linear with respect to the parameters of the various system configurations; in the remaining cases, the access decision time is constant.

The highest value for the access decision time we measured was 272 ms, in the case of an AC request of type *access to a resource*, in a basic system configuration, in case of a delegation scenario, with 10K permissions assigned to the role of the user making the request. This value would represent a 15% overhead with respect to the average networking time (see [75]) for fetching a complex Web page, as reported in the discussion for **RQ1** on page 106. Such an overhead is still acceptable when enforcing access control policies in large systems.

The highest value for the execution time we measured was 2017 ms, in the case of an AC-related event of type *user authentication*, in a system configuration with a precedence policy, with 25K active sessions, with a known user location. As discussed for **RQ1** on page 106, this value would still be below the think time threshold (7 s) provided by the TCP-W [76] benchmark.

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

We define the access decision time within the *proxy* as the time difference from the time the *proxy* receives an AC request from the user until the time the *proxy* receives an access decision from the *authorization server*.

We address **RQ3** by computing the communication overhead, 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. To compute it, we subtract the access decision time measured within the *authorization server* (indicated with the mark (•) in the plots in the previous sections) from the access decision time measured within the *proxy* (indicated with the mark (•) in the plots in the previous sections).

The answer to **RQ3** is that the overall overhead communication is less than 60 ms for the various system configurations. This means that the maximum access decision time within the proxy is 107 ms, in the case of an AC request of type *access to a resource*, for a system configuration with a history-based DSoD, when considering the AC configuration provided by our industrial partner. This overhead would represent less than 6% with respect to the average networking time (see [75]) for fetching a complex Web page. Furthermore, this value is far below the recommended threshold (200 ms) indicated by our industrial partner. These results confirm the applicability in practice of our approach in a real system.

6.4 Related work

This section provides an overview of the XACML architecture for enforcing access control policies and reviews some existing enforcement mechanisms for RBAC policies.

An enforcement architecture based on the XACML language has been standardized by the OASIS community [14]. This architecture is essentially composed of a policy enforcement point (PEP) and a policy decision point (PDP). Once a user sends an access request, the latter 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. While the PEP is integrated in the application server, the PDP is an external authorization server.

We followed a similar architecture while designing MORRO, where the PEP corresponds to the proxy and the PDP corresponds to the model-driven authorization server.

For enforcing RBAC policies, some proposals have been inspired by the XACML architecture presented above. Sohr et al. [22] implement the PDP as a model-driven authorization engine. RBAC policies are expressed as OCL contraints using the USE tool, a validation tool for UML models and OCL constraints. To make an access decision, the authorization engine checks whether the current system state, represented as an UML object diagram, satisfies the RBAC policies expressed as OCL constraints. This work is very close to our contribution. However, we do not use the system state at the time when the request has been made but we build a target system state to evaluate an access request. Moreover, in this work, the RBAC policies are only enforced as a response to a user access request of type *role activation* or *access to a resource*. Zhang et al. [5] propose a rule-based language and a framework to enforce delegation and revocation policies. The proposed architecture is implemented as a composition of services. A delegation/revocation service is used to handle delegation and revocation requests from users. A role service is responsible for retrieving and updating the RBAC data.

Other proposals deal with the generation of aspects (AOP [77]) from policy specifications; the generated aspects are inserted into the application to be executed at run time. Mourad et al. in [29] propose the use of BPEL aspects to enforce access control policies in the context of web service composition. Kallel et al. [78] generate enforcement aspects in AspectJ from an RBAC specification written in TemporalZ. Mariscal et al. [79] introduce a new UML artifact, called *role-slice* which is used to generated aspects. Mustafa et al. [80] propose an authorization engine based on the RBAC96 model. Policies written in a Z specification are translated into a Java Modeling Language (JML) specification to be checked by an JML runtime assertion checker. This approach supports the set of policies that can be defined on the RBAC96 model. Another approach has been proposed in [81] by Martinez et al. It deals with the generation of a PDP infrastructure from a

specification written in a policy language, using ATL model transformations. An adaptive enforcement approach for the RBAC-based delegation model has been proposed in [82, 83]. Access control policies are transformed into a component-based architecture model. Each resource and role is mapped to a proxy component that will be used to make access decision. In addition to policy enforcement, the proposed framework supports run-time adaptation [84] for RBAC policies. Thus, policies are enforced in case of a delegation request or a change in the access control policies.

