KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Managing Delegation and Conflicts of Interest in Role-Based Access Control

Nezar Nassr

Supervisor:
Prof. dr. ir. Eric Steegmans

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2023

Managing Delegation and Conflicts of Interest in Role-Based Access Control

Nezar NASSR

Examination committee:
Prof. dr. ir. Paul Sas, chair
Prof. dr. ir. Eric Steegmans, supervisor
Prof. dr. ir. Bart Jacobs
Prof. dr. ir. Yolande Berbers
Prof. dr. ir. Bart De Decker
Prof. dr. ir. Jeroen Boydens
Prof. dr. ir. Mehmet Aksit
(University of Twente, The Netherlands)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2023

© 2023 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Nezar Nassr, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

This dissertation is achieved as a result of my research during my doctoral journey. As I mark this significant milestone in my life, I would like to thank all people who have inspired me and guided me along the way.

I would like to thank my supervisor, Eric Steegmans, for giving me the opportunity to pursue my PhD under his supervision. I also thank him for the support and guidance during the journey. He has also given me the freedom to choose the direction of my research.

My special thanks goes to, Bart Jacobs, who provided me with exceptional help and guidance during the PhD, mainly during the implementation phase and the writing of the thesis. It has been a pleasure working with him and learning from him.

I would also like to take this opportunity to thank all the members of my supervisory committee for their support and interest in my research.

I express immense gratitude to my family for their endless support. My mother and father for their encouragement, my wife and kids for the patience and continuous support.

Abstract

Access control is a field in information security, which is used in any organization to regulate access to their most sensitive data and resources. Role-Based Access Control (RBAC) is one of the most used access control models in organizations. Tremendous amount of research was conducted on RBAC, which has generated many enhancements and broader coverage of access control policies over the years. The main advantage of RBAC over any other existing access control model, is that it simplifies the management of access rights by introducing the concept of *role*, a reusable element used to map users to their access privileges (permissions). Roles represent groups of permissions used by users often playing the same role in the organization.

Despite robustness of basic RBAC in managing assignment of access rights to users via the role concept, it is not sufficient as a comprehensive access control model. Two main access control features, which are delegation of access rights and mitigation of conflict of interest are necessary to guarantee applicability of RBAC in a wider range of organizations.

Access rights delegation is a mechanism of performing a takeover on a user's access rights. Delegation gives authority of a user on another user's access privileges to perform functions of the user originally assigned to the delegated access rights. While extensions were made to basic RBAC to provide delegation of access rights features, such extensions are not accepted by most organizations due to not considering the lines of authorities in organizations in the delegation process.

Conflict of interest is one of the major risks that organizations face. Conflict of interest could happen when the decision of the user in the workplace is influenced by his personal interests. This issue can lead to more serious consequences such as corruption, fraud, and cause reputation damage for the organization. A robust access control model must provide capabilities for deterring such risks before they could happen. Existing RBAC extensions related to this field are

focusing mainly on just one aspect of conflict of interest, which can be deterred by separation of duty policies. However, this covers a subset of what can happen in terms of conflict of interest.

This dissertation provides a new approach to access rights delegation and mitigation of conflicts of interest. The access rights delegation model incorporates organization structures to include the lines of authority in an organization in the delegation process, which complies with organizations policies towards approvals of the delegation requests by the line managers of both the delegator and the delegatee. More so, a revamped version of the basic RBAC model, which enables definition of sophisticated context policies is provided to serve as a basis for the conflict of interest mitigation policies. Our approach to conflict of interest policies goes beyond separation of duties, it utilizes algebraic expressions to defuse a wider range of conflict of interest. The algebraic expressions support definitions of expressions involving users, actions, and workflow steps.

Beknopte samenvatting

Toegangscontrole is een domein binnen de beveiliging van informatie dat door organisaties gebruikt wordt om toegang te verlenen tot gevoelige informatie en subsystemen. Toegangscontrole gebaseerd op rollen (RBAC) is één van de meest verspreide modellen voor toegangscontrole in organisaties. Er werd door heen de jaren reeds heel veel onderzoek gedaan naar RBAC, dat geresulteerd heeft in aanzienlijke verbeteringen alsook in een bredere ondersteuning van strategieën voor toegangscontrole. Het grote voordeel van RBAC in vergelijking met andere modellen voor toegangscontrole zit in de notie van een rol dat het beheren van toegangsrechten aanzienlijk vereenvoudigt. Een rol maakt het immers mogelijk om eindgebruikers te mappen op privileges die ze hebben om bronnen te consulteren. Een rol impliceert een reeks permissies voor eindgebruikers binnen hun organisatie.

Ondanks de robuustheid van het het basismodel voor RBAC, is het niet krachtig genoeg voor een uitvoerige en fijnkorrelige toegangscontrole binnen organisaties. Zo biedt het basismodel van RBAC geen ondersteuning voor het delegeren van toegangsrechten. Ook moet het mogelijk zijn om soepel in te spelen op belangenconflicten die zich kunnen stellen tussen eindgebruikers.

Het delegeren vsan toegangsrechten is een mechanisme om toegangsrechten tijdelijk of permanent over te dragen naar andere gebruikers. In de literatuur werden diverse uitbreidingen voorgesteld aan het basismodel voor RBAC om delegatie mogelijk te maken. In de praktijk worden deze uitbreidingen niet toegepast omdat ze los staan van de hiërarchische organisatie van het personeel binnen de organisatie.

Het oplossen van belangenconflicten vormen een grote uitdaging voor organisaties. Dergelijke conflicten doen zich typisch voor wanneer beslissingen conflicteren met persoonlijke belangen van eindgebruikers. Dit kan resulteren in corruptie of fraude, en kan daardoor de organisatie grote schade toebrengen. Een robuust mechanisme voor toegangscontrole moet in staat zijn om te anticiperen

op dergelijke risico's. Bestaande uitbreidingen aan RBAC richten zich in hoofdzaak slechts op één facet van belangenconflicten, met name conflicten die kunnen vermeden worden door plichten van eindgebruikers strikt gescheiden te houden.

Dit werk introduceert een nieuwe aanpak voor het delegeren van toegangsrechten en voor het bestrijden van belangenconflicten. Het delegeren van toegangsrechten wordt geënt bovenop de structuur van de organisatie. Meer in het bijzonder wordt bij het delegeren van toegangsrechten terdege rekening gehouden met de gezagslijnen binnen de organisatie. Hierdoor kunnen verzoeken om toegangsrechten te delegeren veel beter afgestemd worden op de verantwoordelijkheden binnen de organisatie van zowel de verlener als de ontvanger. Dit werk stelt een vernieuwde versie voor van het RBAC model, waarmee fijnkorrelige strategieën kunnen uitgewerkt worden om belangenconflicten hetzij te voorkomen, hetzij op te lossen. De aanpak die in dit werk wordt voorgesteld gaat veel verder dan het strikt gescheiden houden van verantwoordelijkheden van eindgebruikers. Het model maakt gebruik van algebraïsche uitdrukkingen om een veel uitgebreidere set van belangenconflicten te kunnen aanpakken. Meer in het bijzonder voorziet het model in uitdrukkingen waarin gebruikers, acties en processen dominante concepten zijn.

Contents

Abstract iii

Contents vii

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Key Elements of Robust Access Control 2
1.2 Role-Based Access Control (RBAC) 6
1.3 Problem Statement 8
1.4 The Organizational Supervised Delegation Model 9
1.4.1 Delegation: The Need and Relation with Authority . . . 9
1.4.2 Characteristics of Delegation in the Context of Basic RBAC 10
1.4.3 Characteristics of Delegation in the Context of Extended RBAC 13
1.4.4 The Supervised Delegation Model (OSDM) 14
1.5 Mitigation of Conflicts of Interests 14
1.5.1 Separation of Duty (SoD) 14
1.5.2 Non-SoD Conflicts of Interest 15
1.5.3 Mitigation of Conflicts of Interest 16
1.6 The Role-Oriented Access Control Model (ROAC) 16
1.6.1 The Core ROAC Model 17
1.6.2 Hierarchical ROAC 18
1.7 Summary of Contributions 18
1.8 Outline of the Dissertation 21

2 Related Work 25
2.1 Access Control 25
2.1.1 Discretionary Access Control 26

2.1.2 Mandatory Access Control 27
2.1.3 Attribute Based Access Control (ABAC) 27
2.2 Role-Based Access Control 29
2.3 Background and Motivation 32
2.4 Hierarchical Role-Based Access Control 37
2.5 Parameterized Role-Based Access Control 40
2.6 Role Delegation Models 42
2.7 Conflicts of Interest and Authorization Policies 44
2.7.1 Overview of the algebra of Li and Wang 46
2.8 Administrative Models of Role-Based Access Control 48
2.8.1 ARBAC97 49
2.8.2 ARBAC99 51
2.8.3 ARBAC02: Role Administration Using Organization Structure 51
2.8.4 Role Hierarchy Administration 52
2.9 Chapter Conclusion 53

3 Organizational Supervised Delegation Model (OSDM) 55
3.1 Introduction 56
3.2 Overview of Organizational Structures 58
3.3 Related Work 59
3.4 The Organizational Supervised Delegation Model 61
3.4.1 Extensions to RBAC 62
3.4.2 Delegation in OSDM 64
3.4.3 A UML/OCL Formal Model of OSDM 67
3.4.4 Revocation in OSDM 70
3.5 Discussion 70
3.6 Conclusion 71

4 ROAC: A Role-Oriented Access Control Model 73
4.1 Introduction 74
4.2 Background and Motivation 76
4.3 The Role-Oriented Access Control Model Overview 78
4.4 ROAC Reference Data Model 80
4.5 Generalization in the Role-Oriented Access Control Model . . 84
4.6 Discussion 88
4.7 Conclusion and Future Work 89

5 Mitigating Conflicts of Interest by Authorization Policies 91
5.1 Introduction 92
5.2 Related Work 95
5.3 Overview of the Role Oriented Access Control Model (ROAC) . 97
5.4 Conflicts of Interest Policies 98

5.4.1 Extensions to the ROAC Model . . . 99
5.4.2 Specification of Conflicts of Interest Policies . . . 101
5.4.3 Conflict of Interest Policy Enforcement . . . 107
5.5 Discussion . . . 108
5.6 Conclusion . . . 109

6 Comparison, Limitations, and Verification 111
6.1 Comparison with ABAC . . . 111
6.1.1 Expressiveness . . . 112
6.1.2 Least Privilege . . . 113
6.1.3 Complexity . . . 113
6.1.4 Maintainability . . . 113
6.1.5 Dynamicity . . . 114
6.1.6 Auditability . . . 114
6.1.7 Applicability . . . 115
6.1.8 Policy Specification . . . 115
6.1.9 Authorization Decision . . . 115
6.1.10 Policy Conflicts . . . 116
6.1.11 Conflict Detection and Resolution . . . 116
6.1.12 Hierarchies . . . 117
6.2 Limitations . . . 118
6.2.1 Limitations of OSDM . . . 118
6.2.2 Limitations of the ROAC Model . . . 120
6.2.3 Limitations of Conflicts of Interest Mitigation . . . 123
6.3 Verification . . . 125
6.3.1 Verification of Correctness . . . 126
6.3.2 Verification of Safety . . . 127
6.3.3 Verification of Liveness . . . 128

7 Conclusion and Future Work 131
7.1 Summary . . . 131
7.1.1 Authority Delegation . . . 131
7.1.2 Conflicts of Interest Policies . . . 132
7.1.3 The Core ROAC Model . . . 133
7.1.4 The Hierarchical ROAC Model . . . 135
7.2 Future Work . . . 135
7.2.1 Standardization of Policy Specification . . . 136
7.2.2 Centralized Access Control System . . . 136
7.2.3 Artificial Intelligence . . . 136

A A Motivating Example 137
A.1 Introduction . . . 137
A.2 Policy Elements and Relations . . . 138

A.2.1 Elements 138
A.2.2 Relations 140
A.3 Core Model Policies 141
A.3.1 Users 141
A.3.2 Roles 141
A.3.3 Permissions 141
A.3.4 User-Role 143
A.3.5 Role-Permission 143
A.3.6 User-Permission 144
A.4 Hierarchical Policies 144
A.4.1 Role Hierarchy 144
A.4.2 User Hierarchy 145
A.5 Delegation Policies 145
A.6 Conflict of Interest Policies 146

B ROAC Formal Model 149
B.1 The Core ROAC Model 149
B.1.1 The Basic ROAC Model (ROAC0) 149
B.1.2 The Parameterized ROAC Model (ROAC1) 157
B.1.3 Parameters 157
B.1.4 The Context-Aware ROAC Model (ROAC2) 164
B.2 A UML Formal Model of ROAC 173
B.2.1 Validators: 176
B.2.2 Parameters: 176

C The Hierarchical ROAC Formal Model 179
C.1 Role Hierarchy 179
C.2 User Hierarchy 183
C.3 UML Model 187
C.3.1 Elements & Relations Inheritance 188
C.3.2 Role Hierarchy 188
C.3.3 User Hierarchy 189

D Delegation Formal Model 193
D.1 Python Model 193
D.1.1 DelegationRelation 193
D.1.2 DelegationPolicy 194
D.2 UML Model 198

E Conflicts of Interest Formal Model 201
E.1 The History Data Structure 201
E.2 Enforcement of Task Steps Order 203
E.3 Enforcement of Algebraic Expressions 206

E.3.1 Examples of leaf evaluation 208
E.4 Example Policy Enforcement 211

Bibliography **215**

Publications **227**

List of Figures

1.1 The comprehensive ROAC Model 17

2.1 A basic ABAC authorization scenario 30
2.2 The ANSI Core RBAC 32

3.1 Software development department users hierarchy of the organizational structure. 59
3.2 Modeling the hierarchy of users of an organization structure by a general tree data structure 63
3.3 The activity diagram of the delegation process 66
3.4 The class diagram of OSDM 68

4.1 UML diagram of the ROAC model. 80
4.2 An example role and permission. 85
4.3 Name conflict resolution in ROAC. 86

5.1 The UML diagram of the ROAC model and its extensions. . . . 98
5.2 Activity diagram showing the remittance payment business process. 99

A.1 Bank branch user hierarchy 146

B.1 Relationship among the core ROAC models 150
B.2 Relationship among the elements of ROAC0 150
B.3 Relationship among relations of ROAC0 153
B.4 The ROAC1 model 158
B.5 The ROAC2 model 166
B.6 The activity diagram of the ROAC2 authorization process . . . 169
B.7 The UML Meta Model diagram of the ROAC model 175
B.8 An Object diagram of sample users, roles, permissions and relations 177
B.9 The UML diagram of validator 178
B.10 The UML diagram of parameter 178

C.1 Example of matrix organization structure 183
C.2 A directed graph modeling the users hierarchy of a matrix organization structure. 185
C.3 Generalization arrows of inheritance and hierarchy 188
C.4 Inheritance of ROAC elements and relations 189
C.5 UML class diagram of role hierarchy. 190
C.6 Object diagram of an example multiple role hierarchy. 191
C.7 The UML Class diagram of user component. 191

D.1 The UML diagram of OSDM 199

E.1 The UML diagram of the ROAC model with history data structure 202
E.2 Binary tree representation of expression (7.3) 207
E.3 Evaluation of algebraic expression for policy of a payment task exceeding 500K 214

List of Tables

1.1 Access control mappings 7

2.1 Example of a DAC access matrix 26

6.1 Comparison between ROAC and ABAC 118

A.1 Sample users and their attributes 138
A.2 Sample roles and their target assignees 138
A.3 Sample permissions mapped to the operations they protect . . 139
A.4 Sample user-role assignments 140
A.5 Sample role-permission assignments 140
A.6 Sample role policies 141
A.7 Sample user-role policies 143
A.8 Sample role-permission policies 143
A.9 Sample user-permission policies 144

E.1 User-role assignments 208
E.2 Parameter bindings of the task instance variables 212

Chapter 1

Introduction

The emergence of the concept of digital transformation has increased the reliance on software applications by different organizations. Digital transformation means that services are transformed into a digitized form. This has changed the way operations in organizations are conducted. For example, most organizations are deploying their applications on the cloud. Most organizations provide services that used to require their customer presence in person, these services in recent years are being delivered online. Most governments provide a wide range of services digitally. Financial institutions are closing more branches day after day [39], as their clients can do most of their services online. Furthermore, an increasing number of organizations allow their employees to work remotely, which means that organizations must make their infrastructures accessible from the outside.

Digital transformation created new requirements for access control. Existing traditional access control concepts are not adequate to satisfy the needs of organizations. One aspect of digital transformation is guaranteeing business continuity. Therefore, delegation of authority is of paramount importance to insure smooth transition of duties to other users in case of unavailability of a key resource. Furthermore, the reliance on digital services has caused a vast increase in fraud, and the year over year fraud cases are skyrocketing. According to the UK office of national statistics, there was a 56% increase in fraud and computer misuse at 6.9 million incidents in the period between October 2020 to September 2021 [3]. Fraud can happen by outsiders, insiders, or by outsiders with help from insiders. However, in case of insider fraud, the damage is usually more severe than outsider fraud. Access control can play a significant role in deterring insider fraud, which can be mainly done by defusing conflicts of

interests.

In this dissertation, we focus on two main access control dimensions, which are delegation of authority and mitigation of conflicts of interest, both features are studied in the context of role-based access control (RBAC). This introductory chapter starts by providing the key elements of modern access control, then it introduces role-based access control (RBAC). A brief problem statement is also provided. Then we dive into some details related to delegation of duties and conflicts of interest. This is followed by an overview of the contributions and an outline of the dissertation.

1.1 Key Elements of Robust Access Control

Organizations need to implement many internal and external controls to protect their resources. Statistics from financial institutions reveal concerning numbers of insider threats. It is crucial that any security controls that are designated to prevent or mitigate cyber threats should start by having in place a strong and efficient access control mechanism to regulate, control, and monitor access to all internal resources. There are several security controls that need to be realized by the underlying access control system. We can identify the following points as important and must be adhered to:

Least Privilege

This concept suggests that users should be given the minimum access rights needed to accomplish their roles. The least privilege concept restricts access of any resource that is not necessary for an individual user, consequently minimizing the risk of insider fraud. Access rights must be fine grained to the minimal action or resource existing in a software application in order to provide users with the minimum rights they need.

We can identify two challenges in implementing the least privilege concept in access control systems:

- The least privilege concept results in a tremendous number of access privileges, therefore, the access control system must provide an efficient mechanism for simplifying the management of access privileges.

- Some required features in access control have an adverse impact on the least privilege concept. Therefore, much overhead is incurred when adding new features or extending existing features.

Expressiveness and Context

When access rights are granted to users, the access control system should allow for different levels of access to users who are assigned the same access rights. In most organizations, users with similar roles have different authority levels according to factors such as seniority. For example, two tellers in a bank might be assigned the same access rights, but with different granularity levels; imagine a senior teller who is allowed to handle transactions with an amount threshold of 100K, while a junior teller who is assigned the same access rights, is allowed to a threshold of 50K. The underlying access control system must allow this kind of granularity without the need to introduce a new access right each time a user needs a different level of access to an existing access right. Without such expressive power, the access control system would explode with similar access rights but with different levels. This leads to an inconceivable management of access rights.

An efficient access control system would not only define access permissions but would also encapsulate the context for how and when resources are accessed. For example, some sensitive operations cannot be done by a single user and might require multiple users to be involved. The access control system should take into consideration the surrounding context of the environment in which it is deployed. Temporal properties for access rights are one major requirement. For example, if a user is on vacation or on sick-leave, access rights of that user should be frozen. The access control system should also consider the organizational chart of the organization as it represents lines of authorities, which are crucial for some access control features such as role delegation.

We can identify the following challenges when implementing expressiveness and context in access control systems:

- *Auditability*: expressiveness and context are usually achieved using policy rules and parameterization, which is more difficult to trace and audit than access permissions. Therefore, more efforts are needed to facilitate tracing and before-the-fact auditability.

- *Manageability*: context policies can be added gradually as required. Therefore, the access control system should provide a mechanism for enabling building on existing policies, and to be able to generalize policies on various levels, such as on all users granted to a specific role, on a user level, on a permission level, and on the relation level.

- *Specification and enforcement*: organizations usually segregate enforcement and specification. Therefore, the access control

system must support both phases and segregation between them. Furthermore, to provide unambiguous mechanisms for enforcement of designed policies.

Hierarchical Access Rights

Most organizations have a hierarchical, or pyramidal reporting structure, with a few senior individuals at the top of the hierarchy, and an increasing number as you go down the hierarchy. Usually, superiors inherit the roles of their subordinates. However, in many situations, superiors are not involved in the day-to-day functions of their subordinates. To implement this kind of hierarchy while complying with the least privilege principle, an access control model should provide selective inheritance of access rights. Yet, in case of the same line of authority, seniors usually inherit the full access rights of juniors of the same role. Role hierarchy in access control reduces the administrative burden of the model through modularity.

Access control systems attempt to project organization structures via access rights hierarchies. However, this does not reflect the real-world scenarios. Therefore, the main challenge in implementing the hierarchy concept is to change the concept itself. More specifically, to support two different hierarchies:

- *Organization structure*: the hierarchical policies must be able to reflect real organization structures, which must include an organization chart and administrative methods for finding subordinates of a given superior and vice versa.

- *Inheritance*: some roles in organizations can contain subsets of permissions of other roles. Therefore, inheritance of access rights can simplify their management. However, the concept must support selective inheritance in order to handle exceptions and maintain a maximum level of the least privilege concept.

Conflicts of Interest

Operations in organizations are usually managed through systems that implement workflows. For example, a cross-border payment in a financial institution represents a good example as it follows a sophisticated workflow. The payment is initiated from the bank client through a mobile or an online banking application, etc., which then goes into the core banking application where the ordering account is debited, and the beneficiary account is credited. The account holders: the ordering, the correspondent,

and the beneficiary customers are then screened against sanction lists and checked against fraud. Thereafter, the payment instruction is sent through the SWIFT network. Many banks implement an authorization step prior to execution of the transaction. After that, the payment can traverse another workflow, called a payment corridor, which might involve multiple corresponding bank nodes until it reaches the beneficiary bank.

Some of the activities in the workflow, such as payment authorization, investigating of sanction alerts, fraud, etc. are done by human users, others are automated. In addition, many organizations impose separation of duty requirements, which means multiple users must participate in one task, or in multiple steps in the workflow. Separation of duties has several notions. The Four-eyes principle is commonly used when two users are required to participate in the same action. For example, one user releases an alert caused by fraud analysis, then another user confirms or rejects the tentative action. The first user is usually called a maker, while the second is called a checker. This kind of redundancy is usually applied for sensitive tasks, for example, in case of high value transactions. The concept is sometimes extended beyond 4-eyes to 6-eyes or further. Separation of duties is often mandated to mitigate user errors, to deter insider fraud, and to defuse any unknown conflicts of interest.

We can identify the following challenges when implementing separation of duties in access control:

- The separation of duties concept is not enough to achieve the goal behind it. We believe a more generalized concept is needed, which we refer to as *conflicts of interest*. This concept should include more policy types aiming to deter other risks, which cannot be remedied by separation of duty policies in their current form.

- Systems currently utilize workflows in order to implement business processes. Therefore, the concept must integrate well with workflow systems to indicate where separation should occur and in which order.

- Separation of duties must be able to reduce the cost of its expensive enforcement as it requires extra resources from the organization. Therefore, an access control system must include a mechanism for finer grained and efficient enforcement of such polices.

- Separation of duty policies are often complex and very dangerous. Therefore, a solid expression language is needed to facilitate their definition. Moreover, a mechanism must be provided in order to

automatically enforce policies from the expressions, such that no unexpected alterations happen at enforcement phase.

Delegation of Duties

Delegation of duties is essential for organizations to ensure continuity of business when a human resource is absent. It involves assigning a given set of access rights from one user to another and is usually a temporary mandate.

Delegation of duties is a complex undertaking. Early on, delegated access rights should not be assigned without approval from a line of authority of both the delegatee and the delegator. Moreover, users cannot delegate their roles directly as most access control models suggest. Users do not have the authority to grant their access rights to others. Furthermore, delegated access rights must be controlled by properties such as temporal aspects. One main question is frequently asked: do we need to delegate all access rights of an absent user or only a subset?

The main challenges in implementing delegation of duties is its complexity due to the following reasons:

- Delegation has a lot of characteristics. A good delegation model must support all of them.

- Delegation must include a workflow supported by policies and decision matrices.

This dissertation focuses solely on requirements we believe necessary to realize delegation of duties and mitigation of conflicts of interest. However, organizations must implement other security controls that are out of the scope of this work in order to have maximum protection.

1.2 Role-Based Access Control (RBAC)

Access control determines who can access what and when. The main idea is to restrict access to resources from authorized subjects, i.e., principals. Access control existed in life thousands of years prior to the invention of computers. Access to properties has been restricted by means of keys, guards, etc.

In information security, access control represents access restriction of resources in an information system to authorized subjects. Resources can be data records,

objects, operations, functionalities, actions, peripherals, etc. Subjects are entities requesting access to an information system. Principals are subjects that can be uniquely identified by an account. Principals can represent human users or software processes seeking access to an information system. For example, a reconciliation application needs to access a payment system to reconcile payments. Access control restricts access of principals to an information system by using access rights or privileges. Principals can access a resource if they possess an access right to that resource. An access right regulates how a resource can be accessed e.g., read, write, and modify.

Access control is a mapping among principals, access rights, and resources. The following table demonstrates an example mapping:

Table 1.1: Access control mappings

User	Access Right	Permission
Sam	Read	File: xyz.txt
Sam	Write	File: xyz.txt
Sam	Modify	File: xyz.txt
John	Read	File: xyz.txt
John	Execute	File: xyz.txt

Several access control paradigms were introduced to manage the mappings among principals, access rights, and resources. The most known paradigms are discretionary access control, mandatory access control, attribute-based access control (ABAC), and role-based access control (RBAC).

Role-Based Access Control (RBAC) has gained a vast popularity in software applications and from research as well. Standard RBAC consists of three main policy elements: users, roles, and permissions. Users represent principals. Roles usually represent a job function in an organization, such as clerk, professor, etc. Permissions represent access rights or privileges to (or how to) access resources.

The mapping between principals and access rights in RBAC is achieved through roles. Permissions are assigned to roles, and users are assigned to roles. Both relations are many-to-many. A permission can be assigned to different roles and a role can be assigned to different users. A user can also be assigned multiple roles.

The main advantage of RBAC is that it simplifies the administration of access rights. Access rights (permissions) required by a job function are grouped together under one role, then the same role can be assigned to several users. This can greatly reduce the administration burden of access rights. Roles can be revoked from users and permissions can be revoked from roles.

RBAC received a lot of attention from researchers and from the industry, which led to plenty of enhancements and extensions to basic RBAC. Extensions included new features such as role hierarchies, separation of duty, and role delegations, while enhancements improved expressiveness through role parameters, as well as addition of spatial and temporal properties among others.

1.3 Problem Statement

One side-effect of the least privilege principle in RBAC is that some access rights get assigned to a small set of users. This might lead to an immense impact on the respective business unit, should these users not be available. Role delegation models were proposed on top of RBAC to address such limitation. However, existing role delegation models increase the complexity of RBAC models dramatically, as most of them add new relations for roles to control how a role can be delegated, such as the *can_delegate* relation. Furthermore, these relations do not take into account the organizational chart. For example, a relation that enables a professor role to delegate a teaching assistant role will allow a professor in computer science to delegate a teaching assistant role in chemistry. Most delegation models suggest that a role is delegated by its assignee, which is not valid in real world scenarios. More so, none of the existing models propose an approval mechanism for role delegation in-line with what happens in real life cases, where lines of authority have to approve any transfer or assignment of duties of their subordinates.

There are several extensions to RBAC tackling separation of duties. However, these models yet lack attention for the underpinning conflicts of interest. Distribution of a task accomplishment on multiple users cannot necessarily prevent conflicts of interest. More customized policies are needed to defuse a conflict between someone's interests and the organization's interests. Someone's authority must be restrained when an underlying action involves his family members or relatives. A bank teller must be restrained from being a beneficiary of transactions he creates or authorizes. Implementation of such policies in the current form of RBAC is very challenging. RBAC does not provide any possibilities to define context policies on the model elements nor on the relations. One cannot specify that a user cannot use his role outside his business hours, which means that the RBAC model is extremely limited in terms of deterrence policies. Specification and enforcement of conflicts of interest policies require an underlying access control model that has enough flexibility to support their definition.

1.4 The Organizational Supervised Delegation Model

In this section, we explain in which situations delegation is needed and the characteristics of delegation. Then we introduce our new delegation model, which is the organizational supervised delegation model (OSDM).

1.4.1 Delegation: The Need and Relation with Authority

Delegation is usually needed in many circumstances in organizations, examples are:

- Backup of roles [108]; when a key resource is absent, if his tasks cannot wait until he comes back. In this case, other users get delegated his duties to achieve at least the tasks that cannot be delayed.

- When users participate in activities that have a fixed duration, e.g. projects, users get delegated access rights related to tasks in that project. Then, the delegation ends by finishing the tasks within the project.

- At the moment a job becomes too complex, too diverse, or too voluminous for one user, the need for delegation arises to have someone else sharing the workload. In its simplest form, imagine the sole user with objectives and with no time to accomplish them [65]. In this case, the heavy load on the user might be temporary, so the management finds an available resource to share the load, or they decide to hire a new resource. However, in the latter case, the organization might delegate some responsibilities from the overloaded user to a temporary resource until the hiring of a new resource is completed.

- Centralization of authority. When an organization needs to reorganize functions and distribute functions from higher job positions to lower job positions in the organizational structure [108].

- Collaboration of work. Users need to collaborate with others to achieve specific tasks [108].

In organizations context, management is defined as the process of getting results achieved through subordinates. The manager has certain defined objectives (i.e., results) to be accomplished. Therefore, the manager is bearing the responsibility for the tasks to be done to achieve the results. Delegation facilitates that process by assigning responsibilities, delegating authority, and

exacting accountability by employees. When delegated duties, the delegatee must also have the authority to achieve the delegated responsibilities. However, the manager is still ultimately responsible. By assigning some of his or her responsibilities, the manager transfers or creates accountability. If the delegatee does not exercise the responsibility properly, the manager can always withdraw the authority. Delegation without control is abdication [65].

Therefore, the role of the manager in delegation is, with no doubt, a key role. However, more complicated authority exists in organizations, especially in matrix structures. In which, users report to two different managers: a functional manager and a project or product manager. The responsibilities in such setup are distributed over the two managers. Usually, the functional manager is responsible for the whole objectives of the user, he also approves his vacations, besides other responsibilities. Product or project managers are responsible for the tasks within the product or project in which the user is participating. Users might participate in different projects or products at the same time. More so, affiliation of a user in a project or product can be temporary. Once a project is completed, the user gets affiliated with another project, possibly, with a different project manager.

In functional and divisional organization structures, authority is clear, and therefore, the accountable for delegation of tasks is the direct line manager. However, in matrix organization structures, it is difficult to generalize delegation accountability. A key element in the authority matrix is if both delegator and delegatee are under the same manager.

1.4.2 Characteristics of Delegation in the Context of Basic RBAC

The following definitions explain the different characteristics of delegation as defined in [11]:

Totality:

Totality refers to if the role is completely or partially delegated. Total delegation happens when a role is completely delegated. The delegatee gets all permissions of the delegated role. In partial delegation, the delegatee gets a subset from the permissions of the delegated role.

Delegation in flat roles:

1. Total delegation:

In this case, the delegator delegates a role with all its permissions to the delegatee. The delegatee must not be a member in that role before the delegation. The delegated role is assigned to the delegatee with delegation relation instead of the original role assignment relation. This is important to identify delegated roles from roles that were originally assigned to users by the system administrators. The delegatee can start using the role after this step, and the delegator retains or not the power to use the delegated role according to monotonicity (see the next definition).

2. Partial delegation:

   In this case, the delegator only grants a subset of permissions of a given role to the delegatee. In existing delegation models, a temporary role is created and is assigned to the permissions to be delegated. The temporary role is then assigned to the delegatee with delegation. The delegatee can start using the delegated permissions after this step, and the delegator retains or not the power to use the delegated permissions according to monotonicity.

Delegation in hierarchical roles:

1. Total delegation:

   In this case, the delegator delegates a role with all its permissions to the delegatee. The delegatee then has the power to use the role plus all the ancestor roles of the delegated role. The delegated role is assigned to the delegatee with delegation and the delegator retains or not the power to use the delegated role according to the monotonicity of the delegation.

2. Partial delegation:

   This case is similar to the grant partial delegation in flat roles structure. However, the temporary role includes permissions that are directly assigned to the delegated role as well as permissions implicitly assigned through the role-role hierarchy relations.

Monotonicity:

Monotonicity refers to the status of the user-role relation between the delegator and the delegated role. In monotonic delegation, the delegator keeps possession of the delegated role. However, in non-monotonic delegation, the delegator loses power on the delegated role after delegation, either temporarily or permanently.

Permanence:

Permanence refers to the time duration of delegation. It has two types: permanent and temporary delegation. Permanent delegation means that the delegatee is permanently assigned to the delegated role (or subset of its permissions). Therefore, the delegatee cannot take the role back. The delegator can get the role by a new user-role assignment. Example of permanent delegation is when a user quits the organization or permanently changes his position within the same organization. Temporary delegations means that the delegation is bounded in time. The user loses access to delegated permissions upon expiry of delegation.

Levels of Delegation:

This term specifies if a delegated role or set of permissions can be further delegated. In single step delegation, delegated access rights cannot be further delegated. While in multi-step delegation, they can be further delegated to other users.

Multiple Delegation:

This refers to the number of people to whom a delegated role or subset of its permissions can be delegated at any given time.

Lateral Agreements:

Two types of agreements can be defined between the delegator and delegatee. Bilateral agreements specify that both delegator and delegatee need to agree on the delegation before it can be effective. In unilateral agreements, the delegatee does have a right to accept or reject the delegated role.

Administration:

Administration describes the actual administrator of the delegation. In self-acted delegation, the delegator is considered as the administrator of the delegation. In agent-acted delegation, an agent (third party user) is nominated to conduct the delegation. This can be used when the delegator is not available.

Cascading Updates

Once the delegation of a role or a subset of its permissions is achieved and the delegatee starts to use the delegated privileges, changes might afterwards occur to the delegated role or to the original user-role assignment of the delegator. Examples are the role is revoked from the delegator, new permissions are assigned to or revoked from the delegated role, changes to parameter values assigned to the delegator or changes to the role hierarchy of the delegated role. These changes are only applicable in case of monotonic delegation, in which the delegator keeps possession of the delegated role or permissions.

1.4.3 Characteristics of Delegation in the Context of Extended RBAC

Extending RBAC with parameterization and context policies incurs more challenging characteristics such as:

Handling of Parameters:

Existing delegation models tackle delegation in the context of RBAC, there is no existing delegation model that tackles delegation in parameterized RBAC. Delegation of a role or a subset of its permissions results in assignment of the delegated access rights to the delegatee. However, since delegation is meant to substitute a user (i.e. the delegator), parameter values needs to be treated within the delegation process itself.

Context Policies:

Context policies add another complexity layer to the delegation model. Since the delegatee is authorized on behalf of the delegator. Therefore, the context of the delegator rather than the delegatee needs to be validated when the delegatee activates a delegated role or permission. The same applies for the user-role context, where the delegatee-delegated role context is validated. The role, permission, and role-permission context policies are also validated.

1.4.4 The Supervised Delegation Model (OSDM)

Our extension related to delegation of authority starts by incorporating the organizational chart into the underlying RBAC model. This draws the lines of authority in the organization. Hence a new kind of hierarchy is introduced on top of users. A protocol is defined to control delegation according to the lines of authority. The delegation request can be initiated by three different parties, the delegator, the delegatee, or the line manager of the delegator. Once the request is initiated, the delegation request is sent for approval to the line managers of the delegator and the delegatee. The delegation operation is performed when both approvals are received.

1.5 Mitigation of Conflicts of Interests

In this section we explain the different types of separation of duties, then we provide some context on non-SoD risks. Afterwards, we introduce our approach to mitigation of conflicts of interest.

1.5.1 Separation of Duty (SoD)

We can categorize separation of duty into two different types: Static and dynamic separation of duties.

Static Separation of Duty

Static separation of duty (SSD) represents constraints on the assignment of users to roles. A common example of static SoD is defining mutually disjoint user assignments with respect to sets of roles. This means that if a user is assigned to one role, that user is prohibited from being assigned to a another mutually exclusive role. An SSD policy can be centrally specified and then uniformly imposed on specific roles [9].

A major drawback of static SoD is that it needs different sets of users for mutually exclusive tasks. In many cases, organizations cannot afford such cost. Imagine two tellers in a bank branch doing transactions, static SoD requires extra staff to approve their work. This drawback is addressed by dynamic SoD discussed in the next subsection.

Dynamic Separation of Duty

In dynamic separation of duty, if two task steps *T1* and *T2* are mutually exclusive, then both cannot be executed by the same user in the same task workflow instance. However, the user can perform task step *T1* for some workflow instances, while performing task step *T2* for other workflow instances. For example, if a check workflow is two task steps: *prepare check* and *issue check*, then the same user cannot issue a check prepared by himself [18].

Dynamic SoD addresses one major limitation of static SoD discussed in the previous subsection. In case of the two tellers of the bank branch, the bank can define *creation* and *approval* tasks as mutually exclusive. Then both tellers can be assigned the *creation* and *approval* permissions. Therefore, each teller can approve a transaction created by the other teller.

Dynamic SoD addresses the need of separated user sets for separated actions. However, both techniques can succeed in workflows, in which the mutually exclusive actions cannot be repeated. For example, consider a transaction workflow that has four different tasks: *create, modify, verify,* and *authorize*. The *verify* and *authorize* are mutually exclusive. The condition for dynamic SoD to work in this case, is when the transaction cannot be modified after verification or authorization.

1.5.2 Non-SoD Conflicts of Interest

By simply distributing an activity or a workflow over several persons does not necessarily lead to the elimination of conflicts of interest risk. For example, existing separation of duty models do not handle tasks in which the authorized user is a stakeholder. A user possessing permission to create a transaction, can create a transaction in which he is a beneficiary. Neither static SoD nor dynamic SoD can mitigate such risks. This is a major concern as it represents a conflicts of interest risk. Moreover, separation of duty does not help when a medical paper is reviewed by two reviewers who are financed by a pharmaceutical company proposing the paper. In this case, the work is distributed on two persons (or more) but still it does not prevent conflicts of interest. What is required in this case is a policy preventing researchers financed by an organization from reviewing papers proposed by that organization.

Separation of duty looks at blind distribution of tasks without the ability to rule out task participants who might have conflict of interest with the task. In addition to the extra costs of redundancy when distributing activities on multiple users. Therefore, more concrete policies are required to delimit conflicts of

interest. SoD policies should only be used when the risk cannot be mitigated by other policies. Therefore, providing more flexibility for addressing different types of conflicts of interest can reduce the cost of policy enforcement in organizations.

1.5.3 Mitigation of Conflicts of Interest

Our approach to mitigating conflicts of interest is based on the extension of an existing algebra for defining separation of duties policies. Our model supports parameterization of algebraic expressions and the usage of workflow variables in algebraic expressions. Incorporation of workflow into the algebra mitigates cases where the work can be manipulated after participation of involved parties in the workflow, consequently exposing the whole process to risk. As an example, if a policy requires two different users to participate in a high value transaction, the transaction is modified after both users have done their actions. Policies and constraints are defined in roles and permissions business logic, and hence to isolate policy definitions from the application code and business logic. The incorporation of organization charts into the RBAC model provides vital capabilities for mitigating conflicts of interest. Lines of authorities can be referenced easily for approval or participation in workflows, should a sensitive task require such involvement.

1.6 The Role-Oriented Access Control Model (ROAC)

Definition of sophisticated conflicts of interest policies requires that the underlying access control model supports definition and enforcement of such policies. Therefore, we propose a novel access control model: The Role-Oriented Access Control Model (ROAC).

The Role-Oriented Access Control Model (ROAC) is an access control framework, which incorporates concepts of the object-oriented paradigm in the definitions of its elements i.e., users, roles & permissions, and the relations between them, i.e. user-role assignment and role-permission assignment. Transplanting object-oriented concepts into access control elements provides a vast scalability and flexibility. As well, it caters for customizations needed by organizations when an access control model is applied.

The ROAC model is a comprehensive access control model, which has a standard definition for its elements and relations. The inheritance concept in the object-oriented concept enables for easy extension and customization of the elements by using inheritance. We should differentiate here between inheritance of object-

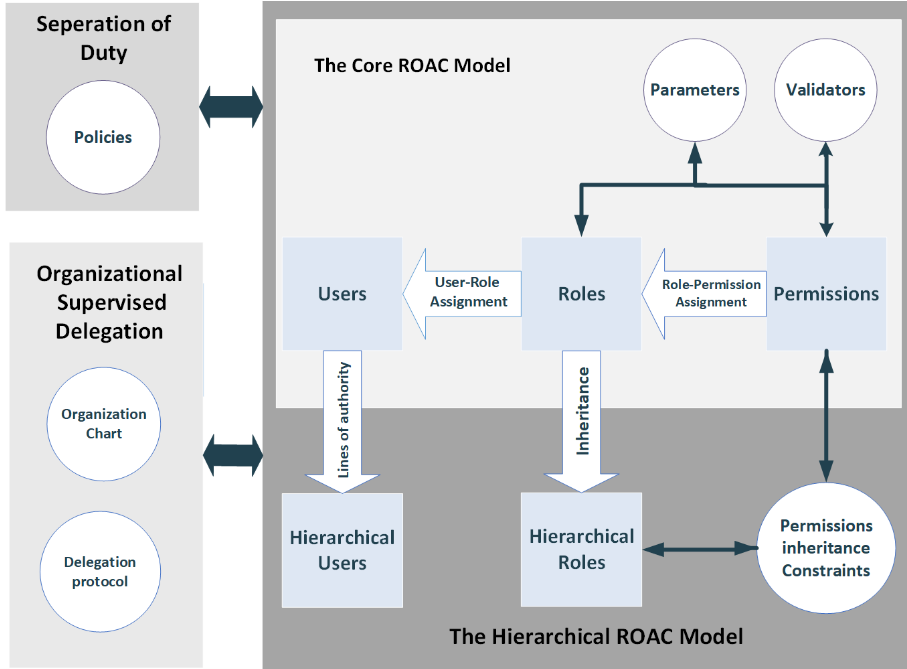


Figure 1.1: The comprehensive ROAC Model

oriented that we can apply for extensions and customizations from hierarchical roles. Object-oriented concepts also enable a reliable mechanism to remedy issues of existing RBAC models, such as expressiveness, context, conflicts of interest, and roles delegation. Fig. 1.1 shows the comprehensive ROAC model. In the following subsections, we provide a brief description of the different features of the ROAC model.

1.6.1 The Core ROAC Model

The core ROAC model plots the model elements and their relations. ROAC supports the parameterization of roles and permissions with attributes. Therefore, it is possible to setup multiple instances of the same role with different levels of granularity. More so, ROAC associates behaviors with roles and permissions. In specifying permissions, business logic can be defined in elements and relations to implement specific requirements for granting authorizations and controlling how the parameters are evaluated and compared against the values from resources. Elements and relations can be equipped with access

to external systems like databases and audit log systems to either extract or provide feedback. The core RBAC model includes other administrative functions such as revocation of roles and permissions.

1.6.2 Hierarchical ROAC

Hierarchical ROAC supports multiple inheritance, by allowing a role to inherit from multiple roles. Inheritance of role permissions in ROAC is based on selective inheritance rather than the *is-a* inheritance. To support this concept, ROAC classifies permissions of roles during inheritance in two sets, inclusive and exclusive. Only inclusive permissions of the super-role are granted to the sub-role.

1.7 Summary of Contributions

The main contribution this research achieved are: the organizational delegation model and conflicts of interest mitigation policies. Moreover, we propose enhancements on the RBAC model to support the definitions of more sophisticated conflicts of interest policies.

The contributions provided in this dissertation can be summarized as follows:

Supervised Organizational Delegation

The delegation model defined in this dissertation is based on the lines of authority in organizations, which is modeled through user hierarchies. Our model addresses limitations of existing delegation models, which mainly rely on a relation to authorize delegation that is used to determine which user can delegate to whom. The delegation relation of existing models (*can-delegate*) brings some disadvantages to the delegation model, the points below list these shortcomings, and show how our model addresses them:

1. The delegation relations in existing delegation models add complexity to the access control model. Large organizations typically have a rather large number of roles. By adding delegation relations for most of these roles, the entire system may explode. This is blocking for large organizations since they require huge efforts for defining and maintaining such a huge number of relations.

We address this limitation by eliminating the delegation relation. Lines of authority of the organization structure are used to determine who can delegate a role. This does not add any extra relations to the access control system.

2. The delegation relations cannot express precise conditions on who can delegate a specific role. As an example, consider a delegation relation that states that a professor can delegate the teaching assistant role. This means that a professor in the faculty of arts can delegate the teaching assistant role to a user in the computer science department.

   We address this limitation by utilizing the organization structure for modeling hierarchies. Approval requests are sent upwards the hierarchy. This guarantees that line managers can only approve requests related to their team roles.

3. The delegation relations may become inconsistent if updates to RBAC relations are allowed such as updates to the role hierarchy [28]. Such updates may occur when new activities are deployed in the organization. This adds huge efforts for the maintenance of the relations, specifically in cases of updates to roles such as adding or removing permissions, as well as updates to the hierarchies of roles. Such updates are likely to happen in organizations.

   The delegation mechanism of our model is not sensitive to the role structure. Furthermore, it is not sensitive to changes in role assignments to users or permission changes of the roles. It mainly relies on the organization hierarchy for initiation and approval of delegation.

4. Role delegation models suggest that a user who is delegating an access right has to be assigned to it [28]. This is not necessarily valid, since it is possible that the user possessing the access right to be delegated is absent. Furthermore, it is not guaranteed that another user who possesses the same access right is available, especially in case of emergency.

   The delegation initiation policy in our model has flexibility that allows the delegation to be initiated by different users, such as HR and the managers.

Users Hierarchy

Our delegation model requires correct projection of organization hierarchies of the underlying access control system. However, existing access control models attempt to project organization hierarchies via the role hierarchy concept. Role hierarchies cannot project organization hierarchies, as the role hierarchy concept

does not necessarily implement partial orders on the users. Two users assigned to hierarchical roles are not necessarily on different levels in the organization hierarchy, such as senior and junior users. Moreover, many organizations use the role hierarchy concept to group common permissions, and therefore, some super roles are not assigned to any user.

Our concept to user hierarchies incorporates users hierarchy according to positions and seniority in the organization. The users hierarchy projects the correct lines of authority and reporting levels of users. Furthermore, it caters for different organization types. Such as functional organizations, where each user reports to one manager, and matrix organizations, in which a user can report to multiple managers such as line manager, project manager, and product manager.

A Mechanism to Defuse Conflicts of Interest

Our approach to mitigation of conflicts of interest incorporates the workflow steps in the policy definitions, which eliminates ambiguities arise when enforcing policies. Furthermore, it eliminates attempts of scam to wrap around defined policies, by manipulating, for instance, the resource after meeting the defined policies. The conflicts of interest policies can be parameterized in an if-then-else fashion. Different policies can be applied depending on values of parameters. This enables specification and enforcement of more sophisticated policies depending on the severity of the action. For example, in doing a transaction, if the amount is small, then it is done by one teller. If the amount is medium, then it is approved by a second teller, if it is high value then a manager must get involved, and so on.

Other Contributions:

Context Policies:
The ROAC model is a context aware access control model. Context can be defined in any element or relation to realize complex policies. Context is validated prior to any access decision. Context policies can be defined on the level of the access control elements (i.e., users, roles, and permissions), and on the level of the relations (i.e., user-role and role-permission assignments).

Expressiveness:
Not only does the ROAC model add parameters to roles and permissions, but it also provides business logic in permissions to validate the parameters. Therefore, the way the parameters are matched against the values from a resource

(e.g., creation of a transaction) is modeled. Existing parameterized RBAC models do not provide any business logic with parameters. Therefore, only an equal operator is considered. The business logic provided with parameters can calculate and convert values. For example, if the parameter is a threshold in *EUR* currency and a *USD* transaction is encountered, the equivalent to *EUR* can be calculated. The relations of user-role and role-permission assignment can also be parameterized.

Authorization Decisions:
In RBAC, authorization decisions are determined based on possession of a permission or not. However, in the ROAC model it is not sufficient to possess the permission to get access to the corresponding protected object. An access decision is determined after validating parameters and context policies. This approach supports multi-factor authorization decisions.

Selective Hierarchies:
The ROAC model treats roles hierarchies as they are in real-world organizations. Unlike the *is-a* inheritance followed by existing models, ROAC gives the ability to choose which permissions are transferred to the sub-role from the super-role.

A Comprehensive Model:
The ROAC model features all aspects of access control in one model. It provides the needed features to define core policies, hierarchical roles, user hierarchy, role delegation, and conflicts of interest. The access control models provided in the literature focus only on a subset of the features required in access control. Building a comprehensive model out of existing models is a challenging task as most of them are tailored for addressing specific shortcomings of existing work without having a holistic view on the full features.

1.8 Outline of the Dissertation

The remainder of the thesis consists of a literature review chapter, four core chapters, a concluding chapter, and four appendixes.

Chapter 2 provides a state-of-the-art summary of related work, which presents a comprehensive background information related to access control. We start by providing brief information about two access control models, discretionary and mandatory access control, that were common prior to RBAC, and are still used such as in operating systems. Then role-based access control models (RBAC) are reviewed with focus on models and extensions related to our work such as delegation of authority, conflicts of interest, and context policies.

Chapter 3, Organizational Supervised Delegation Model (OSDM), provides an extension to RBAC to incorporate the organization structure into the underlying access control model. This extension is necessary to project correct lines of authority in organizations. The OSDM delegation model relies on user hierarchies for providing an approval mechanism on delegation. The delegation starts by an initiation through a request to delegate or to get delegated, then approvals are collected. Afterwards, the delegation is committed, and the access rights are in possession of the delegatee. This chapter is derived from the paper "OSDM: An Organizational Supervised Delegation Model for RBAC" by Nezar Nassr, Nidal Aboudagga, and Eric Steegmans. Published in the proceedings of the 15th International Conference on Information Security (ISC'12),September, 2012. Passau, Germany. [1].

Chapter 4, ROAC: A Role-Oriented Access Control Model, transplants object-oriented concepts in RBAC, which aims to support the definition of context policies on the level of elements and relations of the model. The chapter also describes parameterization of roles. Furthermore, the chapter addresses role hierarchies by proposing an advanced form of inheritance of permissions through selective inheritance. The chapter is based on the paper: "ROAC: A Role-Oriented Access Control Model" by Nezar Nassr and Eric Steegmans. Published in the proceedings of the 6th International Workshop on Information Security Theory and Practice (WISTP), Jun 2012, Egham, United Kingdom. [2].

Chapter 5, Mitigating Conflicts of Interest by Authorization Policies, extends an existing separation of duties algebra to widen its context to cover conflict of interest policies. The extensions integrate workflow steps, parameterization of expressions, and workflow order. The chapter is based on the paper "Mitigating conflicts of interest by authorization policies" by Nezar Nassr and Eric Steegmans. Published in the proceedings of 8th International Conference on Security of Information and Networks, September 2015, Sochi [3].

Chapter 6, Comparison, Limitations, and Verification, provides three main additional topics. It includes a comparison between the ROAC model and ABAC. Then, the chapter discusses some limitations of the OSDM delegation model, the ROAC model, and the conflict of interest mitigation policies. The chapter also includes some suggested mechanisms for verification of conflict of interest mitigation expressions.

Chapter 7 offers concluding remarks, summarizes the contributions of the thesis and identifies several avenues for future work.

[1] https://link.springer.com/chapter/10.1007/978-3-642-33383-5_20
[2] https://link.springer.com/chapter/10.1007/978-3-642-30955-7_11
[3] https://dl.acm.org/doi/10.1145/2799979.2800013

The formal model and Python implementation are provided in the appendixes.

Chapter 2

Related Work

This chapter reviews existing Role-Based Access Control (RBAC) models and their features. We focus mainly on the features that our model has to fulfill. We start by introducing access control, then we review some existing role-based access control models. Thereafter, we review some models of role hierarchies proposed in the literature. Following that expose, we show how parameterized RBAC has improved the expressive power of roles. Furthermore, we study features and requirements that were proposed to complement existing RBAC models, such as role delegation models and authorization policies. Then we briefly review administrative models of RBAC.

2.1 Access Control

Security has been of paramount importance for software systems since their inception. As a consequence, a substantial amount of research has been focusing on defining security standards and architectures for operating systems, distributed systems, databases, middle-ware and etc. The demand for information security has even increased after the deployment of applications on the Internet. Distributed business applications are required to be protected in terms of security to prevent and deter unauthorized access to confidential information. Moreover, they demand a way of regulating the user access to the system. A business application must determine who can access a software system as well as how a user can access the system.

An Access Control Mechanism (ACM) can be defined as: "The logical component

that serves to receive the access request from the subject, to decide, and to enforce the access decision" [44].

Since the introduction of the first discretionary access control model by Lampson [51]. Many access control models were proposed in the literature, of which only a few have gained wide acceptance. However, most of the access control models in the literature are floating around four main models which are Discretionary Access Control (DAC), Mandatory Access Control (MAC), Role-Based Access Control (RBAC), and Attribute Based Access Control (ABAC). Most of the other access control models are extensions or customizations of these models.

In this section, we provide a brief overview of Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Attribute Based Access Control (ABAC). Role-Based Access Control (RBAC) is explained in the next section.

2.1.1 Discretionary Access Control

Discretionary Access Control (DAC) defines and controls access between named users and named objects such as files and programs. Each resource must have an owner who controls propagation of the object's access privileges to other users. Access permission to an object by users not already possessing access permission shall only be assigned by authorized users [100]. Object access privileges are operations that can be executed on that object such as read, write and execute. Access privileges can be granted or revoked by owners of the object. DAC has been widely used in operating systems and databases.

DAC can be implemented by an access matrix, in which users are given access rights, e.g., read or write access to objects by the object's owners. The matrix has a row for each subject and a column for each object. Cells in the matrix contains the access rights. Authorization is defined by a set of commands which has a body and a condition. Conditions specify the access rights that are required to exist in the matrix before the command body can be executed. [41, 81]. The following table shows an example of an access matrix.

Table 2.1: Example of a DAC access matrix

	File1.txt	File2.exe
UserA	read, write	read
UserB	read	read

The access matrix size can become very large if the system contains many users and many objects. This can cause complexities in performance and management

of the matrix. The access control matrix can be implemented by different mechanisms such as access control lists (ACL) and capabilities.

2.1.2 Mandatory Access Control

Mandatory Access Control (MAC) restricts access to objects by a system-wide policy. Unlike in DAC, individual users can grant access in compliance with the policy and not based on ownership. Subjects and objects are assigned security levels. Access to objects by subjects is determined by the security levels of both subjects and objects. In a Mandatory Access Control (MAC) policy, subjects and objects can receive two kinds of security labels which are classification or clearance of the object and its formal category. Classification and clearance mean the allocation of an appropriate level of security to the object (e.g., for documents: confidential or restricted). Formal category represents the category of the object such as HR, finance, marketing, etc. [15].

Subjects and objects have security levels which are pairs of classification and categories. A subject can access the object if its security level dominates the security level of the object. Classification is hierarchical while categories represent an unordered set of category elements. The security level of the subject dominates the security level of the object if the classification of the subject is in the same level as that of the classification of the object or higher in the hierarchy and the categories set of the object is a subset of the categories set of the subject [15].

2.1.3 Attribute Based Access Control (ABAC)

ABAC [105] is one of the most widely used access control models. It has gained a lot of focus from research and a wide acceptance in the industry. ABAC supports both mandatory and discretionary access control needs. ABAC is defined by the National Institute of Standards and Technology (NIST) as "Attribute Based Access Control (ABAC): An access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions." [44]. Attributes represent properties or characteristics and are defined as name-value pairs. Subject attributes are related to the user requesting access such as username, job function, seniority, etc. Object attributes are related to the object itself such as type and sensitivity or related to the actions required on the object such as read, write, and execute. The environment conditions are

often called contextual attributes, which specify context such as what time the access request is initiated, location such as remote or on-premise.

An example of an attribute is that the age value assigned to *User1* is *30 { user1, (Age, 30) }*. An example policy on age is *{User, (Age >= 20)}*, which means that the user is granted access if his age is greater than or equal to 20.

When a subject requests access in ABAC, the access control decision is based on the attributes and a set of policies that are specified in terms of those attributes and conditions. Under this arrangement, policies can be created and managed without direct reference to potentially numerous users and objects. More so, users and objects can be provisioned without reference to the policy [44].

ABAC can cater for fine grained access policy needs. Attributes can be used to define custom access per users, even if they are doing the same job function. It can also take the environment context into account, which is a powerful capability in an access control system. Furthermore, a standard policy definition language (XACML [71]) has been standardized for defining ABAC policies and for defining authorization decisions.

Fig. 2.1. (from [44]) depicts a sample scenario in which a subject requests access to a protected object. The authorization decision is evaluated based on the rules, subject attributes, protected object attributes, and environment conditions.

Attribute-based access control (ABAC) is context-aware, it avoids the need for protected operations to be directly assigned to subject requesters or to their roles before the request is made. Instead, policies are activated when a subject requests access. An access control decision in ABAC is based on the assigned attributes of the requester, the assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions [44].

ABAC is very expressive for defining complex and custom policies. However, it suffers from the following two main shortcomings:

Complexity:
Defining policies in ABAC requires that a potentially large number of attributes to be managed and understood. Furthermore, selection of attributes is a complex task, as attributes have no meanings until they are associated with a user or an object [24]. Moreover, ABAC can also lead to a *rule explosion*, somewhat in the same way as RBAC. As a system with a large number of attributes would have an explosive combinations of possible rules [1].

One major issue with the high number of rules generated by ABAC is maintainability. Policies in organizations keep changing and by the introduction of new policies and updating existing policies, administration issues might

emerge due to conflicting policies, erroneous policies, and unnecessary policies.

Another complexity of ABAC is related to the lack of standardization of users' access. In RBAC, users who do the same job functions might be assigned to the same roles. However, in ABAC, it is more difficult to have a standard group of rules of the same users within the same job function. In many cases, new rules will be added to users for specific access requests, this yields in different policies associated to users with the same function.

Auditability:
One of the main security requirements is auditability. In access control, the term *before the fact audit* is used to refer to the ability to determine what a user can access. In RBAC, this is quite straight forward, we can check the roles assigned to a given user, then we can enumerate the permissions assigned to those roles. However, in ABAC, this is extremely difficult due to the enormous number of rules in the access control system. For example, to audit if a user access is in line with the least privilege concept, or to audit correctness of polices from a huge number of rules becomes a real challenge.

ABAC is an identity-less access control system and users may not be known before access control requests are made, it is often not possible to compute the set of users that may have access to a given resource. Even in cases where the identities of all users and their assigned attributes are known. In order to calculate the resulting set of permissions for a given user, all objects would need to be checked against all relevant policies [93]. Furthermore, the rules need to be checked in the same order in which the system applies them, as a result, it could be impossible to determine risk exposure for any given employee position [1]

2.2 Role-Based Access Control

Much research has been focusing on defining security standards and architectures for software systems and middle-ware. In fact, authorization models have occupied a considerable portion of security-based research. Mandatory Access Controls (MAC) and Discretionary Access Controls (DAC) where the dominant access control models used in military and civilian applications, until Role-Based Access Control (RBAC) [37] and Attribute-Based Access Control (ABAC) [105] have emerged. Since then, RBAC and ABAC and their related access control models have dominated access control.

The core model of standard RBAC, according to the ANSI [9], is expressed in terms of five elements which are: users, roles, permissions, objects, and

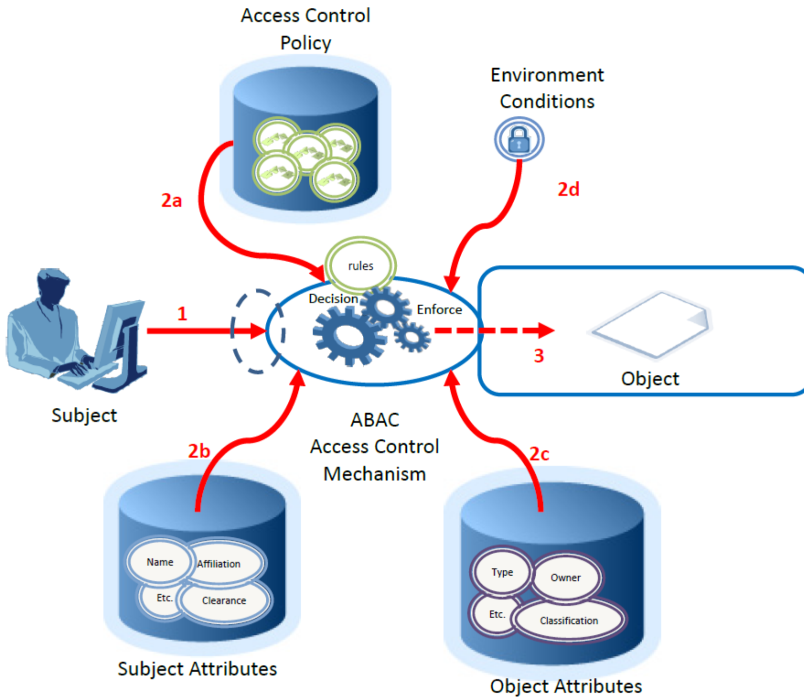


Figure 2.1: A basic ABAC authorization scenario

operations. Objects represent protected system resources to whom access is restricted. An object is an entity that contains or receives information, e.g., data records, files, peripherals, etc. Operations are executable program images that when invoked, provide some functions. In this context, operations are used to access a protected object. For example, read, insert, or update of a database record. Therefore, objects and operations are what needs to be protected by an access control system.

The notion of RBAC entails that users are assigned to roles and permissions are assigned to roles, and hence users get their access privileges in function of what roles they already have. In fact, this approach greatly simplifies the management of access control and is very practical in large organizations. Roles can be assigned to users and revoked from users when necessary. Furthermore, it facilitates updating role privileges by assigning required permissions to roles.

Permissions are privileges to perform an operation on one or more protected

objects. In many organizations, permissions can be mapped to a wider context, such as a small functionality in a software system that involves several operations on different objects. Roles represent job functions within the context of an organization, with some associated semantics regarding the authority and responsibility. Users are subjects requesting access to a system. They can be human users or application users. The center of gravity of RBAC relations is the role, which maps users to permissions. Core RBAC has two relations: user-role assignment and role-permission assignment. Both relations are many-to-many relationships, and so is the indirect relation between users and permissions.

The core RBAC elements and relations are shown in Fig. 2.2.

The simplest formal model of RBAC can be represented by the following relations:

- A set of users $U = u_1,\ u_2,\ .....u_n$,
- A set of Roles $R = r_1,\ r_2,\ ....r_n$,
- A set of permissions $P = p_1,\ p_2,\ ......,\ p_n$,
- A many-to-many user to role assignment relation:
  $UA = (u_1,r_1),(u_2,r_1),(u_2,r_2),(u_2,r_3).....(u_n,r_n)$, and
- A many-to-many permission to role assignment relation:
  $PA = (r_1,p_1),(r_2,p_1),(r_2,p_2),(r_2,p_3).....(r_n,p_n)$.

RBAC supports the principle of least privilege, which entails that users are assigned the minimum privileges required for achieving their functions. Ensuring least privilege requires identifying what the user's job is, determining the minimum set of privileges required to perform that job, and restricting the user to a domain with those privileges and nothing more. Through the use of RBAC, enforced minimum privileges for general system users can be easily achieved [37].

RBAC research can be broadly classified into two main categories: improvements to features existing in standard RBAC and extensions to standard RBAC. Improvements to standard RBAC have been mainly focusing on improving role hierarchies of the standard RBAC model and on improving expressiveness of roles by parameterization. Extensions to standard RBAC have been focusing on adding new features to RBAC such as supporting cross domain roles, role delegation models, etc.

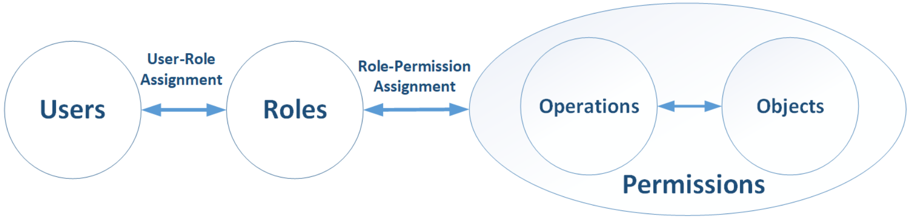


Figure 2.2: The ANSI Core RBAC

Several new access control models have emerged which in general address either limitations of RBAC or are targeted for special environments. We give hereafter a brief description of some of them.

2.3 Background and Motivation

RBAC is based on four principles: abstract privileges, separation of administrative functions, least privilege, and separation of duties [87].

Despite the robustness of RBAC, it received great academic attention from researchers. The literature shows many notable contributions that address limitations and suggest improvements to RBAC. However, in its current form, RBAC does not seem to have enough power to express a wide range of security requirements, nor to capture fine access control granularity when put into practice [7]. Shortcomings of standard RBAC other than expressiveness are scalability, incompleteness, and blind authorization decisions. Many models were proposed to address the shortcomings of RBAC. Yet, these contributions are considered as silos. Compiling a comprehensive RBAC model from all extensions is, by itself, a great challenge.

The RBAC96 Model

Since the wide acceptance of RBAC, which has emerged in the 1990s, RBAC96 [87] was a good shift in the direction of dominance of RBAC. A crucial insight of RBAC96 was the realization that RBAC can range from very simple to very sophisticated, by providing a family of models rather than a single model. A single model is too complex for some needs and simple for others. A graded family of models enables selection of the suitable model for a particular situation [83]. RBAC96 distinguishes between the concept of user groups and roles. Groups are treated as a collection of users and not as a collection of permissions. A

role is both a collection of users and a collection of permissions as well. Roles serve as intermediaries to bring these two collections together [87].

RBAC96 defines a family of four conceptual models. $RBAC_0$ is the base model of RBAC96, it indicates the minimum requirements for any system that professes to support RBAC. All the three remaining models of RBAC96 include $RBAC_0$. $RBAC_0$ consists of a set of users, a set of roles, a set of permissions, and a set of sessions. Users are assigned to roles in many-to-many relations and permissions are assigned to roles in many-to-many relations. Sessions are established when a user activates one or more of its roles to seek an authorization [87].

The model $RBAC_1$ adds role hierarchies to the $RBAC_0$ base model. Role hierarchies are reviewed in the next section.

The $RBAC_2$ model introduces the concept of authorization constraints to the $RBAC_0$ base model. Constraints are a powerful mechanism for laying out higher-level organizational policies [87]. Constraints provide means for further restricting authorizations such as separation of duties where sensitive tasks are distributed on several users to discourage fraud and corruption. We review constraints and authorization policies in detail in section 2.7.

$RBAC_3$ consolidates $RBAC_1$ and $RBAC_2$ to provide both role hierarchies and constraints in one model. There are several issues that arise by bringing these two concepts together. For example, a constraint that indicates that two roles are mutually exclusive so that they cannot be assigned to the same user. This might cause conflicts when assigning hierarchical roles to both users. Care has to be taken when using $RBAC_3$, constraints defined over roles must be also considered in the hierarchies as well [87].

The NIST RBAC Standard

The NIST RBAC model is a standardization of RBAC features that have achieved acceptance in the industry. The initial standard was published in 2004 and was distributed as INCITS 359-2004 [4]. The standard includes three key components: core RBAC, hierarchical RBAC, and constrained RBAC, which includes constraints for static separation of duty relations and dynamic separation of duty relations. The core RBAC defines the RBAC elements, element sets, and relations. The core RBAC also introduces the concept of role activation as part of a user's session within a computer system. The hierarchical RBAC component adds relations for supporting role hierarchies defining a seniority relation between roles. Static separation of duty relations (SSD) adds exclusivity relations among roles with respect to user assignments. The SSD relations model component defines exclusivity relations in both the presence

and absence of role hierarchies. The dynamic separation of duty relations define exclusivity relations with respect to roles that are activated as part of a user's session.

The standard was then revised in 2012, where two new standards were provided, INCITS 359-2012 [5] and INCITS 494-2012 [6]. INCITS 359-2012 was a revision of INCITS 359-2004, which included a more comprehensive and flexible framework for RBAC. INCITS 359-2012 provided enhancements to role activation, session, and role hierarchies. The INCITS 494-2012 is referred to as RBAC policy-enhanced standard, which provided a framework and functional specifications to handle the relationship between roles and dynamic constraints. Constraints can be used to restrict the use of certain permissions or roles based on specific conditions, such as context and environment (e.g. time of the day). This greatly enhances the granularity of the model and provides context-aware access control policies. The INCITS 494-2012 standard combines the best features of RBAC and ABAC, by allowing dynamic attributes in RBAC.

The Or-BAC Model

The Or-BAC [50] model introduces the concept of organization and the concept of context. An organization in Or-BAC represents a business unit, a company, etc. The specification of the security policy is completely parameterized by the organization so that it is possible to handle simultaneously several security policies associated with different organizations. The Or-BAC model aims at supporting the definitions of security policies that are not restricted to static permissions but also include contextual rules related to permissions, prohibitions, obligations, and recommendations [50].

The Or-BAC model considers a ternary relation between organizations, subjects, and roles. Subjects correspond to users or organizations. Or-BAC makes it possible to break down an organization into several sub-organizations. In Or-BAC, the entity *Role* is used to structure the link between subjects and organizations. To explain the ternary relation between organizations, subjects, and roles, consider a user called *Alice* who is a cardiologist working in the *Hospitalia* organization, the relation is modeled as follows: *Employee(Hospitalia; Alice; cardiologist)* [50, 63].

In Or-BAC, the definition of permissions, obligations, prohibitions, and recommendations involve different parameters, which are organizations, contexts, roles, activities, and views. Activities are tasks defined within an organization. Views are used to put together objects that apply the same authorization. Contexts are used to specify the concrete circumstances where organizations grant role permissions to perform activities on views. The relationship

Permission corresponds to a relation between organizations, roles, views, activities, and contexts. The relationships: *Prohibition*, *Obligation*, and *Recommendation* are defined similarly. If *org* is an organization, r is a role, v is a view, a is an activity, and c is a context, then *Permission(org; r; v; a; c)* means that organization *org* grants role r permission to perform activity a on view v within context c [50, 63].

To demonstrate an example of a permission definition, if *Hospitalia* is an organization, r is a role, v is a view and a is an activity, then *Permission(Hospitalia; r; v; a)* means that organization *Hospitalia* grants role r permission to perform activity a on view v [50].

The TRBAC Model

The Temporal Role-Based Access Control Model (TRBAC) [16] extends RBAC with temporal properties and constraints for enabling and disabling roles. Furthermore, TRBAC supports individual exceptions, and the possibility of specifying temporal dependencies among actions that are expressed by means of role triggers. A request by a user to activate a role is authorized if the user assigned to the role, the role is enabled at the time of the activation request, and no exceptions have been specified for the user for that particular role. Triggers are active rules that are automatically executed when the specified actions occur. For example, with TRBAC it is possible to enable a role during a time interval and disable it outside that time interval. Role triggers can also be used to constrain the set of roles which a particular user can activate at a given time instant [16].

In TRBAC, the enabling and disabling of roles is achieved by firing of a trigger which takes effect either immediately or after an explicitly specified duration of time. Enabling/disabling actions can be prioritized. This can help in solving conflicts, such as two simultaneous actions for enabling and disabling the same role. In this case, the action with the highest priority is executed. Exceptions can be specified to restrict role enabling and disabling. Exceptions makes it possible to selectively enable or disable a role only for specific users and keep it active for other users [16].

The ROBAC Model

The Role and Organization Based Access Control model (ROBAC) [110] is a family of models that address the issue where classic RBAC does not scale up well for applications spanning multiple organizations. In this kind of environments,

privacy issues are the main concern. ROBAC extends RBAC by basing access decision on both role and organization [109].

ROBAC utilizes both role information and organization information during the authorization process. The user-role assignment relation of RBAC is therefore changed. Users are assigned to roles and organization pairs instead of roles only. Moreover, permissions in ROBAC are defined as operations over object types instead of operations over objects only. A user can access an object if and only if the user is assigned to a role and organization pair, and the role is assigned a permission for accessing the object's type. Furthermore, the object being accessed must belong to the organization assigned to that user [110].

Like RBAC96, ROBAC has four models. $ROBAC_0$ is the base ROBAC model. $ROBAC_1$ extends the $ROBAC_0$ base model with role hierarchies and organization hierarchies. $ROBAC_1$ defines organization hierarchies in a similar way to role hierarchies. $ROBAC_1$ suggests that an object belongs to the organization assigned to the user in the role, organization pair, or any of its subordinate organizations when determining an authorization decision. $ROBAC_1$ considers the organization hierarchy as a hierarchy of multi-organizations, e.g., organization hierarchy of some different schools, rather than the ordinary organization structure and hierarchy explained by the organizational behavior concept, in which organization hierarchy shows different teams and management levels in an organization.

$ROBAC_2$ adds constraints to $ROBAC_0$ and $ROBAC_3$ consolidates both $ROBAC_1$ and $ROBAC_2$. Again, $ROBAC_3$ inherits problems of $RBAC_3$ described previously in this chapter. Some additional constraints on role hierarchy (RH) and organization hierarchy (OH) may need to be applied.

The GEO-RBAC Model

The GEO-RBAC [17] model is an access control model with spatial and location-based capabilities. GEO-RBAC secures access to spatial data in location-aware applications. GEO-RBAC extends the RBAC model with the concept of spatial role and supports the homogeneous representation of all spatial aspects involving roles, objects, and contextual information.

A spatial role in GEO-RBAC represents a geographically bounded organizational function, e.g., in a specific city. The boundary specifies the spatial extent in which the user is to be located for being enabled to play such a role. Besides a physical position, users are also assigned a logical position, which can be computed from real positions representing the feature in which the user is

located. The role is enabled if the user activating the role is located inside the spatial boundary of that role.

Like RBAC96, GEO-RBAC consists of three models which are: core, hierarchical, and constrained GEO-RBAC.

Group Based RBAC (GB-RBAC)

The Group Based RBAC (GB-RBAC) [58] incorporates the groups element into RBAC. Like RBAC, GB-RBAC consists of users, roles, permissions, and sessions. GB-RBAC also adopts the role hierarchies and role-permission assignment from RBAC. However, GB-RBAC adds groups, group-user assignment, and group-role assignment. Users can be assigned to one or more groups.

Furthermore, GB-RBAC incorporates two kinds of roles: system roles and group roles. System level roles can be assigned directly to users, while group roles are assigned to users via the group scope. This means the user can be assigned to the role if he is assigned to the group that has the role in the set of roles assigned to that group. The user holds permissions to access resources defined with the group-level role. GB-RBAC introduces the concept of default group role set, which is a set of roles that a user can obtain automatically when he is assigned to the group, without intervention of the security administrator.

2.4 Hierarchical Role-Based Access Control

Role hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility [87]. In standard RBAC, role hierarchies support multiple inheritance; meaning that a role can inherit permissions from multiple roles. The general role hierarchy concept in standard RBAC has two main properties; firstly, the possibility to derive roles from multiple roles, and secondly, the role hierarchy concept provides a uniform treatment of user/role and role/role relations. Users can be included in the role hierarchy, using the same relation to denote the user assignment to roles.

Standard RBAC [9] supports two different types of role hierarchies, the general and limited role hierarchies. General role hierarchy provides support for an arbitrary partial order to serve as the role hierarchy, and to include the concept of multiple inheritances of permissions and user membership among roles [9]. More so, standard RBAC supports the limited role hierarchy concept, in which hierarchies are limited to the single immediate descendant [9]. The role hierarchy

concept in standard RBAC suggests that when a senior role inherits from a junior role, all permissions of the junior role are transferred to the senior role.

Sandhu [87, 82] has introduced the concept of the private role, which is a role that cannot be further extended. It allows to keep some permissions private to a role and prevent their inheritance in the hierarchy when the role is inherited by a superior. In situations where users have private documents that they need to protect from their superiors, a new private role has to be introduced for each user. This results in an increased number of roles in the system. This counter-balances the advantage gained by using hierarchies which is reducing number of roles in the system [64].

The most familiar form of collaborative working is hierarchical in nature. In organizational hierarchies, the superior may not take part in the details of a task, but rather acts as the instigator of the task [10]. In other words, the most typical form of hierarchy in organizations is the supervision hierarchy [64]. More so, in some situations it is required to keep a role private and inhibit others from extending it. This has led to the development of new role hierarchy schemes.

Xuexiong et al. [104] have proposed an approach to tackle excessive inheritance that occurs when users get more permissions than they should have by permission inheritance. They resolve the issue by segregating role permissions into private permissions and public permissions. Then only public permissions are transferred through inheritance to superiors. If a role r has a set of permissions P, then P is divided into two sets P_{prv} for private roles and P_{pub} for public roles. When a senior role r_s inherits from r, only P_{pub} are transferred to r_s. The main drawback of this approach is that the private permissions of a role won't be inherited by any other role. In organizations, it might be the case that classification of permissions into private or public is different across superiors of a junior role. In this situation, it won't be possible to define the inheritance for such roles.

Cuppens et al. [30] proposed a hierarchical model for the Or-BAC model (reviewed above). The role hierarchy proposed involve both permissions inheritance and prohibitions inheritance. Furthermore, the authors identified two different relationships for role hierarchy, which are: relationships of specialization/generalization and relationships of organizational hierarchy. An example of the specialization/generalization relationship is the *cardiologist* role which is a specialization of role *physician*. An example of the relationship of organizational hierarchy is the role *department director* which is defined as hierarchically higher than the role *team leader*. Permissions are inherited downward in both hierarchies meaning that the more specialized role (e.g. cardiologist) inherits from the less specialized role (e.g. physician) and the senior role inherits from the junior role. Prohibitions are inherited downward

in the specialization/generalization hierarchy (as for permissions), whereas they are inherited upward in the senior/junior role hierarchy, meaning that the junior role inherits prohibitions from the senior role.

Jajodia et al. [48] proposed an interesting solution to the problem of inheritance of all permissions of a role in the hierarchical RBAC model. They label each role in the hierarchy with pairs such that the role label and the two other components of the pair (object and action) jointly determine a set of authorization triples *(o,s, (sign)a)*, where *(o,s, −a)* is an authorization triple, o is the object to be accessed, s is the subject(role) label , *sign* can be negative or positive, and a is the action to be executed on object o. The triple *(o,s,+a)* means that authorization subject s can execute action a on authorization object o. Similarly, the triple *(o,s,−a)* states that authorization subject s cannot execute action a on authorization object o. The negative sign in the action in the authorization triple will restrict the permission to execute this action.

Moffett and Lupo [64] identified the possible uses of role hierarchies in simplifying access rules, while remaining within the constraints of organizational control principles. They identified three role hierarchies: The $is - a$ role hierarchy, which is based on generalization, the *activity* role hierarchy, which is based on aggregation, and the *supervision* role hierarchy, which is based on the organizational hierarchy of positions. In the $is - a$ role hierarchy, each role is more general than the sub-role. Some of the roles in the $is - a$ hierarchy can be virtual, which means that no user occupies them; they are only defined to capture qualities, which are shared by real roles further up the $is - a$ hierarchy. The activity hierarchy depends on aggregation, which is also known as the *part of* relationship. Complex roles are aggregated from other roles, since activities in an organization might be composed from other smaller activities. The activity hierarchy is partially ordered by subsets of activities. This makes it possible to define a role hierarchy based on activities. For example, *ResponsibleFor* and *Does* are relationships between roles and sets of activities. If a role is *ResponsibleFor* an activity, then either it does it directly or it Delegates responsibility for it to another role. The activity hierarchy is then composed of a hierarchy of roles where the higher role is responsible for a superset of the activities of the lower role. The supervision hierarchy is the most interesting type of hierarchy identified by the authors, since the nature of relationships in organizations is supervisory between superiors and their subordinates. The role hierarchy in RBAC does not correspond to a conceptual relationship between the roles of an organization and, in particular, it does not reflect the supervision hierarchy on which most organizations are based. The supervision hierarchy is derived from the organizational chart, which depicts the positions hierarchy of the organization [64].

2.5 Parameterized Role-Based Access Control

Abstract RBAC suggests that each role has a fixed set of permissions. Two users who are assigned to the same role, consequently, have identical permissions. However, in organizations, different levels of access might be required for users playing the same role. Implementing this requirement is RBAC can only be achieved by adding more roles and more permissions.

Abdallah et al. [7] provide an example of the role *AccountHolder*, which has to be given to thousands of online/mobile banking users. Each user is only entitled to access his account. How could RBAC provide such expressiveness? it can only be done at the following cost, as stated by the authors:

- The implementation of the role *AccountHolder* is instantiated to a large number of roles to cater for every use, which presents a huge burden on the intellectual manageability of access rights.

- Implementing the *AccountHolder* role in RBAC reduces scalability of the access control system. A role definition is needed per user, even though the different roles are just instances of one role. However, they are treated as different roles instead of being grouped under one definition.

- Very complex and inconsistent administration burden, as similar roles treated differently and managed separately.

Lack of expressiveness in role definition has received attention from researchers. Indeed, slight nuances in a role can only be modeled by introducing a different role for each such nuance. Parameterized RBAC [7, 38, 47] is an advanced form of RBAC. It addresses a major drawback of basic RBAC, which is, its lack of expressiveness in defining roles. Parameterized RBAC includes the components of core RBAC, parameters data, and new parameterized permissions.

One of the good attempts to address lack of expressiveness of RBAC by using parameterized roles was defined by Jaeger et al. [47]. The formal definition of parameterized RBAC was introduced by Abdallah et al. [7]. Parameterized RBAC provides finer granularity by creating instances of RBAC elements according to the contexts of their use [7]. This is achieved by associating parameters with roles and permissions. Parameters are used to define the granularity level of the role.

Fischer et al. [38] proposed the object-sensitive RBAC (ORBAC), which is a generalized RBAC model for object-oriented languages. ORBAC addresses the lack of expressiveness of RBAC by using parameterized roles. In ORBAC,

privileged operations are parameterized by a set of index values, which are used to distinguish the granularity level of the roles between users. A privileged operation can only be invoked if both the required role is assigned to the user who invokes the operation and the role's index values matches the operation's index values.

Parameterized roles include additional information that can be consulted each time the role is used to determine an authorization request when invoking an operation or accessing an object. That information corresponds to parameters as they are known in ordinary programming languages, hence the name parameterized RBAC. In assigning roles to end users, actual values are assigned to each of those parameters. When consulting roles, actual parameter values are available. Obviously, in a parameterized context, instances of the same role might not necessarily behave in identical ways because of differences in parameter values. Permissions are parametrized as a result of the instantiation of the roles that would be parameterized.

The advantages most commonly ascribed to RBAC models still apply should the roles be modeled as parameterized RBAC. Core RBAC elements, such as roles, would depend on the values of a parameter. To extend RBAC into a parameterized model, parameters must be associated with roles and permissions then depend on both the role and values of parameters [7].

In the example of the *AccountHolder* role, the role can be parameterized by *AccountNumber* then the value of the parameter is passed to the role with each session. This can guarantee that a user is granted access only to his account. One role is defined, then instantiations are made according to the parameter. The privileges of two users holding the same *AccountHolder* role are not identical.

Despite the fact that Parameterized RBAC addressed expressiveness issues of core RBAC, it fails to completely address the problem. Parameterized RBAC enhances expressiveness to a certain limit. However, it fails to provide different levels of granularity of a role or a permission. Parameterized RBAC does not provide any means to define operators or how the parameter value are matched. The only possibility is to match using an equal operator. In many organizations, it is required that roles be customized according to users and their functions, since users assigned same job functions are not necessarily doing identical functions and do not necessarily have same levels of responsibilities. For example, consider the role *teller* that gives bank tellers permissions to execute transactions. It might be necessary to define several levels of the role for tellers in the bank according to the transaction amounts they are allowed to handle. To address this problem with parameterized RBAC, we need again to define multiple *teller* roles with different granularity levels, such as *SmallAmountTeller*,

MediumAmountTeller and *LargeAmountTeller*, etc.Parameterized RBAC can still be adapted to capture such fine grained authorizations by dramatically increasing the number of distinct roles. Parameterized RBAC cannot be used to define just one role of type *teller* with an amount as a parameter, because paramerized RBAC cannot match the amount using less or equal operator.

Another major disadvantage of parameterized RBAC, is that it does not provide any facility for dynamic calculation of parameters values. Imagine the *SmallAmountTeller* role in a bank that is assigned to a user which enables him to do up to *10,000*. What is the currency used, and what if a different currency is encountered? If the reference currency used is *EUR*, can we allow him to do *20,000 YEN*? as *20,000 YEN* is much less than *10,000 EUR*.

2.6 Role Delegation Models

Several role delegation models [12, 27, 106, 108, 42] studying delegation in the context of role-based access control have emerged. However, delegation was studied before RBAC was proposed, and there were some predecessors of RBAC delegation such as: the access matrix models which introduced the concept of copy flag, which allows users to delegate rights [41]. Wood and Fernandez [103] introduced the idea of reverting the rights to the upper level after revoking a low-level delegation. Graph-based delegation was introduced in [39]. A variety of delegation approaches were also introduced in [62]. In this section, we focus on delegation in the context of RBAC.

Delegation in RBAC can have several characteristics depending on the requirements of the environment where delegation is applied. The main characteristics of delegation were explained by Barka et al. [11]. These key characteristics include permanence, monotonicity, totality, administration, levels of delegation, multiple delegation, lateral agreements, cascading revocation, and grant-dependency revocation [11].

The first work that studied delegation in RBAC was achieved by Barka and Sandhu [12, 11]. They proposed the RBDM0 delegation model [12] which studied delegation in flat roles structure. RBDM0 focused on *grant total delegation*, which means that the delegator keeps the power to use the role after delegation and covers only the delegation of roles. RBDM0 does not support partial role permissions delegation. RBDM0 controls user-user delegation by means of the *can-delegate* relation. The *can-delegate* relation takes the form of *(a, b)* $\in$ *can-delegate*. It means that a user who is an original member of the role *a* can delegate his role to a user who is an original member of role *b*. Revocation in RBDM0 can happen in two ways: firstly, by time outs. Delegations are

revoked when the delegation period expires. Secondly, any original member of the delegated role can revoke the membership of any delegate member in that role.

RDM2000 [106] was the first delegation model to address delegation with hierarchical roles. It also supports multi-step delegation. The *can-delegate* relation in RDM2000 takes the form: *can-delegate* $\subseteq R \times CR \times N$, where R are sets of roles, CR are prerequisite conditions, and N is the maximum delegation depth. The meaning of $(r, cr, n) \in$ *can-delegate* is that a user who is a member of role r (or a role senior to r) can delegate role r (or a role junior to r) to any user whose current entitlements in roles satisfy the prerequisite condition cr without exceeding the maximum delegation depth n.

The permission-based delegation model (PBDM) [108] was the first to address permission delegation (partial delegation). PBDM supports role as well as permission delegations with features of multi-step delegation and multi-option revocation. PBDM comprises in two models: PBDM0 and PBDM1. In PBDM0, permission delegation involves three steps. Firstly, a temporary delegation role is created by the delegator. Secondly, the permissions to be delegated are assigned to the temporary role with permission-role assignment. Thirdly, the delegator assigns the temporary role to the delegatee by user-role assignment. Revocation in PBDM0 includes three cases: by revoking the delegated role, by removing one or more pieces of permissions from the delegated role, or by revoking the user-delegation role assignment.

PBDM1 extends PBDM0 with two main features. Firstly, it adds support for role-role delegation, which supports delegating specific permissions of a role to another role rather than to another user. Secondly, it adds means for controlling delegation; to restrict delegation only to authorized users. This is achieved by the *can-delegate* relation, which takes the form: *can-delegate* $\subseteq DBR \times \textit{Pre-con} \times \textit{P-Range} \times M$, where DBR are sets of delegable roles, *Pre-con* are prerequisite conditions, *P-Range* is the delegation range that specifies which permissions can be delegated, and M is the maximum delegation depth.

Crampton et al. [28] proposed a new model for dealing with transfer delegation. In transfer delegation, the delegator loses the power of using the access right after delegation is completed. They also proposed two relations for controlling delegation: the *can-delegate* and the *can-receive* relations. The advantages of using different relations for controlling delegation include flexibility, greater control, ease of management, and is less error prone. They also included constraints on the *can-delegate* and *can-receive* relations to ensure that the relations do not give the authority to a delegator to delegate a right that is not available to him.

The capability based delegation model [42] is an interesting work based on the capability based access control model (CRBAC) presented in the same paper. The CRBAC model integrates a capability-based access control mechanism into the RBAC96 model. Roles and permissions are assigned to capabilities and capabilities are assigned to users. Delegation is achieved by creating a new capability, then by assigning delegable authority (roles or permissions) to the capability, then the delegator sends the capability to the delegatee. Unlike the other delegation models, delegation is authorized by a permission that the delegator must possess for creating the capability.

2.7 Conflicts of Interest and Authorization Policies

The notion of conflicts of interest has existed in the real word since a long time. However, it has been brought to the lights after starting the concept of banking, commercial, and medical organizations. Yet, conflicts of interest did not receive adequate attention in the context of access control. Most researchers have focused on separation of duty policies, which do not adequately model conflicts of interest policies.

In this section, we review some existing authorization policy models. These models focus on separation of duty since the literature lacks to any broad authorization policy models that discuss wider range of conflicts of interest policies.

Clark and Wilson [23] showed how separation of duty is a fundamental principle of commercial and military data integrity control. The paper of Sandhu [80] was among the first to describe a mechanism for the purpose of enforcing separation of duties in computerized information systems, before role-based access control existence.

Li et al. [54] proposed the statically mutually exclusive roles (SMER) constraints to enforce static separation of duty (SSoD) policies. They have shown that directly enforcing SSoD policies is intractable, while enforcing SMER constraints is efficient. Furthermore, they have characterized the kinds of policies for which precise enforcement is achievable and shown what constraints precisely enforce such policies. They have also presented an algorithm that generates all singleton SMER constraint sets, each of which minimally enforces a role-level static separation of duty requirement. SMER constrains are limited to static separation of duty constrains and hence unable to model a large set of conflicts of interest policies.

Bertino et al. [18] proposed a language for defining constraints on role assignment

and user assignment to tasks in a workflow. The constraint language supports both static and dynamic separation of duties. They have also devised algorithms to check the consistency of the constraints to consistently assign roles and users to tasks in the workflow.

RCL2000 [8] is a role-based constraints specification language built on RBAC96 [87] components. RCL2000 encompasses obligation constraints in addition to the usual separation of duty and prohibition constraints. RCL2000 can express both static and dynamic separation of duty constraints. RCL2000 does not show how constraints written in this language can be efficiently enforced. However, RCL2000 fails to express history or time-based constraints, which are increasingly being used [26]. This prevents defining a wide variety of conflicts of interest policies.

Constraints enforcement have even received less attention from researchers. Crampton et al [26] have introduced the concept of a constraint evaluation structure that is used by the constraints enforcement mechanism to determine whether granting a request would violate a constraint. Two particular constraint evaluation structures form part of the run-time model they introduce in order to enforce dynamic constraints. They have built a model for historic information which is used to record information than is required for enforcing historic constraints.

Probably the best work that has been proposed in this field is the work of Li and Wang [56], in which they propose an algebraic language for formal specification of high-level security policies. It combines qualification requirements with quantity requirements motivated by separation of duty considerations. The algebra has two unary and four binary operators, and is expressive enough to specify a large number of diverse policies [56]. The language is used for high level policy specification. As an example of the algebra, a policy that requires either a manager or two different clerks is expressed using the term *Manager* ∪ *(Clerk* ⊗ *Clerk)).* The algebra focuses on separation of duty policies, it provides high level security policy specification. It is not designed for a specific authorization model. Furthermore, it does not verify whether a workflow is compliant with a high-level security policy specified in the algebra. It assumes zero knowledge of the policy designer about the workflow steps. Specified policies in the high-level design can be ambiguous in the enforcement design. Furthermore, its expressiveness is limited and cannot address the definition of many conflicts of interest policies.

The work of Li and Wang [56] was extended in [14]. The authors addressed some problems which they reported in the algebra. Firstly, they addressed the problem related to the fact that no general mapping from the algebraic terms onto workflows or to dynamic enforcement mechanisms existed. In particular, a

link between the satisfaction of sub-terms and the actions executed in workflows was missing. Furthermore, they addressed the problem of how changing role assignments affect the enforcement of SoD constraints during workflow execution. They constructed formal models of workflows, access-control enforcement, and SoD constraints using the process algebra CSP [14]. This extension of the algebra also inherits the problems of expressiveness and ambiguity of the algebra.

2.7.1 Overview of the algebra of Li and Wang

The main advantages of the alegbra over other existing SoD work, is that it captures requirements for user attributes. Existing SoD models requires a k different users to be involved in a sensitive task. However, there are other minimal qualification requirements for these users, such as te position of the users involved, e.g. task *T1* must be done by a bank branch manager. This is not covered in other existing work, partly due to the lack of a concise-yet-expressive language for specifying such high-level security policies.

The terms of the algebra are composed from operands and operators. The operands can be roles e.g. *Manager* or users e.g. *Alice*. Operators of the algebra are: $+$, $\neg$, $\sqcap$, $\sqcup$, $\odot$, and $\otimes$. The unary operator $\neg$ has the the highest priority, followed by the unary operator $+$, then by the four binary operators ($\sqcap$, $\sqcup$, $\odot$, and $\otimes$), which have the same priority. The key word *All* refers to a user holding any role.

The operator $\neg$ means logical *NOT*. The term $\neg$*Alice* means that the user taking the action must not be *Alice*. The operator $+$ means multiple. For example, $Manager^+$ means one manager user or more.

The operator $\sqcap$ represents a logical *AND* when applied on roles, which evaluates to just one user, $r1 \sqcap r2$ means that the user must have roles $r1$ and $r2$. And represents set intersection when applied on sets of users. e.g. $\{User1, User2\}$$\sqcap$\{User2, User3\} results the set of users $\{User2\}$.

The operator $\sqcup$ represents a logical *OR* when applied to roles, which evaluates to just one user, $r1 \sqcup r2$ means that the user must have role $r1$ or $r2$. And represents set union when applied to sets of users. e.g. $\{User1, User2\} \sqcup \{User2, User3\}$ results the set of users $\{User1, User2, User3\}$.

The operator $\odot$ represents a logical *AND* that evaluates to one or different users. For example, $(Teller \odot BranchManager)$ means two users, one must be a *Teller* and the second must be a *BranchManager*. If one user is both a *Teller* and a *BranchManager*, that user by himself also satisfies the requirement.

The operator $\otimes$ represents a logical *AND* that evaluates to different users. For example, $(Teller \otimes BranchManager)$ means two different users, one must be a *Teller* and the second must be a *BranchManager*. If one user is both a *Teller* and a *BranchManager*, that user by himself *does not* satisfy the requirement.

Example Terms:

- The term $(Manager \sqcup \neg\{Alice, Bob\})$ requires a user that is a *Manager*, but is neither *Alice* nor *Bob*; the sub-term $\neg\{Alice, Bob\}$ implements a blacklist.

- The term $((All \otimes All) \otimes All)$ requires three different users, no matter what roles they are assigned.

- The term $((SecuritiesClerk \sqcup Teller) \otimes BranchManager)$ requires a user who is either a *Teller* or a *SecuritiesClerk* and another user who is a *BranchManager*.

The notion of *configurations* is used to assign meanings to the roles used in the terms. User sets are then used to satisfy the terms. For example, consider the term $(Teller \odot BranchManager)$, and consider the set of users {*Alice, Bob*}, where *Alice* is assigned to both roles *Teller* and *BranchManager*, and Bob is assigned to the role *Teller*, then both sets of users {*Alice*}, and {*Alice, Bob*} satisfy the term.

Other examples:

- The term $(\{Alice, Bob, Carl\} \otimes \{Alice, Bob, Carl\})$ is satisfied by any two users from the set of the three users.

- The term $((Teller \odot SecuritiesClerk) \otimes BranchManager)$ is satisfied by a set of users consisting from a *Teller*, a *SecuritiesClerk*, and a *BranchManager*. If a single user is assigned to both *Teller* and *SecuritiesClerk* roles, then the term can be satisfied by that user and a another user assigned to the *BranchManager* role.

- The term $((Teller \sqcup SecuritiesClerk) \otimes (BranchManager \sqcap \neg OperationsDirector))$ is satisfied by a set of users consisting from two different users, one who is either a *Teller* or a *SecuritiesClerk*, and another who is a *BranchManager*, but not an *OperationsDirector*.

The authors of the algebra suggest a mechanism to enforce a policy (Task, $\varnothing$), a prerequisite is to identify the steps in performing the task by maintaining a history of each instance of the task, which includes information on who have performed which steps. For any task instance, one can compute the set of users who have performed at least one step of the task instance (denoted as U_{past}). Before a user u performs a step of the instance, the system checks to ensure that there exists a super-set of $U_{past} \cup \{u\}$ that can satisfy $\varnothing$ upon finishing all steps of the task. If u is about to perform the last step of the task instance, it is required by the policy that $U_{past} \cup \{u\}$ satisfies $\varnothing$.