In addition to the limited support for the various types of authorization policies (see taxonomy introduced in chapter 2), a limitation shared by the approaches mentioned above is that they only enforce policies in case of an access request. They do not support the usage control concept, meaning that their systems cannot react to changes in the RBAC configuration.

Other proposals deal with context-based usage control in RBAC. Kirkpatrick et al. [85] propose a proximity-based enforcement mechanism for the GEO-RBAC model using the XACML architecture. The access control data are derived from a location device and a role manager. The latter maps each user to the set of her active roles based on her current location. The PDP is composed of a resource manager which evaluates a user request based on her location and the set of her active roles. 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 DSoDCR, are enforced. Once a user sends an access request, the location device detects where the user is located in a specific area and the role manager computes the set of active roles based on the detected location and forwards these information to the resource manager to evaluate the access request. An authorization framework for enforcing temporal policies, based on the X-GTRBAC language and its model GTRBAC has been proposed by Bhatti et al. in [67]. Policies written in the X-GTRBAC language are enforced using a Java-based GUI application. Ben David et al. [86] 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 [87] 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 the 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, the proposed approach was not implemented and only assignment and activation relations have been considered.

Other proposals [23, 24, 25, 26, 27, 28] extend the XACML architecture by introducing a second decision point SDP. Access decisions are cached in the SDP

and they are reused if the request matches one of the cached authorizations. Various implementations have been proposed for the SDP using authorization recycling [23, 24], authorization recycling, CPOL [25] and bloom filter [26, 27]. Although these approaches improve the access decision process in terms of time execution, more memory will be consumed.

We remark that none of the proposed approaches provides a full support for the authorization policies introduced in chapter 2. Each enforcement mechanism implements a limited set of policies supported by its corresponding model. Moreover, the majority of the proposed approaches considers only AC requests of type *access to a resource* and *role activation*. AC requests of type *role delegation* and *role revocation* are only supported by Zhang et al. [5]. As for the usage control, only context-based events, i.e., *user authentication* and *user change location*, are supported in [67, 85]. Furthermore, only few of the aforementioned approaches [22, 25, 81] 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.

6.5 Summary

In this chapter we presented a model-driven enforcement framework for policies written in the GemRBAC-DSL language, which leverages the OCL operationalization of the policies. The idea is to 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. In addition to making an access decision, our approach adopts the usage control (UCON) [69] concept; the access decision is re-evaluated when a new update, from an access control point of view, occurs at the system level. Therefore, policies are enforced both when an AC request is made and when an AC-related event is triggered; we provide the checking algorithms for both cases.

We extensively evaluated the applicability and the scalability of the proposed framework on an industrial Web application developed by our partner. The experimental results show that:

- An access decision can be made within the *authorization server*, on average, in less than 64 ms; the corresponding access decision time, measured at the proxy level, is 107 ms. This overhead would represent less than 6% with respect to the average networking time (see [75]) for fetching a complex Web page. Furthermore, this value is far below the recommended threshold (200 ms) indicated by our industrial partner. These results confirm the applicability in practice of our approach in a real system.

- The execution time for processing a notification of an AC-related event is less than 512 ms. When interpreted in the context of Web applications, this value is far below the average think time (7 s) defined on TCP-W [76], a common benchmark for Web applications.

- Both the access decision time within the *authorization server* and the execution time for processing a notification of an AC-related event are—in the majority of the cases—linear with respect to the parameters of the various system configurations; in the remaining cases, the access decision time is constant. These results show the scalability of MORRO with respect to the various parameters (e.g., the number of users, roles, permissions, sessions, role enabling contexts) that characterize RBAC configurations.