Despite the clear advantage of the algebra, which helps formalizing SoD expressions, it suffers from a major shortcoming. The algebraic terms are defined on the level of the whole task. For example, given a task *T1*, which consists of different steps, and the term $\varnothing_1$, then $\varnothing_1$ is satisfied by a user set, no matter in which step the users are involved. Such kind of enforcement is left for the workflow designer to determine in which steps specific users get involved. This is a major security issue, as security controls must be separated from the application business logic. Therefore, not associating the workflow steps with the algebraic terms is a major issue. One other security requirement that is not an application business requirement, is applying different constraints depending on parameters of the task. The policies 22, 23, 24, 25 in our motivating example (Appendix A) cannot be expressed by the algebra. The transaction amount is considered key in determining the SoD policy that needs to be applied on the transaction workflow. Therefore, different algebraic expressions might be needed depending on parameters of the task.

2.8 Administrative Models of Role-Based Access Control

Authorization systems in large organizations usually contain tremendous numbers of roles, permissions, roles to permissions associations, role hierarchies, and users to roles assignments. Organizations usually face two main challenges when managing their access control systems. Firstly, the management of the access control elements (e.g., roles, permissions, and users). More so, managing relations between them is a very complex and time consuming task. Hence organizations are seeking an effective mechanism for managing them. It should mainly be easy to manage and less error prone. Secondly, given the tremendous number of elements and relations, security and safety become a major threat. Who can add roles to the system? Who can assign a role to a user and how? How revocation of roles from users is achieved? Can a user change his own privileges? What happens if we assign a new permission to a role? Will all role

members be assigned automatically to this permission? Do all the users really need this permission? And many other questions arise!

Administrative models of role-based access control address the issues of the administrative operations for creation and maintenance of RBAC elements and their interrelationships. These operations include creation and deletion of roles, creation and deletion of permissions, assignment of permissions to roles and their revocation, creation and deletion (or disabling) of users, assignment of users to roles and their revocation, definition and maintenance of the role hierarchy, definition and maintenance of constraints, and all of these in turn for administrative roles and permissions. Furthermore, the specification of administrative review functions for performing administrative queries, system functions for creating and managing RBAC attributes on user sessions, and making access control decisions. These functions should be flexible and extensible to meet expectations of organizations [9, 84].

Role-based administrative models in the literature can be classified into two main groups: centralized and decentralized administrative models. In centralized models, security administrators perform all administrative tasks. While in decentralized models, administrative tasks are distributed among different administrators in a controlled manner and adds a separate administrative role hierarchy in the original RBAC model [109].

In this section we review some role-based administration models.

2.8.1 ARBAC97

The ARBAC97 model [86] was the first comprehensive model for role-based administration of roles in RBAC which provides decentralized administrative capabilities to RBAC. ARBAC97 consists of three components which are: The user-role assignment component called URA97, the permission-role assignment component called PRA97 (permission-role assignment ’97) and the role-role assignment (RRA97) component, which has several components that are determined by the kind of roles that are involved. Although these three components are defined in ARBAC97, the URA97 and RR97 components are explained in two dedicate research papers. We explain the three components of ARBAC97 hereafter.

The URA97 Administrative Model

The URA97 (user-role assignment 97) model was defined by Sandhu and Bhamidipati [84]. URA97 is defined in the context of the RBAC96 model

[87]. URA97 focuses exclusively on the user to role assignment administrative activity as well as revoking users from roles. URA97 imposes strict limits on individual administrators regarding which users can be assigned to which roles and by whom. These restrictions are implemented by using prerequisite conditions. URA97 uses the *can-assign* relation to determine if an administrative user can assign a regular role to a given user. Similarly, the URA97 model controls user-role revocation by means of the *can-revoke* relation. Authorization to assign and revoke users to and from roles is controlled by administrative roles. URA97 applies only for regular roles. Assignment of users to administrative roles is centralized under the chief security officer who has complete control over all aspects of RBAC96.

The PRA97 Administrative Model

The PRA97 model deals with role-permission assignment and revocation. PRA97 is similar to URA97, both models use a prerequisite condition to determine if an administrative user is authorized to assign permissions to roles or revoke permissions from roles. The *can-assign* and *can-revoke* relations are used for this purpose. Revocation in PRA97 is weak, which means that permissions can still be inherited after revocation. It applies only to the role from which the permission was revoked. Strong revocation of permissions can be applied by cascading down the role hierarchy [86].

The RRA97 Administrative Model

The RRA97 model (Role-Role Assignment) [88] focuses on decentralized role-based administration of role hierarchies. The RRA97 model distinguishes three kinds of roles for role-role assignment, which are abilities, groups, and UP-roles. Abilities are roles that can only have permissions and other abilities as members. An ability is a collection of permissions that should be assigned as a single unit to a role. Groups are roles that can only have users and other groups as members. A group is a collection of users (e.g., a team) who are assigned as a single unit to a role. UP-Roles membership can include users, permissions, groups, abilities, and other UP-roles.

RRA97 uses the URA97 model to produce the GRA97 (group-role assignment) model. The group-role assignment and revocation are respectively authorized in GRA97 by the *can-assign* and the *can-revoke* relations. Similarly, the ability-role assignment and revocation are respectively authorized in ARA97 by the *can-assign* and the *can-revoke* relations.

The role-role creation, deletion, edge insertion, and edge deletion in the hierarchy are all authorized in UP-RRA97 (UP-Role-Role Assignment) by the *can-modify* relation.

2.8.2 ARBAC99

The ARBAC99 model for administration of roles [89] extends the ARBAC97 administration model with enhancements to the URA and PRA sub-models. ARBAC99 incorporates the concept of mobile and immobile memberships in roles. Immobile membership grants the user the authority to use the permissions of a role but does not make that user eligible for further role assignments. Mobile membership enables the user to use the permissions of a role and makes that user eligible for further role assignments.

In URA99, administrative users determine if a user can be assigned to a given role by the mobile and immobile memberships. URA99 uses the *can-assign* and *can-revoke* relations to authorize role assignment and revocation. The *can-assign* relationship is split into two relations: the *can-assign-M* and the *can-assign-IM*. The relation *can-assign-M(x, y, Z)* deals with mobile membership, and means that a member of administrative role x can assign a user whose current membership, or non-membership, in regular roles satisfies the prerequisite y to a regular role that belongs to the set of roles Z as a mobile member. The immobile relation *can-assign-IM(x, y, Z)* enables administrative users to assign users to roles as immobile members.

The PRA99 model extends PRA97 with mobile and immobile memberships of permissions in a similar way to URA99.

2.8.3 ARBAC02: Role Administration Using Organization Structure

The Role Administration Using Organization Structure (ARBAC02) [73] was proposed to overcome the weaknesses of ARBAC97. The ARBAC97 model has some significant shortcomings. Firstly, URA97 requires multi-step user assignments. Roles higher in the role hierarchy may require more assignment steps. This may require intervention of two or more security officers. Secondly, The URA97 model introduces redundant user-role assignment (UA) records as result of the multi-step user assignment. Thirdly, URA97 causes a more complicated role hierarchy, since prerequisite roles are part of the role hierarchy. Furthermore, the PRA97 model suffers the similar above-mentioned

shortcomings. In addition, PRA97 cannot restrict which permissions from a role can be assigned to another role [73].

ARBAC02 uses the organization structure as new user and permission pools instead of prerequisite roles in a role hierarchy. Furthermore, ARBAC02 uses organization structure for permission-role assignment. ARBAC02 adopts the same notation of the *can-assign* and the *can-revoke* relationships from ARBAC97. The difference is that, the prerequisite roles conditions are replaced by organization units. The organization unit is a group of people and functions (permissions) to achieve the given mission [73]. This simplifies the user assignment and avoids unnecessary user-role assignment records.

For permission-role assignment, the ARBAC02 model suggests that common permissions are assigned to lower roles in the role hierarchy and higher roles inherit common permissions, while special permissions are assigned to higher roles. This avoids duplicate assignments of the same permission through the inheritance line of the role hierarchy [73].

2.8.4 Role Hierarchy Administration

Crampton and Loizou [25, 29] introduced the concept of administrative scope which they use to develop a family of models for role hierarchy administration (RHA). Administration scope associates each role in the role hierarchy with a set of roles over which it has control. The administrative scope of a role is determined by the role hierarchy and changes dynamically as the hierarchy changes.

The RHA family of models consist of four sub-models. The RHA1 is the base model and is the simplest of them. It contains operations for managing role hierarchy, which are: *AddRole*, *DeleteRole*, *AddEdge*, and *DeleteEdge*. RHA1 can be added to RBAC without the need of any additional relations.

The RHA2 sub-model extends RHA1 by using administrative permissions. A role is permitted to perform hierarchy operations if the role has appropriate administrative permissions assigned to it. RHA2 can also be applied without introducing additional relations and offers finer granularity over RHA1.

The RHA3 sub-model introduces the *admin-authority* relation. If $(a, r) \in$ *admin-authority*, then a is called an administrative role that controls r. The *admin-authority* relation induces an extended hierarchy on the set of roles which includes the original hierarchy.

The RHA4 sub-model extends RHA3 by adding possibility to administer the *admin-authority* relation. RHA3 shows when and how the *admin-authority*

relation can be updated by hierarchy operations and by the actions of administrative roles.

2.9 Chapter Conclusion

This chapter outlined the current state of research in role-based access control models. The review of existing models has demonstrated both advantages and weaknesses of existing work.

The chapter started by introducing two access control paradigms that were widely dominant prior to RBAC, which are Discretionary Access Control (DAC) and Mandatory Access Control (MAC). The Attribute-Based Access Control model was also reviewed.

The chapter has also provided background study on role-based access control (RBAC), we reviewed the most relevant work in relation with our work, as the outcome of existing research on RBAC is massive.

The basic RBAC model has been presented then a series of enhancements on top of basic RBAC were also reviewed. We started by introducing RBAC96 which includes a family of four RBAC models that can be consolidated into a more comprehensive model. Other enhancements were also introduced such as temporal RBAC. More attention has been given to parameterized RBAC, since it is considered a major step towards more expressive RBAC.

The chapter then reviewed extensions to basic RBAC such as hierarchical RBAC models, role-delegation models, and authorization policies with focus on conflicts of interest policies.

Finally, the chapter presented an important aspect that is considered as part of any access control model, which is administrative models.

Chapter 3

Organizational Supervised Delegation Model (OSDM)

This chapter is published in the proceedings of the 15th international conference on Information Security (ISC'12),September, 2012. Passau, Germany

The dynamic nature of operations in organizations has led to an interest in roles and permissions delegation to enable a seamless continuity of business. Delegation involves assigning a given set of access rights from one user to another. In existing role delegation models, delegation is often authorized and controlled by a relation that specifies who can delegate to whom. The usage of such relations in delegation models has some disadvantages; such as complexity of maintenance, error proneness, inconsistencies, and inabilities to define some organizational policies related to delegation. In this chapter, we propose a new delegation model that depends on organizational lines of authority to authorize and control delegation. The main advantages of this approach are that it simplifies the management of delegation authorization and complies with organizational behavior. Furthermore, it eliminates inconsistencies related to changes to roles and permissions.

3.1 Introduction

Role-based access control (RBAC)[37] has become the dominant authorization mechanism used in a wide range of organizations. RBAC has gained wide acceptance since it greatly simplifies the management of access rights. Moreover, RBAC attempts to simulate organizational structures at a high level by its hierarchical model. In RBAC, roles are assigned to users and permissions are associated to roles. Users represent staff in organizations. Roles represent the job functions of the users or sub-functions in some cases. Permissions are privileges for accessing objects or performing activities.

High dependability of organizations on access control systems and the dynamic nature of operations have shown a demand for dynamism in the access control systems in place. RBAC supports the principle of least privilege [87], which entails that users are assigned the minimum privileges required for achieving their functions. However, this has led to situations where some specific privileges are assigned only to very few users. Despite the clear advantages of this approach, it restrains access to important resources to a small closed group. At times, none of those users may be available. This may hamper certain activities within the organization at stake. These situations have led to requirements of increased dynamism of the access control systems in place.

Role delegation is a mechanism of performing a takeover on a user's access rights. Delegation gives authority of a user on another user's access privileges to perform functions of the user originally assigned to the delegated access rights. Existing delegation models suggest that delegation can take two forms: administrative delegation and user delegation. In administrative delegation,

an administrative user assigns access rights to a user while the administrative user is not necessarily assigned to the delegated role. In user delegation, a user assigns a subset of his available rights to another user [28].

Most of the existing delegation models [12][27][106][108][42] use a relation to authorize delegation, that is used to determine which user can delegate to whom. The delegation relation often takes the form *can-delegate(R, some conditions)*, where R is the role to be delegated, the conditions specify who can delegate the role, in addition to other parameters such as depth of delegation. The conditions often take the form: a user who has role x can delegate role y. The delegation relation (*can-delegate*) brings some disadvantages to the delegation model:

1. The delegation relations in existing delegation models add complexity to the access control model. Large organizations typically have a rather large number of roles. By adding delegation relations for most of these roles, the entire system may explode. This is blocking for large organizations since they require huge efforts for defining a huge number of relations.

2. The delegation relations cannot express precise conditions on who can delegate a specific role. As an example, consider a delegation relation that states that a professor can delegate the teaching assistant role. This means that a professor in the faculty of arts can delegate the teaching assistant role to a user in the computer science department.

3. The delegation relations may become inconsistent if updates to RBAC relations are allowed such as updates to the role hierarchy [28]. Such updates may occur when new activities are deployed in the organization. This adds huge efforts for the maintenance of the relations, specifically in cases of updates to roles such as adding or removing roles, as well as updates to the hierarchies of roles. Such updates are likely to happen in organizations.

4. User delegation models suggest that a user who is delegating an access right has to be assigned to it [28]. This is not necessarily valid, since it is possible that the user possessing the access right to be delegated is absent. Furthermore, it is not guaranteed that another user who possesses the same access right is available, especially in case of emergency.

In this chapter we propose a novel form of delegation, called the organizational supervised delegation model (OSDM). The idea came after surveying some access control models and policies in some organizations. Our surveys included one of the largest European banks, a European university and a software

provider. We have discussed the applicability of the delegation authorization relations provided in existing delegation models [12][27][106][108][42]. All the surveyed organizations commented negatively on these relations, and mainly regarding complexity and the huge number of relations to be defined, which makes managing them extremely hard. Furthermore, organizations often adopt a different approach for authorizing delegation. We have found that any user-role assignment or role (or permissions) delegation must be approved by the line managers of the users. When roles are delegated, the delegation request is initiated by a user, then approved by another user then executed by a user or a process. No user can delegate a role assigned to himself without approval. The user who approves delegation is not necessarily assigned to the role or to the permissions to be delegated. OSDM depends on organizational hierarchies to find users who must approve delegation, according to lines of authority defined in the organizational structure. OSDM addresses the above-mentioned limitations of existing role delegation models.

The remainder of this chapter is organized as follows: In the second section we show an overview of organizational structures. In the third section we review existing role delegation models. The fourth section presents our proposed delegation model. Section five is a discussion on OSDM. Finally, section six concludes our work.

3.2 Overview of Organizational Structures

Organizational structures outline the planned pattern of positions of individuals, job duties and activities to be achieved. They also describe the lines of authority among different parts of the organization [91]. Organizational structures are modeled using organizational charts that depict the relationships between different positions and the hierarchy that represents authority depending on the rank of users. Organizational structures can take several forms. Most of structures used are hierarchical, matrix, and flat organizational structures. In a flat structure, all employees report directly to a single manager. In hierarchical structures, each individual reports to one and only one manager at the next higher level [40]. Authority is clear in hierarchical organizations, and managers have absolute authority on their teams.

The matrix structure involves dual authorities, where individuals can report to two managers. Employees often have a functional manager and participate in projects that have a project manager. Users report some activities to the project manager and some activities to their functional manager. Authority of functional managers and project managers in the matrix structure varies

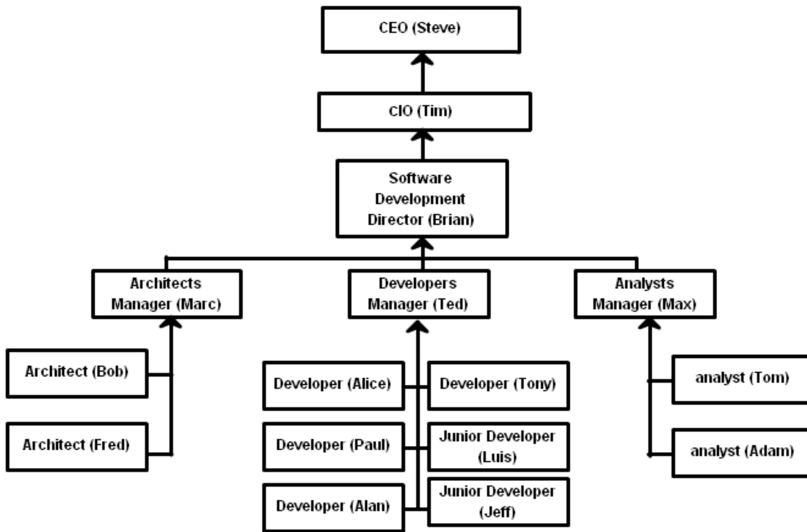


Figure 3.1: Software development department users hierarchy of the organizational structure.

according to the type of the matrix. Resource assignment is normally controlled by the functional manager. Therefore, we are interested in functional managers for our delegation model, since we care about who has the power to approve resource assignment and therefore delegation.

We have developed an example user hierarchy to be used for explanation throughout the rest of the paper. Fig. 3.1 depicts the users hierarchy of a software development department in a technology department of a hierarchical organizational structure.

3.3 Related Work

In the last few years, several role delegation models [12][27][106][108][42] studying delegation in the context of role based access control have emerged. However, delegation was studied before RBAC was proposed, and there were some predecessors of RBAC delegation such as: the access matrix models which introduced the concept of copy flag, which allows users to delegate rights [41]. Wood and Fernandez [103] introduced the idea of reverting the rights to the upper level after revoking a low-level delegation. Graph-based delegation was

introduced in [39]. A variety of delegation approaches were also introduced in [62]. In this paper, we focus on delegation in the context of RBAC. Delegation in RBAC can have several characteristics depending on the requirements of the environment where delegation is applied. The main characteristics of delegation were explained by Barka et al. [11]. These key characteristics include permanence, monotonicity, totality, administration, levels of delegation, multiple delegation, lateral agreements, cascading revocation, and grant-dependency revocation [11].

The first work that studied delegation in RBAC was achieved by Barka and Sandhu [12][11]. They proposed the RBDM0 delegation model [12] which studied delegation in flat roles structure. RBDM0 focused on *grant total delegation* which means that the delegator keeps the power to use the role after delegation and covers only the delegation of roles. RBDM0 does not support partial role permissions delegations. RBDM0 controls user-user delegation by means of the *can-delegate* relation. The *can-delegate* relation takes the form of *(a, b)* ∈ *can-delegate*. It means that a user who is an original member of the role *a* can delegate his role to a user who is an original member of role *b*. Revocation in RBDM0 can happen in two ways: firstly, by time outs. Delegations are revoked when the delegation period expires. Secondly, any original member of the delegated role can revoke the membership of any delegate member in that role.

RDM2000 [106][107] was the first delegation model to address delegation with hierarchical roles. It also supports multi-step delegation. The *can-delegate* relation in RDM2000 takes the form: *can-delegate* $\subseteq R \times CR \times N$, where R are sets of roles, CR are prerequisite conditions, and N is the maximum delegation depth. The meaning of $(r, cr, n) \in$ *can-delegate* is that a user who is a member of role r (or a role senior to r) can delegate role r (or a role junior to r) to any user whose current entitlements in roles satisfy the prerequisite condition cr without exceeding the maximum delegation depth n.

The permission-based delegation model (PBDM) [108] was the first to address permission delegation (partial delegation). PBDM supports role as well as permission delegations with features of multi-step delegation and multi-option revocation. PBDM comprises in two models: PBDM0 and PBDM1. In PBDM0, permission delegation involves three steps. Firstly, a temporary delegation role is created by the delegator. Secondly, the permissions to be delegated are assigned to the temporary role with permission-role assignment. Thirdly, the delegator assigns the temporary role to the delegatee by user-role assignment. Revocation in PBDM0 includes three cases: by revoking the delegated role, by removing one or more pieces of permissions from the delegated role, or by revoking the user-delegation role assignment.

PBDM1 extends PBDM0 with two main features. Firstly, it adds support for role-role delegation, which supports delegating specific permissions of a role to another role rather than to another user. Secondly, it adds means for controlling delegation; to restrict delegation only to authorized users. This is achieved by the *can-delegate* relation, which takes the form: $\textit{can-delegate} \subseteq \textit{DBR} \times \textit{Pre-con} \times \textit{P-Range} \times M$, where *DBR* are sets of delegable roles, *Pre-con* are prerequisite conditions, *P-Range* is the delegation range that specifies which permissions can be delegated, and M is the maximum delegation depth.

Crampton et al. [28] proposed a new model for dealing with transfer delegation. In transfer delegation, the delegator loses the power of using the access right after delegation is completed. They also have proposed two relations for controlling delegation. The *can-delegate* and the *can-receive* relations. The advantages of using different relations for controlling delegations include flexibility, greater control, ease of management and is less error prone. They also included constraints on the *can-delegate* and *can-receive* relations to ensure that the relations do not give the authority to a delegator to delegate a right that is not available to him.

The capability based delegation model [42] is an interesting work based on the capability based access control model (CRBAC) presented in the same paper. The CRBAC model integrates a capability-based access control mechanism into the RBAC96 model. Roles and permissions are assigned to capabilities, and capabilities are assigned to users. Delegation is achieved by creating a new capability, then by assigning delegable authority (roles or permissions) to the capability, then the delegator sends the capability to the delegatee. Unlike the other delegation models, delegation is authorized by a permission that the delegator must possess for creating the capability.

We discuss the drawbacks of the reviewed models and how OSDM addresses them in section 5.

3.4 The Organizational Supervised Delegation Model

We have shown in the previous sections that existing delegation models depend on the delegation relation for determining who can delegate a given role or a set of permissions. We have also shown the drawbacks of using the delegation relation for authorizing delegation. Our surveys of different organizations have revealed that no user is directly allowed to delegate his role to another user without the approval of the direct line managers of both users. Another problem with existing role delegation models is that they require that the user performing the delegation to possess the role to be delegated. This conflicts with one of the

major reasons for using delegation, which is when the user is absent without early notification due to an emergency situation, and hence, blocks urgent activities usually performed by that user.

In this section, we provide our delegation model that aims at addressing issues of existing delegation models. We start by extending RBAC to enable the user hierarchy feature, that is used to find the users who must approve the delegation. Then we provide a detailed description of the delegation model followed by the formal model of the organizational supervised delegation model (OSDM). The last subsection explains how revocation is performed in OSDM.

3.4.1 Extensions to RBAC

In RBAC, roles are assigned to users and permissions are assigned to roles. Roles represent job functions or sub-functions in organizations, while permissions are privileges to access objects or execute operations. The main advantage of RBAC is that it simplifies the management of access rights, since users can be reassigned from one role to another. New permissions can be assigned to roles as new applications and systems are incorporated, and permissions can be revoked from roles as needed [87].

RBAC has paid attention to simulating organizational structures after the concept was originally proposed for user group structures in [36]. RBAC implements role hierarchies that are a natural means of structuring roles in organizations [9]. Role hierarchies are partial orders that express inheritance relations among roles. Although role hierarchies in RBAC can reflect some points from organizational structures, the functional role hierarchy constructed through the existing role engineering approaches does not reflect organizational structures, because they do not take into account the structural characteristics of organizations [52]. Hierarchies in RBAC implement the *is-a* relationships between hierarchical roles. In most organizations, superiors do not need full access to the permissions of their inferiors. It is not necessary that a manager inherits from his inferiors since managers are often performing completely different tasks from their subordinates. In consequence, the application of the *is-a* inheritance in these situations results in the assignment of undesired and unnecessary privileges to superiors. This conflicts with the least privilege concept. In many situations, senior users have supervision relationships with junior users. Organizations are seeking flexibility when defining hierarchies in the access control model that can reflect these nuances [70]. More so, in organizational hierarchies, lines in the hierarchy mean different levels in the structure. While in RBAC it is possible that hierarchical roles are assigned to users in the same level in the hierarchy. For example, in the users structure

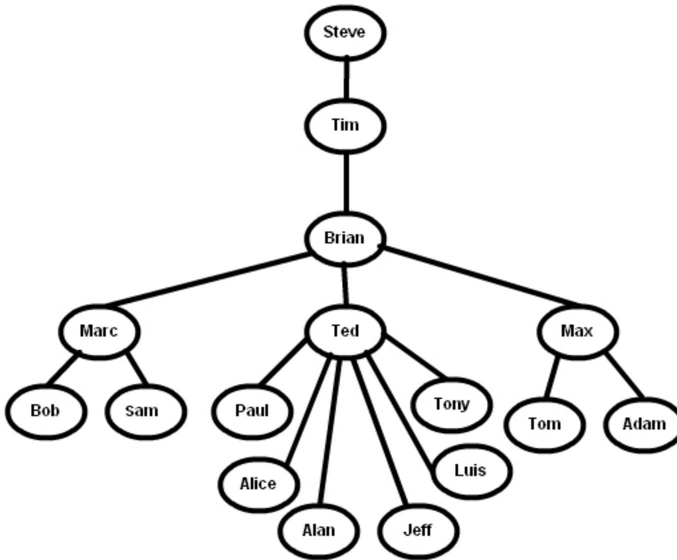


Figure 3.2: Modeling the hierarchy of users of an organization structure by a general tree data structure

depicted in Fig. 3.1, the senior developer role might inherit from the role of the junior developer, although both are in the same level in the hierarchy.

From the description above, it is clear that the role hierarchies in RBAC cannot reflect organizational structures, and therefore, we cannot take the role hierarchies as basis for modeling the organizational hierarchies. Our delegation model depends on the organizational structure in order to find the user who must approve the delegation request. It is impossible to know the manager of a given user in RBAC, since the organizational structure is not available or is misleading in RBAC. Therefore, we propose an extension to RBAC to define hierarchies over users. The hierarchy of users must respect the user hierarchies in organizational structures. In our proposed extension, users hierarchies are represented as a general tree, where each user has one parent node corresponding to his direct responsible (line manager). As an example, the structure depicted in Fig. 3.1 can be represented by the tree model shown in Fig. C.2.

Modeling user hierarchies in RBAC with general trees makes it easier for finding the responsible (line manager) of any user, who is in charge of approving the delegation request. The direct line manager is the parent node of the user node, while the next level manager is the parent node of the direct manager

node. This complies with hierarchies in organizational structures. Even in the matrix structure, the functional managers are responsible for approving task assignments to their employees. Therefore, the parent nodes in matrix structures represent functional managers.

The proposed model incurs some amendments on the administrative RBAC model [9]. The *AddUser* command now has an extra parameter called *Parent* that corresponds to the user's direct responsible. If the node to be added is a parent node (responsible), then the children nodes can be attached to this parent node as sub-trees. The *DeleteUser* is also modified. If the user node to be deleted has no children, it means that the user is not responsible for other users, then it can be deleted outright. If the user to be deleted is a parent of some other nodes, which means that he is a responsible of some other users. Then we promote the children users and attach them to the parent of the deleted node. The root node can only be replaced and cannot be deleted. Other administrative commands remain unchanged.

The extended RBAC model is composed from the following elements and relations:

- U, R, P: are sets of users, sets of roles and sets of permissions, respectively.
- $PA \subseteq P \times R$, a many-to-many permissions to roles association.
- $RA \subseteq U \times R$, a many-to-many users to roles assignments.
- $RH \subseteq R \times R$, a partial order on R. Also represented by $\geq$. If $r, r' \in R$, then $r \geq r'$ means r is higher in hierarchy than r' and that r inherits all permissions of r'.
- $UH \subseteq U \times U$, a partial order on U. Also represented by $>$. If $u, u' \in U$, then $u > u'$ means u is higher in hierarchy than u' and that u has authority on u'.

3.4.2 Delegation in OSDM

The central notion of OSDM is that delegation must be approved by the line managers of the delegator and the delegatee. OSDM supports both role delegation and permission delegation. More so, OSDM supports delegation in both flat and hierarchical roles. In general, the delegation process in OSDM is accomplished in three main steps: firstly, a delegation request is initiated, secondly, the delegation request is sent for approval, and finally, the delegated access rights are assigned to the delegatee.

There are three types of situations in which delegation takes place. Firstly, backup of roles. When the user is absent, the function needs to be achieved by others. Secondly, centralization of authority. When an organization needs to reorganize functions and distribute functions from higher job positions to lower job positions in the organizational structure. Thirdly, collaboration of work. Users need to collaborate with others to achieve specific tasks [108]. However, user delegation models discussed in the literature requires that the user performing the delegation must possess the ability to use the access right [28]. This is not valid in all cases, since in the backup of role case, the user is absent and cannot initiate the delegation. Therefore, we enable different users to initiate the request. Specifically, the delegation request can be initiated by three different users:

- By the delegator (the user that delegates the role). The delegator asks to delegate his role or permissions to another user.
- By the delegatee (the user that is to be delegated the access right). The delegatee asks to acquire an access right from another user.
- By the line manager of the delegator, either by the direct line manager or by a higher level line manager.

Consider the hierarchy depicted in Fig. 3.1, if *Alice* needs to delegate one of her access rights to *Bob*, then the delegation request can be initiated either by *Alice* herself, by *Bob*, or by one of *Alice's* line managers, usually *Ted*. In case of absence of *Ted*, the request can be initiated by *Brian*, *Tim* or *Steve*.

Once the delegation request is initiated, the delegation request becomes pending for approval. The request must be approved by the line manager of the delegator and the line manager of the delegatee. The reason for this is that the line manager of the delegator is responsible for the achievement of his tasks. The approval of the line manager of the delegatee is required because he is responsible for assigning the resources to the tasks. The line managers entitled for approval receive a request to approve the delegation. This means that the delegation authorization is performed by the parent nodes of the delegator and the delegatee. Once both of them have sent their approvals, the delegation operation step can be accomplished; in which the delegated access rights are assigned to the delegatee. Usually, the delegation request is sent to the first line managers of the delegator and the delegatee, but in case of their absence, the request can be redirected on demand to the higher-level line managers of the delegator or the delegatee. In case that the delegator and the delegatee are supervised directly by the same first line manager, then only one delegation request is sent to the line manager. If a user's access right is to be delegated to his direct

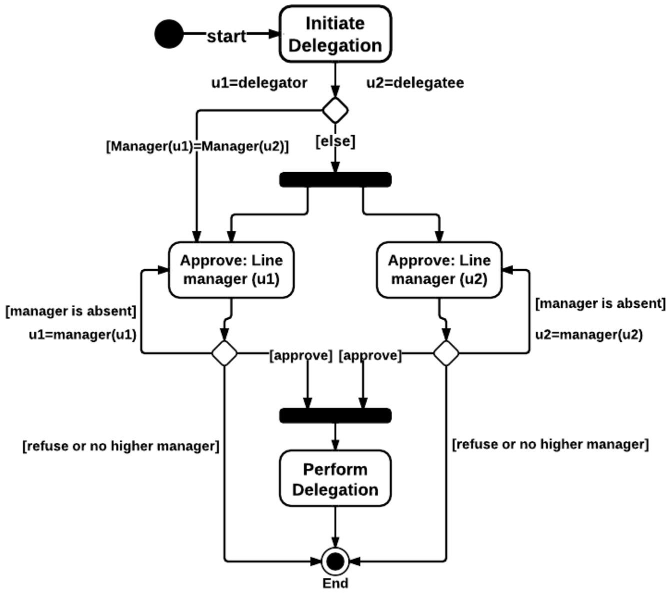


Figure 3.3: The activity diagram of the delegation process

line manager, then one approval request is sent to the line manager of the delegatee. The steps of the delegation process in OSDM are depicted in the activity diagram shown in Fig. 3.3.

To explain the approval step of delegation, we continue our example. When the delegation request is initiated for delegating some access rights from *Alice* to *Bob*, the request for approval is sent to both *Ted* and *Marc*. In case of absence of either *Ted* or *Marc* the delegation request can be sent to *Brian*, *Tim* or *Steve*. If *Ted* asks to acquire an access right from *Alice*, then only one request is sent to *Brian* for approval. If *Alice* needs to delegate some access rights to *Tony*, then only one approval request is sent to *Ted*.

Once the delegation request is approved, then the third step of delegation can be executed. The delegation operation step depends on the characteristics of delegation to be implemented. The identified characteristics of delegation comprise totality, permanence, monotonicity, administration, levels of delegation, multiple delegation, lateral agreements, cascading revocation, and grant-dependency revocation [11]. The delegation operation using these characteristics has been described by several papers [12][27][106][108][11]. In OSDM, we focus on total and partial grant delegation with flat and hierarchical roles.

We adapted the approach discussed in [12][106][108] for the delegation operation. We start with delegation operations on flat roles then we move to hierarchical roles.

Delegation in flat roles:

1. Grant total delegation: In this case, the delegator delegates a role with all its permissions to the delegatee. The delegatee must not be a member in that role before delegation. The delegated role is assigned to the delegatee with delegation relation instead of the original role assignment relation. This is important to identify delegated roles from roles that were originally assigned to users by the system administrators. The delegatee can start using the role after this step, and the delegator retains the power to use the delegated role.

2. Grant partial delegation: In this case, the delegator only grants a subset of permissions of a given role to the delgatee. A temporary role is created and is assigned to the permissions to be delegated. The temporary role is then assigned to the delegatee with delegation. The delegatee can start using the delegated permissions after this step, and the delegator retains the power to use the delegated permissions.

Delegation in hierarchical roles:

1. Grant total delegation: In this case, the delegator delegates a role with all its permissions to the delegatee. The delegatee then has the power to use the role plus all the roles in the hierarchy junior to the delegated role. The delegator is explicitly assigned to the roles junior to the delegated role. The delegated role is assigned to the delegatee with delegation and the delegator retains the power to use the delegated role.

2. Grant partial delegation: This case is exactly the same as the grant partial delegation in flat roles structure.

3.4.3 A UML/OCL Formal Model of OSDM

In this subsection, we include the formal model of OSDM for completeness. We use the Unified Modeling Language (UML) [76] and the Object Constraint Language (OCL) [74] to formalize the definitions of OSDM. Fig. 3.4. shows the UML class diagram of OSDM. The diagram projects the relationships between the different classes of the extended RBAC model according to the definitions in

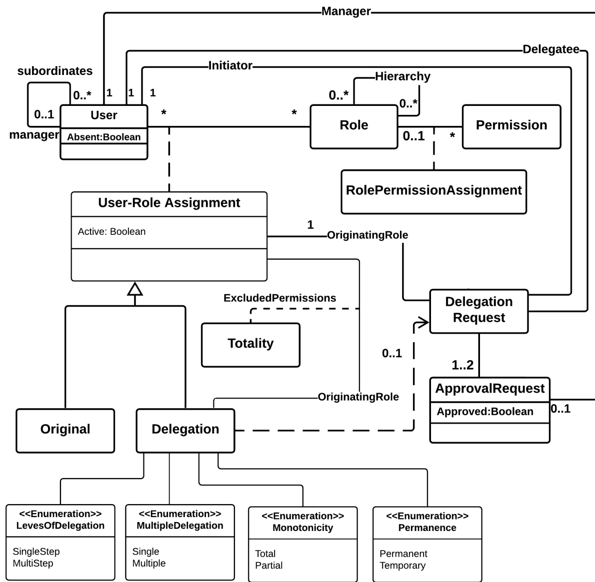


Figure 3.4: The class diagram of OSDM

section 4.1. Furthermore, the diagram depicts the classes required for delegation. The *DelegationRequest* class represents the initiated delegation request. The *ApprovalRequest* represents the requests for approving the delegation request and then assigning the delegated role to the user by delegation. The diagram also depicts the two types of role assignments; the original and delegation assignments.

The following definitions formalize OSDM constraints in OCL:

Definition 1:
In OSDM, the line manager of a user is the user higher in the hierarchy (parent node). In case of absence of the direct line manager, then the line manager is the next level manager.

```
context::User:getLineManager()
post: if self.Manager.Absent = false
then result = self.Manager
else result = self.Manager.getLineManager()
endif
```

Definition 2:
Self-delegation is not allowed in OSDM; the delegator and delegatee cannot be the same user.

```
context::Delegation
inv: no_self_delegation: self.User <> OriginatingRole.User
```

Definition 3:
The role assigned to the user by the delegation relation must be exactly the role in the delegation request; which is the delegated role.

```
context::Delegation
inv: same_role: self.Role = OriginatingRole.Role
```

Definition 4:
The delegation request is initialized by the delegator, the delegatee or the line manager of the delegator.

```
context::DelegationRequest
inv: InitiatorRule: self.Initiator = self.Delegatee or
self.Initiator = OriginatingRole.User.getLineManager()
or self.Initiator = OriginatingRole.user
```

Definition 5:
The delegation request can be initialized only if the delegatee is not assigned to the delegated role.

```
context::DelegationRequest
inv: Delegator_not_member_in_delegated_role:
self.delegatee.role ->asSet() ->excludes(OriginatingRole.role)
```

Definition 6:
Proper delegation requests must have been forwarded to a manager of both users involved in a delegation request.

```
context::DelegationRequest
inv: requests_initiated:
ApprovalRequest.Manager ->
forSome(manages(self.Delegatee)) and
ApprovalRequest.Manager ->
forSome(manages(self.Delegation.User))
```

Definition 7:
The role must be assigned to the user once the delegation Request is approved.

```
context::DelegationRequest
inv: delegate_role: if ApprovalRequest.Approved = true
then delegation ->allInstances ->
includes(delegation|delegation.role = self.OriginatingRole
and delegation.user = self.delegatee)
endif
```

If multi-step delegation is not allowed, then the link between *DelegationRequest* and *UserRoleAssignment* in Fig. 3.4 must be changed to be between *DelegationRequest* and *Original* class. The link between *UserRoleAssignment* and *DelegationRequest* indicates that the delegation request can only be initiated if the role to be delegated is assigned to the delegator.

3.4.4 Revocation in OSDM

Revocation is the step that ends delegation and deassigns the previously delegated access rights from the delegatee. Revocation can be performed when the reason for delegation becomes invalid. For example, if the delegation was performed because of user absence and the user returns back to duty. Revocation in OSDM is achieved as follows:

- In case of grant total delegation, the revocation is simply achieved by deassigning the delegated role from the delegatee.
- In case of grant partial delegation, revocation is achieved either by deassigning the delegated temporary role, or by revoking one or more permissions from the temporary role.

In both cases, revocation must be initiated by the delegator, the delegatee or the line manager of the delegator. The revocation must also be approved by the line manager of the delegator. The approval of the line manager of the delegatee is not necessary for revocation, since the line manager of the delegator is responsible for the accomplishment of his tasks, whereas the line manager of the delegatee is involved in cases where tasks are to be assigned to his employees.

3.5 Discussion

Controlling delegation is the mechanism in delegation models that determines the security of the delegation model. Although the proposed delegation control

relations in existing role delegation models provide a means for authorizing delegation, they still suffer from disadvantages that will turn organizations away from using them. Defining such relations introduces complexities and is prone to error. If the definition of a relation is missing then it prevents delegating a role, while an erroneous relation enables delegation to none entitled entities. More so, relations complicate updates to RBAC, organizations need to revise the relations after each update. Even if organizations can tolerate this great overhead caused by relations, they will not be able to express some constraints on delegation such as specifying that manager m should approve delegation of role r of user u. This is due to the fact that conditions in delegation relations depend on roles. This could create inconsistencies, given it is possible that two managers having the same role becomes able to delegate a specific role. Other models such as CRBAC [42] uses a permission to authorize delegation. This eliminates complications introduced by the *can-delegate* relation, but it opens security breaches since any user that has a permission to create a capability can delegate his roles and permissions. Furthermore, role hierarchies in RBAC do not reflect organizations structures as shown in [52][70][64]. This has led to the idea of OSDM, which models organizational structures in RBAC and then utilizes them in controlling and authorizing delegation.

The advantages of OSDM other than addressing the above limitations of existing delegations, are that it complies with organizational policies towards delegation authorization. OSDM also provides a means for reflecting the organizational hierarchies and lines of authority in RBAC. In organizational structures, the functional line managers of users are responsible for resource allocation. Which means that roles are assigned to users based on the agreement of their line managers. This is also valid for delegation.

The implementation of the model is straightforward once the organizational chart is available to be projected and maintained in the access control system of the organization.

3.6 Conclusion

The central contribution of this chapter is a new roles and permissions delegation model for role-based access control, the organizational supervised role delegation model (OSDM). This model provides a new means for controlling and authorizing delegation based on the organizational hierarchy. The development of the OSDM model was motivated by surveying some organizations and verifying their delegation and role assignment mechanisms in place. The survey has

concluded that such actions are usually approved by managers according to lines of authority in the organization.

The model starts by extending RBAC to adopt changes required for implementing organizational hierarchies. Mainly by adding support for user hierarchies. This enables implementing authority relations among different users by modeling the hierarchy using a general tree data structure. The user hierarchy helps in finding users who need to approve delegations and revocations according the lines of authority in the organization. In existing delegation models, delegation is authorized by using a delegation relation that defines who can delegate a given role. We have explained disadvantages of this approach that could prevent organizations from using delegation models based on such relations.

The delegation request can be initiated by three different parties, the delegator, the delegatee, or the line manager of the delegator. Once the request is initiated, the delegation request is sent for approval to the line managers of the delegator and the delegatee. The delegation operation is performed when both approvals are received. OSDM supports both role and permission delegations, as well as flat and hierarchical role structures. Revocation in OSDM takes similar steps to delegation. Firstly, a request for revocation is to be initiated by the delegator, the delegatee or the line manager of the delegator. Afterwards, the revocation request needs to be approved by the line manager of the delegator, before the revocation operation is performed.

The future work on OSDM will focus on defining API classes for the model, and validation on a case study or by implementing our approach at one of the surveyed organizations. More so, we will be looking at extending the model to support parametrized roles delegation. We will also be looking at more efficient modeling of user hierarchies that can implement special lines of authorities and constraints.

Chapter 4

ROAC: A Role-Oriented Access Control Model

This chapter is published in the proceedings of the 6th International Workshop on Information Security Theory and Practice (WISTP), Jun 2012, Egham, United Kingdom.

Role-Based Access Control (RBAC) has become the de facto standard for realizing authorization requirements in a wide range of organizations. Existing RBAC models suffer from two main shortcomings: lack of expressiveness of roles/permissions and ambiguities of their hierarchies. Roles/permissions expressiveness is limited since roles do not have the ability to express behavior and state, while hierarchical RBAC cannot reflect real organizational hierarchies. In this chapter, we propose a novel access control model: The Role-Oriented Access Control Model (ROAC), which is based on the concepts of RBAC but inspired by the object-oriented paradigm. ROAC greatly enhances expressiveness of roles and permissions by introducing parameters and methods as members. The hierarchical ROAC model supports selective inheritance of permissions.

4.1 Introduction

The deployment of software applications on distributed networks and on the web has exposed them to many new security threats. One major risk is that an application can be accessed by unauthorized users in an easier way than in the past. Governments and commercial organizations are continuously seeking strong access control models that can help them prevent unauthorized access to their systems. Therefore, they maintain their reputation as safe institutions where confidential information is safeguarded. For example, WikiLeaks could have been prevented should better access controls have been in place [35]. Role based Access Control (RBAC) [37] has been used by organizations to protect resources in their software systems against unauthorized access. RBAC has become the dominant access control model that is widely accepted in enterprise, health, and governments systems.

RBAC is based on four principles: abstract privileges, separation of administrative functions, least privilege and separation of duties [87]. RBAC is expressed in terms of users, roles, permissions, objects, and operations [9]. Permissions are assigned to roles and roles are assigned to users. Permissions are privileges to access objects or to execute operations. RBAC models often support role hierarchies. This feature is known as *hierarchical* RBAC. Role hierarchies define partial orders on roles; this is analogous to inheritance in the object-oriented paradigm. The central advantage of RBAC is that it simplifies the management of access rights and offers a high level view on security in organizations by bridging the gap between functional requirements of organizations and the technical authorization aspects of their security policies [87], [7].

Despite the robustness of RBAC, it has received a great academic attention

from researchers. The literature shows many notable contributions that address limitations and suggest improvements to RBAC. However, in its current form, RBAC does not seem to have enough power to express a wide range of security requirements and capture fine access control granularity when put into practice [7]. Two main shortcomings of standard RBAC are its lack of expressiveness when defining roles [7], [38] and ambiguities that may arise in hierarchical role models [50]. Hierarchies in standard RBAC only support the *is-a* hierarchy which does not reflect real organizational hierarchies as we will see later. On the other hand, parametrized RBAC [7], [38],[47] has been proposed to address the lack of expressiveness by associating parameters to roles. Shortcomings related to role hierarchies were addressed by many initiatives. More discussions regarding this are contained in the next section.

Existing RBAC models consider roles as entities of a simple type that cannot have member attributes and operations, except parameters as suggested by parametrized RBAC. This provides a relatively simple and straightforward type for roles, but it lacks flexibility when defining roles. Roles in RBAC are blind in the sense that they are not aware of the application environment. They cannot access data in the system or perform any actions. Roles cannot hold variables, status, methods, etc. More so, the generalization concept in existing RBAC models does not reflect real organizational hierarchies. In most organizations, superiors do not need full access on permissions of their subordinates, hence application of the *is-a* inheritance in these situations results in assignment of undesired privileges to superiors. This conflicts with the least privilege concept of RBAC. In many situations, senior users have supervision relations to junior users. Organizations are seeking flexibility when defining hierarchies in the access control model that can reflect the nuances above.

In this chapter, we propose the Role Oriented Access Control model (ROAC) as a novel access control model. ROAC addresses limitations of existing RBAC models through benefiting from the object-oriented concepts. ROAC makes analogies between roles and classes in object-oriented programming languages, then utilizes their concepts for constructing a new robust and extendible access control model. The main contributions of ROAC are:

1. To the best of our knowledge, ROAC is the most expressive access control model yet defined. ROAC greatly enhances the expressiveness of access control through associating variables and methods to permissions and roles. This architecture provides a means to defining one role and then defining multiple instances from the role with different levels of granularity. More so, application code is able to invoke methods to validate role parameters that are defined as part of role permissions. This helps separating the access control management from the application logic. This all results

in stronger security and minimizes the risk of different interpretations of parameters among developers of the application.

2. ROAC greatly enhances RBAC hierarchies by adopting standard object-oriented inheritance concepts. At the same time, it extends hierarchical facilities with supervision relationships among roles. It also offers means for selective inheritance of permissions of junior roles by senior roles. In this way, ROAC better reflects organizational hierarchies. In other words, ROAC supports both the *is-a* and selective inheritance.

3. ROAC addresses scalability issues of existing RBAC models. In addition to the points mentioned above, ROAC provides a new kind of parameters, referred to as static parameters. Static parameters have common values for all instances of the role. Static parameters help updating all instances of the role at once. For example, if an organization often switches between two roles, it must be able to disable one type of role and enable the other type. A static parameter can then be introduced to specify whether all instances of the role are enabled or not. Moreover, by using validators, organizations can also provide assertions over the parameters and static parameters. Validators can also implement authorization policies that can be checked before authorizing operations. In ROAC, roles and permissions can hold state. Roles can connect to databases and can have data structures to hold data. This can be of great usage for auditing and tracking of authorizations.

The remainder of this chapter is organized as follows: In the next section we review related work, then in the third section we overview the ROAC model. In section four, we provide the data model of ROAC. In the fifth section we explain the generalization model of ROAC. Section six provides a discussion about how ROAC can implement next generation RBAC concepts and the trade-off between complexity and fine granularity. Finally, section seven concludes our work and highlights future tracks.

4.2 Background and Motivation

RBAC has received a lot of attention from academic researchers and from commercial organizations. This has led to many improvements to the standard RBAC model [9]. RBAC research can be broadly classified into two main categories: improvements to features existing in standard RBAC and extensions to standard RBAC. Improvements to standard RBAC have been mainly focusing on improving role hierarchies of the standard RBAC model and on improving

expressiveness of roles by parametrization. Extensions to standard RBAC have been focusing on adding new features to RBAC such as supporting cross domain roles, role delegation models, etc. In this paper, we focus on improvements to RBAC.

In standard RBAC, role hierarchies support multiple inheritance; meaning that a role can inherit permissions from multiple roles. The general role hierarchy concept in standard RBAC has two main properties; firstly, the possibility to derive roles from multiple roles, and secondly, the role hierarchies concept provides a uniform treatment of user/role and role/role relations. Users can be included in the role hierarchy, using the same relation to denote the user assignment to roles. More so, standard RBAC supports the limited role hierarchy concept, in which hierarchies are limited to the single immediate descendent [9]. The role hierarchy concept in standard RBAC suggests that when a senior role inherits from a junior role, all permissions of the junior role are transferred to the senior role.

The most familiar form of collaborative working is hierarchical in nature. In organizational hierarchies, the superior may not take part in the details of a task, but rather, acts as the instigator of the task [10]. In other words, the most typical form of hierarchy in organizations is the supervision hierarchy [64]. More so, in some situations it is required to keep a role private and inhibit others from extending it. Sandhu [87], [82] has introduced the concept of the private role, which is a role that cannot be further extended. In situations where users have private documents that they need to protect from their superiors, a new private role has to be introduced for each user. This results in an increased number of roles in the system. This counter-balances the advantage gained by using hierarchies, which is reducing number of roles in the system [64]. Xuexiong et al [104] proposed an approach to tackle excessive inheritance that occurs when users get more permissions than they should have by permission inheritance. They resolve the issue by segregating role permissions into private permissions and public permissions. Then only public permissions are transferred through inheritance to superiors. If a role r has a set of permissions P, then P is divided into two sets P_{prv} for private roles, and P_{pub} for public roles. When a senior role r_s inherits from r, only P_{pub} are transferred to r_s. The drawbacks of this approach are that the private permissions of a role won't be inherited by any other role. In organizations, it might be the case that private permissions are different between two superiors of a junior role. In this situation, it won't be possible to define the inheritance for the two roles.

Lack of expressiveness in role definition has received attention from researchers as well. In many organizations, different users may require different granularity levels of the same role. For example, two tellers in a bank might have the same role that enables them to perform transactions. But the maximum amount of the

transactions both can perform might be different depending on their seniority. Standard RBAC can be adapted to capture such fine-grained authorizations by dramatically increasing the number of distinct roles. Parameterized roles [7], [38], [47] were proposed to address the lack of expressiveness of roles. One of the good attempts to address lack of expressiveness of RBAC by using parameterized roles was defined by Jaeger et al. [47]. The formal definition of parameterized RBAC was introduced by Abdallah et al. [7]. Parameterized RBAC provides finer granularity by creating instances of RBAC components according to the contexts of their use [7]. This is achieved by associating parameters with roles. Parameters are used to define the granularity level of the role. In the example of the bank tellers presented previously, the teller role can be parameterized by an amount limit parameter. Then each teller can be assigned a maximum amount limit when assigned to the role.

Fischer et al. [38] proposed the object-sensitive RBAC (ORBAC), which is a generalized RBAC model for object-oriented languages. ORBAC addresses the lack of expressiveness of RBAC by using parametrized roles. In ORBAC, privileged operations are parametrized by a set of index values, which are used to distinguish the granularity level of the roles between users. A privileged operation can only be invoked if both the required role is assigned to the user who invokes the operation and the role's index values matches the operation's index values.

Parameterized RBAC was the first initiative to address lack of expressiveness in role definitions, but parametrized RBAC is still not sufficient to express many authorization requirements. In the example discussed above, it is not possible to check the amount against currencies and to find the amount value against the home currency of the bank. Expressiveness of RBAC can be further improved should we introduce possibilities to make validations on parameters. In addition, we provide a new type of parameters that can have values common to all instances of roles. In our proposed access control model (ROAC), we address these limitations and further improve the concept of roles and permissions.

4.3 The Role-Oriented Access Control Model Overview

In the previous section, we showed that parameterized RBAC was proposed to address standard RBAC's lack of expressiveness when defining roles. The proposed approach adds some flexibility when defining roles. In parameterized RBAC, computations involving parameters of roles must be performed at the application side. This is similar to plain old record types of *structs* in procedural programming languages. Object-oriented programming languages

have introduced the notion of encapsulation that is wrapping data and methods within classes in combination with implementation hiding [34]. The idea here is to transplant those ideas to the definition of roles. With parameterized RBAC it is possible, for example, to specify an amount limit and a currency as parameters to the teller role. But it cannot provide further possibilities to compute the amount against the home currency. As an example, if we pass to the teller role 1000 as an amount and *EUR* as a currency, the amount is not equivalent to 1000 with *USD* currency. Moreover, it provides no way of adding static parameters to roles, i.e. when the static parameter is changed it takes effect on all instances of the role. If an organization requires to disable a role from the access control system, but the organization cannot delete it, since it is associated with records in their audit trail. Deleting the role causes inconsistencies within the system. A better way to cope with this issue is to flag the role as deleted.

In ROAC, we address limitations of existing RBAC models by adjusting and transplanting concepts of object-oriented programming languages to the context of roles and permissions. Roles and permissions in ROAC are analogous to objects in object-oriented programming languages. Like objects, roles and permissions can hold data (variables) and operations (methods). Similarly, objects can inherit from other objects typically expressing an *is-a* relation, roles can be organized into hierarchies with different relationships between super nodes and their sub-nodes.

The core ROAC model consists of three main elements: users, roles, and permissions. Users are principals requiring access to a software system. Roles project job functions within organizations. Roles can be further fine grained to represent sub-functions e.g. a job function can be a *teller* and a sub-function can be *AccountHolder*. Permissions are privileges to execute operations or access objects in the system. Users are assigned to role instances and permission instances are assigned to roles. Since permissions usually correspond to operations and/or objects in a software system, parameters and validators should be included in permissions and propagated back to roles when permissions are assigned to roles. This means that roles combine all parameters of their assigned permissions. The values of parameters are set during users to roles assignment. The structure of the ROAC model is shown in Fig. 4.1.

ROAC hierarchical model supports two hierarchies; the *is-a* hierarchy and the supervision hierarchy. In the *is-a* hierarchy, senior roles inherit all permissions and definitions of junior roles. The *is-a* hierarchy in ROAC does not necessarily reflect role hierarchies defined in the standard RBAC model, it could be also used for deriving new roles and re-using definitions of existing roles. The other kind of hierarchy supported by ROAC is the supervision hierarchy. The supervision hierarchy reflects organizational hierarchies. The hierarchical ROAC model is

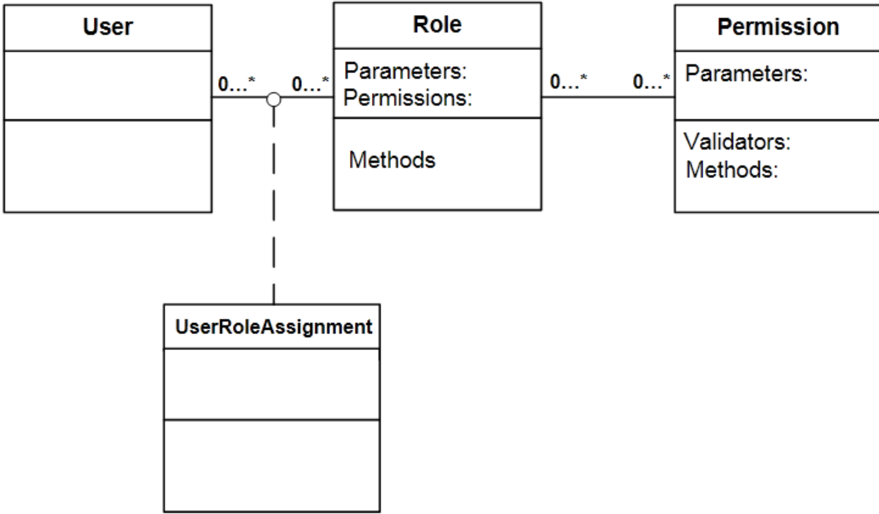


Figure 4.1: UML diagram of the ROAC model.

explained in more details in the fifth section.

4.4 ROAC Reference Data Model

The central notion of ROAC is that instances of roles and permissions are considered as objects, therefore, they are able to encapsulate data and perform operations. In this section, we summarize the main features of ROAC in a reference data model.

In the ROAC model, we extend the principle of roles and permissions to become analogous to object-oriented classes. Both roles and permissions are equipped with variables and methods. Parameters are firstly defined in permissions and then propagated back to roles. Parameters are attributes (also called fields or data members) as in object-oriented languages. Once permissions and roles are defined, instances of both roles and permissions can be created. Roles are assigned in a many-to-many relationship with permission instances. Roles can also have extra parameters defined that are not in permission instances assigned to the role instances. These parameters are of type static. Static parameters can be defined in permissions and in roles. Once a value is set for a static parameter, it takes effect for all instances of the role or permission. Static parameters are similar to static variables in object-oriented languages. Static

variables in object-oriented languages store values for the variables in a common memory location, all objects of the same class are affected if one object changes the value of a static variable [59]. Static parameter values can be initialized when static parameters are defined, and their values can be changed by static *setter* methods. Roles and permissions can also have private attributes which are variables defined to be used in methods or validators internally.

Definition 1: Role and Permission Parameters.

Role and permission parameters are attributes of roles and permissions. Parameters are declared by specifying the parameter name, data type, and modifiers.

Methods are either used as validators or administrative functions such as setters and getters. Validators are methods defined in permissions for computing the authorization decision. The simplest form of a validator is an empty validator. An empty validator grants authorization on an operation in a software system to any user that possesses a role that is assigned the permission that contains the validator definition. Extended form of validators takes inputs from the environment and may connect to external systems to compute the authorization decision. Validators always return a *Boolean* value, *True* if it grants authorization and *False* otherwise. The convenient operations that validators most often perform are to check parameter values extracted from user/role assignment against parameters passed to operations in software systems protected with the ROAC model. Validators can also implement authorization policies. The choice of parameters, static parameters, and validators often depends on the organization and the type of operations and objects they need to protect. Static parameters can hold temporal information about the roles. These temporal properties can be validated by validators. It could be useful in an organization to add a new role in their access control system, and they decide to start using the role on a specific date. The organization can define the role with a static parameter *StartDate* and assign the role to users. Then they can validate the *StartDate* before granting access on an operation. There are many scenarios where static parameters can help organizations maintain dynamic properties of their access control system.

Methods in ROAC are of great importance. Methods can be defined in roles and in permissions. The purpose of methods in ROAC is to provide administrative functions over roles and permissions and to handle operations on role and permission parameters.

Definition 2: Permission Validators.

A validator is a permission member operation that provides an authorization decision. The definition of a validator consists of a signature and a body. The signature specifies the name of the validator and its input parameters. Validators always return Boolean values. *True* if authorization is granted and *False* if denied. The body of the validator is the implementation of validator, which consists of a sequence of programming statements implementing the authorization conditions.

Definition 3: Permission and Role methods.

Methods in permissions and roles are member operations. Method definitions consist of a signature and a body. The signature specifies method name, input parameters, and a return value. The body of the method represents the method's business logic implementation by a sequence of programming statements.

Definition 4: ROAC Permissions.

A permission is a datatype characterized by operations and attributes. Operations and attributes definitions are the same for all instances of a given permission. Permission non-static attribute values are specific to instances derived from a given permission. A permission determines an access authorization on one or more objects or one or more operations in a software system. Permissions in ROAC consist of parameters, validators, and methods. Parameters are attributes, while validators and methods are operations.

Definition 5: ROAC Roles.

A Role is a datatype characterized by operations and attributes. Attributes in roles correspond to role parameters propagated from permissions assigned to the role, and to static attributes of the role. Role operations correspond to role methods that provide administrative operations.

Definition 6: ROAC Data-Types.

Datatypes in ROAC correspond to the type of parameters in permissions and roles. Datatypes supported by ROAC depend on the programming language at stake, which usually are primitive data types and reference data types (objects).

Definition 7: User-Role Assignment.

Let U be a set of user instances from user, R be a set of role instances created from different roles. Let M be a set of parameters of roles and V be a set of possible values for parameters. The user-role assignment is a many-to-many relation, given by the following mapping:
UA = ([u,r] (m_1=v_1, ..., m_n=v_n)) , u ∈ U , r ∈ R, $m_1..m_n$ ∈ M,$v_1..v_2$ ∈ V

Definition 8: Role-Permission association.

Let R be a set of different roles, let P a set of permissions instances, let M be a set of permissions parameters and let R' be a role instance created from R. The role-permission association is given by the following mapping:
RA = (r,p($m_1..m_n$)) r ∈ R ,$m_1..m_n$∈M , r' = r(p_{pre},p) r'∈R', p_{pre} is the existing role permissions

We have until now defined the different elements of the role-oriented access control model. We now discuss how interactions between the different elements are established. Afterwards we use an example to explain these interactions.

In ROAC, users are assigned to roles and permissions are assigned to roles. Actually, one of the major advantages of RBAC is simplification of permissions management. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed [87]. This is a great advantage that can be provided if user-role and role-permission assignments can be achieved dynamically.

We designed relations between ROAC elements to be implemented dynamically. In the user-role assignment, users are associated to role instances by administrative methods. The role definition is not changed during this process. Parameter values of roles are set during the user-role assignment. This enables organizations to define different parameter values for different users, and therefore, provide different granularity levels of roles. The role-permission assignment is also achieved similarly. If a new permission is to be added to the access control system, it does not need redefinition of roles that need to be assigned the new permission. Roles have data structures that contain all permissions assigned to roles. The enumeration can be dynamically updated by administrative role methods for associating new permission instances to roles. Roles also have data structures that contain the parameters of permissions. Parameters of a role are the set of parameters of all permissions associated to the role. The parameters data structure is updated each time a new permission

instance is associated to the role. As well, since multiple permissions may share similar parameters, such as an *amount* value of a bank teller role permissions; all similar permission parameters are considered as one parameter. The only condition is that those parameters must share the same name and datatype. It might happen that a new permission is added to a role in a live environment where the role is already assigned to users, so an administrative function is also provided to set and update particular parameter values for particular users. More so, depending on authorization requirements, more administrative methods can be added to roles. When permissions are assigned to roles, static parameters are not propagated back to roles. Since static parameters are corresponding to the permission and their values are common to all instances of the permission.

Fig. 4.2. shows an example of a role definition and a permission definition. The role reflects a junior *teller* role in a bank. The permission is a privilege for withdrawing money from a bank account. The *Withdraw* permission has two parameters: *AmountLimit* represents the maximum amount of a transaction the teller can perform and *ListOfCurrencies* represents the allowed currencies for the teller. The static parameters of the role are: *StartDate*, which determines when the role is activated and *ExpiryDate*, which determines when the role expires and is retired. The *Disabled* flag determines whether the role is enabled or disabled, the *withdraw* permission has also a *disabled* flag. The *Withdraw* permission has one validator to validate the *Amount* specified in the transaction against the *AmountLimit* of the role and to check if the currency of the transaction is in the *ListOfCurrencies* of the role. The *ValidateHomeAmount()* validator converts the currency of the transaction to the home currency of the bank, and then, it compares the transaction amount with the *AmountLimit*. This computation is required as the home amount value depends on the currency of the transaction. For example, if the user has an *AmountLimit=10000*, and the home amount is *EUR*, and the amount of the transaction is 20000 with the *YEN* currency. Then the transaction should be authorized. The static methods defined in the role are used for setting and getting values of static parameters and for modifying and querying permissions. The *ValidateHomeAmount* validator may check if the permission is enabled or not before deciding to authorize.

4.5 Generalization in the Role-Oriented Access Control Model

In the object-oriented paradigm, inheritance is a mechanism that implements *is-a* relationships between classes. Inheritance allows hierarchically related classes to reuse and absorb features by inheriting class members (variables and

Teller

Parameters:

ParametersList =
(MaximumAmount, MoneyAmount<Decimal, Currency>),
(AllowedCurrencies, Set<Currency>),
// Also parameters of other Permissions.

StartDate: DateTime static = 01/01/2018
ExpiryDate: DateTime static = 01/01/2030
Disabled: Boolean static = false

Permissions:

PermissionsList = {Withdraw, ..., ...}

Methods:
//getters and setters.
.
.

Withdraw

Parameters:

ParametersList =
(MaximumAmount, MoneyAmount<Decimal, Currency>),
(AllowedCurrencies, Set<Currency>)

Disabled: Boolean static = false

Validators:
validateAmountCurrency(Amount, Currency): Boolean
//Calculate the Amount against refrence currency (REF_Amount)
// Return true if REF_Amount <= MaximumAmount
// Return false if REF_Amount > MaximumAmount

Methods:

CalculateRefAmount(amount: Decimal, cry: Currency): Decimal

//getters and setters

Figure 4.2: An example role and permission.

methods). Most existing RBAC models support role hierarchies based on a similar inheritance mechanism found in object-oriented languages.

The advantages most commonly associated with inheritance in the object-oriented paradigm are: malleability and reusability. Malleability facilitates program construction, maintenance, and extension through factoring the definitions common to a set of classes into a single class called the super-class and then any change required in the common behavior can be done only once in the super-class. Reusability facilitates the reuse of code and data by defining abstractions in terms of existing abstractions. This greatly reduces the development efforts by reusing existing software components [21].

Hierarchical ROAC supports multiple inheritance by allowing a role to have more than one parent. Despite the advantages of multiple inheritance, it introduces a new complexity. Two or more parents may define identifiers with the same name [21]. Roles have multiple members such as parameters, static parameters, and validators. The definition of roles in ROAC might introduce some challenges as encountered when defining inheritance in object-oriented languages. One challenge is name conflicts. When two junior roles are to be inherited by a third senior role, where the two junior roles have two parameters (or static parameters) that have identical names. This problem has been exposed in object-oriented languages and there have been some approaches put together to address this problem. In object-oriented languages, strategies for resolving conflicts in multiple inheritance are divided into two main categories, depending

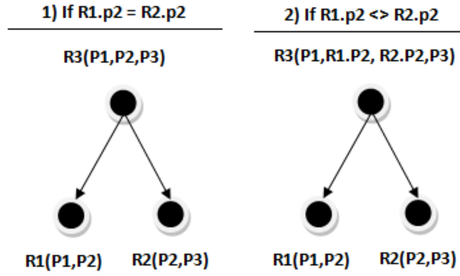


Figure 4.3: Name conflict resolution in ROAC.

on whether resolving the conflict requires interaction with the user or not [33]. In the category where no interactions are required from the users, object-oriented languages automatically resolve the conflict. They rank the objects parent and take the property with highest rank. They use linearization to construct a total ordering of all classes. Linearization solves runtime conflicts without human interventions, but it has two drawbacks: it masks ambiguities between otherwise unordered ancestors, and it fails with inheritance graphs that it deems inconsistent [21]. The other technique used to solve name conflicts requires interventions from users, such as explicit designation as in C++, exclusion as in *CommonObjects*, and renaming as in Eiffel [33]. Renaming gives the developer the power to decide on properties names and to choose appropriate names. It also avoids complexity and inefficiency of linearization. In our approach to role inheritance, we adopt the renaming approach. If a role is inheriting from two roles that have the same parameter or static parameter names, then we rename the parameter if the two parameters are different and we retain parameter names if they are identical and hence combined into one parameter. In this case, the two parameters must have identical data-types. Fig. 4.3 shows an example of how conflicts are solved in ROAC by renaming. In part one of the figure, role R_3 is inheriting roles R_1 and R_2. R_1 has two parameters P_1 and P_2. R_2 has two parameters P_2 and P_3. If P_2 of R_1 is identical to P_2 of R_2, then R_3 inherits only three parameters P_1, P_2 and P_3. In part two of the figure, P_2 of R_1 is different from P_2 of R_2. Then R_3 inherits four parameters, and P_2 of R_1 and P_2 of R_2 must be renamed as shown in Fig. 4.3.

Generalization in ROAC has two sides: roles and permissions definition inheritance and permissions inheritance. In the roles and permissions definition inheritance, the objective is re-usability by factoring the definitions common to a set of roles or permissions into a single role or permission. Permissions instances associated to roles are not considered in roles definitions inheritance. In permissions inheritance, senior roles can inherit subsets of permissions from

the junior roles. In many organizations, the actual hierarchies are supervision hierarchies rather than *is-a* hierarchies. As an example, in a bank, the branch manager could inherit the *teller* role, but he might not need to inherit the permission of initiating payments of the *teller* role. While it might be required that other senior users inherit the teller role, and require the permission of initiating payments, but they do not need the permission of initiating transfers. Therefore, the inheritance model must enable selective inheritance of roles, to enable selecting permissions from junior roles.

In the object-oriented paradigm, encapsulation is a technique used for hiding data within classes and preventing outsiders from manipulating class members directly. Some object-oriented languages such as Java define access control rules that restrict the members of a class from being used outside the class. This is achieved by access control modifiers. The encapsulation model in object-oriented languages is not satisfactory for access control. As in access control, it is required to be more selective regarding permissions when performing inheritance hierarchies. As a consequence, we designed ROAC with two sides inheritance. Firstly, the inheritance for roles and permissions, in which only the definition of the role or permission is transferred to sub-nodes. This is useful for re-using already defined roles and permissions. Another advantage is that a basic role and a basic permission can be defined and equipped by all common administrative methods needed to manipulate administrative functions over roles and permissions. Then all other roles and permissions in the system can inherit from the basic role and the basic permission. Secondly, the permission inheritance has to be defined, which is applicable only for roles. In permission inheritance, permissions of super-roles are transferred to sub-roles. Permission inheritance supports selective inheritance, where a set of role permissions can be excluded from being transferred through inheritance. Permissions can be excluded by providing the permissions *exclusion list* when defining permissions inheritance. For example, let X be a role defined with a set of permissions (p_1,p_2,p_3,..) and let Y be a descendent of X, the exclusion list is (p_1, p_2).

Our target is to provide a mechanism for specifying which permissions can be inherited from a junior role by a senior role. In ROAC, permissions list is defined as a data structure in the role. We can now specify what permissions can be inherited by which senior roles. This enables us to implement supervision as well as the *is-a* hierarchies.

4.6 Discussion

RBAC supports three well-known security principles: least privilege, separation of duties, and data abstraction [87]. RBAC suggests that users are assigned to roles, roles are assigned to permissions, and recommends that roles are assigned only the minimum set of permissions required for tasks needed by members of the roles.

The advances in software systems and the high dependability of organizations on software systems have increased the demand for more requirements on access control. Sandhu and Bhamidipati [85] offered five founding principles for next-generation access control including next-generation RBAC, summarized as ASCAA for Abstraction, Separation, Containment, Automation and Accountability. The first two are included in RBAC96 [87]. Containment includes three principles: least privilege, separation of duty from RBAC96, and incorporates usage limits. Usage limits are constraints on how users can use roles. ROAC directly supports the user limits concept. Conditions on role usage can be easily implemented in roles by specifying them in permission validators, and using static parameters to set values for global parameters. As an example, if it is required to restrict the number of times a role can be exercised in a time frame, we can define two static parameters: one for the time frame and the other for number of exercises, then in the validators we can assert this condition. Similarly, we can define a time frame where the role can be exercised and the role becomes inactive outside the time frame. Similarly, the automation principle can be implemented in ROAC. Constraints can be defined in administrative methods for user-role assignment. For example, expiry of assignment can be defined by using parameters to hold the expiry dates and then implementing the condition in the administrative method that is used to assign users to the role. Different conditions can be implemented for each role. Accountability can be implemented in a combination of three ways. Firstly, sensitive operations require enhanced audit trail, secondly, by notification that requires sensitive operations to trigger a message to an appropriate user, and finally, by escalating the authentication required for sensitive operations [85]. The first and second ways can be incorporated in ROAC. Developers can add any required definitions for roles in validators. Audit information can be supplied to validators in applications and then validators can store them in databases or send them to audit trail systems.

ROAC is an expressive access control model that helps large organizations to provide fine granularity of roles while reducing the number of roles. However, this is applicable when multiple roles can be reduced to single role by using parameters. There is a trade-off between simplifying the management of access rights and providing fine granularity [7]. Therefore, organizations should pay

attention to the design of roles in a way that provides more granularity but reduces the number of roles. The hierarchical form of ROAC can be used to reflect organizational hierarchies which also simplify the management and the view of roles.

We have validated the ROAC model by simulating an implementation using the Java programming language. We have tested the implementation on a security service that was implemented by the authors of the paper.

4.7 Conclusion and Future Work

The main contribution of this chapter is proposing a new access control model, the role-oriented access control model (ROAC). In ROAC roles and permissions are defined as object-oriented classes, where they can have member attributes and operations. ROAC has many advantages compared to existing access control models. One of the main advantages is expressiveness and the possibility to reflect precise organizational hierarchies by ROAC. Another advantage is that organizations can implement any specific requirements for granting authorizations on operations by using validators. The permissions and roles implementation can contain access to external systems like databases and audit log systems to either extract or provide information.

We discussed some related work on existing RBAC models. We explained how existing models attempted to tackle shortcomings of access control models that are encountered when they are put into practice. We focused mainly on two points which are expressiveness of RBAC models and hierarchical RBAC models.

ROAC concepts were validated by an implementation using the Java programming language. In the implementation we created an API that can be used for creating roles and permissions as well as defining relations between the different elements of ROAC such as user/role assignment, role/permissions assignments, and the administrative functions of ROAC. Moreover, the implementation simulated the hierarchical ROAC model. Our future direction from this point is to provide a full feature access control system based on ROAC. Our ideas are to encapsulate separation of duty, role delegation, as well as other features. The target is to make an API that can be used by organizations and researchers, which they can use for constructing their customized access control systems.

Chapter 5

Mitigating Conflicts of Interest by Authorization Policies

This chapter is published in the proceedings of 8th International Conference on Security of Information and Networks, September 2015, Sochi

In many organizations, there are numerous business processes that involve sensitive tasks that may encourage corruption. Conflicts of interest policies are defined in an organization to deter corruption before it can happen. Existing research generally focuses on separation of duties, yet lacks attention for the underpinning conflicts of interest. Moreover, separation of duty is only one particular kind of conflicts of interest. Other kinds do exist and must be resolved as well.

In this chapter a novel approach is proposed to define conflicts of interest policies and to facilitate their enforcement. Our work provides an expressive mechanism that can be applied for a wide range of conflicts of interest that go beyond separation of duty policies. Furthermore, we show how policies can be enforced in the context of the role-oriented access control model (ROAC), which we extend to provide a stronger basis for the enforcement of conflicts of interest policies.

5.1 Introduction

Fraud and corruption are threatening most organizations. The major risk is when different sensitive activities are performed by a single person. This kind of concentration of authority can tempt individuals to become fraudulent or corrupt, especially when a conflict of interest exists between an individual's interests and an organization's interests. Since prevention is always a better approach than a cure, a fraud and corruption prevention mechanism can help organizations maintain their security and reputation before scandals happen and eventually get revealed to the public. Authorization policies can help preventing fraud and corruption by integrating mechanisms that eliminate conflicts of interest.

A conflict of interest is defined as a set of conditions under which professional judgment concerning a primary interest (such as a patient's welfare or the validity of research) tends to be unduly influenced by a secondary interest (such as financial gain) [101]. Separation of duties (SoD) is a well-known security mechanism that can be used to prevent some of the risks imposed by conflicts of interest. Separation of duty is used to formulate multi-person control policies, requiring that two or more different people be responsible for the completion of a task or a set of related tasks.

SoD can discourage fraud by spreading the responsibility and authority for an action or task over multiple people [98]. Separation of duty policies alone are not sufficient to eliminate conflicts of interest in organizations. Indeed, there are many activities with potential conflicts of interest in organizations that

can be done only by one person. Moreover, simply distributing an activity over several people does not necessarily lead to the elimination of fraudulent behavior. For example, a bank can distribute the task of granting a loan to two bankers, but this does not guarantee that the decision is not biased if the loan is going to one of the decision maker's family members. In this case, a clear policy should be implemented to prevent decision makers from participating in the committee of granting a loan to their family members. Furthermore, consider an example of preventing a municipality council member from voting for a deal for a company where he has an interest (e.g. shares). Separation of duty looks at blind distribution of tasks without the ability to rule out task participants who have conflicts of interest with the task, in addition to the extra costs of redundancy when distributing activities on multiple users. Therefore, more concrete policies are required to delimit conflicts of interest.

To remedy the problems caused by conflicts of interest and enact effective policies to prevent them, an understanding of how conflicts of interest operate at the individual level is required as a first step [66]. For example, how to prevent CEOs of companies from manipulating the prices of shares? How to prevent bankers from helping money launderers? The next step is to design policies that can prevent conflicts of interest. According to [56], the design of a policy comprises two steps, (1) a high-level policy design in which a security policy designer evaluates the risks, effects, and sensitivity of the task and determines which users should be involved in the task, and (2) the low-level enforcement design in which a system designer works out a mechanism to model and control the execution of the business task in compliance with a given high-level security policy.

Authorization models are a step in the direction of preventing problems of conflicts of interest. Role based access control (RBAC) [37] is the most popular authorization mechanism. RBAC supports the principle of least privilege [87], which entails that users are assigned the minimum privileges required for achieving their functions [68]. This concept restrains the access to important resources only to authorized users. However, those authorized users still should be further restricted from using authority for achieving their personal interests. Most existing models of authorization policies e.g. [56] [8] [14] [26] [54] focus on separation of duty (SoD) policies. However, separation of duty alone is not sufficient for eliminating conflicts of interest. For example, separation of duty does not help when a medical paper is reviewed by two reviewers who are financed by a pharmaceutical company proposing the paper. In this case, the work is distributed on two persons (or more) but still it does not prevent conflicts of interest. What is required in this case is a policy preventing researchers financed by an organization from reviewing papers proposed by that organization. One of the reasons that restricts the capabilities of existing

models for authorization policies is that they are based on RBAC which lacks flexibility when defining roles. Roles in RBAC are blind in the sense that they are not aware of the application environment. They cannot access data in the system or perform any actions. Roles cannot hold variables, status, methods, etc. [70].

In this chapter we propose a novel form of authorization policies, in which we address the limitations of existing policies. We discuss both high-level policy design and low-level policy enforcement design. The main contributions of our proposed model are:

- We provide a language for high-level conflicts of interest security policies design, the language is an extension of the SoD algebra proposed in [56], which is limited to SoD policies. We adapt the algebra to conflicts of interest policies. We address expressiveness problems of the algebra by supporting parameterization and the usage of workflow variables. Furthermore, we address ambiguity issues of the algebra, by offering possibilities to associate workflow steps in expressions definitions.

- We discuss how to enforce a high level policy design. We use the object constraint language (OCL)[74] for that purpose. Policies are enforced by using an extended version of the Role-Oriented Access Control Model (ROAC) [70]. The ROAC model is based on object-oriented principles and gives more flexibility for defining policies.

- We extend the ROAC model to specify organizational hierarchies. We use organizational hierarchies in solving several conflicts of interest.

- We also extend the ROAC model with a history data structure, such that records exist of all the workflow steps taken so far. Those records are available for consultation by conflicts of interest policies.

The remainder of this chapter is organized as follows: Section 2 reviews existing authorization policies models. Section 3 gives an overview of the role-oriented access control model. Section 4 presents our proposed conflict of interest policies model. In section 5, we discuss implications of our proposed approach. Finally, section 6 concludes our work.

5.2 Related Work

The notion of conflicts of interest has existed in the real world for a long time. However, it has been brought to the lights after starting the concept of banking, commercial and medical organizations. Yet, conflicts of interest did not receive adequate attention in the context of access control. Most researchers have focused on separation of duty policies, which do not adequately model conflicts of interest policies.

In this section, we review some existing authorization policy models. These models focus on separation of duty since the literature lacks to any broad authorization policy models that discuss wider range of conflicts of interest policies.

Clark and Wilson [23] showed how separation of duties is a fundamental principle of commercial and military data integrity control. The paper of Sandhu [80] was among the first to describe a mechanism for the purpose of enforcing separation of duties in computerized information systems, before role-based access control has emerged.

Li et al. [54] have proposed the statically mutually exclusive roles (SMER) constraints to enforce static separation of duty (SSoD) policies. They have shown that directly enforcing SSoD policies is intractable, while enforcing SMER constraints is efficient. Furthermore, they have characterized the kinds of policies for which precise enforcement is achievable and shown what constraints precisely enforce such policies. They have also presented an algorithm that generates all singleton SMER constraint sets, each of which minimally enforces a role-level static SOD requirement. SMER constraints are limited to static SOD constrains and hence unable to model a large set of conflicts of interest policies.

Bertino and Ferrari [18] proposed a language for defining constraints on role assignment and user assignment to tasks in a workflow. The constraint language supports both static and dynamic separation of duties. They have also devised algorithms to check the consistency of the constraints to consistently assign roles and users to tasks in the workflow.

RCL2000 [8] is a role-based constraints specification language built on RBAC96 [87] components. RCL2000 encompasses obligation constraints in addition to the usual separation of duty and prohibition constraints. RCL2000 can express both static and dynamic separation of duty constraints. RCL2000 does not show how constraints written in this language can be efficiently enforced. However, RCL2000 fails to express history or time-based constraints, which are increasingly being used [26]. This prevents defining a wide variety of conflicts of interest policies.

Constraints enforcement has even received less attention from researchers. Crampton et al. [26] introduced the concept of the constraint evaluation structure that is used by the constraints enforcement mechanism to determine whether granting a request would violate a constraint. Two particular constraint evaluation structures form part of the runtime model they introduce in order to enforce dynamic constraints. They built a model for historical information used to record information than is required for enforcing historical constraints.

Probably the best work that has been proposed in this field is the work of Li and Wang [56] [57], in which they propose an algebraic language for formal specification of high-level security policies. It combines qualification requirements with quantity requirements motivated by separation of duty considerations. The algebra has two unary and four binary operators, and is expressive enough to specify a large number of diverse policies [56]. The language is used for high level policy specification. As an example of the algebra, a policy that requires either a manager or two different clerks is expressed using the term *(Manager* $\cup$ *(Clerk* $\otimes$ *Clerk)).* The algebra focuses on separation of duty policies, it provides high level security policy specification. It is not designed for a specific authorization model. Furthermore, it does not verify whether a workflow is compliant with a high-level security policy specified in the algebra. It assumes zero knowledge of the policy designer about the workflow steps. Specified policies in the high-level design can be ambiguous in the enforcement design. Furthermore, its expressiveness is limited and cannot address the definition of many conflicts of interest policies. Our example payment, shown in the next section, cannot be expressed by the algebra, since it does not support parameterized expressions.

The work of Li and Wang [56] was extended in [14]. The authors addressed some problems, which they reported in algebra. Firstly, they addressed the problem related to the fact that no general mapping from the algebraic terms into workflows or to dynamic enforcement mechanisms existed. In particular, a link between the satisfaction of sub-terms and the actions executed in workflows was missing. Furthermore, they addressed the problem of how changing role assignments affect the enforcement of SoD constraints during workflow execution. They constructed formal models of workflows, access-control enforcement, and SoD constraints using the process algebra CSP [14]. This extension of the algebra inherits also the problems of expressiveness and ambiguity of the algebra.

5.3 Overview of the Role Oriented Access Control Model (ROAC)

To define policies for conflicts of interest, the authorization model employed must be very expressive. Furthermore, it must be able to connect to environments beyond the authorization system (e.g. human resources) to access information that could be useful for policy enforcement. As an example, consider a policy that prevents a banker from approving a loan for his or her spouse. Usually, this kind of social information is typically not available in the authorization system itself. Instead of copying that kind of information from external systems, it is much better to consult those external systems directly.

The ROAC model [70] incorporates concepts of object-oriented programming languages in the definitions of roles and permissions. Users, roles, and permissions are all defined as object-oriented classes. The ROAC model is an extended version of RBAC. ROAC extends RBAC with three main features. Firstly, it supports the parameterization of roles with attributes. In this way, it is possible to set up multiple instances of the same role with different levels of granularity. Secondly, it associates behavior with roles and permissions. In specifying permissions, special kind of methods called *validators* can be defined. Validators implement specific requirements for granting authorizations on operations. They can access external systems like databases and audit log systems to either extract or provide information. Validators are methods defined in permissions for computing the authorization decision. Validators always return a Boolean value, true if it grants authorization and false otherwise. The convenient operations that validators most often perform are to check parameter values extracted from user/role assignment against parameters passed to operations in software systems. Thirdly, the ROAC model provides a new extension of the role hierarchy, it supports selective inheritance of permissions in roles to implement supervision relationships among roles.

We selected the ROAC model since it allows the encapsulation of complex policies in roles and permissions. We adapted two extensions of the ROAC model, which are the user hierarchies and the history data structure. Extensions to the ROAC model are shown in section 4. Fig. E.1 shows the UML diagram of the extended ROAC model. The diagram shows that each instance of an activity can be controlled by a different user/role assignment in which role parameters are set (e.g. a role parameter of maximum amount limit of a bank teller role). Furthermore, each authorization action is logged in the history, information is extracted from the activity instance and the user/role assignment.

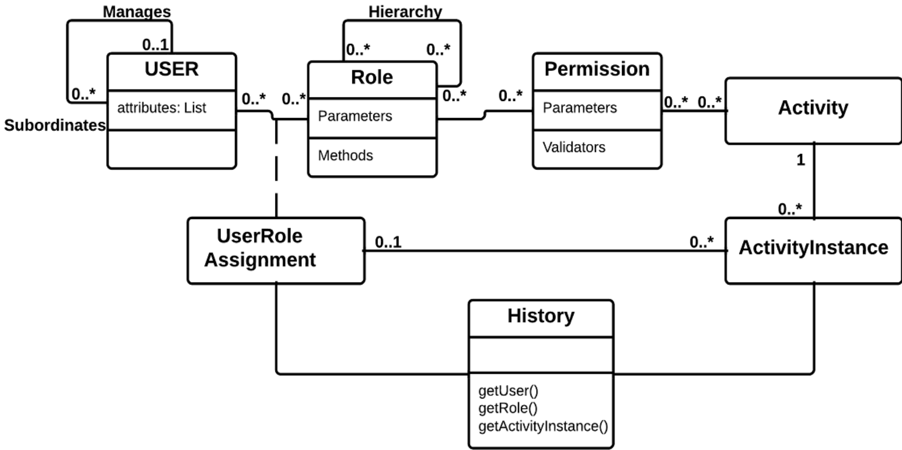


Figure 5.1: The UML diagram of the ROAC model and its extensions.

5.4 Conflicts of Interest Policies

Defining conflicts of interest policies can be a very complex task. The process generally involves a two-step approach; the high-level policy design and the low-level enforcement design [57]. There exists no standard set of conflicts of interest policies that can be applied in all organizations. Usually, the definition of policies depends on the nature of the operations within an organization. However, many conflicts of interest policies defined in organizations implement some type of separation of duties (SoD).

In this section, we start by extending the ROAC model to adapt it for usage within conflicts of interest policies, then we show how policies can be defined and enforced. We illustrate the definition of policies with a remittance payment business process. The first step in this process is initiation of a payment by a user. In this step, a message is created that contains the details of the payment. After that, the payment is sent for authorization to a different user than the one who initiated the payment. Once the payment has been authorized, it can be sent to its destination if the amount is less than one million. If the amount is greater than one million, it is sent for approval to a third user that must be different from the users who were involved in the previous steps of the business process. In addition, according to the policy of the bank, users cannot initiate, authorize, or approve a payment in which they or people close to them are involved as beneficiaries in the remittance. The business process of the remittance payment is shown in Fig. 5.2.

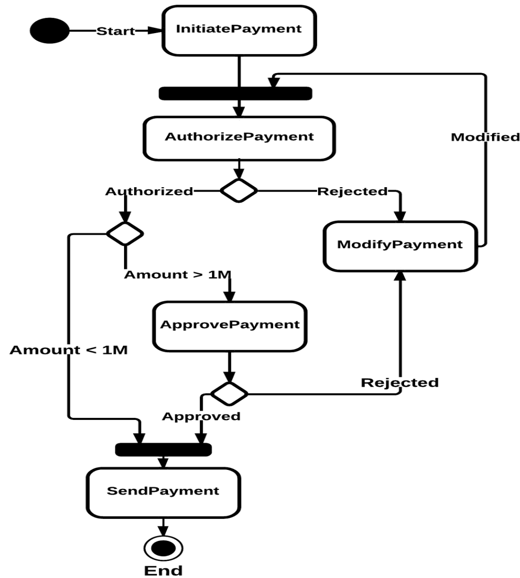


Figure 5.2: Activity diagram showing the remittance payment business process.

5.4.1 Extensions to the ROAC Model

In order to support the definition of a wide range of conflicts of interest policies, we need to have the most expressive authorization model that can be encountered. Therefore, the ROAC model needs to be extended to support customizations that are needed to set up fine-grained conflicts of interest policies. We extended the ROAC model with two main new features. Firstly, we added user hierarchies, which reflect correct organizational hierarchies that are, in many cases, referenced when defining policies. This makes it possible, for example, to express that a manager must approve some tasks performed by the members of his team. We also added a new component to the ROAC model, which is the history data structure. The data structure is required to record events about previous activities in a workflow. This makes it possible to impose that the same person cannot be involved in several steps of some authorization processes.

Users Hierarchy:

Role hierarchies in RBAC can reflect some aspects of an organizational structure. However, typical functional role hierarchies that are constructed in existing role engineering approaches do not reflect organizational structures. They do not take into account the structural characteristics of organizations [52]. This also applies to the ROAC model. The ROAC model provides an extension to the hierarchical model of RBAC, in which it transplants object-oriented inheritance concepts and supports permission-based inheritance. However, it is still unable to reflect precise organizational hierarchies.

In most organizations, superiors do not need to have all the permissions of their inferiors. It is not necessary for a manager to inherit from his inferiors since managers often perform completely different tasks than their subordinates. In many situations, senior users have supervision relationships over junior users [70]. Furthermore, users who are assigned to roles in different levels in the role hierarchy can be at the same level in the organizational hierarchy. For example, a senior developer role typically inherits from the junior developer role, however, both the senior and junior developers themselves may very well be at the same level in the organizational hierarchy.

The organizational hierarchy is of paramount importance when designing security policies. In many cases, the permissions and roles assigned to the user cannot by themselves determine a correct authorization decision. For example, if a manager is required to approve an activity of one of his subordinates, the permission to approve that activity is not enough. We must also check that the manager is approving the activity of one of his team members and not of a user from another team.

We can reflect more precise organizational hierarchies in ROAC by adapting the concept of user hierarchies introduced in [68]. This adaptation leads to modeling user hierarchies with general tree type data structures. This makes it easier to find the responsible (line manager) of any user. The direct line manager is the parent node of the user node, while the next level manager is the parent node of the direct manager node [68]. When a user is added, he is positioned at the appropriate level in this hierarchy. The administrative model of ROAC is also modified to add necessary methods for handling user hierarchy.

The History Data Structure:

Many conflicts of interest policies reference historical events. The activities that were performed and the identity of who has performed them are recorded and made available for consultation by security policies requiring history. Examples of policies that require historical data are operational and historical separation of duties. For example, in our remittance payment business process, it is required that the user approving the payment is being different from the user who initiated the payment. To monitor this, we need to record the user who has initiated the payment instance. When a user subsequently attempts to approve the payment, we can consult the history to verify that the two activities are not performed by the same user.

We extended the ROAC model with a historical data structure, which contains records of events for each activity instance. The events contain the activity instance and the user who has invoked it. The activity instance can be identified by a unique identifier. For example, each payment instance should have a unique reference number (usually called the transaction reference). In the future, there may be a need to extend the history data structure with more fields such as the role and/or the permission involved. The ROAC model with its extensions is shown in the UML diagram in Fig. E.1.

The history data structure is populated after each successful authorization. The activity data can be recorded in the data structure by the permission validators. We show this in section 4.3. The history data structure is optional if the application logs contain equivalent information as recorded in the history data structure, and the information is made accessible to the authorization system.

5.4.2 Specification of Conflicts of Interest Policies

In this section, we explain the high-level policy design. The first step in the high-level policy design is identifying conflicts of interest in the organization. The policy designer plots the high-level design, in which, he specifies how existing conflicts of interest can be avoided. The policy designer has to understand organizational objectives and articulate major policy decisions to support these objectives. The policy is specified at a high-level by the security policy designer and the actions of the security administrator should be subject to this policy [8].

Li and Wang [57] proposed an algebra for defining the high-level policies

(hereafter referred to as SoDA), but it suffers from some shortcomings as explained in section 3. We therefore extend the algebra proposed in [57] such that we are able to parameterize algebraic expressions, use workflow variables, specify workflow steps, and utilize the user hierarchies. We give a brief introduction to the algebraic expressions of Li and Wang in the next paragraph, and then we explain our extensions.

Definition 1 (SoDA Algebraic Expressions): The components of Li and Wang [57] algebraic expressions are:

- Atomic terms: which can be any of the following three forms: a role $r \in R$, a set of users $S \subseteq U$, or the keyword *All*. Where *All* refers to the set of all users.

- Operators: the algebra of Li and Wang [57] has two unary operators ($\neg$ and $+$) and four binary operators ($\otimes$, $\cup$, $\cap$, $\odot$). The unary operator $\neg$ has the highest priority, followed by the unary operator $+$. The rest of operators has the same priority. Operators $\cup$ and $\cap$ are commutative.

- A SoDA expression ϕ is a group of atomic terms separated by operators.

An SoD policy that requires three different users can be expressed using the term ($All \otimes ALL \otimes All$). Variables that can be used in the algebra are roles and sets of users. They consider a role as simply a named set of users, rather than the role concept in RBAC and ROAC. The expression ($Manager \cap Accountant$) requires a user that is both a *Manager* and an *Accountant* to satisfy it. The expression ($Manager \cap \neg$ {*Alice, Bob*}) requires a user that is a manager, but is neither *Alice* nor *Bob*; the sub-term $\neg$ {*Alice, Bob*} implements a blacklist. The expression ($Physician \cup Nurse$) requires a user that is either a *Physician* or a *Nurse*. The expression ($Manager \odot Clerk$) is satisfied by either two different users a *Manager* and a *Clerk* or one user that is both a *Manager* and a *Clerk*. The expression ($Manager \otimes Clerk$) requires two different users a *Manager* and a *Clerk*, one user that is both *Clerk* and *Manager* does not satisfy the expression [57].

Parameterization:

The SoDA algebra of Li and Wang [57] is designed for separation of duty policies. In many organizations, however, policies may require different actions depending on some parameters such as permission and role parameters or variables of a workflow activity. The values of role and permission parameters are set during

user/role assignments, while workflow variables are used to hold data needed internally by the process, or to exchange data between processes. For example, specifying a maximum amount for a bank teller role, which means that he can perform transactions up to the specified amount limit. The corresponding workflow variable is the payment amount. In our payment example, the algebra cannot specify that the payment must be approved by two different users if the amount value is greater than one million, while one user is sufficient for approving the payment if the amount is less than one million. In most cases, parameterization is needed to fine tune policy definitions. We therefore extend the algebra with facilities for parameterizing expressions to define different actions according to these parameters, which we call *parameterized algebraic expressions.*

Definition 2 (Parameterization of expressions):
A parameterized algebraic expression takes the form:
$\phi(x_1, x_2....., x_n)$, where $x_1, x_2, .., x_n$ are parameters. The parameterized SoDA expression is defined as a conditional expression of the form:

$$\phi(x_1, x_2, ..., x_n) = \begin{cases} \phi_1 & , p1(x_1, x_2 ..., x_n) \\ \phi_2 & , p2(x_1, x_2 ..., x_n) \\ ...... & \\ \phi_n & , Otherwise \end{cases} \tag{5.1}$$

where p_1, p_2, ... , p_n are logical expressions. ϕ_1, ϕ_2, ..., ϕ_n are SoDA expressions.

To demonstrate how parameters can be used in policy definitions, we show a policy example for the payment example shown in the activity diagram of Fig. 5.2. The policy definition is modeled by the following parameterized algebraic expression:

$$\phi(Amt.) = \begin{cases} Clerk \otimes Manager & , Amt. \leq 1M \\ Clerk \otimes Manager \otimes & \\ \qquad Manager & , Otherwise \end{cases} \tag{5.2}$$

Workflow variables:

Variables in the algebraic expressions of [57] can be either users or roles (sets of users). Using this kind of variables, we cannot specify all kinds of restrictions. As an example, we are not able to specify a well-known conflict of interest policy stating that a user cannot authorize a payment if he is a sender or beneficiary of the payment. Therefore, we extend the algebra with a new kind of variables

that are the workflow variables. In the remittance payment example, we can use the workflow variables such as *sender*, *beneficiary* and *amount*.

Definition 3 (Extension of SoDA terms with workflow variables): Atomic terms can take any of the following four forms: a role $r \in R$, a set of users $S \subseteq U$, a workflow variable $v \in V$, where V is the set of workflow variables, or the keyword *All*.

We can re-write expression (5.2) as follows:

$$\phi(Amt.) = \begin{cases} Clerk \otimes (Manager \cap \neg \\ \quad \{Payment.sender, \\ \quad Payment.beneficiary\}) & , Amt. \leq 1M \\ Clerk \otimes (Manager \cap \neg \\ \quad \{Payment.sender, \\ \quad Payment.beneficiary\}) \\ \quad \otimes Manager & , Otherwise \end{cases} \tag{5.3}$$

Another kind of variables that can be used in expressions are variables that can be derived from the ROAC administrative model. For example, in expression (5.3), we can specify that the *Manager* approving the payment has to be the line manager of the *Clerk* who approved the payment. In this case, we can use the administrative model of ROAC to determine the line manager of the *Clerk* user. Expression (5.3) can be re-written as follows:

$$\phi(Amt.) = \begin{cases} Clerk \otimes (Manager \cap \neg \\ \quad \{Payment.sender, \\ \quad Payment.beneficiary\} \cap \\ \quad \{Clerk.LineManager\}) & , Amt. \leq 1M \\ Clerk \otimes (Manager \cap \neg \\ \quad \{Payment.sender, \\ \quad Payment.beneficiary\} \cap \\ \quad \{Clerk.LineManager\}) \\ \quad \otimes Manager & , Otherwise \end{cases} \tag{5.4}$$

Workflow steps:

Another shortcoming of the algebra presented in [57], is the ambiguity encountered in the enforcement design step. Expressions used during high level policy design could be ambiguous. For example, the policy that two users must be involved in a remittance payment does not mean that the remittance business process is secure. It is important to specify in which steps of the

business process the users must get involved. The high-level security design must provide a clear and unambiguous description of the policy. According to [57], policy designers are not required to have detailed knowledge of the actual steps through which the tasks are carried out. But this is not always correct, since in most cases the business process has clear milestones that are known by the security policy designers. Furthermore, the policy designer needs to understand the environment for which he is designing policies in order to provide unambiguous policy definitions. The policy designer must understand organizational objectives and articulate major policy decisions to support these objectives.