Although the MORRO 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. Furthermore, though we considered the GemRBAC+CTX model and its corresponding OCL constraints in the application of our approach, the latter is generic and does not depend on the GemRBAC+CTX model: it can be applied to any other access control model that can be expressed in UML and whose policies can be expressed in OCL.

Part IV

Finale

Chapter 7

Conclusions and Future Work

7.1 Conclusions

RBAC is the de facto standard for access control in enterprise information systems: by assigning permissions to roles, it decouples users from permissions, simplifying the administration and deployment of access control in the enterprise. In addition to role-to-permission and role-to-user assignments, authorization policies can be defined to restrict unauthorized access to critical resources.

The first RBAC model [3] supported a very small set of authorization policies. Thus, many extensions of this model have been proposed in the literature to enable the specification of complex policies such as delegation, revocation and contextual policies. However, there is no unified framework that can be used to define all these policies in a coherent way, using a common conceptual model. In addition to models, a policy specification language is needed to facilitate the definition of these complex policies. The lack of expressiveness of existing RBAC models/languages hinders the definition of an enforcement mechanism built on top of them, which can prevents the design and implementation of an enforcement mechanism derived from policies, in order to make an access decision.

In this dissertation we have tackled the issues of both *specification* and *enforcement* of RBAC policies by pursuing the following research goals:

- formalizing RBAC policies to enable the operationalization of the access decision procedure;
- expressing RBAC policies using a high-level policy specification language;
- enforcing RBAC policies at run time in an efficient manner.

In the rest of this chapter we summarize the contributions presented throughout this dissertation (Section 7.2), point out their limitations (Section 7.3), and discuss future research directions (Section 7.4).

7.2 Contributions

In this dissertation we have addressed the challenges in specifying and enforcing RBAC policies by making the following contributions:

1. the **GemRBAC+CTX** conceptual model, a UML extension of the RBAC model that includes all the entities required to express the various types of RBAC policies found in the literature, with a specific emphasis on contextual policies. For each type of policy, we provided the corresponding formalization using the Object Constraint Language (OCL) to operationalize the access decision for a user's request using model-driven technologies.

2. the **GemRBAC-DSL** language, a domain-specific language for RBAC policies designed on top of the GemRBAC+CTX model. The language is characterized by a syntax close to natural language, which does not require any mathematical background for expressing RBAC policies. The language supports *all* the authorization policies captured by the GemRBAC+CTX model. We defined the language semantics using a model-driven approach, by mapping each type of GemRBAC-DSL policy to the corresponding OCL constraint(s) defined on the GemRBAC+CTX model. GemRBAC-DSL has been adopted by our industrial partner; in its first use for the specification of the RBAC policies for a real-world application, it has allowed the security engineers to define 19 new types of contextual policies.

3. **MORRO**, a model-driven framework for the run-time enforcement of RBAC policies expressed in GemRBAC-DSL, built on top of the GemRBAC+CTX model. MORRO provides policy enforcement for both access and usage control. The extensive evaluation of the performance of MORRO from the point of view of the access decision time—executed on a real-world Web application developed by our industrial partner, under different configurations—showed that the overhead it adds is acceptable from a practical standpoint and, in the worst case, scales linearly with respect to the main system parameters. These results corroborate the feasibility of embedding the MORRO framework in Web applications.

4. three tools:

   - **GemRBAC-DSL-Editor**, the editor for GemRBAC-DSL;
   - **GemRBAC-DSL-Transform**, the model transformation tool for GemRBAC-DSL;
   - **MORRO**, the run-time enforcement framework;

   have been implemented and released as part of the PhD research work.

7.3 Limitations

The work presented in this thesis is characterized by various limitations and open issues.

Expressiveness. With the emergence of attribute-based access control (ABAC) paradigms [40], more policies can be defined to restrict the user access based on attributes that can be assigned to users, objects and/or permissions; an example of such a policy is "a user can access only resources belonging to her own organization". The RBAC model itself can be seen as an attribute-based model in which the role is defined as a user's attribute.

Although the GemRBAC+CTX model and the GemRBAC-DSL language have been designed in a way to cover the various types of policies proposed in the literature, attribute-based policies are not *fully* taken into account.

Privacy. Preventing unauthorized access to resources based on the user's location is conflicting with protecting the privacy concerns of the user. Contextual policies, and more specifically location-based policies, require a permanent record of the user's position, which may violate the laws of the user's country. Such an issue could hinder the applicability of our run-time enforcement framework, which relies on AC-related data.

Enforcement. To enforce RBAC policies, we represent the system state (from an access control point of view) as an instance of the GemRBAC+CTX model. As the size of the system increases, more memory space will be required by this representation. At the moment, the MORRO framework is not designed to be space efficient, and does not support "garbage collection" or "compacting" operations on the model instance.

7.4 Future Research Directions

This dissertation sets the basis to follow different research directions in the future.

GemRBAC-DSL usability assessment. Although three half-day training sessions were enough for the security engineers of our partner to express their policies using the GemRBAC-DSL language, we plan to further and systematically assess the usability of the language through user studies with practitioners.

ABAC extension. Our next step regarding the specification of policies, will focus on supporting attribute-based access control policies in the GemRBAC+CTX model and in the GemRBAC-DSL language. More specifically, we will base our extension on the Role-Centric Attribute-Based Access Control (RABAC) [88] model, which extends the RBAC model by adding attributes to users and objects.

Space efficiency of the MORRO framework. From the point of view of the enforcement, we will focus on the optimization of the model-driven enforcement framework, especially in terms of space efficiency. We plan to adopt the Kevoree

Modeling Framework (KMF) [89] as an alternative to the Eclipse Modeling Framework (EMF), to improve the support of runtime models for large distributed systems. More specifically, KMF has special mechanisms to decrease the memory footprint and supports different operations on models (i.e., loading, serializing and cloning model entities).

Run-time verification of access control policies. The model-driven enforcement approach proposed in this thesis can be used in the context of run-time verification to check the correctness of the decision made by an existing enforcement mechanism. In order to detect the possible violations of RBAC policies while making an access decision, a monitoring mechanism should be integrated in the enforcement architecture to intercept the access decision. By observing the behavior of the system from an access control point of view, the monitor would send a notification to the *authorization server* whenever a new access decision is made or an AC-related event is triggered.

Bibliography

[1] Ravi Sandhu and Pierangela Samarati. Authentication, Access Control, and Audit. *ACM Comput. Surv.*, 28(1):241–243, March 1996. 2

[2] Fuchs Ludwig, Pernul Gunther, and Sandhu Ravi. Roles in information security – A survey and classification of the research area. *Computers & Security*, 30(8):748–769, 2011. 2

[3] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based Access Control Models. *Computer*, 29(2):38–47, 1996. x, 2, 8, 9, 11, 17, 71, 73, 130

[4] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proc. of RBAC 2000*, pages 47–63. ACM, 2000. 2

[5] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A Rule-based Framework for Role-based Delegation and Revocation. *ACM Trans. Inf. Syst. Secur.*, 6(3):404–441, 2003. 3, 4, 13, 14, 71, 73, 125, 127

[6] Zhikun Zhang, Jianguo Xiao, Hanyi Li, and Youping Geng. An Extended Permission-based Delegation Authorization Model. In *Proc. of CSSE 2008*, volume 3, pages 696–699, December 2008. 3, 71, 73

[7] Jason Crampton and Hemanth Khambhammettu. Delegation in Role-based Access Control. *Int. JIS*, 7(2):123–136, 2008. 3, 13, 72, 73

[8] Karsten Sohr, Mirco Kuhlmann, Martin Gogolla, Hongxin Hu, and Gail-Joon Ahn. Comprehensive two-level analysis of role-based delegation and revocation policies with UML and OCL. *Inf. Softw. Technol.*, 54(12):1396 – 1417, 2012. 3, 72, 73, 75, 76, 77