We extended the algebra with the ability to specify business process steps in policy definition expressions. To illustrate this idea, consider the business process of Fig. 5.2. The policy expression (5.2) defined above states that a *Clerk* and two different *Managers* need to be involved in the business process. But it does not specify in which step of the business process the *Clerk* and the *Managers* need to get involved. During the enforcement design, different interpretations can be found for this expression. This might yield an insecure enforcement of the policy. For example, the payment of amount less than one million can be created by a *Clerk* then modified by a *Manager*, then it is modified by the *Clerk* who initiated the payment. This complies with expression (2). It does not matter who approved the payment. But it is not secure from a payment business process perspective. The policy designer wants the payment to be approved by a *Manager* in the last step before it is sent to its destination, to make sure that the payment is not manipulated after that.

Definition 4 (extension of SoDA terms with Workflow steps):
Atomic terms that take the form of a role can take the form *r(s)* where $r \in R$ is parameterized with a workflow step $w \in W$, where W is the set of all workflow steps.

Hence, expression (5.2) can be modified according to our extension as follows:

$$\phi(Amt.) = \begin{cases} Clerk \otimes Manager(Authorize) & , Amt. \leq 1M \\ Clerk \otimes Manager(Authorize) & \\ \quad \otimes Manager(Approve) & , Otherwise \end{cases} \tag{5.5}$$

The above expression makes it clear that the *Managers* are involved in the *Authorize* and *Approve* steps.

The term describing the policy for our remittance payment: "Payment is initiated by a user and must be authorized by a *Manager* if the payment amount is less than one million. If the amount of the payment is greater than one million, the payment must be further approved by another *Manager*. In both

cases, the managers approving or authorizing the payment should not be parties in the payment" is modeled by the following term:

$$\phi = \begin{cases} All \otimes (Manager(Authorize) \\ \quad \cap\neg\{Payment.sender, \\ \quad\quad Payment.beneficiary\}) & , Amount \leq 1M \\ All \otimes (Manager(Authorize) \\ \quad \cap\neg\{Payment.sender, \\ \quad\quad Payment.beneficiary\}) \\ \quad\quad \otimes(Manager(Approve) \\ \quad\quad \cap\neg\{Payment.sender, \\ \quad\quad\quad Payment.beneficiary\}) & , Otherwise \end{cases} \tag{5.6}$$

The keyword *All* means any user who has permission to initialize the payment regardless of the role assigned to him.

The new design of the expression eliminates any ambiguities, by specifying explicitly the sensitive business process steps in the high-level policy design. It is also possible to specify that a user cannot be involved in a specific step. For example, it is possible to specify that: if the payment amount is greater than one million, the second manager cannot be the user who initiated the payment.

Users Hierarchy:

We have also extended the ROAC model with user hierarchies. In many organizations, certain user activities within workflows are required to be approved by the line managers of users who have performed certain activities. This requirement is usually set to prevent other managers with authority who belong to other departments from approving the activities of users who do not belong to their teams.

In order to achieve this requirement, we introduce two functions to the SoDA algebra which find the line manager(s) of a user and the team members managed by a line manager.

Definition 5 (extension of SoDA terms with users hierarchy): Two new atomic terms are added to the algebra, *inferior(u)* and *superior(u)*. Where *superior(u)* is the set of nodes (managers) higher than u in the organizational hierarchy, *inferior(u)* is the set of all nodes below the node u in the organizational hierarchy.

For example, if user *Alice* is managing users *Bob* and *John*. Then *superior(Bob)* is *Alice* and *inferior(Alice)* is both *Bob, John*.

The expression (*Manager* $\otimes$ *inferior(Alice)*) is satisfied by a manager and one of the team members of *Alice* who is either *Bob* or *John*.

5.4.3 Conflict of Interest Policy Enforcement

Conflict of interest policies enforcement is triggered once the policies are designed at a high level. Policy enforcement comprises enforcement design and implementation of the policies. In enforcement design, a system designer designs a mechanism to model and control the execution of the business task in compliance with a given high-level security policy [56]. In this step the high-level policy is mapped to the business processes. For example, In our remittance payment example, the policy defined in term (5.5) is mapped to the remittance payment workflow steps in the enforcement design. Thereafter, the designed enforcement policy is implemented.

We use OCL expressions to model policy enforcement design expressions. The reasons behind this are, firstly, to make it easier for implementing the policies, since the ROAC model is object-oriented. Secondly, to eliminate any ambiguities that might be encountered when transforming algebraic expressions to code.

To illustrate this, we consider again our example remittance payment. The payment workflow consists of several steps as shown in Fig. 5.2. The enforcement design of expression (5.5) is as follows:

1- We identify the steps in our workflow that are involved in the enforcement design. According to the expression, the *Authorize* and *Approve* steps must be involved. Furthermore, we need to involve a third user according to the expression, represented by the *All* keyword. This step can be ambiguous, since we can choose from *Initiate*, *Modify* and *Execute* steps. Logically, the *Initiate* step is involved.

2- For all the involved steps, we write the corresponding OCL expression if an OCL expression is required. For our example, it is not required to write any OCL expression for the *Initiate* step, since this step can be done by any user who has permission to initialize the payment, this is the only thing checked at this step. For the other steps, *Authorize* and/or *Approve*, it is required to write OCL expressions to verify for example that the user authorizing a payment has a manager role and was not the user who initialized the payment. Furthermore, to make sure that the user is not a party in the payment. The OCL expressions required for the *Authorize* and *Approve* steps are as follows:

```
context::Permission
inv: Authorize:
Role::Manager in User.AssignedRoles() and
User <> History.getUser(Initiate) and
User not in (Payment.Sender,
Payment.Beneficiary)
```

```
context::Permission
inv: Approve:
Role::Manager in User.AssignedRoles() and
User <> History.getUser(Initiate) and
User <> History.getUser(Authorize) and
User not in (Payment.Sender,
Payment.Beneficiary)
```

Once the policy enforcement design is finished, we can implement the enforcement policy. At runtime, the policy must be validated when granting any authorization decision. The policies in enforcement design are implemented in the validator of each permission mapped to an activity (or a step in the workflow). If a permission is associated with several activities, then each activity that requires a policy must have a dedicated validator in the permission.

5.5 Discussion

An important aspect of security policies is their integrity and containment. An important challenge that is arisen when designing a new model for addressing conflict of interest is the heterogeneous nature of operations at organizations, which incurs different requirements for each organization. The main advantage of our model is that you can develop the policies in object-oriented. This gives a great flexibility to define probably any conflicts of interest policy. However, this adds an extra effort; we need to transform the high-level policy to program code. In other models, it is easier to standardize the implementation of the designed policies and even to automatize them. However, this is not impossible to achieve for our model. This can be done by standardizing the way workflow variables and workflow steps are used with expressions. Furthermore, a key feature that needs to be added to the model is a parser that translates parameterized algebraic expressions to ROAC validators. This requires defining the way the ROAC model communicates and interacts with the workflow. There has been some work proposed in the literature [77] [102] [69] that discusses how authorization models can be integrated in workflows and their specification languages.

Organizations can also minimize conflicts of interest by designing good organizational structures. Organizational structures describe organizational behavior and lines of authority inside organizations. Organizations should be careful when providing permissions for senior directors on functional activities that are executed by their inferiors. Furthermore, organizations should always try to adhere to the least privilege concept. We have reflected the organizational structure in our model by adapting the user hierarchy.

In conflict of interest, the primary interest is determined by the professional duties of an employee, while the secondary interest is encountered when it affects professional decisions in favor of the secondary interest. However, the aim of organizations is not to eliminate or to reduce financial gain or other secondary interests (such as preference for family or the desire for power). It is rather to prevent these secondary factors from dominating or appearing to dominate the relevant primary interest in the making of professional decisions. Since in many cases secondary interest is not illegitimate in itself, and indeed, it may even be a necessary and desirable part of professional practice [101].

5.6 Conclusion

In today's business and corporate environments, wide varieties of security policies are implemented to ensure data integrity and prevent unauthorized activities as well as preventing improper usage of authority. Current authorization models are not expressive enough to support definition of such requirements. Existing work focuses on one type of conflicts of interest policies, which is separation of duties, and hence leaves out a wide range of conflicts of interest policies uncovered.

In this chapter, we proposed a novel approach to conflict of interest policy definition and enforcement. We have extended the algebra proposed in [56] [57] to support parameterization of algebraic expressions and the usage of workflow variables in the algebraic expressions. This offers better flexibility for defining more precise and fine-grained policies and constraints. Furthermore, we addressed ambiguity issues of the algebra, by offering possibilities to associate workflow steps in expressions definitions. We used the role-oriented access control model (ROAC) as the basis authorization model for implementing the conflicts of interest policies. ROAC enables us to define the policies and constraints in the roles and permissions since it is object oriented, and hence to isolate policy definitions from the application code and business logic. We used a remittance payment business process example for explanation of the work. We have also shown how high-level policy expressions can be enforced and how

the algebraic expressions are translated to OCL expressions in the enforcement design phase.

Chapter 6

Comparison, Limitations, and Verification

This chapter describes three additional topics, which are a comparison between the ROAC model and attribute-based access control (ABAC), discussion on the limitations of our research related to the OSDM delegation model, the ROAC model, and the mitigation of conflict of interest policies. Moreover, some suggested mechanisms for verification of mitigation of conflicts of interest policies are presented.

6.1 Comparison with ABAC

The introduction chapter provided a set of mandatory requirements, of which two requirements are of paramount importance for the core access control.

The most important requirement is to comply with the least privilege concept, where users are given the minimum set of access rights needed to accomplish their roles. The concept also requires that access rights are mapped to the smallest access unit. Moreover, operations on protected objects in a software system need to be also fine grained to the smallest possible unit, so that the access rights mapped to them give the least possible access.

The second important requirement is expressiveness of access rights. An access right must provide the flexibility to handle different granularity levels when it is assigned to different users. This requirement prevents adding tremendous

number of access rights to the access control system. On one hand, organizations need different granularity levels of one access right to cater for the differences in seniority between their employees. On the other hand, they need such granularity to account for the same level of an access right that is assigned to different users in segregated zones. For example, several professors have the privilege to teach courses, but they do not teach the same courses.

The expressiveness of access rights definition requirement remedies the consequences of the least privilege concept. However, on top of these mandatory requirements, other features could affect the applicability of an access control system in large organizations. One major feature is to have simplified management of access rights. Organizations usually have many software applications, which results in a tremendous number of access rights. Moreover, some organizations have thousands of users. Therefore, the way access rights are mapped to users must be simple in management and maintainability.

RBAC has remained the dominant access control model in organizations for a long time and remained prevalent. However, many organizations recently started to divert away from RBAC towards attribute-based access control (ABAC) [20]. The main reason for diverting to ABAC is that RBAC cannot express their access control policies. RBAC is restrictive in access policy definitions since accesses are based only on roles and it is difficult to include other characteristics of users and contextual or environmental factors in access control policies [19]. Enterprises needed to include attributes, such as time of day and user location, for distributed and dynamically changing systems. ABAC was identified as a replacement for or adjunct to RBAC [24].

The ROAC model was proposed to address expressiveness limitations of RBAC, by enabling attributes and behavior into the RBAC elements and relations.

In this section, we provide a comparison between ABAC and the ROAC model. Several aspects that affect the applicability of both models are reviewed.

6.1.1 Expressiveness

In ABAC, access control policies are expressed as rules, which represent conditions on user attributes, environment attributes, or object attributes. These rules are evaluated in order to determine the authorization decision. ABAC is highly expressive as any available attributes can be used to express access policies. However, conditions can be expressed on raw attributes, no conversions or behaviors are possible within the rules.

On the other hand, the ROAC model supports both attributes and behavior,

which can be used to provide advanced calculations on attributes (e.g. transfer amount of foreign currency transaction to the local currency).

6.1.2 Least Privilege

In ABAC access policies are defined using attributes. These attributes can be anything about users, environment, or objects. Therefore, access control policies can be fine grained to achieve the required level of the least privilege concept.

On the other hand, the ROAC model relies on the concept of roles, which means users doing similar job functions are assigned similar permissions. Despite the simplicity of management advantage of this approach, users doing similar job functions might not require identical permissions, which might not be relevant to their job. Policies can be defined on levels of elements or relations to further restrict user access. However, this might add more complexity to the defined policies and might cause a role explosion.

6.1.3 Complexity

Defining policies in ABAC requires that a potentially large number of attributes to be managed and understood. Furthermore, selection of attributes is a complex task, as attributes have no meanings until they are associated with a user or an object [20]. Moreover, ABAC can also lead to a rule explosion, somewhat in the same way as RBAC. As a system with a large number of attributes would have an explosive combination of possible rules [1].

The ROAC model uses RBAC elements and relations to manage access control. Policies that cannot be expressed by users, roles, permissions, user-role and role-permission relations are expressed using attributes and behavior in the elements and relations. A large percentage of access control requirements of an organization can be expressed by the RBAC elements and relations. Only the remaining part is defined in terms of attributes and behavior.

ABAC is far more complex than ROAC, and the complexity depends on the number of access policies that can be expressed by RBAC elements and relations.

6.1.4 Maintainability

One major issue with the high number of rules generated by ABAC is maintainability. Policies in organizations keep changing and by the introduction

of new policies and updating existing policies, administration issues might emerge due to conflicting policies, erroneous policies, and unnecessary policies.

The ROAC model groups permissions of similar job functions in a role, then users can be assigned to roles. This offers less maintainability overhead than maintainability of access rules. However, some access policies that cannot be expressed by the elements and relations will get defined using attributes and behavior, this adds complexity to the ROAC model. However, these policies will still be grouped under elements and relations, which makes their maintainability much easier than in ABAC.

6.1.5 Dynamicity

In ABAC, an authorization decision is determined based on the user attributes reflected in objects they want to access. This means that ABAC permissions can be acquired dynamically by virtue of the user's attributes [24].

In the ROAC model, the authorization decision first verifies user policy, user-role policy, then enumerates the roles that are assigned to the user requesting authorization. Afterwards, it checks the role-permission policy and enumerates the user permissions, then it checks the permission policy and the user-permission policy (the parameters). This means that the user must be preassigned to the roles in advance.

6.1.6 Auditability

In ABAC, before-the-fact auditability is extremely difficult due to the enormous number of rules in the access control system. For example, to audit if a user access is in line with the least privilege concept, or to audit correctness of polices from a huge number of rules becomes a real challenge. Furthermore, ABAC is an identity-less access control system and users may not be known before access control requests are made, it is often not possible to compute the set of users that may have access to a given resource [93]. ABAC allows to base an authorization decision on some characteristic of the subject other than its identity [71]. Even in cases where the identities of all users and their assigned attributes are known. In order to calculate the resulting set of permissions for a given user, all objects would need to be checked against all relevant policies [93]. Furthermore, the rules need to be checked in the same order in which the system applies them, as a result, it could be impossible to determine risk exposure for any given employee position [1].

In ROAC, before-the-fact auditability is much easier. Firstly, the model is identity based, therefore, user identities are used in access policy definitions. Secondly, roles assigned to a given user can be easily found, then permissions assigned to those roles can be enumerated. This is a straightforward audit to know which resources or objects a given user can access. However, policies defined in elements and relations (using attributes and behavior) can be more complex to audit.

When the user identity is known, ABAC requires an exhaustive enumeration of the attributes. The full set of access rules, which could number in thousands in some cases, must then be instantiated with user and object attribute values. Because attributes can change dynamically, determining a user's potential permission set will also require instantiating rules with all possible attribute values while a user is active [24]. In the ROAC model, the same strategy can be used for policies defined in elements and relations when auditing which resources a given user can access.

6.1.7 Applicability

ABAC is widely deployed in enterprises, government, and healthcare. However, the ROAC model is not yet deployed in any organization.

6.1.8 Policy Specification

The eXtensible Access Control Markup Language (XACML) [71] is used for expressing ABAC access policies. XACML provides a declarative language for policy definition and an architecture for enforcement of the policies.

The ROAC model does not have a policy specification language. This is a known limitation of the model, and it is discussed in more detail in the next section.

6.1.9 Authorization Decision

Both ABAC and ROAC support multi-factor authorization decisions. However, each of them has a different way of determining the authorization decision. In ABAC, rules are evaluated sequentially one by one, the final authorization decision is calculated according to the rule combining algorithm applied. In the ROAC model, policies are evaluated hierarchically in a specific order as follows:

user policies, user-role policy, role policy, role permission policy, permission policy, user-permission policy (parameters).

Authorization decision evaluation in the ROAC model is more structured, it is clear which policies prevail over others. However, policy designers must categorize the policies in advance into the different levels, e.g., user, user-role, etc. On the other hand, ABAC authorization decisions depend on the rule combining algorithm, and gives more freedom to policy designers to choose which rules prevail.

6.1.10 Policy Conflicts

A policy conflict occurs when an access request matches more than one policy and policies conflict with each other, i.e., yield different decisions [31]. In ABAC, security policies are highly flexible and expressive, but conflicts between policies occur frequently, affecting the security and availability of the system [60]. Such issues happen at the enforcement phase. Availability issues cause rejecting a legitimate user access to a resource, and security issues cause allowing an illegitimate user to access a resource [31].

In ROAC, conflicts can also happen when a user has overlapping access policies, which mainly happen in the context policies defined in elements and relations. As an Example, a user is assigned to the same permission via two roles. Both roles have policies restricting the activation of permission to a specific period of time. A conflict happens if one policy allows the user to activate the permission and the other prevents the user from activating the permission. Both ABAC and ROAC are prone to policy conflicts, the next comparison item we see policy conflict resolution strategies in both models.

6.1.11 Conflict Detection and Resolution

There are two different conflict types that may occur in access policies: static and dynamic conflict. In static conflict, incongruence is found during the initialization phase. In dynamic conflicts, a potential conflict is quite unpredictable and is the result of a run-time action [13].

In ABAC, static conflict detection methods detect conflicting rules before the system runs, therefore, providing a reference for policy maintenance. Dynamic conflict detection is more difficult or even impossible to cover all the possible conflicts in the policy set [60]. Several conflict detection mechanisms has been proposed in the context of ABAC and XACML, for example [60, 97, 32, 49, 99].

XACML uses rules combining algorithms to resolve conflicts at enforcement phase. The rule combining algorithm defines a procedure for arriving at an authorization decision given the individual results of evaluation of a set of rules [71]. XACML 3.0 has 13 algorithms for rules and policy combining. Examples of XACML rule combining algorithms: *deny-overrides*, *permit-overrides*, *deny-unless-permit*, *permit-unless-deny*, *first-applicable*, and *only-one-applicable.*

On the other hand, policy conflict detection has not been studied in detail in the context of the ROAC model. It is up to the policy designer to make sure that the policies are safe and not conflicting. However, conflicts may occur only in the context policies defined in elements or relations. Conflicts do not happen in elements, relations, or parameters, for which a *permit-overrides* resolution strategy is adapted.

6.1.12 Hierarchies

Hierarchies in access control aim at simplifying administration of access rights and to reflect some kind of organization structure. In ABAC, hierarchies are mainly focusing on simplifying management of access rights. Hierarchies can be defined on attributes, which can be classified in a tree structure based on their access control relationship in a system [53]. Hierarchies can also define user and object groups, which are collection of entities of users or objects. Group attributes are attributes assigned to groups of users and objects. These groups get relevant attributes assigned to them, which represent the group characteristics [19]. Hierarchies can then be defined on the groups such as in [19, 92].

In the ROAC model, two types of hierarchies are supported: role and user hierarchies. Role hierarchies represent a role-role relation. Role hierarchies are used to simplify management of access rights. Descendant roles inherit permissions from their ancestor roles. The ROAC model supports the is-a hierarchies and the supervision hierarchy, in which descendant roles inherit a subset of the ancestor role permissions. User hierarchies in ROAC represent the lines of authorities of the organization by implementing partial orders on users. The user hierarchy enables finding the line manager of a user and the subordinates of a manager. Both ABAC and ROAC implement hierarchies that simplify the management of access rights. However, the hierarchies in the ROAC model are more advanced than ABAC, since it supports the supervision relationship and user hierarchies.

Table 6.1 summarizes the differences between ABAC and ROAC.

Table 6.1: Comparison between ROAC and ABAC

Feature	**ABAC**	**ROAC**
Expressiveness	Medium	High
Least Privilege	High	Medium
Complexity	Complex	Moderate
Maintainability	Moderate	Advanced
Dynamicity	Advanced	Basic
Auditability	Difficult	Moderate
Applicability	Applied	Not applied yet
Policy Specification	XACML	None
Policy Conflicts	High	High
Authorization Decision	Sequential on rules	Hierarchical
Conflict Detection & Resolution	Advanced	Basic
Hierarchies	Basic	Advanced

6.2 Limitations

Despite the clear contributions of our research towards improvements on role delegation, mitigation of conflicts of interests, and context-based RBAC policies, there are identified limitations in the provided models. Limitations related to the ROAC model are mainly the absence of some features such as policy specification language and policy conflict resolution. These features are necessary due to the increased complexity of the ROAC model, which are introduced for increasing the flexibility and expressiveness of the model. Limitations in the delegation model are mainly non-tackled problems that might arise when applying the model is some specific scenarios. Limitations related to mitigation of conflicts of interest are related to the algebra that is used for specification of the policies, which has a limited expressive power.

6.2.1 Limitations of OSDM

The OSDM delegation model provides a mechanism for supervising delegation by lines of authority in the organization. OSDM supports both role delegation and permission delegation. Despite the clear contributions of the model, it still suffers from some limitations, which we discuss in this subsection.

Delegator Context Policies

In OSDM, when a role is delegated, the context policies of the role, role-permission, permissions, user-permission (parameters) are taken into account and are validated in the authorization decision. However, context policies on the level of the delelgator user and the user-role assignment are not considered and not validated. To give an example, delegator *u1* delegates his role *r1* to delegatee *u2*. Consider a context policy defined on user level for delegator *u1* limiting activation of all his roles to time frame 07:00 AM to 05:00 PM. Consider a policy defined for delegatee *u2* limiting activation of any role to 07:00AM to 9PM. If delegatee *u2* tries to activate the delegated role *r1* at 8:00 PM, the authorization decision will allow him to activate the role, because the authorization decision takes the delegatee user context policy into consideration in this case. The same applies for user-role context policies.

In delegation, the authority of the delegatee should not exceed the authority of the delegator, and the model should be extended to apply the delegator user and user-role context policies for the delegated roles rather than the delegatee context policies.

Delegation of Managerial Roles

In most organizations, directors and C-level managers are managed by the highest authority line in an organization (usually called the CEO). In case of absence of the CEO, the OSDM model might block the delegation of authority of an absent user managed by the CEO, and therefore, causing a deadlock. In most organizations, management roles are delegated in a different way than other roles in the organization. In most cases, there are predefined rules on who takes over a given managerial role if its assignee is absent. Therefore, the OSDM model might have a limitation that affects its applicability for delegation of managerial roles. The model should be extended with a mechanism that automatically allows the delegation of certain roles according to predefined rules, which bypasses approvals from managers of managers.

Restricting Usage of Delegated Roles

Delegation can be required in organizations in different scenarios. Such as backup of roles, disasters, decentralization of authority, collaboration, etc. In the case of using delegation in the context of backup of roles, this means that the delegated role should be used to substitute the job of the delegator who is absent in this case. In some cases, the access rights and policies might give the

delegatee more authority than he originally had. The delegatee can get more permissions or get higher parameter values than parameter values assigned to him through the original user-role relation. For example, a user *u1* working in a bank, has the right to do both cross-border and domestic transactions is absent and his role needs to be delegated to a user *u2* who is allowed only to do domestic transactions. Consider that *u1* has an amount limit parameter set to *1 million*, and that *u2* has the same parameter value set to *100K*. When the role of *u1* is delegated to *u2*, user *u2* becomes able to do transactions of amounts up to *1 million*. The OSDM model differentiates between original and delegated parameters of the user, however, user *u2* can use his delegated role to perform domestic transactions of up to *1 million* instead of his original role and parameter value, which limit him to a maximum of *100K* amount. An open research problem is this context is: how to restrict usage of a delegated role or permissions to the usage of delegated tasks and how to prevent a user from applying delegated access rights on his original duties.

6.2.2 Limitations of the ROAC Model

We provided a comparison between ROAC and ABAC in the previous section. We have seen that the ROAC model is lagging behind in two main features, which are the absence of a policy specification language and policy conflict detection and resolution. We focus on these two limitations in this subsection.

Policy Specification Language

Applying security policies comprises two main steps: the high-level policy specification and the low-level policy enforcement. Policy specification means how policies can be expressed, whether in natural languages or in formal languages. Natural language specification has the advantage of ease of comprehension by human beings, but may be prone to ambiguities, and the specifications do not lend themselves to the analysis of properties of the set of constraints [8]. A security policy is enforced through the deployment of certain security functionalities within the application [43]. In policy enforcement, policies must be connected to the end application functionalities, which involves two main tasks: mappings between permissions and protected operations & objects, then implementing a mechanism for evaluating the specified policies and computing the authorization decision.

Policy specification is very key in designing access control requirements. Policy specification needs to be segregated from policy enforcement. The access control policies are usually specified by the security analysts and officers who are not

necessarily aware of the technical details behind the application functionalities, they are not necessarily software developers.

Moreover, policies must be separated from the implementation of the system, this enables the policy to be modified dynamically in order to cover the changes in the strategy for managing the system and controlling the behavior of users without changing the implementation of the underlying components of the system [96]. Therefore, we identify two main requirements for applying security policies, a policy specification language, which must be expressive enough to express the different policies. Then, the specified policies must be automatically enforced by enabling automated evaluation of the authorization decision from the specified policies. Furthermore, to enable enforcing the policies on the application functionalities in a dynamic way. Formal specification of security policies provides several important advantages. First, formal specification minimizes the possibility of misunderstanding between policy designers and system designers. Second, formal specification facilitates the analysis of security policies [56]. Third, formal specification enables automated enforcement of specified policies. Automated enforcement eliminates ambiguities and misinterpretations of specified policies by application developers.

The eXtensible Access Control Markup Language (XACML) [71] is the most used policy specification language. It is used to specify policies of different access control models such as DAC, MAC, RBAC, and ABAC. However, XACML is mostly used for specifying ABAC policies. XACML is an OASIS standard that describes both a policy language specification and an access control decision request/response language. XACML uses XML as the basis for the language due to the ease of its syntax and semantics. The request/response language calculates the authorization decisions by evaluating the specified access rules. The authorization decision is one of four values: *Permit*, *Deny*, *Indeterminate* (an error occurred, or some required value was missing, so a decision cannot be made) or *Not Applicable* (the request can't be answered by this service). XACML has a policy enforcement point (PEP), which is responsible for handling access requests from subjects to access resources. The PEP formulates the request based on the different attributes, the request is then evaluated by another component called the policy decision point (PDP), which returns the authorization decision to the PEP, which then allows or denies the subject's request to access the resource [72].

In the ROAC model, there is no formal access control policy specification language that is supported. Policies are specified in natural language, which could cause ambiguities and be misinterpreted by the system engineers or the developers. This limitation can be remedied by supporting a policy specification language such as XACML. However, efforts are needed to translate the specified policies to the ROAC object-oriented code. A request/response language is also

necessary. A request containing the user details and the protected operation or object details must be evaluated according to the specified policies, then a response should be returned with the authorization decision result.

Policy Conflicts

One obstacle to accurate access-control policies is human error; policy authors are prone to making specification errors that could lead to incorrect policies. During specification, access control policies are divided into smaller units called access rules. Conflicts happen when one user combines access rules that when evaluated give conflicting access decisions. For example, a user has an access rule granting him access to one resource and has another access rule that denies the same users access to the same resource. This might sound like a trivial error by a policy author, however, in reality access policies are very complicated and in many cases are hierarchical, which makes it difficult for policy authors to specify conflict free policies. The access rule denying access to the resource might not be associated directly with the user, rather to a group of users in which he belongs. Access rules within organizations change with time, new rules are added, and existing rules get updated. Policy authors might leave their posts and new policy authors join. Furthermore, the high number of access rules in organizations might be difficult to manage. Research in policy conflicts is focusing in two main fields, which are policy conflict detection and policy conflict resolution.

Policy conflicts detection is the process in which conflicts between the policies or access rules are detected. Once conflicting policies or rules are detected, the policy resolution algorithm decides what should be done to resolve the conflict. This is usually done through policy and rule-combining algorithms. There exist several conflict resolution strategies, such as *denial takes precedence* and *the most specific authorization takes precedence* in the literature of access control models [22]. Policy conflict resolution strategies can be static or dynamic. In the static policy detection and resolution, the rules are verified without generating access requests. While in dynamic detection and resolution, conflicts are detected at runtime during evaluation of access requests, the resolution strategy is also applied at runtime.

In XACML, grouping can be defined on policies and access rules. A *PolicySet* can contain a set of policies, and a policy can contain multiple access rules. XACML resolves conflicts using policy combining algorithms, correctly combining the results from evaluation of different policies into one decision. Evaluation of policies and rules may yield different access control decisions. XACML defines strategies for reconciling the decisions of each rule or policy. This is done through

a collection of combining algorithms. Each algorithm represents a different way of combining multiple decisions into a single decision. The combining algorithms can be defined on the policy level *(Policy Combining Algorithms)* or on the rule level *(Rule Combining Algorithms)*. An example of a combining algorithm is the *Deny Overrides Algorithm*, which states that no matter what, if any evaluation returns *Deny*, or no evaluation permits, then the final result is also *Deny*. The combining algorithms are used to build up increasingly complex policies [72].

Policy conflict detection and resolution has not been analyzed in detail in the context of the ROAC model. A rigid strategy for conflict resolution in the elements, relations, and parameters is adapted, which is *permit-overrides*. However, for context policies, in which conflicts are more likely to happen, the resolution strategy is left for the policy author, who must decide on how conflicts are handled. This limitation affects the security and applicability of the model.

6.2.3 Limitations of Conflicts of Interest Mitigation

Our work related to mitigation of conflicts of interest is based on an extension of an algebra, which enables the formal specification of high-level separation of duty policies (SoD). We extended the algebra in order to increase its expressiveness. Algebraic expressions are extended with new variables, which come from the protected operations. Furthermore, workflow steps can be associated with terms. The algebraic expressions were also extended with the possibility to do conditional parameterization in order to enforce different policy levels according to conditional expressions. The algebra was also extended with the possibility to express managers of users in order to set policies requiring managers to participate in activities of their subordinates. A control on the workflow steps execution order was also included to ensure that the conflicts of interest policies are projected correctly on the workflow steps.

The conflicts of interest policy specification algebra provides a means for specification of conflict of interest policies in an easy and unambiguous way. However, the algebra has a limited expressive power, which fail at specifying several conflict of interest mitigation policies.

Spatial and Temporal Context

Spatio-temporal information is of paramount importance in defining access control policies. Authorization policies can change depending on where and when a resource is accessed.

The algebra we use for specification of conflicts of interest policies cannot use spatio-temporal information in specification of the policies. Spatial and temporal variables need to be added to the algebra such as *TimeOfAccess* and *AccessLocation* that contain respectively the timestamp and the geographic location of the subject requesting access to the protected resource. The algebra should also include operators for matching spatial and temporal information. To illustrate with an example, an organization might require an extra step of approval on a transaction from the organization's premises if all users involved in the workflow are working remotely. The organization might specify more restrictive policies if the transaction includes actions from remote users and outside usual business working hours. Such policies utilizing spatial and temporal information are key in fraud and conflict of interest policies specification.

Rule-Based Policies

User behavior is of great importance when defining conflict of interest policies. Policies to defuse conflicts of interest before it could happen need to consider every possible piece of information about the user, the protected operation, the environment, and the user behavior. On the one hand, policies should contain limitations of usage for users. For example, in a conflict of interest policy, policy authors should be able to specify that a user can do a maximum of a certain number of approvals within a time frame. On the other hand, the specification language should support expressing detection and blocking of certain behavior that can be modeled with business rules. For example, a policy might specify that the user needed for a given workflow step should be any user who is a manager, has been working for the company for at least 10 years, and has a minimum of grade A in latest appraisal. This kind of policy cannot be expressed in the extended algebra.

Behavioral and Predictive Policies

Machine learning can be used to analyze large data sets containing user access information. Afterwards, many beneficial models related to mitigation of conflicts of interest can be developed. Firstly, models to detect anomalous user behavior. Machine learning can compare a user's behavior to the past behavior of the user himself or compare a user behavior to his peers to detect anomalous behavior. For example, a user is approving a transaction at an unusual time for the user, a user is approving only high value transactions, while his peers are approving different value transactions, etc. Secondly, machine learning has a predictive power, which can be used to develop models to predict conflicts of interest before they happen.

Machine learning and advanced behavior analysis of users is not possible to express using the extended algebra. This needs more research and probably to build predefined machine learning and behavior detection models, then enable specifying policies using these models.

6.3 Verification

Access control is used in organizations to regulate and control access to protected objects by subjects. Organizations centralize their access control policies to facilitate their management and audit. There are three factors that increase the complexity of access control policies in organizations. Firstly, subjects require access to a huge number of protected objects and operations distributed across many different systems. Secondly, changes and updates are continuously added to software systems. Thirdly, access control policies are regularly updated with changes to existing policies and addition of new policies or removal of existing policies. Identifying discrepancies between policy specifications and their intended function is crucial because enforcement of policies by applications is based on the premise that the policy specifications are correct [45].

Access control policies are a critical component in security. Issues in specified policies such as faulty policies, misconfiguration, or flaws can result in serious vulnerabilities [45]. Vulnerabilities that might happen due to such reasons might enable unauthorized access to protected objects in software systems. In order for policies to achieve the desired protection, the specification and enforcement of the policies must be correct. Another issue that is usually encountered when specifying access control policies is inconsistencies, which can also result in vulnerabilities. Inconsistencies occur when different incompatible policies can apply in a specific situation [95]. Furthermore, high expressiveness of the underlying access control system increases the potential for error in policy implementation and conformance between the specification, and enforcement may not be guaranteed [61].

Policy specifications must undergo rigorous verification and validation through systematic verification and testing to ensure that the policy specifications truly encapsulate the desires of the policy authors [45]. A policy is a set of rules that define the expected behavior of the system enforcing that policy [94]. Policy testing is a technique that can be used to verify policies through test cases. Testing can be used to ensure correct enforcement of specified policies and to ensure correctness and safety of specified policies. Policy testing requires definition of exhaustive test cases that covers all possibilities and variations, which might result in a huge number of test cases. Test cases must be applied

each time policies are updated, and new test cases must be generated to cover any updates to the policies. The main challenge in defining test cases is to make sure that the cases cover all nuances of the policies. An efficient approach to policy analysis is by using systematic verification. Verifying the conformance of access control policies is a non-trivial and critical task. One important aspect of such verification is to formally check different aspects such as safety, correctness, liveness, inconsistency, and incompleteness of the policies.

Analysis of security policies has received attention from researchers, many papers were proposed that address different topics such as verifying safety, inconsistencies, irrelevancy, incompleteness, redundancies, etc. [61, 45, 94, 67, 55, 95] . However, policy verification has not been a focus in our research. In this section, we provide some verification mechanisms in the context of conflicts of interest mitigation policies. We focus on three main topics: correctness, liveness, and safety of policies. We focus on verification in the context of specified policies. Conflicts of interest mitigation policies are specified using algebraic expressions. Policies are enforced automatically using a policy enforcement engine which is provided in appendix E.

6.3.1 Verification of Correctness

Syntax Verification

Syntactic verification of conflicts of interest mitigation algebraic expressions is the first step required to validate correctness of the expressions. This can be achieved by syntactic analysis on the expressions through a syntactic verifier, which receives an expression, parsing it and representing the expression in a hierarchical format. The syntactic verifier must be able to reject an expression should its syntax be incorrect. Validation should be done on the operators, ensuring that only allowed operators are used, then validating correct usage of operators, e.g. to validate that binary operators are used between two operands and that unary operators and used with a single operand.

The syntactic verifier should also classify the operands into their correct categories, for example keyword, role, user, manager of user, workflow step, and operation variable. Any operand that cannot be classified to one of these categories should cause the whole expression to be rejected.

Semantic Verification

Conflicts of interest mitigation policies are specified by algebraic expressions, which are then automatically enforced by the enforcement engine. A conflict of interest mitigation algebraic expression is considered semantically correct if it does its intended functionality at enforcement stage, i.e. meets the requirements specified in the policy. Semantic verification of policy expressions is not a trivial task, especially when validating the end functionality of the policy. However, there are several semantics that can be verified to ensure a level of correctness of the policy. We give some examples of such semantics hereunder.

- Verifying that operands of category roles specified in the expression exist in the access control system. This can be done by taking all the operands that the parser classified as roles and ensuring that each of them is a role existing in the access control system, the role has already permissions assigned to it, and the role is assigned to at least one user.

- Verifying that a role specified in expression more than once that evaluates to multiple users is assigned to at least the number of required users in the access control system. For example, a role expressed twice with operator $\otimes$, evaluates to two different users, e.g. $Clerk \otimes Clerk$, the expression requires two different users assigned to the *Clerk* role. There should be at least two users assigned to the *Clerk* role in the access control system.

- If a user's line manager specified in expression, verify that the user has a line manager.

- Verify that a workflow step specified in an expression exists in the workflow.

- Verify that a protected operation variable specified in an expression exists in the protected operation.

- Verify that parameterization variables exist in the protected operation.

- Verify that a user associated to a workflow step in the algebraic expression, has a permission that enables him to perform the specified workflow step.

6.3.2 Verification of Safety

Safety means that the conflict of interest mitigation policy expression satisfies the specified safety requirements. Implicit in this description of safety is that

there is no violation of the constraints specified in the safety requirements and it is assured that the expression will eventually be in the desired situation after it took actions in compliance with the policy specification [45]. In this subsection we suggest some mechanisms to help policy authors ensure safety of their specified policies.

Verifying that a SoD policy requires multiple users in all situations

This can be achieved by developing a SoD verifier that accepts the SoD level as input, which is a number representing the minimum number of users that must participate in a workflow according to the SoD policy. The verifier then recursively substitutes the operands with all their possible user values in the expression. For example, substitute a role with all users assigned to it, substitute the ALL keyword with all users in the system, etc. The verifier ensures that with all different combinations of substituted values, the minimum number of users that satisfy the expression must be at least equal to the SoD level. The SoD verifier can help policy authors to ensure a certain level of safety of their expressions, however, testing all possibilities of expression evaluation can be an expensive operation in terms of computation cost.

List of all users who could be part of a workflow according to the expression

Safety is the fundamental property of a conflict of interest mitigation policy, which ensures that the policy will not result in the leakage of permissions to unauthorized users. Thus, a policy is said to be safe if no privilege can be escalated to unauthorized or unintended users [46]. Policy authors might need to review the list of all users who can participate in a workflow according to the conflict of interest mitigation specified policy. This can give indications on policy safety if they find unintended users in the list. This can be achieved by returning a list of all users in the access control system that can satisfy the algebraic expression in all possible combinations, which the expression can evaluate to.

6.3.3 Verification of Liveness

Liveness means that a policy can be satisfied given the access control elements, relations, and context policies defined on the level of the elements and relations. And that, there is no deadlock that can cause the workflow to wait forever for user actions to satisfy the expression due to impossible satisfaction of the policies.

To give an example of a policy expression that causes a workflow deadlock, consider a step in the workflow requiring a bank branch manager role, the bank branch manager role has a context policy that restricts the member access to role activation between 9:00 AM till 5:00 PM. Consider that the permission to perform the desired workflow action restricts activation to between 6:00 PM and 10:00 PM, this means that the branch manager will never be able to perform the required action in the workflow, and therefore, a workflow instance will remain unaccomplished forever.

Deadlocks might also happen if the algebraic expression can never be satisfied given the elements, relations, context policies, parameterization, and protected operation variable values. Therefore, an expression liveness verifier is needed to ensure expression satisfiability and that the user permissions and context policies allow them to perform the required actions. Semantic verification on the policies can give indications about satisfiability of expressions, however, it does not cover all different aspects of expression satisfiability such as protected operation variable values and context policies of elements and relations. This can be achieved by automatic generation of test cases to test liveness of the policy expression, which takes into account the different time interval workflow steps can be executed, different values of protected operation variables, and different possibilities for workflow steps order. The test cases should also cover different combinations of users that can satisfy the expression.

Chapter 7

Conclusion and Future Work

The following sections summarize the contributions of this dissertation and discuss some future research directions that can be further studied.

7.1 Summary

In this research work, we developed two main extensions to RBAC, which are delegation of duties and mitigation of conflicts of interest. Furthermore, a new form of role-based access control was also developed that facilitates the definition of more sophisticated context policies.

7.1.1 Authority Delegation

The central contribution of this dissertation is a new roles and permissions delegation model for role-based access control: the organizational supervised role delegation model (OSDM). This model provides a new means for controlling and authorizing delegation based on the organizational hierarchy. The development of the OSDM model was motivated by surveying some organizations and verifying their delegation and role assignment mechanisms in place. The survey has concluded that such actions are usually approved by managers according to lines of authority within the organization.

The work starts by remedying a limitation of existing RBAC models, which is the inability of projecting lines of authorities within organizations using the

roles hierarchy. This is due to the fact that role hierarchies are designated at simplifying the management of access rights through inheritance of common access rights by different roles. This approach does not take the organization structure into account. Therefore, we started by providing an extension to the underlying access control model, which is the user hierarchy. The user hierarchy feature projects the lines of authorities within organizations. This is achieved through defining the relations between users hierarchically, focusing mainly on the superior-subordinates relation. This enables implementing authority relations among different users by modeling the hierarchy using a graph data structure. The user hierarchy helps in finding users who need to approve delegations and revocations according to the policies and lines of authority in the organization. In existing delegation models, delegation is authorized by using a delegation relation that defines who can delegate a given role. We have explained disadvantages of this approach that could deter organizations from using delegation models based on such relations.

The delegation request can be initiated by different parties, such as: human resources, the delegator, the delegatee, or the line manager of the delegator. Once the request is initiated, the delegation request is sent for approval according to the approval matrix, usually involving the line managers of the delegator and the delegatee. The delegation operation is executed when the necessary approvals are obtained. OSDM supports both role and permission delegations, as well as flat and hierarchical role structures.

Revocation of delegation in OSDM takes similar steps to delegation. Firstly, a request for revocation is to be initiated by a user such as: human resources, the delegator, the delegatee, or the line manager of the delegator. Afterwards, the revocation request needs to be approved according to the approvals matrix before the revocation operation is executed.

7.1.2 Conflicts of Interest Policies

In today's business and corporate environments, wide varieties of security policies are implemented to ensure data integrity and prevent unauthorized activities as well as preventing improper usage of authority. Current authorization models, however, are not expressive enough to support definition of such requirements. Existing work focuses on one type of conflicts of interest policies, which is separation of duties (SoD), and therefore, leaves out a wide range of conflicts of interest policies uncovered. Furthermore, existing SoD models suffer from several limitations, mainly breaches related to missing order of task workflow steps, specification of which task steps to be separated, and ambiguities related to interpreting specified polices at design phase during the enforcement phase.

We proposed a novel approach to conflicts of interest policy specification and enforcement. We extended the algebra proposed in [56] to support parameterization of algebraic expressions, and the usage of task variables in the algebraic expressions. This offers better flexibility for defining more precise and fine-grained policies. Furthermore, we addressed ambiguity issues of the algebra, by offering a possibility to associate task steps in expressions definitions, in which task steps can be associated with variables in operand terms. Moreover, we provided a new approach to control the task workflow execution, and to make sure users are involved at predefined order of task workflow steps. An expression evaluation engine is also provided in Appendix 5, which enforces specified algebraic expressions and task order steps. We used a remittance payment business process example for demonstration of the work, in which we showed how high-level policy expressions are specified and how they are enforced.

An important aspect of security policies is their integrity and containment. An important challenge that arises when designing a new model for addressing conflicts of interest is the heterogeneous nature of operations at organizations, which incurs different requirements for each organization. One advantage of our model is that designed policies are enforced automatically by the expressions evaluation engine. This approach leaves no possibility for miss-interpretations of the specified expressions during enforcement. Therefore, our model standardizes both specification and implementation of conflicts of interest policies. Furthermore, the expression algebra we proposed provides a wider range of variables that enables specification of broader SoD policies than any other SoD model in the literature. In addition, it enables specification of non-SoD conflicts of interest policies.

7.1.3 The Core ROAC Model

Specification and enforcement of conflicts of interest policies requires a great expressiveness power of the underlying access control model. Existing RBAC models suffer from expressiveness issues.

The way permissions are currently described in RBAC suggests that every role is assigned a set of unique permissions. Two users assigned the same role have identical access privileges. Users, roles, permissions, and their relations in RBAC are considered as simple entities, which cannot express any kind of different granularity of the same element or relation. Furthermore, authorization decisions in RBAC are based on the user's association to permissions through roles. This kind of authorization does not support multi-factor decisions (for example, decisions dependent on physical location or temporal context). RBAC

role assignments tend to be based upon more static organizational positions, presenting challenges in various access control environments where dynamic access control decisions are required. Trying to implement these kinds of access control decisions would require the creation of numerous roles that are ad hoc and limited in membership, leading to what is often termed “role explosion” [44], this yields in millions of extra roles and permissions in the access control system. Parameterized RBAC is a notable initiative to address the lack of expressiveness of standard RBAC, but it is still not sufficient to express many authorization requirements. Context policies are an example of policies that cannot be expressed using parameterized RBAC. The ROAC model is proposed to provide a solid basis for expressing sophisticated authorization policies and to serve as the base model for expressing and enforcing conflicts of interest policies.

The core ROAC model presented the fundamental elements of the model and their relations. The ROAC model is empowered with object-oriented concepts. Mainly, providing attributes and behavior to the model elements and behavior. The ROAC model integrates parameters to permissions. Parameterization enables expressing policies on the level of the user-permission relation. A relation that does not exist explicitly in RBAC. The main advantage of our approach to parameterization is that it enables wider coverage on parameters that can be used, it enables both continuous and discrete value parameters. Moreover, permission validators, which are behavior in permissions, enable multi-factor authorization decisions. Context policies can be added in any element or relation of the model. The ROAC model is context aware, which supports the following levels on context polices, they are ordered below top-down from general to specific:

1. Role
2. Permission
3. Role-permission association
4. User
5. User-role association
6. Parameters (user-permission relation)

Any policy that can be defined in a general level can be defined in a more specific level. Furthermore, policies must be defined at the most general possible level to avoid complexity and reduce the number of policies.

7.1.4 The Hierarchical ROAC Model

In standard RBAC, role hierarchies support multiple inheritance; meaning that a role can inherit permissions from multiple roles. The concept of role hierarchies in standard RBAC has two main properties; firstly, the possibility to derive roles from multiple roles, and secondly, the concept provides a uniform treatment of user-role and role-role relations. Users can be included in the role hierarchy, using the same relation to denote the user assignment to roles. Several enhancements were suggested over the standard RBAC hierarchical model. However, the generalization concept in existing RBAC models does not reflect real organizational hierarchies. Existing role hierarchy concepts consider an *is-a* partial order on hierarchical roles. This means that sub-roles inherit all permissions of their super-roles. We believe that this approach cannot satisfy organizations needs. We can identify three different types of role hierarchies in organizations: the superior-subordinate, the senior-junior and the special-general hierarchy. These relations need to be treated differently when defining role hierarchies. In the special-general hierarchy, specialized roles might or might not inherit all permissions of the general roles. In the senior-junior hierarchy, the senior role usually inherits all permissions of the junior role. In the superior-subordinate hierarchy, the role of superiors is to supervise their subordinates. Consequently, they do not do tasks that are done by their subordinates. Therefore, superiors do not require the permissions of the roles of his subordinates.

We provided a novel approach to tackle hierarchies in the context of the Role-Oriented Access Control Model (ROAC). Our approach to role hierarchy implements selective hierarchies, which enable a descendant role to inherit a subset of the ancestor's role permissions. This is achieved by defining permission exclusion sets over the role-role relations. The main idea behind selective role hierarchy is to support the supervision relation between superiors and their subordinates in organizations.

7.2 Future Work

This research proposed extensions to RBAC and an underlying access control model to enable enforcement of the extensions. While this work significantly improves role-based access, there are still areas outside the core focus which represent opportunities for extending the work presented in this dissertation.

7.2.1 Standardization of Policy Specification

We used object-oriented modeling for policies specification and enforcement. However, despite robustness of this approach, policy designers might not be familiar with object-oriented concepts. Therefore, we see a potential enhancement for the model by introducing a mathematical expression language to specify access control policy requirements, and then, to develop a policy enforcement engine for automatically enforcing the specified policies. This is similar to what we did in conflicts of interest policies. However, a standard expression language is needed to cover all other policies such as core, hierarchical, delegation and conflict of interest policies.

7.2.2 Centralized Access Control System

Access control is enforced in applications separately. Then organizations collect access rights from all applications and consolidate them in one centralized access control system. This causes many issues, such as ambiguities, conflicts, etc. To address this issue, more research is needed to integrate access control specification and enforcement in programming languages, e.g., using annotations. Then, APIs need to be developed to communicate with the centralized ROAC access control system.

This can also enable unit testing of application access control. Security testing of applications is usually done prior to software release. This usually causes delays and requires more development efforts of software applications.

7.2.3 Artificial Intelligence

There exist two approaches to policy definition, rule-based and model-based. Rule-based policies represent policies that can be expressed using rules, which are usually based on conditions. However, model-based policies are more sophisticated, which usually rely on machine learning to model behavior of users and spot anomalies. In nowadays businesses, the rule-based approach alone is not enough to cover all access risks. Moreover, it is very costly to organizations, as the policy designer must identify all possible risk areas and model them into rules. This is only possible for the known-known policies, which are risks known to the organization. However, this leaves out a wide range of risks uncovered which are the known-unknowns, and the unknown-unknowns. Artificial intelligence can detect anomalous behavior not known to the policy designer. Therefore, we propose a future line of research to standardize model-based policies in the ROAC model.

Appendix A

A Motivating Example

A.1 Introduction

Access control is integrated into almost all organizations' IT environments to regulate access to their resources. Organizations often use an access control model to implement access control policies that are defined by the organization. The complexity of the policies can differ from one organization to another. However, we can comfortably say that access control policies are getting more and more complex.

Most existing RBAC models fail to satisfy most organizations requirements in terms of access control policies. To show the motivation and demonstrate our work, we use this example access control system that uses access control policies from the financial industry.

The motivating example policies are defined in the context of RBAC, which means that policies related to RBAC elements, and their relations are firstly defined, then a set of context policies are defined on top of the RBAC elements and their relations. The example represents a subset of a banking access control system, which only allows authorized users to access specific banking services.

The context policies on top of the elements and relations covers four different types of policies. The core policies are defined in section 3; represent policies that regulate access to operations that access protected objects. Then, section 4 provides policies related to the organization structure, namely hierarchy of roles and users. Section 5 provides requirements and policies for delegation of access rights within the organization. Finally, section 6 provides policies required to

mitigate conflict of interest within the organization.

A.2 Policy Elements and Relations

A.2.1 Elements

Users

The users in this banking example are of two different types; bank employees and bank clients. Table A.1 shows some sample users, and their attributes:

Table A.1: Sample users and their attributes

User	Name	Type	ID Expiry
User1	Charlie	Bank Client	01/01/2025
User2	Bob	Bank Employee	N/A
User3	Alice	Bank Employee	N/A
User4	Dave	Bank Employee	N/A
User5	Marc	Bank Employee	N/A

Roles

Permissions required for a job function are mapped to a role. Table A.2 shows some sample roles and their target assignees:

Table A.2: Sample roles and their target assignees

Role	Target Assignee
BankClient	Bank clients
Teller	Bank employees
SecuritiesClerk	Bank employees
BranchManager	Bank employees

Permissions

Permissions are mapped to operations, the mapping relationship can have different cardinalities. In this example, we assume a one-to-one mapping to the sample protected operations. The sample permissions are shown in Table A.3.

Table A.3: Sample permissions mapped to the operations they protect

Permission	Protected Operation
CheckBalance	CheckBalance(AccountNumber)
TransferFunds	WireTransfer(Type, OriginatingAccount, BeneficiaryAccount, Amount, Currency)
WithdrawCash	CashWithdrawal (Account, Amount)
OnboardNewClient	ClientOnboarding
ApproveTransaction	TransactionApproval (Account, Amount)

Protected Operations

Any access control system aims at protecting a set of operations in an organization. In this example, we aim at protecting the following operations:

CheckBalance:
This operation is available for both bank employees and bank clients. The *CheckBalance* operation is used to check the account balance of a given account.

WireTransfer:
This operation is available for both bank employees and bank clients. The *WireTransfer* operation allows a user to transfer funds from an account to another. Wire transfers have different types, for the sake of simplicity, we consider three types: domestic, cross-border, and securities.

CashWithdrawal:
This operation is also available for bank employees and bank clients. It enables a user to withdraw cash either from an ATM machine or over the counter.

ClientOnboarding:
This operation is only available for bank employees. It enables a user to onboard a new client to the bank.

TransactionApproval:
This operation is only available for bank employees. It enables a manager user to approve a transaction done by any of his subordinates.

A.2.2 Relations

User-Role Assignment

Users are assigned to roles according to their job functions. Table A.4 shows the user-role assignment of our example users and roles.

Table A.4: Sample user-role assignments

User	Role
User1	BankClient
User2	Teller
User3	Teller
User4	SecuritiesClerk
User5	BranchManager

Role-Permission Assignment

Permissions are assigned to roles according to job functions required permissions. Table A.5 shows the role-permission assignment of our example roles and permissions.

Table A.5: Sample role-permission assignments

Role	Permission
BankClient	CheckBalance
	TransferFunds
	DepositCash
SecuritiesClerk	CheckBalance
	TransferFunds
Teller	CheckBalance
	TransferFunds
	DepositCash
	OnBoardNewClient
BranchManager	CheckBalance
	OnboardNewClient
	ApproveTransaction

A.3 Core Model Policies

A.3.1 Users

Policy 1. *If a bank employee is also a bank client, then they are considered two different users.*

Policy 2. *Each user gets automatically restrained from access if he is on leave e.g. holiday or sickness.*

Policy 3. *According to anti-money laundering regulations, banks need to maintain copies of valid identity cards of their clients. Therefore, a bank client user has an expiry date that is the expiry date of his ID document. Expired users cannot perform any protected operation.*

A.3.2 Roles

Policy 4. *Roles have a remote activation policy, which means some roles can be activated only from bank premises and some roles can be activated from anywhere.*

Table A.6 shows activation policies of our example roles:

Table A.6: Sample role policies

Role	**Activation**
BankClient	Anywhere
Teller	Branch
SecuritiesClerk	Head Office

A.3.3 Permissions

Permission policies are driven from the protected operations policies given below.

CheckBalance

Policy 5. *If the user is a bank employee, working in a branch, and has permission* CheckBlanace*, then he can check the account balance of any client in his branch only.*

Policy 6. *If the user is a bank employee, working in the bank head office, and has permission* CheckBlanace*, then he can check account balance of any client in the bank, given that the account is of an account type that the user is allowed for e.g. loan, current, and savings.*

Policy 7. *If the user is a bank client, then he can check account balance of his account or an account for which he is a guardian.*

WireTransfer

Policy 8. *If the user is a bank employee, then:*

- *The user is allowed for a predefined set of wire transfer types.*

- *The user has a threshold for the maximum limit of a wire transfer he can perform.*

Policy 9. *If the user is a bank client, then he can do any wire transfer type. The user must have a threshold limit for the maximum amount of the wire transfer.*

CashWithdrawal

Policy 10. *If the user is a bank employee having permission to withdraw cash, then:*

- *The user must be a branch bank employee. Head office users cannot perform cash withdrawal.*

- *The user can only withdraw cash from an account in his branch.*

- *The cash withdrawal operation can only be performed from the branch. The user cannot perform this operation if he is working from home.*

- *Each user must be assigned a threshold limit for the maximum amount he can withdraw.*

Policy 11. *If the user is a bank client, then:*

- *The user can withdraw from anywhere i.e. from branch, or any ATM.*

- *The user has a daily limit, a weekly limit, and a limit per withdrawal.*

TransactionApproval

Policy 12. *This operation is only available for bank employees:*

- *The user has a threshold for the maximum limit of the transaction he can approve.*

A.3.4 User-Role

Policy 13. *A bank policy specifies that a user might hold a role for a temporary period of time. In this example, let's consider that* User3 *to role* Teller *ends on his contract end date. The expiry policies for our sample user-role assignments are shown in Table A.7.*

Table A.7: Sample user-role policies

User-Role	**Expiry**
User1-BankClient	N/A
User2-Teller	N/A
User3-Teller	01/01/2024
User4-SecuritiesClerk	N/A
User5-BranchManager	N/A

A.3.5 Role-Permission

Policy 14. *A policy that is often required for some permissions is the activation time interval according to the assigned role. Table A.8 shows the sample role-permission time intervals.*

Table A.8: Sample role-permission policies

Role-Permission	**Activation Interval**
BankClient-TransferFunds	N/A
Teller-TransferFunds	08:00 to 16:00
SecuritiesClerk-TransferFunds	09:00 to 17:00

A.3.6 User-Permission

Policy 15. *There is no direct assignment of users to permissions in RBAC. This relation is achieved through roles assignment to both users and permissions. However, policies are often needed to control users activation of permissions. Table A.9 shows sample policies on the roles assignment to the* TransferFunds *permission:*

Table A.9: Sample user-permission policies

UserPermission	**Policies**	
User1-BankClient-TransferFunds	From Belgium	Amount <= 50,000
	Outside Belgium	Amount <= 10,000
	Fund Types: {Any}	
	Branch: N/A	
User2-Teller-TransferFunds	From Belgium	Amount <= 50,000
	Fund Types: {CrossBorder, Domestic}	
	Branch: Brussels	
User3-Teller-TransferFunds	From Belgium	Amount <= 100,000
	Fund Types: {Domestic}	
	Branch: Brussels	
User4-Secur.Clerk-TransferFunds	From Belgium	Amount <= 100,000
	Fund Types: {Securities}	
	Branch: HeadOffice	
User5-BranchMngr-ApproveTrans.	From Belgium	Amount <= 500,000
	Branch: Brussels	

A.4 Hierarchical Policies

A.4.1 Role Hierarchy

In real world banking, roles are usually assigned a relatively high number of permissions. To reduce the administration burden, two policies are required to reduce the number of direct assignment of permissions to roles representing hierarchical positions.

Policy 16. *The* Teller *role is considered superior to the* SecuritiesClerk *role. It inherits all its permissions. This policy requires elimination of direct assignment of the permissions included in* SecuritiesClerk*. Therefore, the* Teller *gets these permissions indirectly through the role hierarchy.*

Policy 17. *The* BranchManager *role is considered superior to all roles in the branch including the* Teller *and* SecuritiesClerk *roles. It inherits all role permissions excluding a set of permissions that the branch manager is not allowed to possess. This policy requires elimination of direct assignment of the permissions included in roles below the* BranchManager *role.*

The *Teller* role already inherits all permissions from the *SecuiritiesClerk* role. Therefore, there is no need to define hierarchy between *BranchManager* and *SecuirtiesClerk*, since it is included implicitly. A hierarchy is needed, therefore, between the *BranchManager* and the *Teller* roles. The set of permissions that needs to be excluded are: *TransferFunds* and *WithdrawCash*. The *BranchManager* role has one permission directly assigned to it; *ApproveTransaction*, which neither *Teller* nor *SecuritiesClerk* possess.

A.4.2 User Hierarchy

The user hierarchy is important to know who is the manager of a given user. For example, if a manager needs to approve a transaction from one of his subordinates, we need to know the manager of the subordinate so we can redirect the approval request to him.

Policy 18. *This policy requires that the access control system be aware of who is managing whom in the branch. The policy simply requires awareness of the simple user hierarchy depicted in Fig. A.1.*

Note that the user hierarchy does not necessarily project the role hierarchy. For example, the *Teller* role is inheriting the *SecuritiesClerk* role, however, the teller *User2* does not manage the securities clerk *User4*.

A.5 Delegation Policies

Some of the positions in our banking branch example are not redundant. This means that absence of some resources could be a showstopper for the branch operations. Therefore, a delegation mechanism is needed in order to assign access rights of an unavailable resource to another resource. The following policies control the delegation of user access rights:

Policy 19. *If* User2*,* User3*, or* User4 *is absent, then his role can be delegated to any user in the branch. In this case, the delegated user must get the same parameter values assigned to* User2*. Before delegation takes effect, the delegation must be approved by the branch manager and any other user in the branch.*

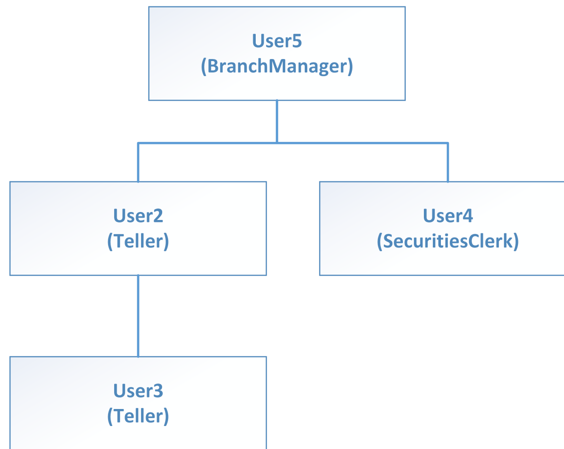


Figure A.1: Bank branch user hierarchy

Policy 20. *If* User5 *is absent, then his role can be delegated to another user, provided that the delegation gets approved by the HR manager of the bank and by the bank branch operations director of the bank head office. In this case, the delegated user must get the same parameter values assigned to* User5*.*

A.6 Conflict of Interest Policies

In this section, we provide some policies that aim at preventing insider fraud in the bank branch.

Policy 21. *No user in the bank branch is allowed to execute any transaction involving his own account, as an ordering customer or a beneficiary customer.*

Policy 22. *Transactions must be subject to the* 2X-Eye *principle. for any* X *above* 1*, the users representing* 2 *to* X *must be the last users to take an action on the transaction in the workflow.*

The *2X-Eye* principle means that multiple users must participate in the transaction workflow. *X* represents the number of users who must participate in the transaction. For example, if *X=2*, then the 4-eyes concept is applied, which means 2 users. If *X=3*, then it is the 6-eye concept meaning 3 users.

In case of 4-eyes, the second user must be the last user to take an action on the transaction. In case of 6-eyes, no other users can take any action on the

transaction after the second and the third users. The idea is to prevent that the transaction gets modified by any other users after it gets approved by the extra users. Users actions are allowed after the first 2-eyes user. For example, if 4-eyes is applicable on a given transaction, then *User1* can create the transaction, *User2* modifies the payment instructions, then transaction gets approved by *User3* and *User4*.

Policy 23. *Any over-the-counter transaction that exceeds* 50K EUR *must get authorized by a user other than the user who created the transaction. The authorizor must be an* n_{th} *line manager of the user who created the transaction. No authorization is required for transactions below* 50K EUR*.*

Policy 24. *Any over-the-counter transaction that exceeds* 100K EUR *must get authorized by branch manager. The authorizor must be assigned the role* BankBranchManager

Policy 25. *Any over-the-counter transaction that exceeds* 500K EUR *must get two different approvals by two users other than the user who created the transaction. The first approval is called* Verifier*, which must be a user who is assigned the role* BankBranchManager*. The second approval is called* Authorizer*, which must be a user in the bank Head Office, and assigned to any of the following roles:*HOOperations*,* RegionalOperationsManager*, or* OperationsDirector*.*

Appendix B

ROAC Formal Model

In this appendix we provide the core ROAC model, which is the flat ROAC model, it does not include any hierarchies on the model elements. We split the definition of core ROAC model into a family of three models. The models are given one by one based on the increasing security functionality of the models.

B.1 The Core ROAC Model

The models are denoted ROAC0, ROAC1 and ROAC2. ROAC0 is a basic model that is similar to the RBAC standard model but modeled in the object-oriented paradigm. It serves as an enabler of the next two models. ROAC1 is the parameterized version of ROAC0, it adds permission parameters and provides an architecture for managing the parameters. ROAC2 is a context aware version of ROAC1, which adds context to the model elements and relations. Each version of the model inherits all characteristics of the previous model. Fig. B.1 shows the relationship among the three ROAC models.

B.1.1 The Basic ROAC Model (ROAC0)

In this basic version of the model, we show the different elements and relations and their modeling in object-oriented. We also give an administrative model which we call *AuthorizationPolicy* that manages the relations and provides the necessary functionalities to determine the authorization decision.

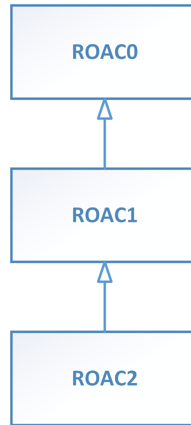


Figure B.1: Relationship among the core ROAC models

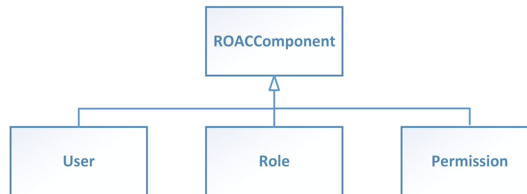


Figure B.2: Relationship among the elements of ROAC0

Policy Elements of ROAC0

ROAC0 has three main elements: users, roles, and permissions. All of these three elements are descendants from the abstract ROAC element. Fig. B.2 shows the generalization relation among the ROAC elements.

The abstract ROAC element is given by the following Python definition:

```
@dataclass(frozen=True, eq=False)
class ROACElement:
  pass
```

The following rules apply on the abstract element:

- Members from a given sub-element (e.g. set of users) are defined as instances from the sub-element definition (e.g. User).

- Decedents can be defined from each sub-element definition by generalizing the sub-element definition.

- The abstract element can contain attributes or behavior that is common among all instances of its sub-elements or their specializations if any.

Definition 1. *Users:*

> *A user in the ROAC model is a principal that is uniquely identifiable in a way making it accountable for its actions. The identity is also used in the authentication process and other security aspects such as non-repudiation. Unique identity can be achieved by using unique usernames. Users can be either humans or computer applications seeking access to information systems. The user blue-print can have attributes and implementation of behavior.*

The user element is defined as follows in Python:

```
@dataclass(frozen=True, eq=False)
class User(ROACElement):
  name: str
  dateOfBirth: date
  # Other user attributes
```

An instance from *User* or its descendants must be defined for each user in the system.

Definition 2. *Roles:*

> *A role is defined as a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role [9]. Role is the centric element in RBAC relations, which represents a group of access rights (permissions) required often by a group of users who play a similar role in the organization.*

The role element is defined as follows in Python:

```
@dataclass(frozen=True, eq=False)
class Role(ROACElement):
  pass
```

Each role must have a unique identity across all its instances.

Definition 3. *Permissions:*

> *A permission is an access right to execute one or more protected operations that give access to protected objects.*

The permission element is defined as follows in Python:

```
@dataclass(frozen=True, eq=False)
class Permission(ROACElement):
  pass
```

Each permission must have a unique identity across all its instances.

Definition 4. *Protected Operations:*

> *A protected operation in the context of the ROAC model is any operation that represents a programmed business logic, which users need to be authorized to execute. Operations can also access protected objects that also require that a user is authorized prior to accessing.*

Relations of ROAC0

ROAC0 has two relations: user-role assignment and role-permission assignment. Both relations are descendants from the abstract *ROACRelation*. In the ROAC model, both relations represent a binary association between instances of two different elements. Therefore, we use the word *association* rather than *assignment* to denote the definition of the relations. The assignments are mappings between sets of elements, which we tackle in the next subsection. Fig. B.3 shows the generalization relation on the ROAC relations.

The abstract ROAC Relation is given by the following Python definition:

```
@dataclass(frozen=True)
class ROACRelation:
  pass
```

The following rules apply on the abstract relation:

- Members from a given relation (e.g. set of user-role associations) are defined as instances from the relation definition (e.g. *UserRoleAssociation*).

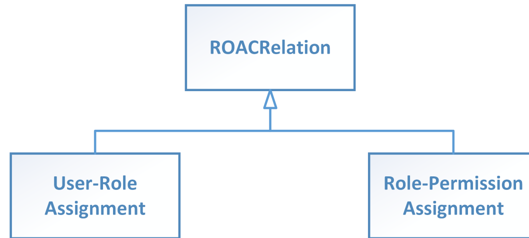


Figure B.3: Relationship among relations of ROAC0

- Decedents can be defined from each relation definition by generalizing the relation definition.
- The abstract relation can contain attributes or behavior that is common among all instances of relations or their specializations if any.

Definition 5. *User-Role Assignment:*

> *The user-role assignment represents a set of user-role associations, which model how roles are assigned to users. A user-role association is a link between an instance of a user to a role instance. The assignment relation is many-to-many. A user instance can be assigned multiple role instances and the same role instance can be assigned to multiple user instances.*

The user-role association relation is defined as follows in Python:

```
@dataclass(frozen=True)
class UserRoleAssociation(ROACRelation):
  user: User
  role: Role
```

Definition 6. *Role-Permission Assignment:*

> *The role-permission assignment is a many-to-many mapping of permissions to roles. It represents a set of binary role-permission associations among role instances and permission instances.*

The role-permission association relation is defined as follows in Python:

```
@dataclass(frozen=True)
class RolePermissionAssociation(ROACRelation):
  role: Role
  permission: Permission
```

The Administrative Model (*AuthorizationPolicy*)

Relations assignments are two sets of user-role associations and role-permission associations. The *AuthorizationPolicy* has two dictionary data structures to contain the individual relations. Both relations are modeled using a dictionary of dictionary.

The definition of class *AuthorizationPolicy* and the relations data-structures in Python are shown below:

```
@dataclass_with_check(frozen=True)
class AuthorizationPolicy:
  userRoleAssociations: frozendict[User, frozendict[Role, UserRoleAssociation]]
  rolePermissionAssociations: frozendict[Role, frozendict[Permission,
    ↪ RolePermissionAssociation]]
```

The *dataclass_with_check* is a decorator, which we defined to allow for adding statements to the end of the constructor without having to define the complete constructor explicitly.

By creating an instance from *AuthorizationPolicy*, the following constraints on the policy elements and their relations apply:

Constraint 1. *If one element is associated to multiple elements, then either the same instance of the elements can be used in the multiple associations, or a new instance can be created and associated to each of the multiple elements. In this case, we consider different instances equal if their identifier is the same.*

A. *If two user instances have the same* identity*, then they are considered equal. Therefore, it is not allowed to have instances with the same* identity *but different other attributes.*

B. *If two role instances have the same* identity*, then they are considered equal. Therefore, it is not allowed to have instances with the same* identity *but different other attributes.*

C. *If two permission instances have the same* identity*, then they are considered equal. Therefore, it is not allowed to have instances with the same* identity *but different other attributes.*

In our Python implementation, we used object identities rather than definition an identity attribute for the sake of simplicity. We pass *eq=False* as an additional keyword argument to the *@dataclass* decorator of *Permission*, *Role*, and *User*. This causes two instances to be considered equal if and only if they are the same object.

Constraint 2. *The key in* userRoleAssociations *is the* User *attribute of the* UserRoleAssociation *object. The key of the sub-dictionary in* userRoleAssociations *is the* Role *attribute of the* UserRoleAssociation *object.*

Constraint 3. *The key in* rolePermissionAssociations *is the* Role *attribute of the* rolePermissionAssociations *object. The key of the sub-dictionary in* rolePermissionAssociations *is the* Permission *attribute of the* RolePermissionAssociation *object.*

The following Python code is added to the constructor of *AuthorizationPolicy* in order to validate constraints: 2 and 3:

```
# Validate Constraint 2
for user in self.userRoleAssociations.keys():
  for role, userRole in self.userRoleAssociations[user].items():
      assert userRole.user == user and userRole.role == role

# Validate Constraint 3
for role in self.rolePermissionAssociations.keys():
  for permission, rolePermission in self.rolePermissionAssociations[role].items():
    assert rolePermission.role == role and rolePermission.permission == permission
```

The *AuthorizationPolicy* class defines a set of administrative methods, which are used to retrieve authorization information from the relations data structures:

User Roles:

The *rolesAssignedToUser* method returns a set of all roles assigned to a given user.

```
def rolesAssignedToUser(self, user: User) -> frozenset[Role]:
  return frozenset(self.userRoleAssociations[user])
```

Role Permissions:

The *permissionsOfRole* method returns a set containing all permissions assigned to a given role.

```
def permissionsOfRole(self, role: Role) -> frozenset[Permission]:
  return frozenset(self.rolePermissionAssociations[role])
```

User Permissions:

The *permissionsOfUser* method returns a set containing all permissions of all roles assigned to a given user. If multiple roles that contain overlapping permissions are assigned to the same user, then the permission appears only once in the set of user permissions.

```
def permissionsOfUser(self, user: User) -> frozenset[Permission]:
  return frozenset(r for rs in self.rolesAssignedToUser(user) for r in self.permissionsOfRole
    ↪ (rs))
```

Assigned Users:

The *getAllUsers* method returns a dictionary of all users that are assigned to roles.

```
def getAllUsers(self) -> frozenset[User]:
  return frozenset(r.user for rs in self.userRoleAssociations.values() for r in rs.values())
```

Assigned Roles:

The *getAllRoles* method returns a dictionary of all roles that have permissions assigned to them.

```
def getAllRoles(self) -> frozenset[Role]:
  return frozenset(r.role for rs in self.rolePermissionAssociations.values() for r in rs.values
    ↪ ())p
```

Assigned Permissions:

The *getAllPermissions* method returns a dictionary of all permissions assigned to roles.

```
def getAllPermissions(self) -> frozenset[Permission]:
  return frozenset(r.permission for rs in self.rolePermissionAssociations.values() for r in rs.
    ↪ values())
```

Authorization Decisions in ROAC0

The authorization decision in ROAC0 is calculated simply by verifying that the set of permissions of the user contains the required permission for executing the protected operation. The permissions set of a given user can be retrieved using the administrative method *permissionsOfUser* defined above.

B.1.2 The Parameterized ROAC Model (ROAC1)

ROAC1 is defined from ROAC0 by adding permission parameters. The parameterized ROAC model not only gives the possibility to define permission parameters, but also adds a business logic for validation of parameters. Validation of parameters specifies how parameters are matched and provides a mechanism for accessing external information that might be needed for the validation.

We start by defining permission parameters, then we define permission validators that are used to validate the parameters. Afterwards, we show updates on the impacted elements and relations, then we show the updated administrative model and authorization decision.

The ROAC1 model is depicted in Fig. B.4.

B.1.3 Parameters

Definition 7. *Parameters:*

> *A parameter in the ROAC model is a composite structural feature that is defined in permissions. Parameters are composed of an identity (e.g. object identity or name), and a data type, which is the data type of the parameter value. The parameter value is bound during the user-role assignment of a role that contains parameterized permissions.*

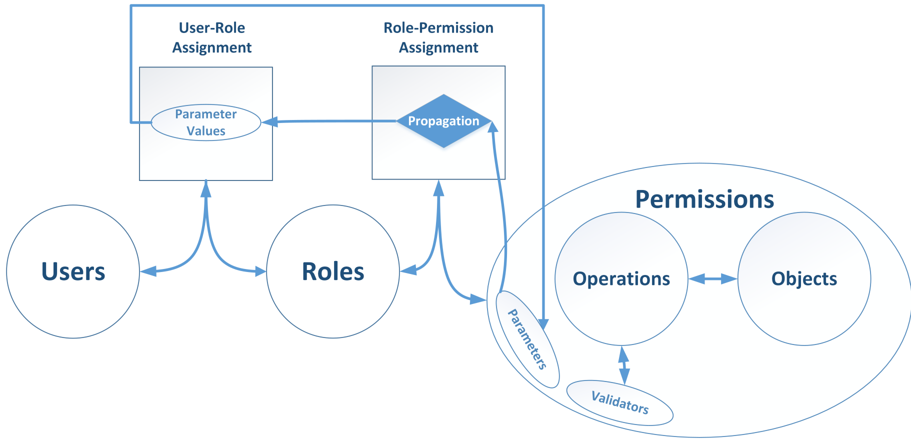


Figure B.4: The ROAC1 model

Parameters represent a policy on the level of user-permission assignment, a relation that does not exist directly in RBAC. Therefore, a parameter propagation mechanism is used to propagate parameters to the user-role assignment relation. Propagation of parameters is given in the next definition.

Definition 8. *Propagation of Parameters:*

> *Propagation of parameters has two different types: back propagation and forward propagation. Back propagation means that, for a given role, all parameters of all permissions assigned to that role are propagated back to the user-role association through assigned roles. And for a given user, who is assigned to one or more roles, values are bound for all parameters propagated in all roles assigned to that user. Forward propagation means that a permission can get the values bound to its parameters for a given user.*

Although parameters are defined in permissions, their values cannot be assigned in permission instances as attributes. Binding parameters in the user-role association enables each user to have a different binding of the same permission parameter.

As parameter bindings are set in the user-role assignment relation and the fact that the authorization decision is calculated locally in permission validators, permissions must have access to the user parameter bindings. For this reason, permissions have a combined association with users through roles, which enables

extracting parameter values for a given user from the user-role relation in a forward propagation.

The permission parameter is defined as follows in Python:

```
@dataclass(frozen=True)
class Parameter:
  name: str
  dataType: type
```

The *ParameterBinding* class is used for setting a value for a given parameter. The constructor of the *ParameterBinding* validates the following constraints before an instance is created:

Constraint 4. *Correctness of parameter bindings is given by the following sub-constraints:*

A. A parameter cannot be bound to a null value.

B. The value bound to a parameter must respect the data type of the parameter.

```
@dataclass_with_check(frozen=True)
class ParameterBinding:
  parameter: Parameter
  value: object

  def __check__(self):
    # Validate Constraint 4.A
    assert self.value is not None

    # Validate Constraint 4.B
    assert isinstance(self.value, self.parameter.dataType)
```

Permissions

The permission definition is impacted by parameterization. The following definitions state the changes to parameterized permissions:

Definition 9. *Parameterized Permissions:*

A parameterized permission *is any permission that has parameters. A parameterized permission is an instance from a parameterized permission*

definition, which must be defined as a descendant of the permission and must override the validator *method. Non-parameterized permissions can be defined as instances from* Permission *directly.*

Definition 10. *Validators:*

A validator is a special kind of behavior defined only in permissions for computing an authorization decision. The validator definition consists of a signature, a body, and a return value. The signature specifies the validator name which is always "validator", in addition to input arguments. The body of the validator contains business logic to determine the authorization decision. This is usually a result of asserting the parameter values against the operation values. The business logic can be a simple matching or a more complex logic that involves conversions, access to external data sources, etc. Validators always return Boolean values. True *if authorization is granted and* false *if denied.*

The updated Python definition of the parameterized permission is given below:

```
@dataclass_with_check(frozen=True, eq=False)
class Permission(ROACElement):
  parameters: frozenset[Parameter]

  def __check__(self):
    assert_no_duplicates(self.parameters, lambda p: p.name)

  def validator(self, *args, **kwargs) -> bool:
    return True
```

The *assert_no_duplicates* method verifies that there are no multiple parameters with the same name in the *parameters* set.

User-Role Association

The user-role association is impacted by parameterization, as parameters bindings need to be set at this step. The following constraints are validated prior to creating any user-role association:

Constraint 5. *If two parameters have the same name, they must also have the same data type.*

Constraint 6. *If multiple permissions that have overlapping parameters (with same name and data type) are assigned to a single role, then*

parameters with same name and data type are consolidated to one parameter in the user-role association.

The updated Python definition of the user-role association is shown below:

```
@dataclass_with_check(frozen=True)
class UserRoleAssociation(ROACRelation):
  user: User
  role: Role
  parameterBindings: frozenset[ParameterBinding]

  def ___check___(self):
    # Validate Constraint 6
    assert_no_duplicates(self.parameterBindings, lambda p: p.parameter)
```

The *assert_no_duplicates* method verifies that there are no multiple parameters with the same name in the *parameterBindings* set.

To satisfy constraint 6, *parameterBindings* are maintained in a set data-structure.

Administrative Model

The administrative model *AuthorizationPolicy* is also impacted by parameterization. New administrative methods are added for managing parameters.

The following constraint on parameters is validated in the constructor of the *AuthorizationPolicy* class:

Constraint 7. *Correctness of parameters are validated according to the following sub-constraints:*

A. When a role is assigned to a user, then all consolidated parameters propagated from permissions of that role must be included in the user-role assignment.

B. If multiple roles, which are assigned permissions with overlapping parameters, are assigned to the same user, then, overlapping parameters must have the same value binding.

```
userRoleAssocSet = frozenset(r for rs in self.userRoleAssociations.values() for r in rs.
    ↪ values())

for userRoleAssoc in userRoleAssocSet:
  # Valiadte Constraint 7.A
  paramsFromBindings = frozenset((binding.parameter for binding in userRoleAssoc.
    ↪ parameterBindings))
  assert self.parametersOfRolePermissions(userRoleAssoc.role).issubset(
    ↪ paramsFromBindings)

  # Valiadte Constraint 7.B
  userParamBindings = self.parameterBindingsOfUser(userRoleAssoc.user)
  for paramBinding in userRoleAssoc.parameterBindings:
    assert paramBinding.value == userParamBindings[paramBinding.parameter]
```

Two new administrative methods are added to the *AuthorizationPolicy* class:

Role Parameters:

The *parametersOfRolePermissions* returns a set containing all parameters of all permissions of a given role.

```
def parametersOfRolePermissions(self, role: Role) -> frozenset[Parameter]:
  return frozenset(r for rs in self.rolePermissionAssociations[role].values() for r in rs.
    ↪ permission.parameters)
```

User Parameter Bindings:

The *parameterBindingsOfUser* returns a set of all parameter bindings of a given user. This method is used in the permission validators to get the parameter values assigned to the user requesting authorization. Validators match the parameter bindings against the corresponding values from the protected operation.

```
def parameterBindingsOfUser(self, user: User) -> frozendict[Parameter, object]:
  return frozendict({r.parameter: r.value for rs in self.userRoleAssociations[user].values()
    ↪ for r in rs.parameterBindings})
```

Authorization Decision in ROAC1

Authorization decision is ROAC1 involves two steps. The first step is the authorization decision of ROAC0, which checks if a needed permission is present.

The second step is verifying the permission parameter bindings of the user. The authorization decision is determined by the permission validator. For non-parameterized permission, the validator always returns a positive decision. However, in parameterized permissions, the decision is determined based on more complex business logic that implements the required policies related to parameterization.

An Example Parameterized Permission

Consider the *TransferFunds* permission in our motivating example. The permission is parameterized by the maximum amount allowed for a wire transfer and the wire transfer type. The policy requires that the wire transfer protected operation gets authorized if the amount is less than or equal to the amount threshold assigned to the user, and if the wire transfer type is one of the wire transfer types assigned to the user. The wire transfer protected operation can accept wire transfers of different currencies. Therefore, the amount given in the wire transfer might not be ready to be matched directly to the amount parameter binding of the user. For example, if the base amount in the bank is *EUR*, then the amount threshold bindings for users are set against *EUR*. If the currency in the wire transfer is *YEN*, then we need first to find the equivalent amount against *EUR*, which is calculated by multiplying the exchange rate of *YEN/EUR* to the wire amount. This business logic must be implemented in the permission validator.

The permission validator gets the parameter bindings of the user from the *AuthorizationPolicy* class using the method *parameterBindingsOfUser*. The following Python code shows the definition of the *TransferFunds* permission:

```
@dataclass(frozen=True, eq=False)
class Treasury:
  exchangeRates = frozendict({"EUR": 1, "USD": 0.82, "YEN": 0.01})

  def exchangeRate(self, currency: str):
    return self.exchangeRates[currency]


@dataclass(frozen=True, eq=False)
class TransferFundsPermission(Permission):
  fundType = Parameter("FundType", frozenset[str])
  fundAmount = Parameter("Amount", float)

  def __init__(self):
    super().__init__(parameters=frozenset([self.fundType, self.fundAmount]))

  def validator(self, authPolicy: AuthorizationPolicy, user: User, wireType: str,
     ↪ wireAmount: float, wireCurrency: str):
    if self not in authPolicy.permissionsOfUser(user):
        return False

    userBindings = authPolicy.parameterBindingsOfUser(user)

    amtBaseCurr = wireAmount * Treasury.exchangeRate(wireCurrency)

    return amtBaseCurr <= userBindings[self.fundAmount] and wireType in userBindings
     ↪ [self.fundType]
```

B.1.4 The Context-Aware ROAC Model (ROAC2)

ROAC2 is the parameterized model ROAC1 plus context policies.

Context is defined in the dictionary as "words around other words that help determine the meaning" and "a time and setting in which an event happens" [2]. In access control, context represents the information taken into account while making the access control decisions, such information is also suitable for applications where access to resources is controlled by exploiting contexts of the resources in the policy [79]. For example, a policy might depend on contextual information such as current time, geo-location from which the request has been initiated, if a user is on holiday, etc.

In ROAC2, context can be defined in any element or relation. When context is defined in an element or a relation, then the same context is shared across all instances of the same element or the relation. Unlike permission parameters, which are considered as custom policies per user assignment to the parameterized

permission. To explain the difference, consider the *CashDeposit* permission of our motivating example, when the permission gets assigned to *User1* through the role *R1*, we can set the parameter value of *Amount* to *10,000*. However, we can set a different amount value when we assign the permission to *User2* e.g. *5000*. Per contra, if we define a context policy for *User*, e.g. a user must not be on holiday when he activates any permission, then the same policy is applied on all user instances.

Context must always be defined at the highest possible parent of an element or relation. In case context needs to be customized for specific instances of a given element or relation, then sub-elements or sub-relations needs to be defined using the inheritance feature of the ROAC model, then specific context is defined in the sub-element or sub-relation. Policies that are shared among all elements can be defined in the *ROACElement*, and similarly context policies that are shared among the two relations can be defined in the *ROACRelation*. For example, if a policy of expiry date is needed for all instances of users, roles, and permissions, then it can be defined in the *ROACElement*. If expiry date is only needed for the instances of users, then that context can be defined in the *User* element. However, if an extra context is applicable only on a subset of users, for example, restricted working hours, that is only applicable on this subset of users, then a sub-user element can be created e.g. *RestrictedUser*, in which the restricted hours intervals can be defined as a context. The same concept applies to other elements and relations.

The ROAC model have different levels of contexts, they are ordered below top-down from general to specific:

1. Role
2. Permission
3. Role-permission association
4. User
5. User-role association
6. Parameters (user-permission relation)

There are two constraints applicable on the above levels:

Constraint 8. *Any policy that can be defined at a general level can be defined in a more specific level.*

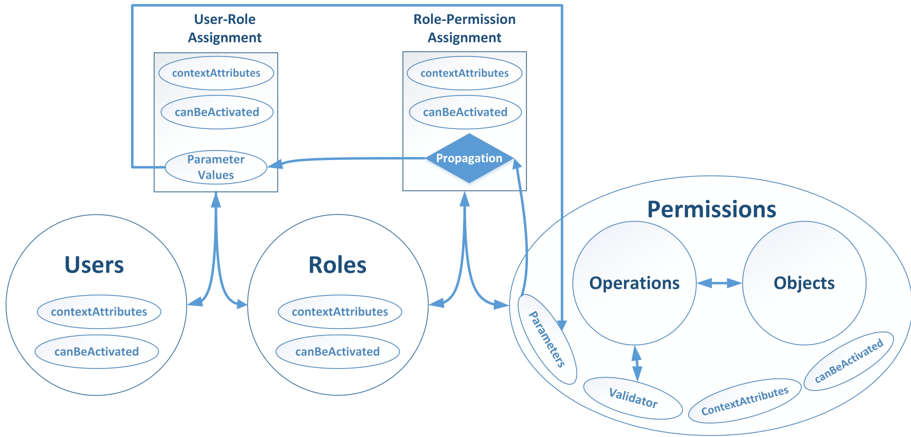


Figure B.5: The ROAC2 model

Constraint 9. *A policy must be defined at the most general possible level, to avoid complexity and reduce number of policies.*

The core ROAC model is depicted in Fig. B.5.

To generate the ROAC2 elements and relations, any element or relation in which context policies needs to be defined must include:

Context attributes: a dictionary of the element or relation context attributes. The dictionary key corresponds to the attribute name, and the value represents the value bound to the attribute. The context attribute dictionary is defined in the super element *ROACElement* and the super relation *ROACRelation*, and therefore, it is inherited by all elements and relations.

The context attributes dictionary is defined as follows in Python:

```
contextAttributes: FrozenDict[str, object]
```

In case attributes are needed for validating context for an element or relation, the *contextAttributes* containing these attributes must be passed when creating an instance of the element or the relation. For example, policy 4 of Table A.6 in our motivating example states that the *BankClient* role can be activated from any ware, when we create an instance from *Role*, we must pass the activation location attribute as shown in the following example:

```
bankClient = Role(contextAttributes=frozendict({"ActivationPolicy": "AnyWare"}))
```

Validation of context: the method *canBeActivated* must be defined in any context aware element or relation. The method validates the context then returns a Boolean value to indicate if the context is valid or not. However, user-permission context policies are validated in the parameterized permission validator. The *canBeActivated* method is defined as follows in Python:

```
def canBeActivated(self, operationAttributes: frozendict[str, object]) -> bool:
  # Context validation business logic
  return True
```

Validation of context in the *canBeActivated* method can depend on attributes that need to be passed from the authorization request to access a given protected operation or the *canBeActivated* method can extract the needed context attributes from the environment, for example, to connect to HR system to check if a user in on holiday. In case attributes need to be passed from the authorization request, the attributes are provided in the *operationAttributes* dictionary argument of the validator method. Therefore, a new argument is added to the validator method signature:

```
operationAttributes: frozendict[str, object]
```

Authorization Decision:

Authorization decision is ROAC2 is determined on multi-levels. The process starts by a user requesting authorization on a protected operation. The first step is to invoke the *canBeActivated* method of the user in order to validate the user context. Upon successful validation of the user context, we can move to the next step, which is validation of the context of all the user-role associations in which the user is associated. Afterwards, the role context is validated of all roles of the active user-role associations. Then, the context is validated for all role-permission associations of all relations in which the user active roles are associated. Thereafter, the context is validated for all permissions of the active role-permission relations in which the user roles are associated. The result of this process is a set of all active permissions of the user.

If the permission to execute the protected operation is in the set of the user's permissions, then we move to the final step which is validating the parameter values assigned to the user against corresponding parameter values from the protected operation, and validating the permission context, which is achieved by invoking the permission validator.

The different steps of the authorization decision are executed sequentially in the given order. If the result of any step in the authorization decision returns no active elements or relations, or the result of the permission validator is *False*, then the whole process is terminated by giving an access denied decision. The activity diagram in Fig. B.6 summarizes how an authorization decision is determined in ROAC2.

The authorization decision business logic that returns the active permissions of the user is implemented in a new method *activePermissionsOfUser* added to the *AuthorizationPolicy* class. Another method *activeRolesAssignedToUser* is also added, which returns the active user roles. The definition of both methods in Python is given below:

```
def activeRolesAssignedToUser(self, user: User) -> frozenset[Role]:
  roles = set()
  if user.canBeActivated():
    for k,v in self.userRoleAssociations[user].items():
      if v.canBeActivated():
        roles.add(v.role)
  return frozenset(roles)
```

```
def activePermissionsOfUser(self, user: User, operationAttributes: frozendict[str, object])
     ↪ -> frozenset:
  activeUserPermissions = set()
  for role in self.activeRolesAssignedToUser(user, operationAttributes):
    for rolePermission in self.rolePermissionAssociations[role].values():
      if rolePermission.canBeActivated(operationAttributes) and rolePermission.role.
     ↪ canBeActivated(operationAttributes) and rolePermission.permission.
     ↪ canBeActivated(operationAttributes):
        activeUserPermissions.add(rolePermission.permission)
  return frozenset(activeUserPermissions)
```

Revocation of Access Rights

Revocation of access is an action that involves removal of access rights. The smallest unit of access right in RBAC is the permission. Therefore, we will

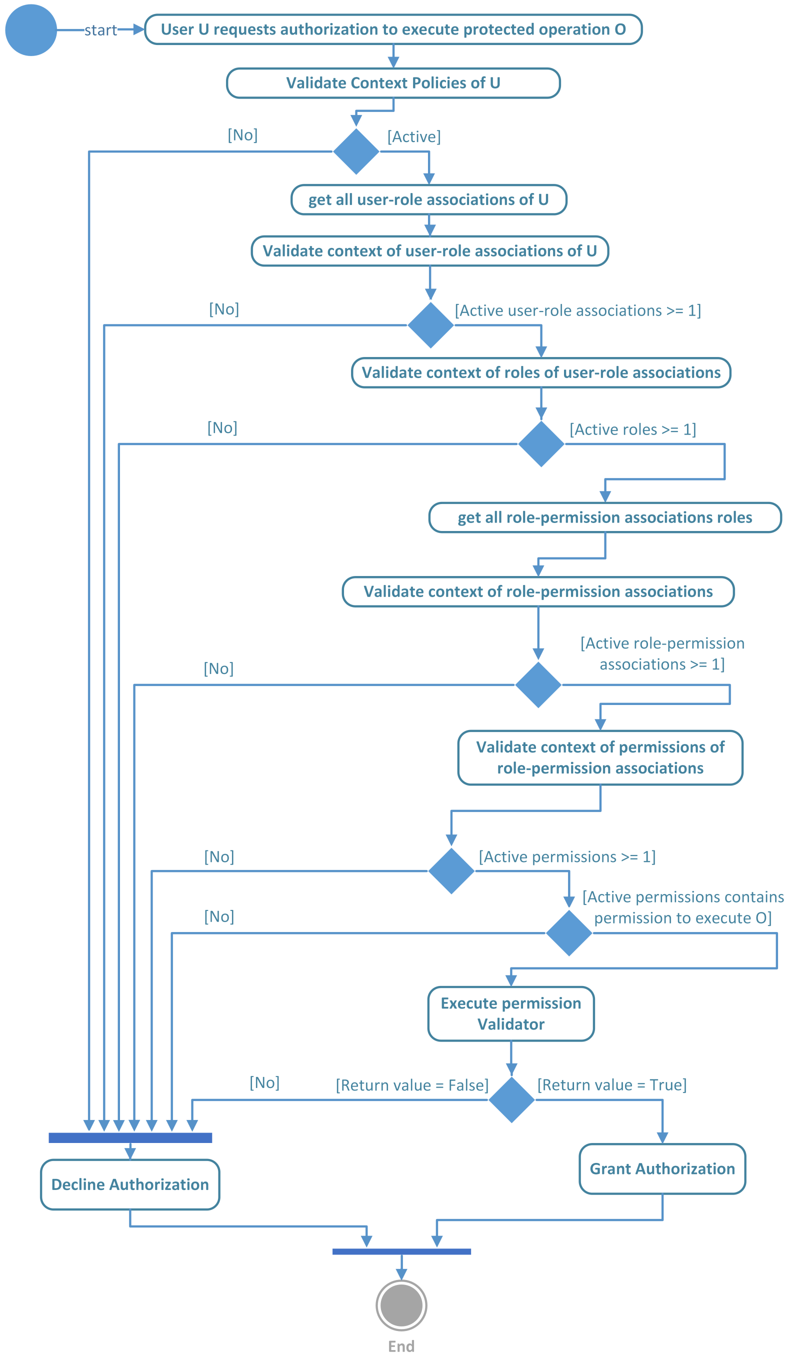


Figure B.6: The activity diagram of the ROAC2 authorization process

consider revocation of permissions. Revocation can be permanent or temporary. Revocation of access rights is needed in various circumstances; we can identify different types of revocations as follows:

- Revocation of one or more permission from a given role.
- Revocation of one or more permissions from a given user.
- Revocation of a permission from the access control system.
- Revocation of a role from the access control system.

Revocation of permissions from a given role:
Revoking a permission from a role is the simplest revocation type in RBAC. Revocation can be elegantly achieved by deassigning the permission from the role, which can be done by updating the role-permission relation. In the ROAC2 model, the concept of logical revocation is used to revoke a permission from the role-permission assignment. This can be achieved by introducing a new Boolean type attribute to *RolePermissionAssociation* called *revoked*. The *canBeActivated* method in the *RolePermissionAssociation* returns *False* if the value of the *revoked* attribute is *True*.

Revocation of permissions from a given user:
Revoking permissions from a given user can be a complex task as there is no direct assignment between users and permissions. However, revocation can be straightforward if permissions to be revoked are constituting the permissions set of a given role. In this case, revocation can be achieved elegantly by revoking the role from the user. In the ROAC model, we use the concept of logical revocation, which means that we indicate the relation as revoked. For this purpose, we add a new Boolean type attribute in the *UserRoleAssociation* relation called *revoked*. The *canBeActivated* method in the *UserRoleAssociation* returns *False* if the value of the *revoked* attribute is *True*. Total revocation of all user access rights can be simply achieved by revoking all roles assigned to that user.

RBAC does not support revocation of a subset of permissions of a user role. However, in the ROAC2 model, such revocation can be achieved in a more complex way. A new attribute can be defined in the *User* element of type *set*, which contains the revoked permissions. Then, the methods *activePermissionsOfUser* and *permissionsOfUser* can exclude the set of revoked permissions from the set of user permissions.