[9] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A Temporal Role-based Access Control Model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, August 2001. 3, 15, 72, 73, 74

[10] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. GEO-RBAC: A Spatially Aware RBAC. In *Proc. of SACMAT 2005*, pages 29–37. ACM, 2005. 3, 15, 72, 73, 74

[11] James B D Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A Generalized Temporal Role-based Access Control Model. *IEEE Trans. Knowl. Data Eng.*, 17(1):4–23, January 2005. 3, 15, 72, 73, 74, 76

[12] Subhendu Aich, Shamik Sural, and A.K. Majumdar. STARBAC: Spatiotemporal Role Based Access Control. In *Proc. of OTM 2007*, volume 4804 of *LNCS*, pages 1567–1582. Springer, 2007. 3, 72, 73, 74

[13] Indrakshi Ray and Manachai Toahchoodee. A Spatio-temporal Role-Based Access Control Model. In *Proc. of DBSec 2007*, volume 4602 of *LNCS*, pages 211–226. Springer, 2007. 3, 15, 73, 74

[14] OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0, 2005. 3, 4, 76, 125

[15] Diala Abi Haidar, Nora Cuppens-Boulahia, Frederic Cuppens, and Herve Debar. An Extended RBAC Profile of XACML. In *Proc. of SWS 2006*, pages 13–22. ACM, 2006. 3, 76

[16] Anne Anderson. XACML profile for role based access control (RBAC). *OASIS Access Control TC committee draft*, 1:13, 2004. 3, 76

[17] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham. ROWLBAC: Representing Role Based Access Control in OWL. In *Proc. of SACMAT 2008*, pages 73–82. ACM, 2008. 3, 76, 77

[18] Rodolfo Ferrini and Elisa Bertino. Supporting RBAC with XACML+OWL. In *Proc. of SACMAT 2009*, pages 145–154. ACM, 2009. 3, 76, 77

[19] Gail-Joon Ahn and Ravi Sandhu. Role-based Authorization Constraints Specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, November 2000. 4, 12, 75, 76

[20] Michael Hitchens and Vijay Varadharajan. Tower: A Language for Role Based Access Control. In *Proc. of POLICY 2001*, volume 1995 of *LNCS*, pages 88–106. Springer, 2001. 4, 76

[21] C. Cotrini, T. Weghorn, D. Basin, and M. Clavel. Analyzing first-order role based access control. In *Proc. of CSF2015*, pages 3–17. IEEE, July 2015. 4, 75, 76

[22] Karsten Sohr, Tanveer Mustafa, Xinyu Bao, and Gail-Joon Ahn. Enforcing role-based access control policies in web services with UML and OCL. In *Proc. of ACSAC 2008*, pages 257–266. IEEE, 2008. 4, 75, 125, 127

[23] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in hierarchical rbac systems. *ACM Trans. Inf. Syst. Secur.*, 14(1):3:1–3:29, June 2011. 4, 126, 127

[24] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in rbac systems. In *Proc. of SACMAT 2008*, pages 63–72, New York, NY, USA, 2008. ACM. 4, 126, 127

[25] Kevin Borders, Xin Zhao, and Atul Prakash. Cpol: High-performance policy evaluation. In *Proc. of CSS 2005*, pages 147–157, New York, NY, USA, 2005. ACM. 4, 126, 127

[26] Mahesh V. Tripunitara and Bogdan Carbunar. Efficient access enforcement in distributed role-based access control (rbac) deployments. In *Proc. of SACMAT 2009*, pages 155–164, New York, NY, USA, 2009. ACM. 4, 126, 127

[27] Jeffrey Hieb, Jacob Schreiver, and James Graham. Using bloom filters to ensure access control and authentication requirements for scada field devices. In Jonathan Butts and Sujeet Shenoi, editors, *Proc. of ICCIP 2012*, pages 85–97, Berlin, Heidelberg, 2012. 4, 126, 127