Revocation of a role or permission from the access control system:
Revoking a role or permission completely from the access control system can be

achieved in two different ways. One possibility is by deassigning the permission to be revoked from all roles using the role-permission revocation explained above. Similarly, a role can be revoked from all users using the user-role revocation.

Alternatively, a new Boolean type attribute can be added to the *Role* and *Permission* elements, called *revoked*. The *canBeActivated* method in the *Role* and *Permission* returns *False* if the value of the *revoked* attribute is *True*. Therefore, users will no longer be able to activate revoked roles or permissions.

Example Policies

To demonstrate how policies are modeled in context aware ROAC, we provide the Python definition of some policies from our motivating example (Appendix A).

User Policies:

The user policies 1, 2, and 3 can be defined in the *canBeActivated* method of *User* as follows:

```
def canBeActivated(self, operationAttributes: frozendict[str, object]) -> bool:
  # Policy 1
  assert self.contextAttributes["userType"] == "Client" or self.contextAttributes["
    ↪ userType"] == "Employee"

  # Policy 2
  assert self.checkIfUserIsOff() == False

  # Policy 3
  if self.contextAttributes["userType"] == "Client":
    assert self.contextAttributes["userIDExpiry"] >= date.today()

  return True
```

The *checkIfUserIsOff* method connects queries in HR system and returns *False* if the user is off.

User-Role Policies:
Policy 12 of user-role can be defined in the *canBeActivated* method of *UserRoleAssociation* as follows:

```
def canBeActivated(self, operationAttributes: frozendict[str, object]) −> bool:
  # Policy 12
  if "UserRoleExpiryDate" in self.contextAttributes:
    if self.contextAttributes["UserRoleExpiryDate"] < date.today():
      return False

  return True
```

User-Permission Policies:
Policy 14 of user-permission shown in table 3.9 for *TransferFundsPermission* can be defined in the *validator* of *TransferFundsPermission* as follows:

```
def validator(self, authPolicy: AuthorizationPolicy, operationAttributes: frozendict[str,
    ↪ object], user: User, wireType: str, wireAmount: float, wireCurrency: str, account:
    ↪ str):
  if self not in authPolicy.activePermissionsOfUser(user, operationAttributes):
    return False

  userBindings = authPolicy.parameterBindingsOfUser(user)

  if userBindings[self.branch] in ["N/A", "HeadOffice", "Any"] or isAccountInBranch(
    ↪ account, userBindings[self.branch]):
    return False

  amtBaseCurr = wireAmount ∗ self.exchangeRate(wireCurrency)

  if isSessionIpInBelgium(operationAttributes["Session"].ip):
    userAmount = userBindings[self.fundAmount]["Belgium"]
  else:
    userAmount = userBindings[self.fundAmount]["OutsideBelgium"]

  return amtBaseCurr <= userAmount and wireType in userBindings[self.fundType]
```

The method *exchangeRate* returns the exchange rate of a given currency. The returned exchange rate is used for calculating the transaction amount against the base currency of the bank.

The method *isSessionIpInBelgium* verifies if an IP address is in Belgium by referencing a geo-location database.

B.2 A UML Formal Model of ROAC

In this section, we include the formal model of ROAC for completeness. We use the Unified Modeling Language (UML) [76] to formalize the definitions of the ROAC model. UML provides four modeling layers: M0, M1, M2 and M3 as they become more abstract. Layer M0 corresponds to the end instances (objects) or in other words, the concrete system. M1 is the model of the system, of which M0 is just one realization. M2 is the language used to describe the model. M3 is the most abstract level, which is used to describe UML itself or any other modeling language [78]. Specifications of the standard UML metamodel are provided by OMG in [76]. We start by providing the ROAC model (layers M2, M3), which is depicted in Fig. B.7, then we provide sample M1 and M0 models, which are depicted in Fig. B.8.

Fig. B.7 depicts the meta model of the core ROAC model. The diagram projects the relationships between the different elements of the core ROAC model according to the definitions of elements and their relations.

The primary goal of the ROAC metamodel (bottom of Fig. B.7 denoted *ROAC Metamodel*) is to define a language for specifying the ROAC model and to define the semantics for how model elements in the ROAC model get instantiated.

Everything in the ROC metamodel is derived from the *Element* parent. *Classifier* is the parent of all the ROAC elements. A *classifier* has a set of *features*, some of which are properties called *attributes* of the *Classifier*. *Attribute* and *Behavior* such as *Method* are sub-elements from the *Feature* element. *Method* is called *Operation* in the standard UML specifications [76] . Therefore, ROAC elements which are instances from *Classifier* have both attributes and behavior.

The *Feature* hierarchy of the metamodel is extended with a new *StructuralFeature* called *Parameter*, that is the permission parameter of the ROAC model. In the standard UML metamodel [76], *Classifier* has a direct association with *feature*. However, in the ROAC model, parameters can only be defined in permissions, therefore, the direct association between *Classifier* and *Feature* in the ROAC meta model has been moved one level down to *Attribute* and *Behavior*, excluding *parameter*. Therefore, *Parameter* is directly associated with *Permission*. The difference between *Parameter* and *Attribute* (both instance and static), is that it goes across the role-permission relation to roles. We provide later-on an OCL modeling for parameters propagation.

Similarly, the *BehavioralFeature* of the standard UML metamodel was also extended with a new behavior called *Validator*. Therefore, the *BehavioralFeature* in the ROAC metamodel has two sub-features, *Method* and *Validator*. The direct association between *Classifier* and *BehavioralFeature* is also downgraded to level

of the *Method*, as validators can be only defined in permissions. Therefore, the diagram includes a direct association between *Validator* and *Permission*.

The relations of the ROAC model are instances from *Association* , as such is an instance from both *Classifier* and *Relationship*. The ROAC relations (user-role and role-permission assignments) have a similar concept to the *AssociationClass* of UML. According to UML specifications [76], "An *AssociationClass* is a declaration of an *association* that has a set of *features* of its own. An association class describes a set of objects that each share the same specifications of *features*, *constraints*, and semantics entailed by the association class as a kind of *Class*, and correspond to a unique link instantiating the *AssociationClass* as a kind of *Association*". The ROAC association classes *UserRoleAssociation* and *RolePemissionAssociation* specify semantic relationships between the three ROAC element instances (*User*, *Role* and *Permission*). Where *UserRoleAssociation* declares that there can be links between instances of *User* and *Role*. While *RolePemissionAssociation* declares that there can be links between instances of *Role* and *Permission*

An object diagram of sample users, roles, permissions and their relations is provided in Fig. B.8. It contains two models. The top model in the figure denoted *Sample ROAC Model*) is instantiated from the ROAC metamodel. The sample ROAC model is specified at layer M1 of the UML language. The bottom model is an object model, specified at layer M0 of UML, which is instantiated from the sample ROAC model.

The three users depicted in the figure are instances from *BankerUser*, which is an instance from *User*. Roles *JuniorTeller* and *SeniorTeller* are both instances from *BankerRole*, which is an instance from *Role*. However, permissions are treated differently, a permission instance is defined at layer M1 for each permission. The instance *JuniorDeposit* is created from permission *Deposit* and assigned to the *JuniorTeller* role, another instance *SeniorDeposit* of the same permission is created and assigned to the *SeniorTeller* role. Both *JuniorDeposit* and *SeniorDeposit* are considered as two different access rights, because an attribute is defined in the permission that restricts access according to *AccountType*, which has different values in both instances. Alternatively, one instance from the permission instance *CheckAccountBalance* is created and assigned to both *SeniorTeller* and *JuniorTeller* roles. The *rolePermissionAssignment* associates all parameters of different permissions instances to the role instance to whom they are assigned. The same role instance can be assigned to multiple user instances, as shown in the assignment of *JuniorTeller* to both *Alice* and *Bob*. The parameter values are set per user-role assignment record, e.g. *Marc* is *SeniorTeller* with *MaxAmount = 100K* and *Branch = "Leuven"*.

Both instances from *Deposit* permission grant access on the same operation

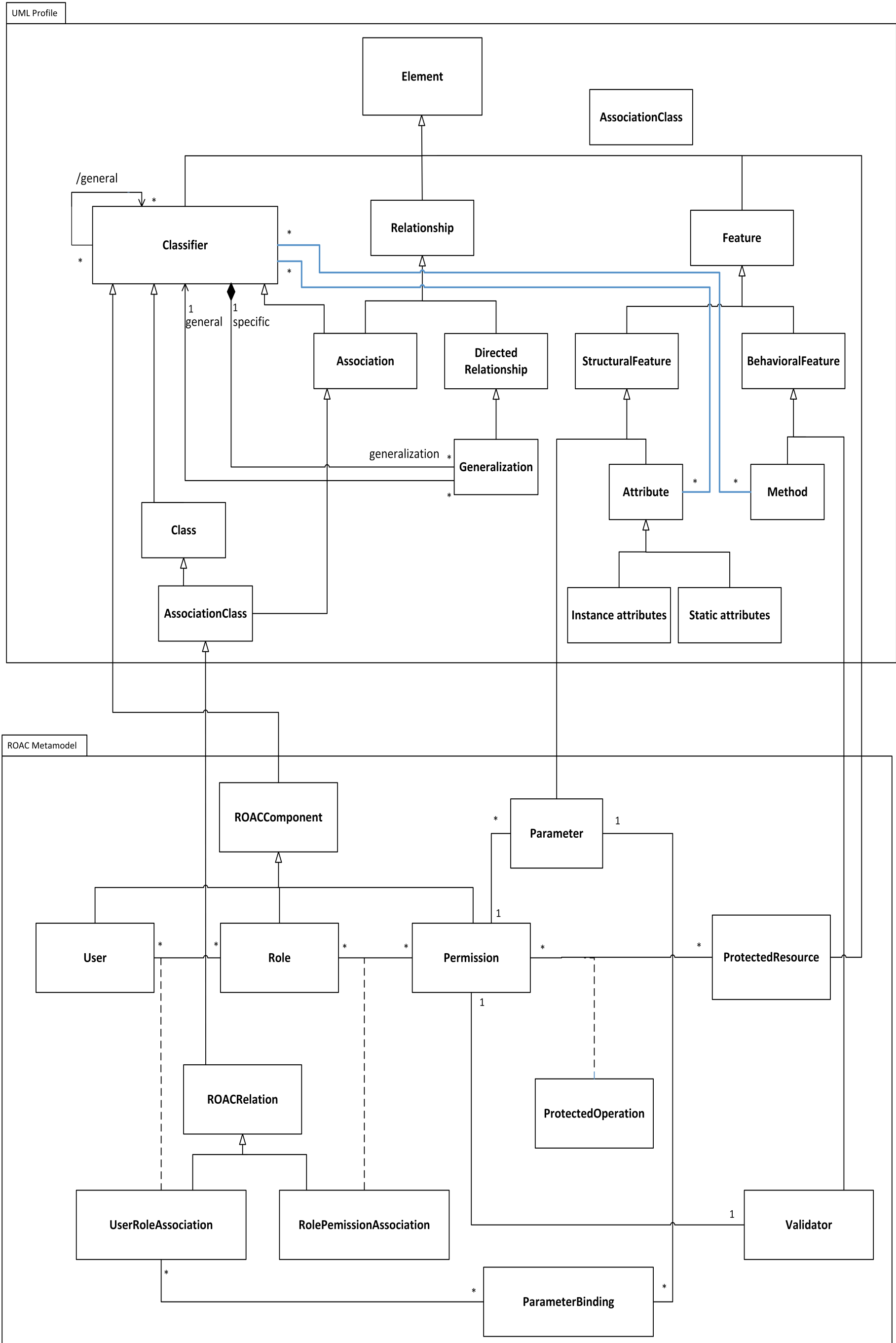


Figure B.7: The UML Meta Model diagram of the ROAC model

instance *DepositIntoCurrentAccount*, which commits a new transaction with each successful deposit operation.

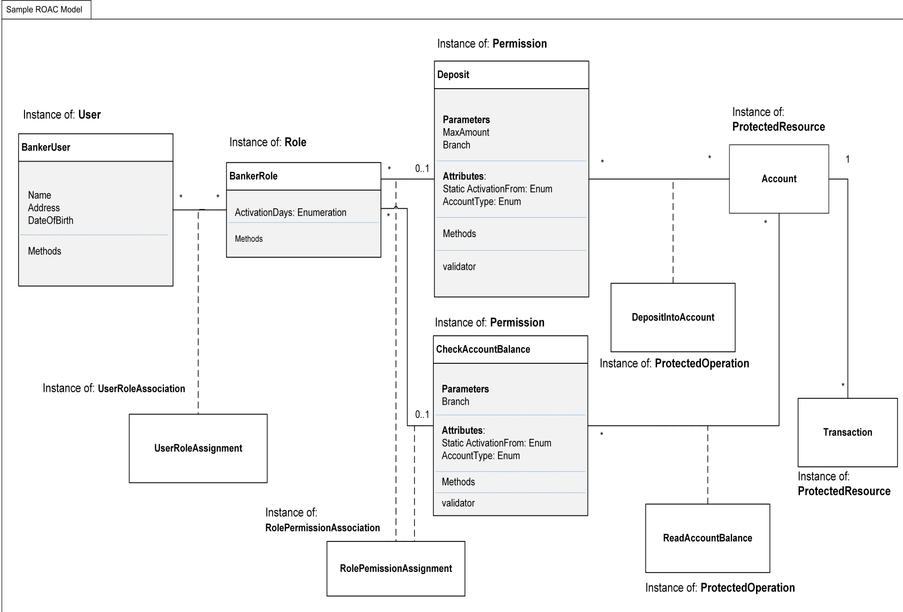
B.2.1 Validators:

The UML diagram of validator is depicted in Fig. B.9.

In the standard UML meta model specifications include an item called *Parameter* that is associated with behavioral features. It refers to a specification of an argument used to pass information into or out of an invocation of behaviors. We renamed it to *argument* in the diagram to avoid confusion with the ROAC permission parameter.

B.2.2 Parameters:

The UML model of parameter is shown in Fig. B.10.



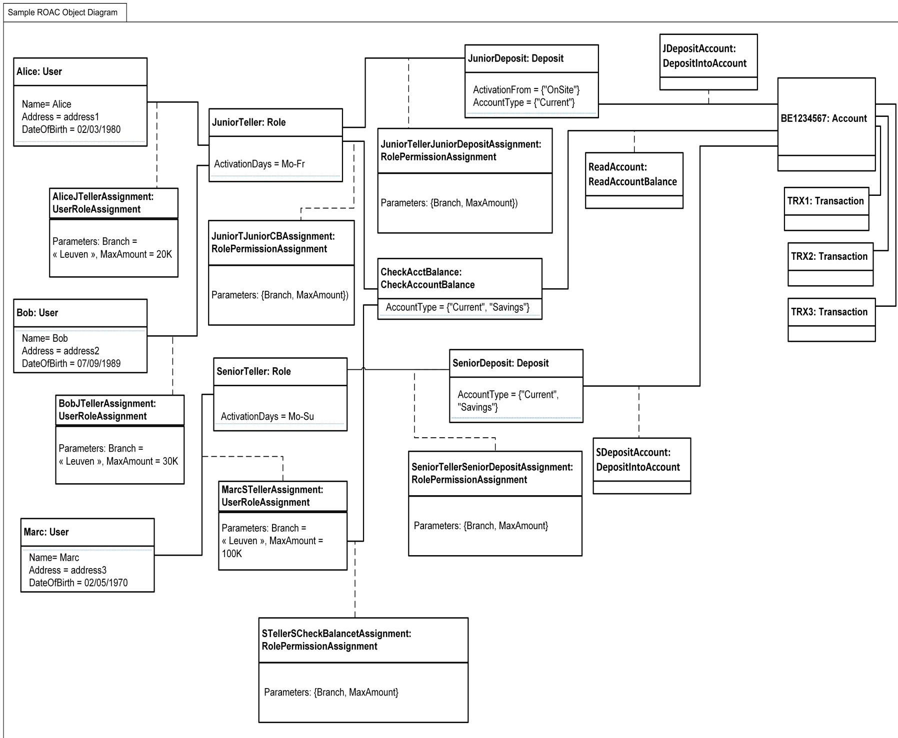


Figure B.8: An Object diagram of sample users, roles, permissions and relations

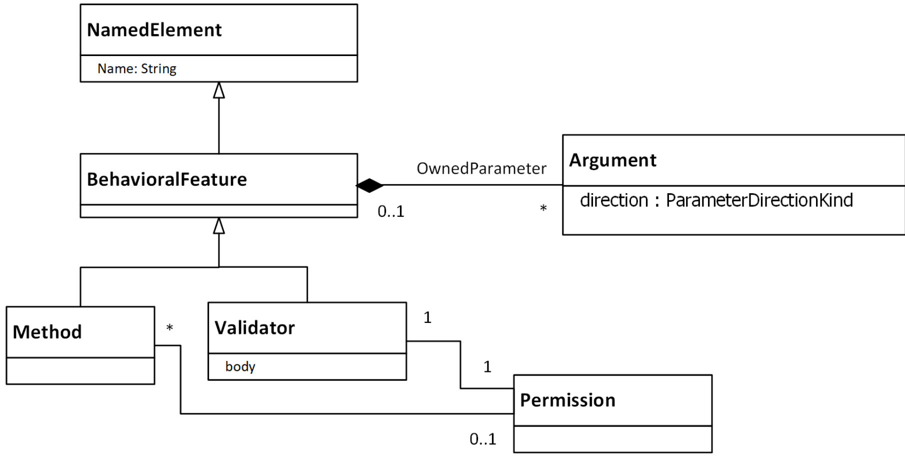


Figure B.9: The UML diagram of validator

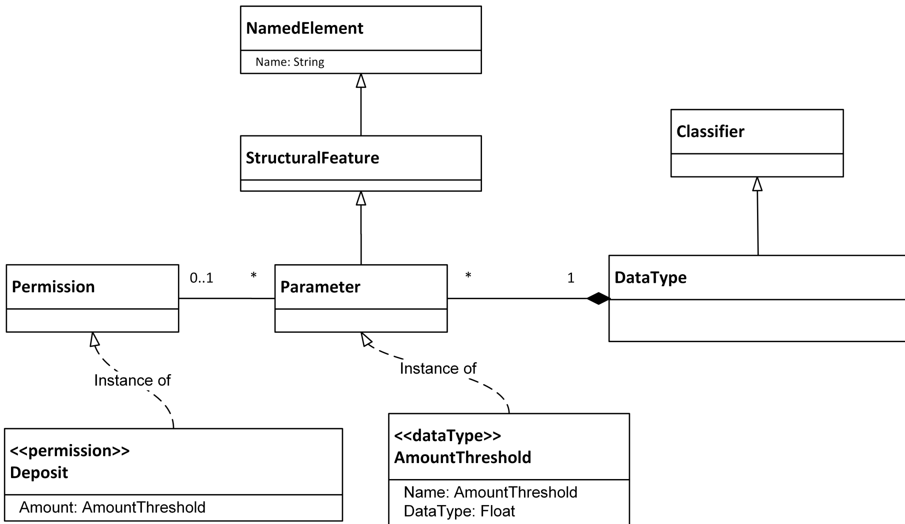


Figure B.10: The UML diagram of parameter

Appendix C

The Hierarchical ROAC Formal Model

In this appendix we provide the hierarchical ROAC model. The model is composed mainly of role and user hierarchies. Both the Python and UML models are also provided.

C.1 Role Hierarchy

The Python model of role hierarchy includes an implementation of role hierarchies, which is modeled as a directed graph data structure. The Python model also includes changes to the *AuthorizationPolicy* administration class.

A dictionary data structure is introduced for modeling the vertices and edges of the role hierarchy. The dictionary key is the hierarchical role, the value is a sub-dictionary of which the key is the hierarchized role, and the value is the exclusion set of permissions.

The following constraints are also validated to ensure correctness of the dictionary.

Constraint 10. *A role cannot hierarchize itself.*

Constraint 11. *The role hierarchy graph cannot include cycles, which means that a hierarchical role cannot be hierarchized by any of its sub-roles.*

The following Python fragment shows the graph dictionary and the validation of the constraints.

```
@dataclass_with_check(frozen=True)
class RoleHierarchy:
  # The dictionary is [role: Role, frozendict[subRole: Role, exclusionSet: frozenset]
  roleHierarchyDict: frozendict[Role, frozendict[Role, frozenset]]

  def __check__(self):
    for role in self.roleHierarchyDict.keys():
      # Constraint 10
      assert role not in self.roleHierarchyDict[role].keys()

      # Constraint 11
      assert role not in self.subRoles(role)
```

The following set of role hierarchy administrative methods are also defined in the *RoleHierarchy* class:

allHierarchicalRoles: this method returns all hierarchical roles that hierarchize other roles.

```
def allHierarchicalRoles(self) -> frozenset:
  return frozenset(self.roleHierarchyDict.keys())
```

allSubRoles: this method returns all roles that are hierarchized by other roles.

```
def allSubRoles(self) -> frozenset:
  return frozenset({k for k, v in self.roleHierarchyDict.items() for kk, vv in v.items()})
```

directSubRoles: this method returns all roles that are directly hierarchized by a given role.

```
def directSubRoles(self, role: Role) -> frozenset:
  if role in self.roleHierarchyDict.keys():
    return frozenset(self.roleHierarchyDict[role].keys())
  else:
    return frozenset()
```

subRoles: this method returns a set of all roles that are hierarchized by the given role. The method uses recursion to depth-search all sub-roles.

```
def subRoles(self, role: Role) -> set:
  directSubRoles = self.directSubRoles(role)
  tmp = set(directSubRoles.copy())
  for role in directSubRoles:
    tmp.update(self.subRoles(role))
  return tmp
```

subRolesExclusionSets: this method returns a dictionary of which the keys represent all roles that are hierarchized by the given role, similar to outcome of the *subRoles* method. The values of the dictionary are sets of excluded permissions.

```
def subRolesExclusionSets(self, role: Role) -> dict:
  if role in self.roleHierarchyDict.keys():
    directSubRoles = frozendict(self.roleHierarchyDict[role].items())
  else:
    directSubRoles = frozendict()

  tmp = dict(directSubRoles.copy())
  for role in directSubRoles.keys():
    tmp.update(self.subRolesExclusionSets(role))
  return tmp
```

An instance of *RoleHierrachy* is defined in *AuthorizationPolicy*, it enables accessing the role hierarchies administrative methods from the administration class.

```
roleHierarchy: RoleHierarchy
```

Changes has been also made to some administrative methods, mainly, the modifications allow to include indirect roles into account in administrative methods.

ROAC0:
The *permissionsOfRole* method is modified so that it returns permissions of the given role, and permissions of sub-roles of the given role. There are two implementations of this method, one for the optimistic mode and another for the pessimistic mode. We use the optimistic implementation as default in *AuthorizationPolicy*.

```
def permissionsOfRole(self, role: Role) -> frozenset[Permission]:
  directPermissions = set(self.rolePermissionAssociations[role])
  indirectPermissions = set()
  for k, v in self.roleHierarchy.subRolesExclusionSets(role).items():
    indirectPermissions.update(set(self.rolePermissionAssociations[k]).difference(v))
  return frozenset(directPermissions.union(indirectPermissions))
```

```
def permissionsOfRolePessimistic(self, role: Role) -> frozenset[Permission]:
  directPermissions = set(self.rolePermissionAssociations[role])
  indirectPermissions = set()
  excludedPermissions = set()
  for k, v in self.roleHierarchy.subRolesExclusionSets(role).items():
    indirectPermissions.update(set(self.rolePermissionAssociations[k]))
    excludedPermissions.update(v)
  return frozenset(directPermissions.union(indirectPermissions.difference(
    ↪ excludedPermissions)))
```

ROAC1:
There are no changes to the ROAC1 model. However, the parameters of hierarchized roles must be provided with the direct role assignment (the *UserRoleAssignment* object). For example if *User1* is assigned *Role1*, which has parameter *P1*. *Role1* hierarchizes roles *Role2*, which has parameter *P2* and *Role3*, which has parameter *P3*. The parameters, *P1, P2, P3* must be provided in the relation *UserRoleAssociation(User1, Role1, params_P1_P2_P3)*.

ROAC2:
The method *activePermissionsOfUser* is modified to validate context of role-permission associations of hierarchized roles.

```
def activePermissionsOfUser(self, user: User) -> frozenset:
  activeUserPermissions = set()
  for role in set(self.rolesAssignedToUser(user)):
    for rolePermission in self.rolePermissionAssociations[role].values():
      if rolePermission.canBeActivated and rolePermission.role.canBeActivated() and
    ↪ rolePermission.permission.canBeActivated():
        activeUserPermissions.add(rolePermission.permission)
  return frozenset(activeUserPermissions)
```

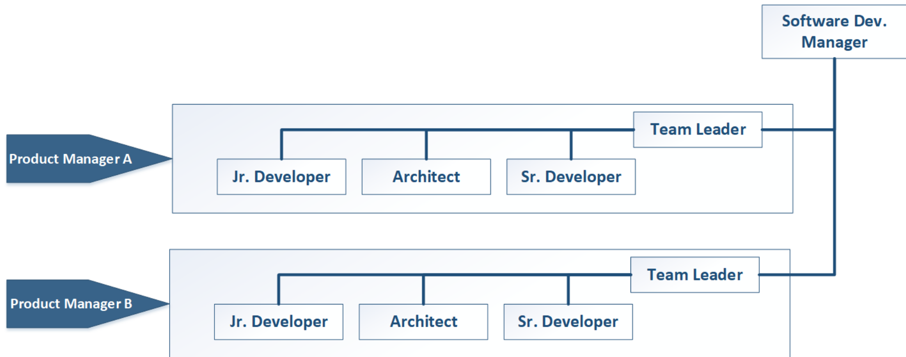


Figure C.1: Example of matrix organization structure

C.2 User Hierarchy

Hierarchical RBAC models attempt to project organization structures on role hierarchy. However, this is not the case in the real world. To explain this further, consider the organization chart in Fig. C.1. The chart has several developer positions. However, it is not necessarily that all these positions are assigned identical roles. The developers might work on different products or projects in the organization. This might include different development environments. Moreover, different technology might be used across different products. Developers who work on a product have access to their products development environment and have no access to development environments of other products, in spite of the fact that all of them are on the same level in the hierarchy and are assigned to the same position in the organization chart.

In a case study of an RBAC based access control system of a German bank [90], the bank defines roles as a combination of the official position, that is defined in the organization structure and the job function. Typical official positions could be that of the ordinary *Clerk*, *Group Manager* or *Regional Manager*. Job functions represent their daily duties such as being a *financial analyst*, *share technician* or *internal software engineer*. In matrix structures, a role might be constructed by combining the positions and products, such as *Software Developer* of *Product A*.

The hierarchical ROAC model is extended to define hierarchies over users. In hierarchical ROAC, hierarchy of users reflects user hierarchies in organization structures. User hierarchies are represented as directed graphs, where each user has at least one parent node corresponding to his direct responsible (line manager). A user might have a second parent node, if the organization structure

encompasses dual reporting lines, such as matrix structures. The directed graph is reduced to a general tree for structures with single reporting lines, such as functional or divisional organizations. Fig. C.2 depicts a sample user hierarchy of the organization structure of Fig. C.1. Dotted edges represent activity reporting lines (related to projects or products), continuous lines represent functional lines of authority (line managers). Some nodes do not have dotted lines reporting since they are not affiliated in one product, such as the *Software Development Manager*. Functional lines of authority are mandatory in our sample structure. Therefore, all nodes are connected to line managers.

Modeling user hierarchies with directed graphs makes it easier for finding the right responsible (or reporting line) of any user. The graph also enables finding the next level manager of a user, i.e. the nth line manager.

Representation of user hierarchies by adopting directed graphs incurs some changes to user administration. The most interesting information that can be absorbed from organization hierarchies into the user hierarchy are the line manager, the activity manager, organization unit, project or product, and the position title. All these fields can be integrated as attributes in the user blueprint. However, the line and activity managers represent edges of the graph. Therefore, administrative functions that add, modify, or delete users must handle the graph edges when performing any action on users.

Python Model

The Python model of user hierarchy utilizes a dictionary data structure for defining the vertices and edges of the user hierarchy. The dictionary key is the user, the value is a sub-dictionary of which the key is the manager and the value is the type of management relation, e.g. *Line* or *Activity*.

The following constraints are also validated to ensure correctness of the dictionary:

Constraint 12. *A user cannot be the manager of himself.*

Constraint 13. *There exists only one user that does not have a line manager (The CEO of the organization). It is possible for users not to have activity managers at all.*

Constraint 14. *The user hierarchy graph cannot include cycles, which means that a subordinate user cannot be a line manager of his own line manager nor can be a manager of his* n^{th} *level manager.*

Constraint 15. *A user can have only one line manager. However, he can have multiple activity managers.*

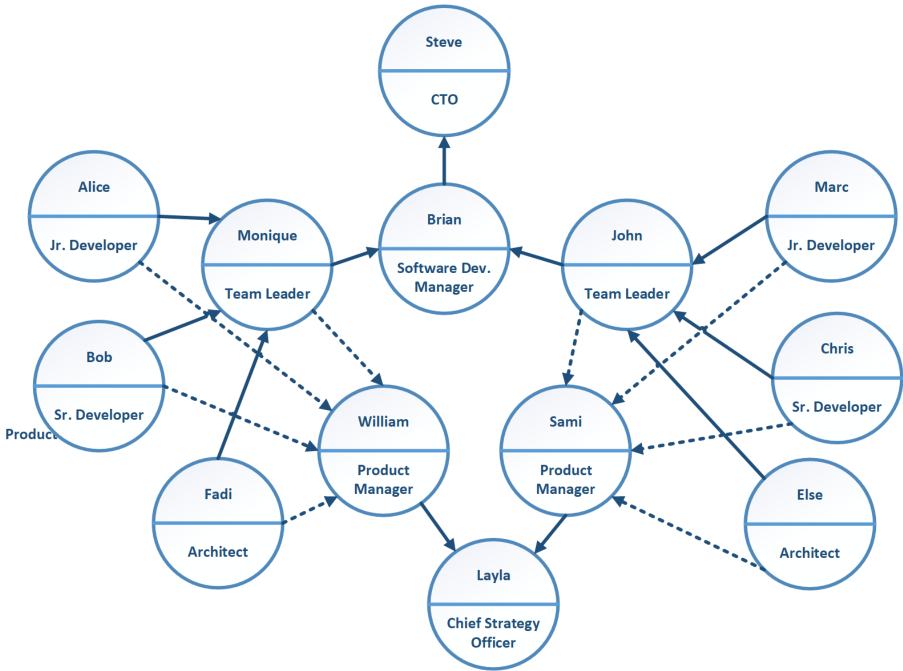


Figure C.2: A directed graph modeling the users hierarchy of a matrix organization structure.

Constraint 16. *The graph edges must be defined between already defined vertices. In other words, all line managers defined in sub-dictionaries must exist as keys in the main dictionary.*

The following Python fragment shows the graph dictionary and the validation of the constraints:

```
@dataclass_with_check(frozen=True)
class UserHierarchyGraph:
  userHierarchy: frozendict[User, frozendict[User, str]]

  def ___check___(self):
    for user in self.userHierarchy.keys():
      # Constraint 12
      assert user not in self.userHierarchy[user].keys()

      # Constraint 14
      for manager in self.userHierarchy[user].keys():
        assert manager not in self.allSubordinates(user)

      # Constraint 15
      assert len({k for k, v in self.userHierarchy[user].items() if v == "Line"}) <= 1

    # Constraint 16
    assert self.allLineManagers().issubset(self.userHierarchy.keys())

    # Constraint 13
    assert len({k for k, v in self.userHierarchy.items() if v == {}}) == 1
```

The *UserHierarchyGraph* includes the following administrative methods:

lineManager: returns the direct line manager of a given user.

```
def lineManager(self, user: User) -> User:
  lineMgr = {k for k, v in self.userHierarchy[user].items() if v == "Line"}
  if len(lineMgr) > 0:
    return lineMgr.pop()
```

activityManagers: returns the set of all activity managers of a given user.

```
def activityManagers(self, user: User) -> frozenset:
  return frozenset({k for k, v in self.userHierarchy[user].items() if v == "Activity"})
```

activityManagers: returns the set of all line managers of the organization.

```
def allLineManagers(self) -> frozenset:
  return frozenset({kk for k, v in self.userHierarchy.items() for kk, vv in v.items() if vv
    ↪ == "Line"})
```

allSubordinatesOfOrganization: returns the set of all users who have line managers (subordinates).

```
def allSubordinatesOfOrganization(self) -> frozenset:
  return frozenset({k for k, v in self.userHierarchy.items() for kk, vv in v.items() if vv ==
    ↪ "Line"})
```

organizationsCEO: returns the CEO of the organization.

```
def organizationsCEO(self) -> User:
  ceo = {k for k, v in self.userHierarchy.items() if v == {}}
  if len(ceo) == 1:
    return ceo.pop()
```

directSubordinates: returns all direct subordinates of a given user.

```
def directSubordinates(self, manager: User) -> frozenset:
  return frozenset({k for k, v in self.userHierarchy.items() for kk, vv in v.items() if kk ==
    ↪ manager and vv == "Line"})
```

allSubordinates: returns all subordinates of a given manager. The method uses recursion to depth-search all the graph.

```
def allSubordinates(self, manager: User) -> set:
  directSubordinates = self.directSubordinates(manager)
  tmp = set(directSubordinates.copy())
  for user in directSubordinates:
    tmp.update(self.allSubordinates(user))
  return tmp
```

C.3 UML Model

ROAC supports two different types of generalizations: hierarchies and inheritance. To distinguish between both different relations in the UML models, we use different UML relationship graphical representations. We dedicate the UML generalization relationship arrow for inheritance which is represented graphically as a solid line ending with a triangular hollow arrowhead pointing to the parent. And a solid filled arrowhead to represent hierarchies. Both representations are shown in Fig. C.3.

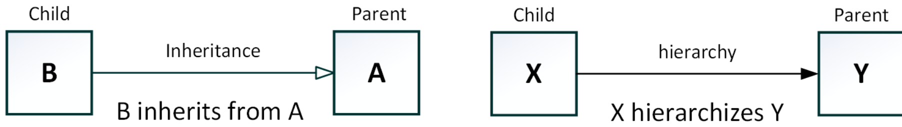


Figure C.3: Generalization arrows of inheritance and hierarchy

C.3.1 Elements & Relations Inheritance

Inheritance in ROAC is provided through the generalization relation of the UML metamodel, which is provided in the UML superstructure [75]. Inheritance is defined on the level of classifier, which is inherited by all ROAC elements and relations as shown in Fig. C.4. Inheritance is a sub-type of *DirectedRelationship*. The inheritance navigability between a specific and a general element or relation is one-way, which is directed towards the general, which means that the specific knows the general but not vice-versa.

C.3.2 Role Hierarchy

The UML class diagram of the role hierarchy in the ROAC model is shown in Fig. C.5. The role-role hierarchy relation is defined via a new association *RoleHierrachy* which defines a one-to-one exclusion set between each role-role relation.

In both pessimistic and optimistic modes, permissions of ancestor roles are not assigned directly to the descendant role, i.e. permissions are not included in the role-permission assignment data structure. However, permissions of ancestor roles are always retrieved from the ancestor roles role-permission assignments according to the activation mode (i.e. pessimistic or optimistic). This mechanism allows descendant roles to maintain an up to date set permissions of ancestor roles. This kind of implicit assignment caters for changes which might be made to ancestor role-permission relations.

Fig. C.6. depicts the object diagram of policies 16 and 17 of our motivating example. Role *BranchManager* hierarchizes both *Teller* and *SecuritiesClerk* roles. The hierrachy excludes permissions *TransferFunds* and *DepositCash* from the hierrachy. The hierrachy permissions of role *BranchManager* are *CheckBalance* and *OnboardClient* . The role *BranchManager* has another permission though direct role-permission assignment *ApproveTransaction*.

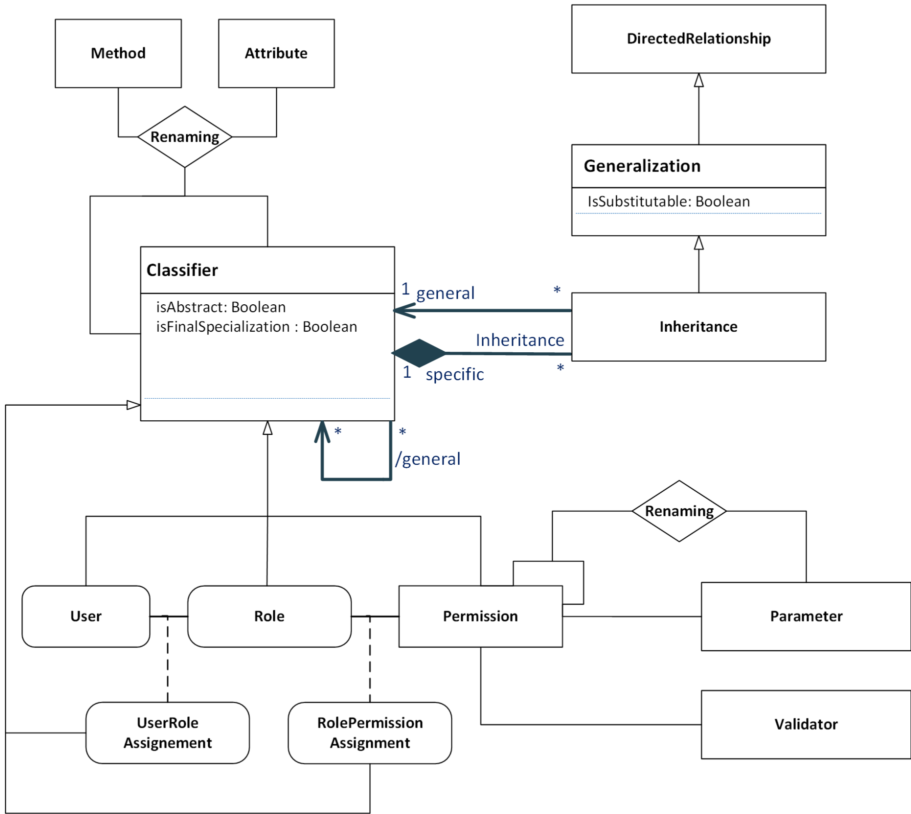


Figure C.4: Inheritance of ROAC elements and relations

C.3.3 User Hierarchy

Fig. C.7 depicts the class diagram of the hierarchical user element. It includes two associations, *LineManager* to *Subordinate* which is a one-to-many relationship and two associations with *Activity* (e.g. project or product). An activity can have multiple member users. User membership in activities is optional. Moreover, the same user can be member in different activities. Unlike elements and relations inheritance, the user hierarchy relation is an association rather than a directed association. Therefore, the navigability between *LineManager* and *Subordinate* is two-way, which means that both know each other.

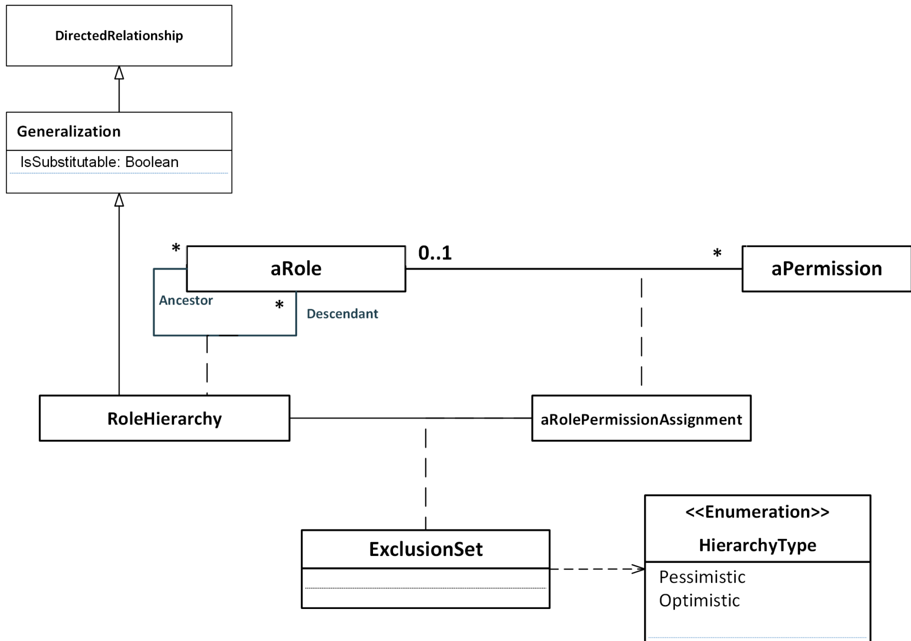


Figure C.5: UML class diagram of role hierarchy.

RA: RolePermissionAss.
SecuritiesClerk: BankerRole
CheckBalance: CheckBalancePermission
RA: RolePermissionAss.
TransferFunds: TransferFundsPermission
SecuritiesClerk: Teller
Optimistic
Exclude: ExclusionSet
RH: RoleHierrachy
RA: RolePermissionAss.
Optimistic
OnboardClient: OnboardClientPermission
Exclude: ExclusionSet
DepositCash: DepositCashPermission
RA: RolePermissionAss.
BranchManager: BankerRole
ApproveTransaction: ApproveTrxPermission
RA: RolePermissionAss.

Figure C.6: Object diagram of an example multiple role hierarchy.

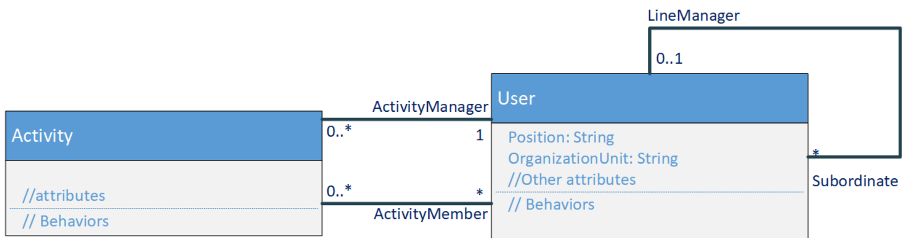


Figure C.7: The UML Class diagram of user component.

Appendix D

Delegation Formal Model

In this appendix, we provide the Python and UML model of OSDM.

D.1 Python Model

D.1.1 DelegationRelation

The *DelegationRelation* contains the delegator, delegatee, expiry date of the delegation relation, and whether the delegation is multilevel or not. Furthermore, it includes a dictionary, of which the keys include the roles to be delegated and the values contain the sets of excluded permissions of each delegated role, in case of partial delegation.

```
@dataclass(frozen=True)
class DelegationRelation:
  delegator: User
  delegatee: User
  delegationRoles: frozendict[Role, frozenset[Permission]]
  expiryDate: date
  multiLevel: bool
```

D.1.2 DelegationPolicy

DelegationPolicy includes the administrative methods of delegation. Furthermore, it includes two dictionary datastructures. The first dictionary is *approvalMatrix*, which includes the approval matrix for all available roles in the access control system. The key of the dictionary is the role, the value is a set of sets containing sets of users. The relation between elements of the outer set is logical *AND* , the relation between elements of the inner set is logical *OR*. Consider *policy 19* of our motivating example, the approval for the *Teller* role requires the branch manager (User5) and any user of the branch. We can model this policy using the *approvalMatrix* dictionary as follows:

```
{Teller:
  {{User5}, {User2, User3, User4}}
}
```

The dictionary can also contain references to the user managers, for example, a policy requires approval of the manager of the delegator and the manager of the delegatee can be modeled as follows:

```
{Teller:
  {{"ManagerOfDelegator"}, {"ManagerOfDelegatee"}}
}
```

In this case, the users are retrieved from the users hierarchy.

The second dictionary is *delegationUserRoleAssociations*, which maps users to their delegation relations.

The *DelegationPolicy* class includes a reference to *AuthorizationPolicy*, which is used supporting the cascading of role-permission relation and context validations. It also includes a reference to the *UserHierarchyGraph*, which is used to get the managers of delegator and delegatee for two purposes, firstly, when validating the users who initiate the delegation. Secondly, when references to the managers are used in the *approvalMatrix* dictionary.

The following code fragment shows the definition of the *DelegationPolicy* class and its instance variables:

```
@dataclass
class DelegationPolicy:
  authorizationPolicy: AuthorizationPolicy
  userHierarchyGraph: UserHierarchyGraph

  approvalMatrix: frozendict[Role, frozenset[frozenset[object]]]
  delegationUserRoleAssociations: dict[User, DelegationRelation]
```

Administrative methods of *DelegationPolicy*:

delegate:

This method adds the delegation relation to the *delegationUserRoleAssociations* dictionary. Prior to affecting the relation, it validates the following constraints:

Constraint 17. *The initiator of the delegation request must be allowed to initiate the delegation request.*

Constraint 18. *The delegated access rights must be part of the roles assigned to the delegator at the time of delegation by direct assignment, role hierarchy, or delegated roles in case multi-level delegation is allowed.*

Constraint 19. *The delegation cannot be affected unless all necessary approvals are made.*

```
def delegate(self, delegationRelation: DelegationRelation, delegationInitiator: User,
    ↪ delegationApprovals: frozenset[User]):

  # Validate Constraint 17
  assert delegationInitiator in (delegationRelation.delegator, delegationRelation.delegatee,
    ↪  self.userHierarchyGraph.lineManager(delegationRelation.delegator), self.
    ↪ userHierarchyGraph.lineManager(delegationRelation.delegatee))

  # Validate Constraint 18
  if not delegationRelation.multiLevel:
    assert set(delegationRelation.delegationRoles.keys()).issubset(self.authorizationPolicy.
    ↪ rolesAssignedToUser(delegationRelation.delegator))
  else:
    assert set(delegationRelation.delegationRoles.keys()).issubset(self.authorizationPolicy.
    ↪ rolesAssignedToUser(delegationRelation.delegator)) or (set(delegationRelation.
    ↪ delegationRoles.keys()).issubset(self.delegatedRolesAssignedToUser(
    ↪ delegationRelation.delegator)))

  # Validate Constraint 19
  for role in delegationRelation.delegationRoles.keys():
    for s in self.approvalMatrix[role]:
      tmp = set(s)
      for item in s:
        if item == "ManagerOfDelegator":
          tmp.remove(item)
          tmp.add(self.userHierarchyGraph.lineManager(delegationRelation.delegator))
        elif item == "ManagerOfDelegatee":
          tmp.remove(item)
          tmp.add(self.userHierarchyGraph.lineManager(delegationRelation.delegatee))
      assert len(delegationApprovals.intersection(tmp)) > 0

  # Add the delegationRelation to the dictionary
  self.delegationUserRoleAssociations[delegationRelation.delegatee] = delegationRelation
```

delegatedRolesAssignedToUser:

This method retruns all roles that are delegated to a given user.

```
def delegatedRolesAssignedToUser(self, user: User) -> frozenset[Role]:
  if self.delegationUserRoleAssociations[user].expiryDate >= date.today():
    return frozenset(self.delegationUserRoleAssociations[user].delegationRoles.keys())
```

delegatedPermissionsOfUser:

This method retruns all permissions that are delegated to a given user.

```
def delegatedPermissionsOfUser(self, user: User) -> frozenset:
  if self.delegationUserRoleAssociations[user].expiryDate >= date.today():
    return frozenset(r for rs, rv in self.delegationUserRoleAssociations[user].
     ↪ delegationRoles.items() for r in self.authorizationPolicy.permissionsOfRole(rs).
     ↪ difference(rv))
```

delegatedActivePermissionsOfUser:

This method validates the context policies of the delegator, delegator-role assignment, delegated role, permission, and role-permission before returning the user permissions.

```
def delegatedActivePermissionsOfUser(self, user: User) -> frozenset:
  userPermissions = set()
  activeUserPermissions = self.authorizationPolicy.activePermissionsOfUser(self.
     ↪ delegationUserRoleAssociations[user].delegator)

  if self.delegationUserRoleAssociations[user].expiryDate >= date.today():
    userPermissions = (r for rs, rv in self.delegationUserRoleAssociations[user].
     ↪ delegationRoles.items() for r in self.authorizationPolicy.permissionsOfRole(rs).
     ↪ difference(rv))

  return frozenset(activeUserPermissions.intersection(userPermissions))
```

delegatedParameterBindingsOfUser:

This method returns a dictionary of all parameters (and their values bound to the user) of all permissions of roles delegated to a given user.

```
def delegatedParameterBindingsOfUser(self, user: User) -> frozendict[Parameter, object]:
     ↪
  if self.delegationUserRoleAssociations[user].expiryDate >= date.today():
    allDelegatorParameters = self.authorizationPolicy.parameterBindingsOfUser(self.
     ↪ delegationUserRoleAssociations[user].delegator)
    allDelegatedPermissionParameters = frozenset(p for rs in self.
     ↪ delegatedPermissionsOfUser(user) for p in rs.parameters)
  return frozendict({k: v for k, v in allDelegatorParameters.items() if k in
     ↪ allDelegatedPermissionParameters})
```

Authorization Decision

The authorization decision, first, it checks the assigned permissions (via direct and hierarchy), in case the needed permissions are not there, or the values bound

to permission parameters are not enough to authorize, it tries to authorize via the delegation relation. The following code fragment shows a sample validator:

```
def validator(self, authPolicy: AuthorizationPolicy, delegation: Delegation, user: User,
    ↪ loanAmount: float, loanType: str):
  if self in authPolicy.activePermissionsOfUser(user):
    userParamBindings = authPolicy.parameterBindingsOfUser(user)
    if loanAmount <= userParamBindings[self.amount] and loanType in
     ↪ userParamBindings[self.loanTypes]:
      return True

  if self in delegation.delegatedPermissionsOfUser(user):
    userParamBindings = delegation.delegatedParameterBindingsOfUser(user)
    if loanAmount <= userParamBindings[self.amount] and loanType in
     ↪ userParamBindings[self.loanTypes]:
  return True
```

D.2 UML Model

In this section, we include the formal model of OSDM for completeness. We use the Unified Modeling Language (UML) [76] to formalize the definitions of OSDM. Fig. D.1. shows the UML class diagram of OSDM. The diagram projects the relationships between the different components and relations of the ROAC model extended with the OSDM model that was explained in the previous section. The *DelegationRequest* represents the initiated delegation request. The *ApprovalRequest* represents the requests for approving the delegation request and then assigning the delegated role to the user by delegation. The diagram also depicts the two types of role assignments: the original and delegation assignments.