[28] Oscar Mortagua Pereira, Diogo Domingues Regateiro, and Rui L Aguiar. Distributed and Typed Role-based Access Control Mechanisms Driven by CRUD Expression. *IJCSTA*, 2(1):1–11, 2014. 4, 126

[29] Azzam Mourad, Sara Ayoubi, Hamdi Yahyaoui, and Hadi Otrok. New approach for the dynamic enforcement of web services security. In *Proc. of PST 2010*, pages 189–196. IEEE, 2010. 4, 125

[30] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. A Comprehensive Modeling Framework for Role-based Access Control Policies. *JSS*, 107:110–126, September 2015. 5

[31] Ameni Ben Fadhel, Domenico Bianculli, Lionel Briand, and Benjamin Hourte. A Model-driven Approach to Representing and Checking RBAC Contextual Policies. In *Proc. of CODASPY2016*, pages 243–253. ACM, 2016. 5

[32] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. GemRBAC-DSL: A High-level Specification Language for Role-based Access Control Policies. In *Proc. of SACMAT2016*, pages 179–190. ACM, 2016. 5

[33] Gail-Joon Ahn and M.E. Shin. Role-based authorization constraints specification using object constraint language. In *Proc. of WETICE 2001*, pages 157–162. IEEE, 2001. 9, 12, 75

[34] Gail-Joon Ahn. Specification and Cassification of Role-based Authorization Policies. In *Proc. of WETICE 2003*, pages 202–207. IEEE, 2003. 11, 12

[35] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. A Role-based Access Control Policy Verification Framework for Real-time Systems. In *Proc. of WORDS 2005*, pages 13–20. IEEE, February 2005. 11

[36] Ninghui Li, Mahesh V Tripunitara, and Ziad Bizri. On mutually exclusive roles and separation-of-duty. *ACM Trans. Inf. Syst. Secur.*, 10(2):5, 2007. 12

[37] Richard T. Simon and Mary Ellen Zurko. Separation of Duty in Role-based Environments. In *Proc. of CSFW 1997*, pages 183–194. IEEE, 1997. 12, 13

[38] M. Kuhlmann, K. Sohr, and M. Gogolla. Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In *Proc. of SSIRI 2011*, pages 108–117, 2011. 12, 72, 73

[39] Mark Strembeck and Jan Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Inf. Softw. Technol.*, 53:456–483, 2011. 13

[40] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-Based Access Control. *Computer*, 48(2):85–88, 2015. 15, 132

[41] OMG. Object Constraint Language. http://www.omg.org/spec/OCL/, 2012. 17

[42] Wei Dou, Domenico Bianculli, and Lionel Briand. Revisiting model-driven engineering for run-time verification of business processes. In *Proc. of SAM 2014*, volume 8769 of *LNCS*, pages 190–197. Springer, September 2014. 23

[43] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, June 2001. 47

[44] Helge Janicke, François Siewe, Kevin Jones, Antonio Cau, and Hussein Zedan. Analysis and run-time verification of dynamic security policies. In *Defence Applications of Multi-Agent Systems*, volume 3890 of *LNCS*, pages 92–103. Springer Berlin Heidelberg, 2006. 47

[45] Eclipse. Eclipse OCL tools. http://www.eclipse.org/modeling/mdt/?project=ocl. 48, 89

[46] Eclipse. Eclipse Epsilon. https://eclipse.org/epsilon/, 2014. 69

[47] James B. D. Joshi, Basit Shafiq, Arif Ghafoor, and Elisa Bertino. Dependencies and Separation of Duty Constraints in GTRBAC. In *Proc. of SACMAT 2003)*, pages 51–64. ACM, 2003. 72

[48] James B. D. Joshi and Elisa Bertino. Fine-grained Role-based Delegation in Presence of the Hybrid Role Hierarchy. In *Proc. of SACMAT 2006*, pages 81–90. ACM, 2006. 72

[49] Maria Luisa Damiani, Claudio Silvestri, and Elisa Bertino. Hierarchical Domains for Decentralized Administration of Spatially-Aware RBAC Systems. In *Proc. of ARES 2008*, pages 153–160. IEEE, March 2008. 72

[50] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A Location-Aware Role-Based Access Control Model. In *Proc. of ICISS 2006*, volume 4332 of *LNCS*, pages 147–161. Springer, 2006. 72, 73, 74

[51] Frode Hansen and Vladimir Oleshchuk. SRBAC: A spatial role-based access control model for mobile systems. In *Proc. of NORDSEC2003*, pages 129–141, 2003. 72, 73, 74

[52] SuroopMohan Chandran and J.B.D. Joshi. LoT-RBAC: A Location and Time-Based RBAC Model. In *Proc. of WISE 2005*, volume 3806 of *LNCS*, pages 361–375. Springer, 2005. 72, 73, 74

[53] Subhendu Aich, Samrat Mondal, Shamik Sural, and ArunKumar Majumdar. Role Based Access Control with Spatiotemporal Context for Mobile Applications. In *Trans. on Comput. Sci. IV*, volume 5430 of *LNCS*, pages 177–199. Springer, 2009. 72, 73, 74

[54] Abdunabi Ramadan, Al-Lail Mustafa, Ray Indrakshi, and France Robert B. Specification, Validation, and Enforcement of a Generalized Spatio-Temporal Role-Based Access Control Model. *IEEE Syst. J.*, 7(3):501–515, September 2013. 73, 74

[55] Frédéric Cuppens and Nora Cuppens-Boulahia. Modeling contextual security policies. *Int. JIS.*, 7(4):285–305, 2008. 73, 74

[56] Indrakshi Ray and Manachai Toahchoodee. A Spatio-temporal Access Control Model Supporting Delegation for Pervasive Computing Applications. In *Proc. of TrustBus 2008*, volume 5185 of *LNCS*, pages 48–58. Springer Berlin Heidelberg, 2008. 74

[57] Ramadan Abdunabi, Indrakshi Ray, and Robert France. Specification and Analysis of Access Control Policies for Mobile Applications. In *Proc. of SACMAT 2013*, pages 173–184. ACM, 2013. 74

[58] Hua Wang, Yanchun Zhang, Jinli Cao, and Jian Yang. Specifying Role-Based Access Constraints with Object Constraint Language. In *Proc. of APWeb 2004*, volume 3007 of *LNCS*, pages 687–696. Springer Berlin Heidelberg, 2004. 75

[59] Indrakshi Ray, Na Li, Robert France, and Dae-Kyoo Kim. Using UML to visualize role-based access control constraints. In *Proc. of SACMAT 2004*, pages 115–124, 2004. 75

[60] Çağdaş Cirit and Feza Buzluca. A UML Profile for Role-based Access Control. In *Proc. of SIN 2009*, pages 83–92. ACM, 2009. 75

[61] Mirco Kuhlmann, Sohr Karsten, and Gogolla Martin. Employing UML and OCL for designing and analysing role-based access control. *MSCS*, 23:796–833, 8 2013. 75

[62] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. of MODELS 2008*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002. 75

[63] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Inf. Softw. Technol.*, 51(5):815 – 831, 2009. 75

[64] D. Basin, M. Clavel, M. Egea, M.A.G. de Dios, and C. Dania. A model-driven methodology for developing secure data-management applications. *IEEE Trans. Sof. Eng.*, 40(4):324–337, April 2014. 75

[65] Dae-Kyoo Kim, Indrakshi Ray, Robert France, and Na Li. Modeling Role-based Access Control Using Parameterized UML Models. In *Proc. of FASE 2004*, volume 2984 of *LNCS*, pages 180–193. Springer Berlin Heidelberg, 2004. 75

[66] J.B.D. Joshi. Access-control language for multidomain environments. *Internet Computing, IEEE*, 8(6):40–50, Nov 2004. 76

[67] Rafae Bhatti, Arif Ghafoor, Elisa Bertino, and James B. D. Joshi. X-GTRBAC: An XML-based Policy Specification Framework and Architecture for Enterprise-wide Access Control. *ACM Trans. Inf. Syst. Secur.*, 8(2):187–227, May 2005. 76, 126, 127

[68] Qun Ni and Elisa Bertino. xfACL: An Extensible Functional Language for Access Control. In *Proc. of SACMAT 2011*, pages 61–72. ACM, 2011. 77

[69] Jaehong Park and Ravi Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004. 81, 127

[70] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999. 89

[71] Spring Tool Suite. Spring Boot. https://projects.spring.io/spring-boot/, 2014. 89

[72] Netflix. Zuul. https://github.com/Netflix/zuul/, 2014. 89

[73] Eclipse. Ecore. https://www.eclipse.org/ecoretools/, 2013. 89

[74] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proc. of OOPSLA 2007*, pages 57–76, 2007. 91

[75] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. In *Proc. of WebApps 2011*, Berkeley, CA, USA, 2011. USENIX. 106, 124, 127

[76] Transaction Processing Performance Council. Tpc benchmark w (web commerce) specification v.1.7. www.tpc.org/tpcw/spec/tpcw_V17.pdf, October 2001. 107, 124, 128

[77] Gregor Kiczales. Aspect-oriented Programming|. *CSUR*, 28(4es):154, 1996. 125

[78] Slim Kallel, Anis Charfi, Mira Mezini, Mohamed Jmaiel, and Karl Klose. From formal access control policies to runtime enforcement aspects. In *Poc. of ESSoS 2009*, LNCS, pages 16–31, 2009. 125

[79] J.A. Pavlich-Mariscal, T. Doan, L. Michel, S.A. Demurjian, and T.C. Ting. Role slices: A notation for rbac permission assignment and enforcement. In *Proc. of DBSec 2005*, volume 3654 of *LNCS*, pages 40–53, 2005. 125

[80] Tanveer Mustafa, Michael Drouineaud, and Karsten Sohr. Towards formal specification and verification of a role-based authorization engine using jml. In *Proc. of SESS2010*, pages 50–57. ACM, 2010. 125

[81] Salvador Martínez, Jokin García, and Jordi Cabot. Runtime Support for Rule-based Access-control Evaluation Through Model-transformation. In *Proc. of SIGPLAN 2016*, SLE 2016, pages 57–69, 2016. 125, 127

[82] Phu Hong Nguyen, Gregory Nain, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon. Model-driven adaptive delegation. In *Proc. of AOSD 2013*, pages 61–72. ACM, 2003. 126

[83] Phu H. Nguyen, Gregory Nain, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon. Modularity and dynamic adaptation of flexibly secure systems: Model-driven adaptive delegation in access control management. *Trans. TAOSD*, 11:109–144, 2014. 126

[84] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. Security-driven Model-based Dynamic Adaptation. In *Proc. of ASE 2010*, pages 205–214, New York, NY, USA, 2010. 126

[85] Michael S. Kirkpatrick and Elisa Bertino. Enforcing spatial constraints for mobile rbac systems. In *Proc. of SACMAT 2010*, pages 99–108, New York, NY, USA, 2010. ACM. 126, 127

[86] Olivier-Nathanaël Ben_David and Benoit Baudry. Toward a Model-driven Access-control Enforcement Mechanism for Pervasive Systems. In *Proc. of MDSEC 2012*, Innsbruck, Autriche, 2012. 126

[87] Gordon Blair, Nelly Bencomo, and Robert B France. Models@run. time. *Computer*, 42(10), 2009. 126

[88] Xin Jin, Ravi Sandhu, and Ram Krishnan. RABAC: Role-Centric Attribute-Based Access Control. In *Proc. of MMM-ACNS 2012*, volume 7531 of *LNCS*, pages 84–96. Springer, 2012. 132

[89] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@ runtime requirements. In *Proc. of Models (2012)*, 2012. 133