If multi-step delegation is not allowed, then the link between *DelegationRequest* and *UserRoleAssignment* in Fig. D.1 must be changed to be between *DelegationRequest* and *Original* class. The link between *UserRoleAssignment* and *DelegationRequest* indicates that the delegation request can only be initiated if the role to be delegated is assigned to the delegator.

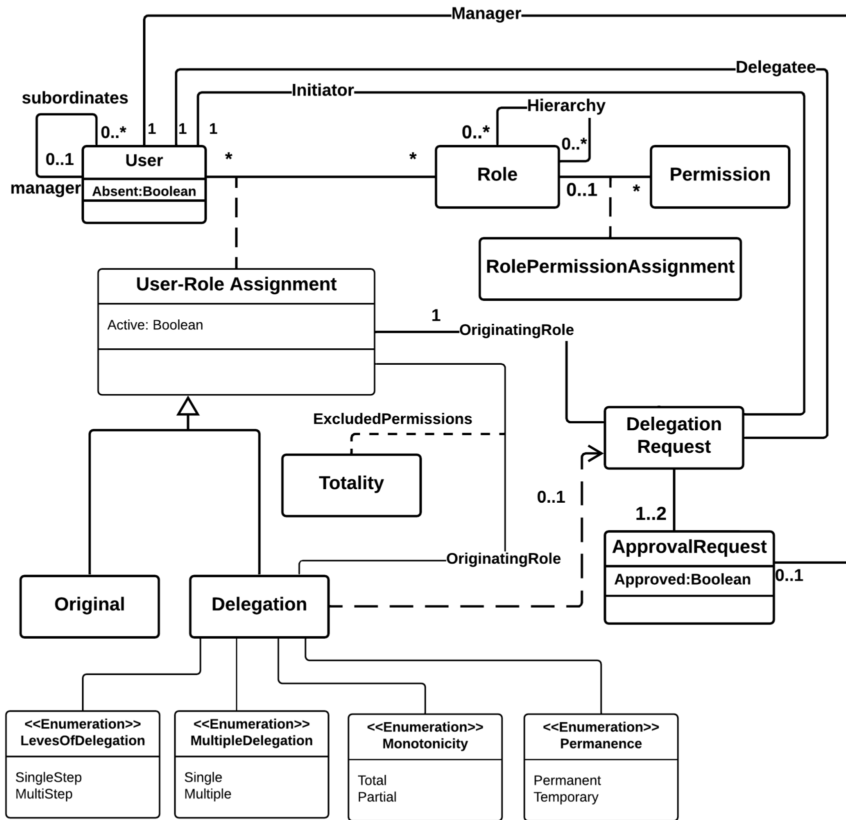


Figure D.1: The UML diagram of OSDM

Appendix E

Conflicts of Interest Formal Model

Conflict of interest policies enforcement is triggered once the policies are designed at a high-level. Policy enforcement comprises enforcement design and implementation of the policies. The approach we take to enforcement of policies segregates the policies enforcement from the task business logic. This is achieved by an expression evaluation engine that automatically enforces the specified policies. Enforcement using the expressions evaluation engine comprises two steps. In the first step, the expression evaluation engine validates the order of the task steps and generates a new task instance history ($U_{ordered}$) from (U_{past}), which is then used by the second step to validate the policy algebraic expression.

E.1 The History Data Structure

Many conflict of interest policies reference historical events. The activities that were performed and the identity of who has performed them are recorded and are made available for consultation by security policies requiring history. Examples of policies that require historical data are operational and historical separation of duties. For example, in the conflict of interest policies presented in our motivating example, it is required in policy 24 that the user authorizing the payment is being different from the user who initiated the payment. To monitor this, we need to record the user who has initiated the payment instance.

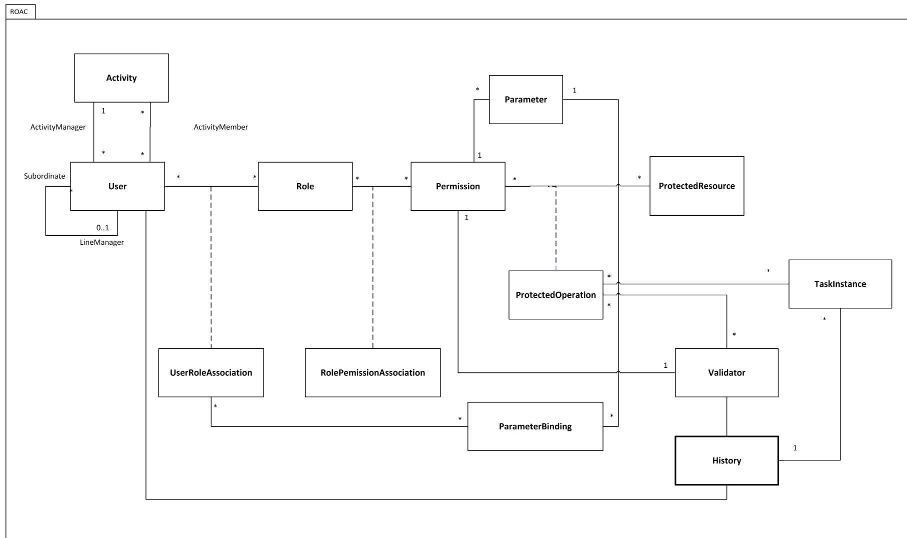


Figure E.1: The UML diagram of the ROAC model with history data structure

When a user subsequently attempts to authorize the payment, we can consult the history to verify that the two activities are not performed by the same user.

We extended the ROAC model with a history data structure, which contains records of events for each task instance. The events contain steps of the task instance and the user who has invoked it. Each task instance can be identified by a unique identifier, for example each payment instance should have a unique reference number (usually called the transaction reference). The ROAC model with the history data structure extension is shown in the UML diagram in Fig. E.1.

The history data structure is populated after each successful task step execution. The task step events can be recorded in the data structure in the permissions validators. After successful completion of a task workflow, the history data structure must have the actual workflow the task instance has followed.

The following constraints are applicable on the history data structure:

Constraint 20. *Each event represents a single workflow step*

Constraint 21. *Events must be ordered according to their execution order.*

Constraint 22. *Each event must contain, the unique reference of the task instance, the user who executed the step, and the step reference or name. Events can contain other optional data elements such as timestamps.*

The following Python code elaborates the definition of the history data structure in the ROAC model. The data structure is modeled by a dictionary of tuple of dictionaries. The dictionary key is the reference of the task instance. The value is a tuple that contains dictionaries of user and task step pairs.

```
Upast: frozendict[str, tuple[dict[User, str]]]
```

Example:

```
Upast = frozendict({
  "12345": (
  {U1: "Initiate"},
  {U2: "Verify"},
  {U3: "Modify"},
  {U4: "Verify"},
  {U5: "Authorize"}
  ),
  "67890": (
  {U1: "Initiate"},
  {U2: "Verify"},
  {U3: "Modify"},
  {U5: "Authorize"}
  )
})
```

E.2 Enforcement of Task Steps Order

Before getting into the details of enforcement of task steps order, we start be defining both workflow sequenced steps and workflow ordered steps.

1- Sequenced steps:

   Sequenced steps means that order of these steps must be respected anywhere in the task workflow for all sequenced steps. For example, if we specify that *verify* and *authorize* are sequenced steps, then *verify* must be always followed by an authorize step. If a workflow includes multiple *verify* steps, of which at least one is not followed by an *authorize* step, then the whole task workflow is considered invalid. The sequenced steps are defined between braces (), e.g. Verify, Authorize

2- Ordered steps:

> Ordered steps means that the order specified of a set of given steps must be respected at least once in the task workflow. Moreover, ordered steps must be specified at the end of the task steps order. For example, if we specify that *verify* and *authorize* are ordered steps, then the task workflow can contain different *verify* and *authorize* steps, the last two steps of the task workflow must be a *verify* followed by an *authorize*.
>
> Ordered steps does not represent any value from SoD perspective if they are followed by *ANY*. The main idea behind ordered steps, is to determine who are the users who must be the last to see the task before it is executed, which aims to guarantee that nothing has changed after their involvement.

The validation of the task steps order firstly validates the sequenced steps in U_{past}, if the order of sequenced steps is not respected, the whole task instance gets rejected. Secondly, the ordered steps are validated. The order must exist at the end of U_{past}. If order is not respected, the workflow is not rejected, however, it stays invalid until new steps are added to U_{past} that satisfy the order. Finally, a new task instance history is generated $U_{ordered}$, in which all steps of U_{past} related to ordered steps that do not respect the ordered steps are reduced. This is achieved by eliminating the task step associated with the user so that such steps wont be used for validation of algebraic expressions.

The history data structure is populated after each successful task step execution. The task step events can be recorded in the data structure in the permissions validators. After successful completion of a task workflow, the history data structure must have the actual workflow the task instance has followed. The following constraints are applicable on the history data structure: Constraint 20. Each event represents a single workflow step Constraint 21. Events must be ordered according to their execution order. Constraint 22. Each event must contain, the unique reference of the task instance, the user who executed the step, and the step reference or name. Events can contain other optional data elements such as timestamps.

Example:

Consider the U_{past} example provided in the previous section, and consider the following task steps order:

```
{ANY, {modify, Verify}, ANY, Verify, Authorize}
```

The second step in U_{past} is a *verify*, which does not respect the order, therefore, it is eliminated. The *verify* step at the fourth step is maintained since it respects the specified order. Therefore, $U_{ordered}$ will be as follows:

```
({U1: Initiate}, {U2: None}, {U3: Modify}, {U4: Verify}, {U5: Authorize})
```

The following Python code listing shows the *validateTaskOrder* method, which validates the task steps order and generates $U_{ordered}$:

```
class ConflictOfInterest:
  Upast: frozendict[str, tuple[dict[User, str]]]

  def validateTaskOrder(self, taskOrder: List, taskID: str) −> tuple[dict[User, str]]:
    UpastTask = list(self.Upast[taskID])
    assert len(taskOrder) >= 1
    UpastActivityList = [v for rs in self.Upast[taskID] for v in rs.values()]
    if len(taskOrder) == 1 and taskOrder[1] == "ANY":
      return self.Upast[taskID]
    else:
      # Validate sequenced steps
      for item in taskOrder:
        if isinstance(item, list) and len(item) > 1:
          for i in range(len(UpastActivityList)):
            if UpastActivityList[i] == item[0]:
              subList = UpastActivityList[i:i + len(item)]
              assert subList == item
              break
      # Validate ordered steps and generate Uordered
      Uordered = list(self.Upast[taskID])
      flatTaskOrder = [item for sublist in taskOrder for item in sublist]
      idxLastAnyOrder = len(flatTaskOrder) − flatTaskOrder[::−1].index("ANY") − 1
      if idxLastAnyOrder == len(flatTaskOrder) − 1:
        return tuple(UpastTask)
      else:
        lastSteps = flatTaskOrder[idxLastAnyOrder + 1:len(flatTaskOrder)]
        idxLastAnyUpast = len(UpastActivityList) − flatTaskOrder[::−1].index("ANY") −
  ↪ 1
        assert lastSteps == UpastActivityList[idxLastAnyUpast + 1: len(UpastActivityList
  ↪ )]
        for i in range(len(UpastTask[0: idxLastAnyUpast])):
          if list(UpastTask[i].values())[0] in lastSteps:
            Uordered[i] = {list(UpastTask[i].keys())[0]: None}
        return tuple(Uordered)
```

E.3 Enforcement of Algebraic Expressions

Validation of algebraic expressions checks if the ordered actual task instance steps ($Uordered$) satisfies the conflict of interest policy expression. A prerequisite to validation is to represent the algebraic expression in a binary tree structure. The operands of the expressions (i.e., ALL, users, roles, task variables) and the unary operators associated to them (e.g. $\neg$) appear in the leaf nodes of the tree, while interior nodes represent the binary operators (i.e., $\sqcap$, $\sqcup$, $\otimes$, and $\odot$).

Fig. E.2. depicts a binary tree representation of expression (7.3).

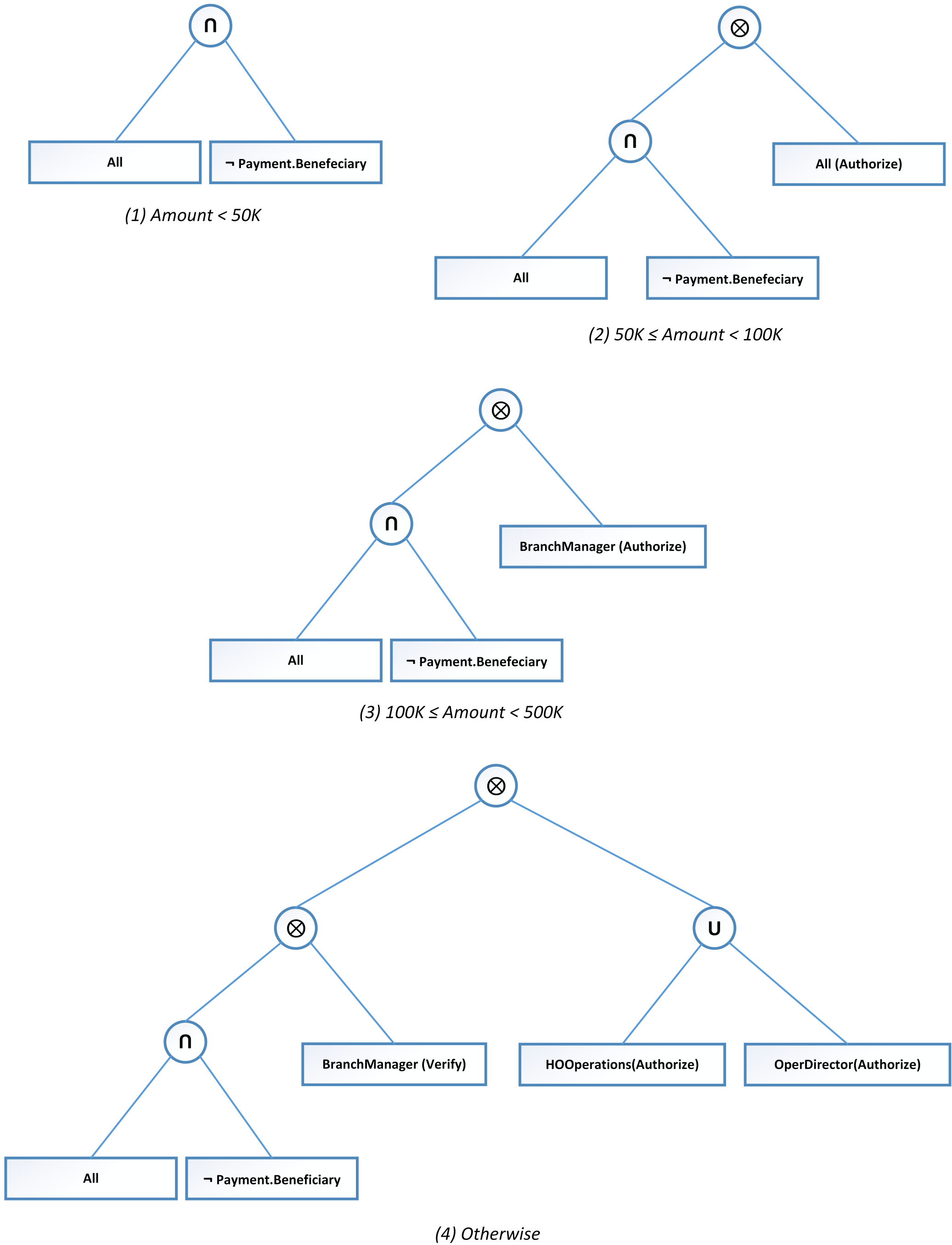


Figure E.2: Binary tree representation of expression (7.3)

The binary tree node is modeled as follows in Python:

```
class Node:
  def __init__(self, value):
    self.left = None
    self.value = value
    self.right = None
```

Since evaluation of the algebraic expression is done against $U_{ordered}$, which includes users and their corresponding task steps, all leaf expressions must be first evaluated to their corresponding configurations (sets of users). Five different operands can be encountered: *All*, a user, a role, a black list of users, a black list of roles, or a black list of task variable expressions. The black lists are defined using the unary operator ¬, e.g., $\neg\{Payment.Sender\}$, $\neg\{BranchManager, Teller\}$, $\neg\{Uer1, User2\}$. The leaf evaluation converts the operands to a set of users by selecting users from $U_{ordered}$ that satisfy the operand term.

E.3.1 Examples of leaf evaluation

Given the following $U_{ordered}$:

```
({U1: 'Initiate'}, {U2: None}, {U3: 'Modify'}, {U4: 'Verify'}, {U5: 'Authorize'})
```

Table E.1 below shows the user-role assignments of all users involved in the task instance (U_{past}).

Table E.1: User-role assignments

User	Roles
U1	SecuritiesClerk
U2	BranchManager, Teller
U3	Teller
U4	BranchManager
U5	OperDirector

(1) The operand term $BranchManager(Verify)$ evaluates to the set of users: $\{U4\}$. U2 is excluded because its *verify* step was eliminated.

(2) The operand term $\neg\{U2, U3\}$ evaluates to the set of users: $\{U1, U4, U5\}$

(2) The operand term $\neg\{BranchManager, OperDirector\}$ evaluates to the set of users: $\{U1, U3\}$.

The following Python code listing shows how the evaluation of leaves is achieved. The leaf is represented as a class of operand data, type of the expression, which must be one of the following possible values: *All*, *Role*, *User*, *UserBlackList*, *RoleBlackList*, *TaskExpressionBlackList*, the task step associated with the operand variable, and the operand value (set of users) which is calculated automatically.

```
class Leaf:
  value: set

  def __init__(self, data: object, expressionType: str, taskStep, parameters: frozendict[
    ↪ str, object], authPolicy: AuthorizationPolicy, Uordered: tuple[dict]):
    self.data = data
    self.type = expressionType
    self.taskStep = taskStep
    self.value = set()

    if expressionType == "ALL":
      self.value = set([v for rs in Uordered for v in rs.keys()])

    if expressionType == "User":
      for item in Uordered:
        if self.taskStep is not None:
          if self.taskStep in item.values() and self.data in item.keys():
            self.value.update(item.keys())
        else:
          if self.data in item.keys():
            self.value.update(item.keys())

    elif expressionType == "Role":
      for item in Uordered:
        if self.taskStep is not None:
          if self.taskStep in item.values() and self.data in authPolicy.rolesAssignedToUser(
    ↪ set(item.keys()).pop()):
            self.value.update(item.keys())
        else:
          if self.data in authPolicy.rolesAssignedToUser(set(item.keys()).pop()):
            self.value.update(item.keys())

     elif expressionType == "UserBlackList":
      if self.taskStep is not None:
        taskUsers = set([v for rs in Uordered for v in rs.keys() if taskStep in rs.values()])
        self.value = taskUsers.difference(self.data)
      else:
        taskUsers = set([v for rs in Uordered for v in rs.keys()])
        self.value = taskUsers.difference(self.data)

    elif expressionType == "RoleBlackList":
```

```
      if self.taskStep is not None:
        taskUsers = set([v for rs in Uordered for v in rs.keys() if taskStep in rs.values()])
        for user in taskUsers:
          if len(authPolicy.rolesAssignedToUser(user).intersection(self.data)) == 0:
            self.value.add(user)
      else:
        taskUsers = set([v for rs in Uordered for v in rs.keys()])
        for user in taskUsers:
          if len(authPolicy.rolesAssignedToUser(user).intersection(self.data)) == 0:
            self.value.add(user)

    elif expressionType == "TaskExpressionBlackList":
      usersBlackList = set()
      for item in self.data:
        usersBlackList.add(parameters[item])
      if self.taskStep is not None:
        taskUsers = set([v for rs in Uordered for v in rs.keys() if taskStep in rs.values()])
        self.value = taskUsers.difference(usersBlackList)
      else:
        taskUsers = set([v for rs in Uordered for v in rs.keys()])
        self.value = taskUsers.difference(usersBlackList)
```

For example, the operand: $\neg\{BranchManager(Verify)\}$ is represented as follows:

```
leaf = Leaf(data=branchManager, expressionType="Role", taskStep="Verify",
    ↪ parameters=None, authPolicy=authPolicy, Uordered=Uordered)
```

After evaluating the leaf nodes to sets of users, the algebraic expression can be validated by the method *evaluateExpressionTree*, which is part of class *ConflictOfInterest*, which was defined above. The method goes through all the nodes of the binary tree recursively and evaluates them.

```
def evaluateExpressionTree(self, root: Node) -> set:
  assert root is not None

  # A leaf node
  if root.left is None and root.right is None:
    return root.value

  # Evaluate left tree
  leftValue = self.evaluateExpressionTree(root.left)

  # evaluate right tree
  rightValue = self.evaluateExpressionTree(root.right)

  # Which operator to apply
  if root.value == 'Intersection':
    return leftValue.intersection(rightValue)
  elif root.value == 'Union':
    return leftValue.union(rightValue)
  elif root.value == 'OTimes':
    assert len(leftValue) > 0
    assert len(rightValue) > 0
    return set([(x, y) for x, y in itertools.product(leftValue, rightValue) if x != y])
  elif root.value == 'ODot':
    assert len(leftValue) > 0
    assert len(rightValue) > 0
    return set(itertools.product(leftValue, rightValue))
```

E.4 Example Policy Enforcement

In this example, we show how to enforce policy (25) of our motivating example. The policy is applicable on workflows of payments exceeding *500K EUR*, and the algebraic expression of the policy is given in the following expression:

$(((ALL \cap \neg\{Payment.beneficiary\}) \otimes BranchManager(Verify)) \otimes$
$(RegionalOperMgr(Authorize) \sqcap OperDirector(Authorize)))$

The task order associated with the expression is: $[ANY, Verify, Authorize]$.

The user-role assignments are given in Table E.1 above.

The actual task history of an example task instance (U_{past}) is given below:

Table E.2: Parameter bindings of the task instance variables

Parameter	**Value**
Amount	600K
Payment.Beneficiary	U3

```
Upast = frozendict({
  "12345": (
  {U1: "Initiate"},
  {U2: "Verify"},
  {U3: "Modify"},
  {U4: "Verify"},
  {U5: "Authorize"}
  )
})
```

The task instance variable parameters are given in Table E.2.

The task parameters can be defined as follows:

```
taskParameters = frozendict({
  "Amount": 600000,
  "Payment.Beneficiary": U3
})
```

The first step of enforcement is validation of correctness of the task steps order ($Upast$) and to generate $U_{ordered}$. The following code listing shows the order validation step:

```
taskOrder = [{'ANY'}, {"Verify"}, {"Authorize"}]
coi = ConflictOfInterest(Upast)
Uordered = coi.validateTaskOrder(taskOrder=taskOrder, taskID="12345")
```

The generated $U_{ordered}$ is:

```
({U1: 'Initiate'}, {U2: None}, {U3: 'Modify'}, {U4: 'Verify'}, {U5: 'Authorize'})
```

After validating the order and generating $U_{ordered}$, we can now validate the algebraic expression. The first step is to define the leaves of the tree. Afterwards,

the expression is represented in binary tree structure, as given in the below code listing:

```
# Define the leaves
leaf1 = Leaf(data=None, expressionType="ALL", taskStep=None, parameters=None,
    ↪ authPolicy=authPolicy, Uordered=Uordered)
leaf2 = Leaf(data={"Payment.Beneficiary"}, expressionType="TaskExpressionBlackList",
    ↪  taskStep=None, parameters=taskParameters, authPolicy=authPolicy, Uordered
    ↪ =Uordered)
leaf3 = Leaf(data=branchManager, expressionType="Role", taskStep="Verify",
    ↪ parameters=None, authPolicy=authPolicy, Uordered=Uordered)
leaf4 = Leaf(data=regionalOperMgr, expressionType="Role", taskStep="Authorize",
    ↪ parameters=None, authPolicy=authPolicy, Uordered=Uordered)
leaf5 = Leaf(data=operDirector, expressionType="Role", taskStep="Authorize",
    ↪ parameters=None, authPolicy=authPolicy, Uordered=Uordered)

# Construct the tree
root = Node('OTimes')
root.left = Node('OTimes')
root.left.left = Node('Intersection')
root.left.left.left = Node(leaf1.value)
root.left.left.right = Node(leaf2.value)
root.left.right = Node(leaf2.value)
root.left.right = Node(leaf3.value)
root.right = Node('Union')
root.right.left = Node(leaf4.value)
root.right.right = Node(leaf5.value)
```

Fig. E.3. shows the resulting evaluation of each node of the tree. The final result of the evaluation is determined by the value of the root node. The expression evaluation fails if the set value of the root node is empty.

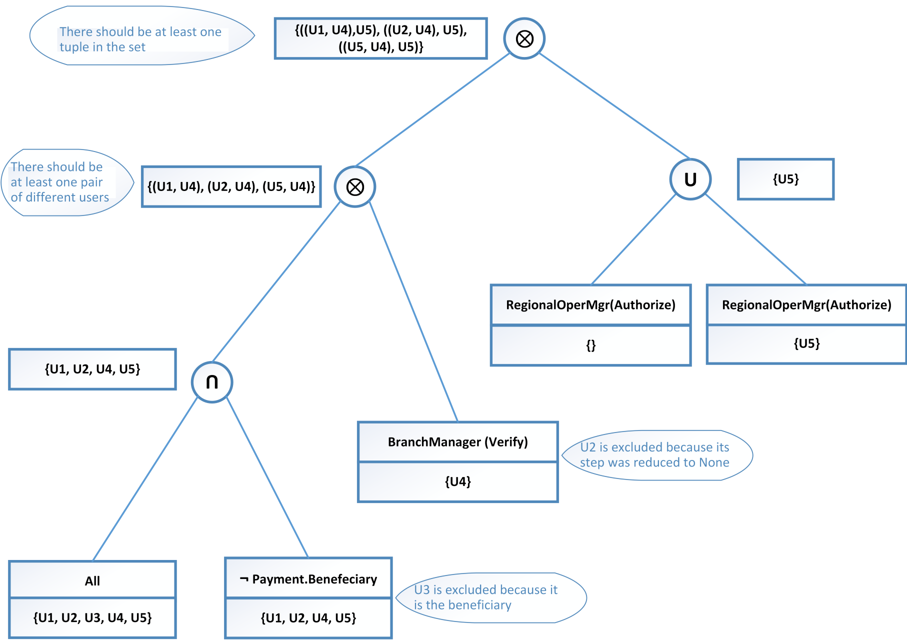


Figure E.3: Evaluation of algebraic expression for policy of a payment task exceeding 500K

Bibliography

[1] *Best Practices in Enterprise Authorization: The Next Generation Access Control with RBAC/ABAC Hybrid Model.* EmpowerID: White Paper.

[2] Cambridge university press: Cambridge dictionary. `https://dictionary.cambridge.org/dictionary/english-french/context`.

[3] Crime in England and Wales: year ending September 2021. `https://www.ons.gov.uk/peoplepopulationandcommunity/crimeandjustice/bulletins/crimeinenglandandwales/yearendingseptember2021`.

[4] *INCITS 359-2004: Role Based Access Control.* American National Standards Institute, Inc.

[5] *INCITS 359-2012: Role Based Access Control.* American National Standards Institute, Inc.

[6] *INCITS 494-2012: Role Based Access Control - Policy Enhanced.* American National Standards Institute, Inc.

[7] Abdallah, A., and Khayat, E. A formal model for parameterized role-based access control. In *Formal Aspects in Security and Trust*, T. Dimitrakos and F. Martinelli, Eds., vol. 173 of *IFIP International Federation for Information Processing.* Springer US, 2005, pp. 233–246.

[8] Ahn, G.-J., and Sandhu, R. Role-based authorization constraints specification. *ACM Transactions on Information and System Security 3*, 4 (nov 2000), 207–226.

[9] ANSI INCITS 359. Standard for role based access control. *American Nat'l Standard for Information Technology* (2004).

[10] Barka, E. *Framework for Role-Based Delegation Models.* PhD thesis, George Mason University, 6 2002.

[11] Barka, E., and Sandhu, R. Framework for role-based delegation models. In *In Proceedings of the 16th Annual Computer Security Applications Conference* (2000), ACSAC '00, IEEE Computer Society, pp. 168–176.

[12] Barka, E., and Sandhu, R. A role-based delegation model and some extensions. In *Proceedings of 23rd National Information System Security Conference* (Baltimore, Maryland, United States, 2000), NISSC, pp. 101–114.

[13] Basile, C., Cappadonia, A., and Lioy, A. Algebraic models to detect and solve policy conflicts. In *Computer Network Security* (Berlin, Heidelberg, 2007), V. Gorodetsky, I. Kotenko, and V. A. Skormin, Eds., Springer Berlin Heidelberg, pp. 242–247.

[14] Basin, D., Burri, S. J., and Karjoth, G. Dynamic enforcement of abstract separation of duty constraints. In *Computer Security – ESORICS 2009* (Berlin, Heidelberg, 2009), M. Backes and P. Ning, Eds., Springer Berlin Heidelberg, pp. 250–267.

[15] Bell, D., and LaPadula, L. Secure computer systems: Unified exposition and multics interpretation. In *Technical Report, MTR-2997, The Mitre Corp., Bedford, Mass.* (1976).

[16] Bertino, E., Bonatti, P. A., and Ferrari, E. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security 4*, 3 (Aug. 2001), 191–233.

[17] Bertino, E., Catania, B., Damiani, M. L., and Perlasca, P. Geo-rbac: A spatially aware rbac. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2005), SACMAT '05, ACM, pp. 29–37.

[18] Bertino, E., Ferrari, E., and Atluri, V. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security 2*, 1 (Feb. 1999), 65–104.

[19] Bhatt, S., Patwa, F., and Sandhu, R. Abac with group attributes and attribute hierarchies utilizing the policy machine. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control* (New York, NY, USA, 2017), ABAC '17, Association for Computing Machinery, p. 17–28.

[20] Chakraborty, S., Sandhu, R., and Krishnan, R. On the feasibility of rbac to abac policy mining: A formal analysis. In *Secure Knowledge Management In Artificial Intelligence Era* (Singapore, 2020), S. K. Sahay, N. Goel, V. Patil, and M. Jadliwala, Eds., Springer Singapore, pp. 147–163.

[21] Chambers, C., Ungar, D., Chang, B., and Holzle, U. Parents are shared parts of objects: inheritance and encapsulation in self. *LISP and Symbolic Computation* (1991), 207–222.

[22] Chinaei, A. H., Chinaei, H. R., and Tompa, F. W. A unified conflict resolution algorithm. In *Secure Data Management* (Berlin, Heidelberg, 2007), W. Jonker and M. Petković, Eds., Springer Berlin Heidelberg, pp. 1–17.

[23] Clark, D. D., and Wilson, D. R. A comparison of commercial and military computer security policies. In *1987 IEEE Symposium on Security and Privacy* (Los Alamitos, CA, USA, apr 1987), IEEE Computer Society, pp. 184–184.

[24] Coyne, E., and Weil, T. R. Abac and rbac: Scalable, flexible, and auditable access management. *IT Professional 15*, 3 (2013), 14–16.

[25] Crampton, J. Administrative scope and role hierarchy operations. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2002), SACMAT '02, ACM, pp. 145–154.

[26] Crampton, J., and Khambhammettu, H. Data structures for constraint enforcement in role-based systems. In *Proceedings of the IASTED Conference on Network and Information Security* (2005), pp. 158–167.

[27] Crampton, J., and Khambhammettu, H. Delegation in role-based access control. In *Proceedings of the 11th European conference on Research in Computer Security* (2006), ESORICS 2006, Springer-Verlag, p. 174–191.

[28] Crampton, J., and Khambhammettu, H. Delegation in role-based access control. *International Journal of Information Security* (2008), 123–136.

[29] Crampton, J., and Loizou, G. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security 6*, 2 (May 2003), 201–231.

[30] Cuppens, F., Cuppens-Boulahia, N., and Miege, A. Inheritance hierarchies in the or-bac model and application in a network environment. In *In Proceedings of the 3rd Workshop on Foundations of Computer Security (FCS'04), Turku, Finland,* (2004).

[31] Davari, M., and Zulkernine, M. Policy modeling and anomaly detection in abac policies. In *Risks and Security of Internet and Systems*

(Cham, 2022), B. Luo, M. Mosbah, F. Cuppens, L. Ben Othmane, N. Cuppens, and S. Kallel, Eds., Springer International Publishing, pp. 137–152.

[32] Davari, M., and Zulkernine, M. Classification-based anomaly prediction in xacml policies. In *Security and Privacy in Communication Networks* (Cham, 2023), F. Li, K. Liang, Z. Lin, and S. K. Katsikas, Eds., Springer Nature Switzerland, pp. 3–19.

[33] Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic conflict resolution mechanisms for inheritance. In *In conference proceedings on Object-oriented programming systems, languages, and applications.* (1992), OOPSLA '92, ACM.

[34] Eckel, B. *Thinking in Java, 2nd Edition.* Pearson Education, 2000.

[35] eWeek. Rethinking access controls: How wikileaks could have been prevented. `http://www.eweek.com/c/a/Security/Rethinking-Access-Controls-How-WikiLeaks-Could-Have-Been-Prevented/1/`.

[36] Fernandez, E. B., Wu, J., and Fernandez, M. H. User group structures in object-oriented databases. In *Proceedings of the IFIP WG11.3 Working Conference on Database Security VII, Status and prospects, Bad Salzdetfurth, Germany* (1994), vol. 60, pp. 57–76.

[37] Ferraiolo, D., and Kuhn, R. Role-based access controls. In *Reprinted from 15th National Computer Security Conference* (1992), Baltimore, Oct 13-16, pp. 554–563.

[38] Fischer, J., Marino, D., Majumdar, R., and Millstein, T. Fine-grained access control with object-sensitive roles. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Berlin, Heidelberg, 2009), Genoa, Springer-Verlag, pp. 173–194.

[39] Griffiths, P. P., and Wade, B. W. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems 1*, 3 (Sept. 1976), 242–255.

[40] Harris, M., and Raviv, A. Organization design. *Management Science 48*, 7 (July 2002), 852–865.

[41] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. Protection in operating systems. *Communications of the ACM 19*, 8 (Aug. 1976), 461–471.

[42] Hasebe, K., Mabuchi, M., and Matsushita, A. Capability-based delegation model in rbac. In *In Proceedings of the 15th ACM symposium on Access control models and technologies* (New York, NY, USA, 2010), SACMAT '10, ACM, pp. 109–118.

[43] Horcas, J.-M., Pinto, M., and Fuentes, L. Closing the gap between the specification and enforcement of security policies. In *Trust, Privacy, and Security in Digital Business* (Cham, 2014), C. Eckert, S. K. Katsikas, and G. Pernul, Eds., Springer International Publishing, pp. 106–118.

[44] Hu, V. C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., and Scarfone, K. Guide to attribute based access control (abac) definition and considerations, 2014.

[45] Hu, V. C., Kuhn, R., and Yaga, D. Verification and test methods for access control policies/models. In *NIST Special Publication 800-192* (2017), National Institute of Standards and Technology.

[46] Hu, V. C., and Scarfone, K. Guidelines for access control system evaluation metrics. In *NISTIR 7874* (2012), National Institute of Standards and Technology.

[47] Jaeger, T., Michailidis, T., and Rada, R. Access control in a virtual university. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. (WET ICE '99) Proceedings.* (1999), pp. 135–140.

[48] Jajodia, S., Samarati, P., Sapino, M. L., and Subrahmanian, V. S. Flexible support for multiple access control policies. *ACM Transactions on Database Systems 26*, 2 (June 2001), 214–260.

[49] Jebbaoui, H., Mourad, A., Otrok, H., and Haraty, R. Semantics-based approach for detecting flaws, conflicts and redundancies in xacml policies. *Computers and Electrical Engineering 44* (2015), 91–103.

[50] Kalam, A., Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miege, A., Saurel, C., and Trouessin, G. Organization based access control. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on* (2003), pp. 120–131.

[51] Lampson, B. W. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University* (1971), pp. 437–443.

[52] LEE, H., LEE, Y., AND NOH, B. A framework for modeling organization structure in role engineering. In *In Proceedings of the 7th international conference on Applied Parallel Computing (PARA'04), Springer-Verlag, . Berlin, Heidelberg* (2004), pp. 1017–1024.

[53] LI, J., WANG, Q., WANG, C., AND REN, K. Enhancing attribute-based encryption with attribute hierarchy. In *2009 Fourth International Conference on Communications and Networking in China* (2009), pp. 1–5.

[54] LI, N., TRIPUNITARA, M., AND BIZRI, Z. On mutually exclusive roles and separation-of-duty. *ACM Transactions on Information and System Security* (2007).

[55] LI, N., AND TRIPUNITARA, M. V. Security analysis in role-based access control. *ACM Transactions on Information and System Security 9*, 4 (nov 2006), 391–420.

[56] LI, N., AND WANG, O. Beyond separation of duty: An algebra for specifying high-level security policies. *Journal of the ACM* (2008).

[57] LI, N., AND WANG, Q. Beyond separation of duty: An algebra for specifying high-level security policies. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, ACM, pp. 356–369.

[58] LI, Q., ZHANG, X., QING, S., AND XU, M. Supporting ad-hoc collaboration with group-based rbac model. In *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on* (Nov 2006), pp. 1–8.

[59] LIANG, D. *Introduction to Java Programming, Comprehensive Version.* Prentice Hall, 2006.

[60] LIU, G., PEI, W., TIAN, Y., LIU, C., AND LI, S. A novel conflict detection method for abac security policies. *Journal of Industrial Information Integration 22* (2021), 100200.

[61] LUONG, T.-N., LE, H.-A., VO, D.-H., AND TRUONG, N.-T. A framework to verify the abac policies in web applications. In *Intelligence of Things: Technologies and Applications* (Cham, 2022), N.-T. Nguyen, N.-N. Dao, Q.-D. Pham, and H. A. Le, Eds., Springer International Publishing, pp. 124–133.

[62] MAJETIC, I., AND LEISS, E. L. Authorization and revocation in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering 9*, 4 (1997), 668–672.

[63] Miege, A. *Definition of a formal framework for specifying security policies. The Or-BAC model and extensions.* PhD thesis, Ecole Nationale Superieure des Telecommunications, Paris, France, 6 2005.

[64] Moffett, J. D., and Lupu, E. C. The uses of role hierarchies in access control. In *Proceedings of the Fourth ACM Workshop on Role-based Access Control* (New York, NY, USA, 1999), RBAC '99, ACM, pp. 153–160.

[65] Montana, P., and Charnov, B. Management: A streamlined course for students and business people. *Barron's Business Review Series* (1993), 155–169.

[66] Moore, D., and Cain, D. *Conflicts of Interest: Challenges and Solutions in Business, Law, Medicine, and Public Policy.* Cambridge University Press, 2005.

[67] Morisset, C., Willemse, T. A. C., and Zannone, N. Efficient extended abac evaluation. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies* (New York, NY, USA, 2018), SACMAT '18, Association for Computing Machinery, p. 149–160.

[68] Nassr, N., Aboudagga, N., and Steegmans, E. Osdm: An organizational supervised delegation model for rbac. In *Proceedings of the 15th International Conference on Information Security* (Berlin, Heidelberg, 2012), ISC'12, Springer-Verlag, pp. 322–337.

[69] Nassr, N., and Steegmans, E. A parameterized rbac access control model for ws-bpel orchestrated composite web services. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for* (Dec 2011), pp. 122–127.

[70] Nassr, N., and Steegmans, E. Roac: A role-oriented access control model. In *nformation Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems* (Berlin, Heidelberg, 2012), I. Askoxylakis, H. C. Pohls, and J. Posegga, Eds., Springer Berlin Heidelberg, pp. 113–127.

[71] OASIS. extensible access control markup language (xacml) version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html`.

[72] OASIS. A brief introduction to xacml. `https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html`, 2003.

[73] Oh, S., and Sandhu, R. A model for role administration using organization structure. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2002), SACMAT '02, ACM, pp. 155–162.

[74] OMG. Object constraint language, version 2.4. `http://www.omg.org/spec/OCL/2.4`.

[75] OMG. Omg unified modeling languagetm (omg uml), superstructure version 2.2. `http://www.omg.org/spec/UML/2.2/Superstructure`.

[76] OMG. Unified modeling language® (omg uml®). `https/www.omg.org/spec/UML/`.

[77] Paci, F., Bertino, E., and Crampton, J. An access-control framework for ws-bpel. *International Journal of Web Services Research 5*, 3 (2008), 20–43.

[78] Pilone, D., and Pitman, N. *UML 2.0 in a Nutshell.* O'Reilly Media, Inc., 2005.

[79] Rajpoot, Q. M., Jensen, C. D., and Krishnan, R. Attributes enhanced role-based access control model. In *Trust, Privacy and Security in Digital Business* (Cham, 2015), S. Fischer-Hübner, C. Lambrinoudakis, and J. opez, Eds., Springer International Publishing, pp. 3–17.

[80] Sandhu, R. Separation of duties in computerized information systems. In *Proc. of the IFIP WG11.3 Workshop on Database Security* (1990).

[81] Sandhu, R. The typed access matrix model. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), SP '92, IEEE Computer Society, pp. 122–.

[82] Sandhu, R. Role activation hierarchies. In *Proceedings of the Third ACM Workshop on Role-based Access Control* (New York, NY, USA, 1998), RBAC '98, ACM, pp. 33–40.

[83] Sandhu, R. Future directions in role-based access control models. In *Information Assurance in Computer Networks*, V. Gorodetski, V. Skormin, and L. Popyack, Eds., vol. 2052 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2001, pp. 22–26.

[84] Sandhu, R., and Bhamidipati, V. The ura97 model for role-based user-role assignment. In *Database Security XI: Status and Prospects, IFIP Advances in Information and Communication Technology* (Boston, MA, 1998), T. Y. Lin and S. Qian, Eds., Springer US, pp. 262–275.

[85] Sandhu, R., and Bhamidipati, V. The ascaa principles for next-generation role-based access control. In *Proceedings 3rd International Conference on Availability, Reliability and Security (ARES)* (2008), ARES '08.

[86] Sandhu, R., Bhamidipati, V., Coyne, E., Ganta, S., and Youman, C. The arbac97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (New York, NY, USA, 1997), RBAC '97, ACM, pp. 41–50.

[87] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[88] Sandhu, R., and Munawer, Q. The rra97 model for role-based administration of role hierarchies. In *Proceedings 14th Annual Computer Security Applications Conference (Cat. No.98EX217)* (1998), pp. 39–49.

[89] Sandhu, R., and Munawer, Q. The arbac99 model for administration of roles. In *Proceedings of the 15th Annual Computer Security Applications Conference* (Washington, DC, USA, 1999), ACSAC '99, IEEE Computer Society, pp. 229–.

[90] Schaad, A., Moffett, J., and Jacob, J. The role-based access control system of a european bank: A case study and discussion. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2001), SACMAT '01, ACM, pp. 3–9.

[91] Schermerhorn, J., Osborn, R., and Uhl-Bien, M. *Organizational Behavior, 12th edition.* Wiley, 2011.

[92] Servos, D., and Osborn, S. L. Hgabac: Towards a formal model of hierarchical attribute-based access control. In *Foundations and Practice of Security* (Cham, 2015), F. Cuppens, J. Garcia-Alfaro, N. Zincir Heywood, and P. W. L. Fong, Eds., Springer International Publishing, pp. 187–204.

[93] Servos, D., and Osborn, S. L. Current research and open problems in attribute-based access control. *ACM Computing Surveys 49*, 4 (Jan. 2017).

[94] Shafiq, B., Masood, A., Joshi, J., and Ghafoor, A. A role-based access control policy verification framework for real-time systems. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (2005), pp. 13–20.

[95] Shaikh, R. A., Adi, K., Logrippo, L., and Mankovski, S. Inconsistency detection method for access control policies. In *2010 Sixth International Conference on Information Assurance and Security* (2010), pp. 204–209.

[96] Sharghigoorabi, M. *Access Control Obligation Specification and Enforcement Using Behavior Pattern Language*. PhD thesis, University of Ontario Institute of Technology (UOIT), Oshawa, Canada, 1 2018.

[97] Shu, C.-c., Yang, E. Y., and Arenas, A. E. Detecting conflicts in abac policies with rule-reduction and binary-search techniques. In *2009 IEEE International Symposium on Policies for Distributed Systems and Networks* (2009), pp. 182–185.

[98] Simon, R., and Zurko, M. Separation of duty in role-based environments. In *Proceedings 10th Computer Security Foundations Workshop* (Rockport, MA, USA, 1997), CSFW '97, IEEE, pp. 183–194.

[99] St-Martin, M., and Felty, A. P. A verified algorithm for detecting conflicts in xacml access control rules. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (New York, NY, USA, 2016), CPP 2016, Association for Computing Machinery, p. 166–175.

[100] STANDARD, D. O. D. Department of defense trusted computer system evaluation criteria. *DoD 5200.28-STD* (1983).

[101] Thompson, D. Understanding financial conflicts of interest. *The new England journal of medicine 329* (1993), 573–579.

[102] Wang, X., Zhang, Y., Shi, H., and Yang, J. Bpel4rbac: An authorisation specification for ws-bpel. In *Proceedings of the 9th International Conference on Web Information Systems Engineering* (Berlin, Heidelberg, 2008), WISE '08, Springer-Verlag, pp. 381–395.

[103] Wood, C., and Fernandez, E. B. Decentralized authorization in a database system. In *Fifth International Conference on Very Large Data Bases, 1979.* (1979), pp. 352–359.

[104] Xuexiong, Y., Qinxian, W., and Changzheng, X. A multiple hierarchies rbac model. In *Communications and Mobile Computing (CMC), 2010 International Conference on* (April 2010), vol. 1, pp. 56–60.

[105] Yuan, E., and Tong, J. Attributed based access control (abac) for web services. In *IEEE International Conference on Web Services (ICWS'05)* (2005), p. 569.

[106] Zhang, L., Ahn, G., and Chu, B. A rule-based framework for role-based delegation. In *In Proceedings of ACM Symposium on Access Control Models and Technologies* (Chantilly, VA, 2001), SACMAT 2001, p. 153–162.

[107] Zhang, L., Ahn, G. J., and Chu, B. T. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security 6*, 3 (2003), 404–441.

[108] Zhang, X., Oh, S., and R., S. Pbdm: a flexible delegation model in rbac. In *In Proceedings of the eighth ACM symposium on Access control models and technologies* (New York, NY, USA, 2003), SACMAT '03, ACM, pp. 149–157.

[109] Zhang, Z. *Scalable Role and Organization Based Access Control and its Administration.* PhD thesis, George Mason University, Fairfax, VA, 2008.

[110] Zhixiong, Z., Xinwen, Z., and Sandhu, R. Robac: Scalable role and organization based access control models. In *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on* (2006), pp. 1–9.

Publications

Nezar Nassr, Eric Steegmans: Mitigating conflicts of interest by authorization policies. SIN '15: Proceedings of the 8th International Conference on Security of Information and Networks, September 2015, pp 118–126, Published by: ACM (was also accepted in INSCRYPT)

Nezar Nassr, Nidal Aboudagga, Eric Steegmans: OSDM: An Organizational Supervised Delegation Model for RBAC. Information Security Conference ISC-2012 , Passau, Germany, published by: Springer-Verlag

Nezar Nassr and Eric Steegmans, ROAC: A Role-Oriented Access Control Model, 6th Workshop in Information Security Theory and Practice (WISTP 2012), London, UK. Published by: Springer-Verlag.

Nezar Nassr, Eric Steegmans: A parameterized RBAC access control model for WS-BPEL orchestrated composite web services. ICITST 2011: 122-127, Published By IEEE

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven