A Formal Model to express Dynamic Policies for Access Control and Trust Negotiation in a Distributed Environment

Thesis presented and defended by

Marwa El Houri

On the 28th of May 2010

To obtain the degree of

Doctorat de l'Université de Toulouse

Delivered by: Université Toulouse III Paul Sabatier (UPS)
Major: Computer Science

Advisors:

Philippe Balbiani
Directeur de Recherche CNRS
Université Paul Sabatier

Yannick Chevalier
Maître de Conférences
Université Paul Sabatier

Reviewers:

Alban Gabillon
Professeur
Université de la Polynésie Française

Olga Kouchnarenko
Professeur
Université Franche-Comté

Members:

AbdelKader Hamerlain
(President of the Jury)
Professeur - Université Paul Sabatier

Laurent Vigneron
Maître de Conférences
Université Nancy 2

École doctorale:
Mathématiques Informatique Télécommunications de Toulouse (MITT)
Unité de recherche:
Institut de recherche en Informatique de Toulouse (IRIT)

To my grandfather
who couldn't see the end of the journey,
I hope you are proud...

Acknowledgements

The actual achievement of this project would have never been possible without the help, support and understanding of many people.

To my advisors Philippe Balbiani and Yannick Chevalier I am grateful for their assistance, useful advices, interesting discussions but mostly for there continuous presence and support throughout this journey.

I would also like to thank the jury members Alban Gabillon, AbdelKader Hamerlain, Olga Koushnarenko and Laurent Vigneron for their valuable advices and remarks.

My deepest gratitude and thanks to a dear friend Fahima Cheikh-Alili for the laughs, the cries and all the support she made available throughout this thesis. Thank you Fahima for this valuable friendship, I am happy that our life paths crossed during this journey.

It is a pleasure to thank François for the good mood he reflects on us all through laughs, music, and chocolate, and Nadine for the support and the constant positiveness. I will always cherish the good times we shared in office 302, the discussions, the good laughs and the coffee breaks. I thank my officemates Sihem, Elise, Srdjan and Emmanuel for making this a pleasant working place and my friends Wassim, Dania, Mounira, Salam, Pablo, Marija, Eric and Madelaine for their friendship and support. Without all of you this journey would have never been the same.

I would also want to thank my family, especially my parents for their great support, for their patience and understanding during the hard times, without their presence this project would not have been possible.

Finally, to Bilal my friend, officemate, love and husband a deep thank you for the constant support, patience and understanding. I am lucky to have you in my life.

Abstract

The development of the Internet and the wide acceptance of the Service-Oriented Architecture as a paradigm for integrating software applications within and across organizational boundaries led to a new form of of distributed structures. In this paradigm, independently developed and operated applications and resources are exposed as services that can communicate with one another by passing messages over HTTP, SOAP, etc. Such a paradigm offers the possibility to orchestrate services in order to create new composed services adapted to a given task. For business, security and legal reasons, it is necessary to control the access to these services as well as the collaboration and exchange of information between them while performing a common task.

The main objective of this thesis is to define a high level logical language that can express complex security policies within an access control framework.

The development of this language is done in three steps. First we define a role based dynamic framework where the state evolution of a service depends on the execution of its functionalities. Next we define an attribute based framework that gives more expressivity in terms of specification of access control conditions and add the notion of workflow that gives an order over the execution of the service functionalities and thus allows the definition of the general behavior of the service. Finally, in order to take into account the collaboration between different services we add a trust negotiation layer that allows each service to define its own exchange policy with respect to other services in the environment.

Having such a unified framework facilitates the safety analysis of the security policy since one can take into account the different factors that influence the access control decisions within the same framework. Thus the second objective of this thesis is to study the main access control features such as delegation and separation of duty properties on the one hand and the security features for the communication between the services at the trust negotiation level on the other hand. As such, in order to show the expressivity of the framework we present a generic encoding for the different concepts of delegation and revocation, we also give the specification for the different aspects of separation and binding of duty constraints and an encoding for the role-based access control model along with some of its extensions. Finally, in order to take into account a real communication network, we give an extension of our framework in order to be able to model for an active intruder that can intervene during a trust negotiation session by intercepting messages or constructing new messages.

Contents

Chapter 1

Introduction

Context

With the development of the Internet, and the spread of information on distributed structures, the collaboration between different services became a necessity. A service is an entity that is responsible for a given set of functionalities. It can be accessed by a human user or by other services. For example, a service connected to a virtual library allows users to search for available books. Another service with connection to a secured banking system offers the possibility for secure payment and a third service provides shipment advantages. If a user requests to buy a book online, a collaboration between these three services is necessary to achieve the user's task. A similar scenario can be found within an organization with decentralized departments.

In order to be able to collaborate, services must be able to communicate, exchange information and reach mutual agreements. There exists standard languages that ensures the security of messages such as SOAP [58] that defines the form of the messages through an envelop and regulate message transmission between services. Also, such services need to have a security policy that states who is authorized to access the service functionalities. Standard languages such as WS-SecurityPolicy [59] and WS-Policy [74] can define a security policy at the level of the messages. That is they are capable of expressing the criteria for a given service to accept or not a message from another service. Moreover, the service functionalities need to be executed in a given order to achieve a global task. This can be provided by BPEL [47], a standard language that specifies the interaction between services and thus allows the expression of a business process through the orchestration of services behavior. However such standard languages are essentially designed for the implementation of security policy and business process design and thus do not offer means to reason about policies.

Objectives and contributions of this thesis

The main objective of this thesis is to present a high level logical language that can express complex security policies within an access control framework.

An access control framework should guarantee the safety of the system with respect to predefined security requirements, but must also allow the feasibility of performing a check for the security of the system. In other words, a model should be able to provide a solution for the safety problem that, given an initial state and a security policy, checks whether it is possible when applying the access control policy to reach an unsafe state. The safety criteria are often defined with respect to the confidentiality or the integrity of the information but can also concern operational constraints, availability of the information, etc. In order to handle such problems, it is essential to define a mathematical framework that is able to express both access control rules and security objectives, and in which it will be possible to validate the properties of the system.

Usually, security policies are defined by a security modeler, i.e. a human being who expresses security requirements. The security language needs to be expressive enough to model such human designed security policies in a formal language, in the best possible manner, in order to avoid ambiguities.

Additionally, access control policies are applied in a specific environment and access control decisions lead to permitting or denying some actions. This has an effect on the policy environment. Thus it is also important to be able to specify the effects of access control authorizations within the access control framework. That is, in addition to providing yes or no answers for access requests, the security language should express the actual "change" that occurs when a request is granted (or denied).

Further, in a time where information is being decentralized, new aspects of access control are needed. An access control framework needs to take into account the distributed nature of the environment and the possibility of collaboration and communication between different access control systems that do not necessarily know each other. Thus it is important to have a security language that can express trust negotiation between the different actors of a collaboration in order to establish mutual trust.

Finally, current organizations often define an access control policy with respect to business processes. That is the collaboration between different services is ordered in a specified manner in order to achieve a final goal. Thus in order to express business processes, the security language should be able to define, in addition to the direct effects of authorized actions, the possibility to specify an order over these actions.

The first objective of this thesis is to formalize a unified framework that considers access control in an environment where entities communicate to accomplish a set of tasks with respect to their security policy. Such entities can be viewed as services, but also as users or organizations.

It is important to point out that several models exist that answer to some of the problems stated above ([45, 13, 52, 18, 12, 11]). However the contribution of this thesis is to provide a unified framework that takes into account the totality

of these features. In fact having a unified framework would facilitate the safety analysis of the security policy since one can take into account the different factors that influence the access control decisions within the same framework.

Thus the second objective of this thesis is to study the main access control features such as delegation and separation of duty properties on the one hand and the security features for the communication between the entities at the trust negotiation level on the other hand. The expression of such properties would lead to the specification of safety criteria for the unified framework in general. In particular we specify security constraints that can be checked at the security level and enforced within the workflow at run time level. We also specify authentication and confidentiality properties to be satisfied during a negotiation session when assuming the presence of a malicious entity.

Outline of the thesis

The thesis is presented in two parts. The first part is devoted for the study of existing access control models and the design of a logical language to express complex access control policies that can model dynamic features in a distributed environment. The second part presents some access control features that can be expressed in this new framework and specifies security properties that contribute to the reasoning about the safety of security problems.

In Chapter 2, we give an extended overview of existing access control models. We are interested in satisfying three main criteria, namely:

- the flexibility of the language to express complex access control policies, i.e. the possibility to express requirements that go beyond the simple subject, object action paradigm,
- the capacity of expressing dynamic access control, i.e. possibility of defining effects to authorized accesses and the modeling of security state evolution with respect to a well defined order of execution and
- the necessity of defining an interaction between different actors in the environment by defining a trust management mechanism allowing the establishment of mutual trust between actors that may not necessarily know each other.

We show the solutions provided by several access control models to some of these criteria and explain their limitations to motivate the need for a unified framework that takes into account the above three criteria. In Chapter 3 we present a first approach to define a dynamic access control model based on roles. The main importance of this model lies in the definition of dynamic policies to express the effects of the authorizations depending on whether or not an authorization was executed. It is extended in Chapter 4 to an attribute based framework able to satisfy the above mentioned three criteria. Also, the development of the unified framework was partially influenced by the analysis of various case studies in order to express security for business processes. In

Chapter 5, we present one such case study that models a car registration process in order to illustrate the expressivity of our framework.

In Chapter 6 we give an overview of the different concepts of delegation and revocation. We then give an encoding to express flexible delegation and revocation in our unified framework. Chapter 7 presents an overview of the separation of duty properties that demonstrates the expressivity of our framework both on the level of specification of security constraints and on the level of enforcement of such constraints at runtime. In Chapter 8 we give an encoding of the role-based access control model with some of its extensions and illustrate the access control features within this model. Chapter 9 gives a different perspective to security by assuming the presence of a malicious entity at the trust negotiation level. As such we give an extension of our logical language that takes into account the different types of communication channels and give the specifications for the authentication and confidentiality properties within these assumptions.

It is worth to mention that this thesis was partially funded by the FP7-ICT-2007-1 Project no. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures[1]".

[1] www.avantssar.eu

Part I

Dynamic access control models

Chapter 2

Related Works

Security has always been considered as a necessity for human beings and access control decisions or authorizations always existed in real life situations whenever humans found the need to protect their assets or belongings. With the rise of technology, and the development of information technology, there was an increase in the dependence on computers in decision making. Thus more effort was invested in research concerning access control in order to regulate access to sensitive information and protect system resources. In general, access control decisions or authorizations refer to yes or no decisions concerning "who can do what" questions.

An access control framework regulates the access to functionalities (be it web services, processes or companies) by authorized users depending on an access control policy. In its simplest form, this relation is represented by a list associating authorized users to resources. When a user provides sufficient proof for his identity, an authorization decision can be achieved with respect to user-resource relationship. Access control is defined through the notions of *users*, *subjects*, *objects* and *actions* (also called operations or tasks in the literature). A user is a human-agent interacting with a machine, a subject is an *active* entity possibly acting on behalf of a user, or on behalf of a machine (a service). An object is a *passive* entity that can refer to a resource, a database or any information and can be used or accessed by subjects and an *action* is an active process performed by a subject on a given object.

In this chapter we give a historical overview of the first access control models in Section 2.1, then we present the role-based access control in Section 2.2. In Sections 2.3 and 2.4 we present two extensions of RBAC, namely FAF and OrBAC that offer more flexibility than the RBAC structure. In Section 2.5 we present XACML, a standard language to express access control policies, and in Section 2.6 we present more recent role-based access control frameworks that handle security policy in distributed domains. Section 2.7 presents SecPAL a dynamic logical language based on attributes and Section 2.8 defines additional dynamic languages. In Section 2.9 we present a glimpse of some trust management models and we conclude in Section 2.10.

2.1 An overview of early access control models

With the increased use of computers to access resources and the need to share resources among users came the need to formalize an access control model that regulates the access to shared resources and protect sensitive information from potential leaks. This was the concern of the US department of defense (DoD) that encouraged, since the 1960s, research on access control. The main concern was to investigate the vulnerabilities that may exist in government systems due to the increased dependence of the defense systems on computers. This problem was also a primordial research subject in universities.

The first access control models can be divided into *discretionary* and *mandatory* access control. This distinction involves the capacity of a subject of modifying access rights. The *mandatory access control* policies also known as MAC originated in the military domain and prevents subjects from modifying access rights. In 1983 the TCSEC in [72] formalized regulations to protect sensitive information in multilevel security systems, it defines MAC as

> (...) a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to information of such sensitivity [72].

In fact, MAC supports the US department of Defense requirements concerning unauthorized access to classified information, and in particular the protection of the confidentiality of sensitive information. The ability for a subject to access or perform an operation on an object is constrained by the system. That is the security policy is centrally controlled by an administrator and thus subjects do not have the ability to override the policy. These policies were formalized in the Bell LaPadula model [15] that preserves confidentiality of information and the Biba model [20] that preserves integrity. Note that such models are static and rigid in the sense that they do not allow an update of the access control policy unless it is manually performed by the policy designer or the administrator.

The *discretionary access control* policies also known as DAC is more flexible and supposes that the subject can manage the rights concerning objects that he has created (more precisely, that he *owns*). DAC is defined by TCSEC as

> (...) a means of restricting access to objects based on the identity of subjects or groups, or both, to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restricted by MAC)[72]

In fact, DAC allows subjects to grant and revoke access rights to other subjects. That is, the control over the access to an object is left to the discretion of the authorized subject (owning that object) without the intervention of the system administrator. This is the case for example in the UNIX operating

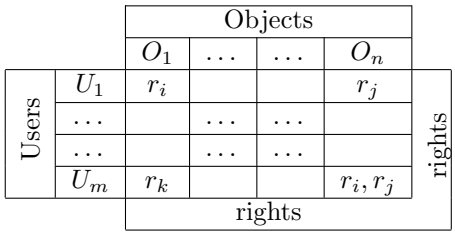


Figure 2.1: Access control matrix in the Lampson model.

system where each user has the possibility to define *read* and *write* rights over his own documents. These policies were formalized first by Lampson [50] by using an access control matrix, then in the HRU model [41] by allowing transitions over access control matrices and the take-grant model [46] by defining state transition graphs.

2.1.1 Discretionary access control

In this section we present three different models that can express discretionary access control, namely the Lampson model, the HRU model and the take-grant model. All three models have the specificity of allowing state transition for the system.

The Lampson model

The earliest work in defining a formal description for access control was by Lampson [50] who introduced the formal notions of subject and object and defined an access control matrix (see Figure 2.1). In this model, a right is a relation between subjects in S and objects in O. The rights that a subject has on an object are represented by an access control matrix M.

Each entry $M(i, j)$ of the matrix represent the rights that the subject i has over object j. The model defined by Lampson is not fixed. In fact it is defined as a transition state machine where each state is of the form (S, O, M). When a subject s loses a right over an object o, the entry $M(s, o)$ will be modified accordingly. This model also allows the addition of new subjects and new objects to the system. However, the update of the matrix becomes more fastidious. When adding a new object o', all entries $M(x, o')$ concerning the object o' need to be modified accordingly. This task becomes more complicated when removing a subject s from the system since this will lead to a complete browsing of the matrix in order to change all the entries $M(s, y)$ concerning this subject.

In summary, the Lampson model defined the access control matrix that was the base for the formalization of the HRU and the take-grant models and many other till nowadays. However the system update in the case of the Lampson model is rather complex.

rights	enter r into $M(s, o)$
	delete r from $M(s, o)$
subjects	create subject s
	delete subject s
objects	create object o
	delete object o

Table 2.1: Primitive operations in HRU model

Command: $c(x1, ..., xk)$		
	if	r_1 in $M(s_1, o_1)$ and
		r_2 in $M(s_2, o_2)$ and
		:
		r_m in $M(s_m, o_m)$
	then	op_1
		op_2
		:
		op_n
end		

Table 2.2: A command in HRU model

The HRU Model

In 1976, Harrison, Ruzzo, and Ullman presented HRU model [41] in order to study the complexity of the *safety problem*. The safety problem can be defined as the problem to determine whether or not a given subject can eventually obtain an access right on a given object. Like the Lampson model, the HRU protection system is a state transition system, made of:

- a set of subjects S,
- a set of objects O,
- a set of access rights R, and
- an $|S| \times |O|$ access control matrix M such that the entry $M(s, o)$ is the subset of R specifying the rights subject s has on object o.

The importance of the HRU model is in the *commands* that formalize the modifications on the access control matrix M. The only operations permitted in the HRU model are *primitive operations* for manipulating subjects, objects, and the access matrix presented in Table 2.1.

The execution of these primitives within a command leads to changes in the matrix by the insertion or deletion of lines and columns, or the modification of entries in the matrix. An access control policy is an arbitrary finite set of rules that are formed from a given set of primitive operations as can be seen in Table 2.2.

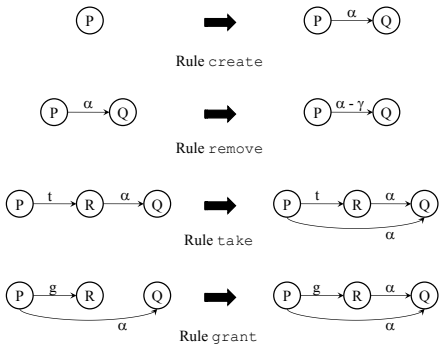


Figure 2.2: Graph rewriting rules for the take-grant model

The HRU model can capture security policies regulating the allocation of access rights. The access matrix describes the state of the systems whereas the commands constitute the transitions. The effect of a command is recorded as a change to the access matrix.

Given a system, an initial configuration Q_0 and a right r, Q_0 is *safe* for r if there is no sequence of commands that when executed from Q_0 lead to the addition of the right r into a position of the access matrix that previously did not contain r. The verification of this property defines the *safety problem*. The authors proved that this problem is undecidable in general. That is there is no way to verify whether an unauthorized user will gain access to a right in an improper manner. This is due to the flexibility of the HRU model that does not control how users pass rights from one another.

The safety problem is decidable if each command consists of a single primitive operation. That is a command can only test one cell of the access matrix. However this reduces drastically the expressivity of the model. In fact such constraint prevents for example the system from the ability to revoke an access right.

The take-grant model

In an attempt to model an expressive access control model for which the safety problem is decidable, Jones et al. [46] defined the take-grant protection model as a directed graph, where vertices are either subjects or objects and the edges are labeled with the rights that the source of the edge has over the destination. The application of rights is modeled by applying graph rewriting rules as seen in Figure 2.2. It can be viewed as a variant of the HRU model in the fact that it restrains the commands by defining them in four categories:

- the *create rule*, that allows a subject P to create a new object Q, and assigns initial rights α of P on Q.
- the *remove rule*, that allows a subject P to remove a right γ it has over an object Q.

- the *take rule*, presented by an edge labeled with t from a subject P to a subject (or object) R, that allows P to take all rights α that R has over Q
- the *grant rule*, presented by an edge labeled with g from a subject P to a subject (or object) R, that allows P having rights α on an object (or subject) Q to grant α to R

A state in the take-grant model is said to be safe between p and q with respect to right α if there is no sequence of graphs $G_1, \ldots G_n$ such that p acquires the right α over q in G_n. Jones et al.[46] defined the necessary and sufficient conditions leading to a safe state. However the problem turns out to be undecidable in case one considers that there is a collaboration between subjects [46].

Discussion

The models presented in this section rely either on access control matrices or on graph rewriting system for directed graphs. They are enriched with primitives that allow changes in access right allocation either in non-deterministic manner [50, 41] or with certain constraints with respect to subjects and objects [46]. This specificity adds a very desired dynamic aspect that allows the representation of the changes permitted by the policy on the environment. Also such models can be used to express distributed access control, since the discretionary policies allow each subject to have his own policy on objects. As such if subjects represent entities in the environment, then the interaction between such entities can be modeled by the policy they share on the objects in the system. However the undecidability of verification of the compliance of the policy with the security requirements presents a very important disadvantage for such models.

2.1.2 Mandatory access control

A *safer* solution for designing access control policies is the mandatory access control. Originating in the military environment, MAC offers a safe structure for a highly centralized access control system that regulates access to multi-level objects by authorized subjects depending on their clearance level. Ideally, the objects are labeled with security labels ranging from *Top secret* for the most sensitive to *public* or *unclassified* for the least sensitive, and the subjects are divided into different clearance levels (depending on their military rank).

The Bell LaPadula model

The first *military access control* model was formalized in 1973 by Bell and LaPadula [15]. The Bell LaPadula model is a mathematical model that allows the analysis of confidentiality related security properties of the model in details. The model is composed of

- a set of subjects S,

- a set of objects O
- a set of access operations A denoting the access rights, and
- a set L of security levels, with a partial ordering.

Access permissions are defined by an access control matrix. In the Bell LaPadula model, a state in the system is represented by a 4-tuple (b, M, F, H), where b denotes the current access set, M the access control matrix, F the level function and H the hierarchy.

Example 2.1.1. One can define a state in the model such that

- Access set b: Bob has the *read* right on the document d for staff.
- Access matrix M contains an entry $M(Bob, d)$ having *read* in the set of access rights.
- Level function F : the highest level for Bob is $(SECRET, STAFF, FINANCE)$, and the object d is classified as $(CONFIDENTIAL, STAFF)$. Thus the current level for Bob would be $(CONFIDENTIAL, STAFF)$.
- Hierarchy : the object d is isolated.

The security policies prevent information flowing downwards from a high security level to a low security level. This is guaranteed by the *no read up* and *no write down* constraints that verify the confidentiality of system resources. In order to communicate, subjects need to follow the two following properties:

- The *Simple Security Property* states that a subject s at a given security level may not read an object o at a higher security level (no read-up). This prevents access to classified information to low level subjects.
- The **-property* states that a subject s at a given security level must not write to any object o at a lower security level (no write-down). This prevents high level subjects from sending information to low level subjects

A consequence of the *-property is that eventually there will be an upgrading of all objects due to the no write down policy. To remedy to this problem, such objects will have to be downgraded manually by an officer or a program that does not comply with the policy.

Note that the Bell LaPadula model only considers the information flow that occurs when a subject observes or alters an object. However, although the confidentiality of resources is preserved, the integrity can be violated. In fact, a subject from a lower clearance level can corrupt an object at this low level, that can be read by higher level subjects. The Biba model [20] came to circumvent the weaknesses of the Bell LaPadula model.

The Biba model addresses the integrity problem by defining an integrity hierarchy for subjects over objects and defining reverse constraints namely a *no*

read down and *no write up* policy. Subjects can only create content at or below their own integrity levels and can view content at or above their own integrity level.

Note that both the Bell Lapadula and Biba models rely on the subject and object classifications, the automatic upgrading of objects confidentiality levels in the case of Bell LaPadula due to the no write down property or similarly the problem of downgrading of object integrity level in the case of Biba due to the no write up property needs the intervention of *outsiders* to remedy to the situation, something that is not desirable when designing a security policy.

2.1.3 The Clark Wilson model

While military models took the lead and gained ground on the commercial market as by providing a high level of confidentiality, the need was for models that emphasize more on integrity as that was the main goal of commercial customers. In 1987, Clark and Wilson [27] presented a different access control model. Although this model is not fully formalized, it arised from the need to find a model flexible enough to commercial security policies that are not necessarily based on security stratification. Their model relies on two basic concepts, the well-formed transactions that constrain the user to change data only in authorized manner and the separation of duty constraints that prevent fraud and abuse of power. The access and modification of data is defined through certification and enforcement rules. On the one hand, *data items* are changed only by *transformation procedures,* which maintain integrity. On the other hand, users may only invoke some transformation procedures, and a pre-specified set of data objects or constrained data items, depending on their duties which allow the enforcement of separation of duty. This model introduces two very important concepts in security models, namely the tractability property, that is the possibility to keep track of executed transactions and the separation of duty property that prevents fraud and preserves integrity in commercial organizations.

2.2 Role based access control

The rise of multi-user and multi-application system made it more complex to manage rights in access control matrices or directed graph as is the case with the MAC and DAC models presented in Section 2.1. As mentioned in Section 2.1.3, unlike military users, commercial users and industries were looking for more flexibility in access control systems while maintaining the integrity of the model. The ideal recipe would be to opt for the flexibility and dynamicity of the HRU model [41], as opposed to the highly stratified MAC model [15, 20], while preserving the safety provided by the Bell LaPadula and Biba models.

The role based access control [36],[37] model has emerged as an alternative solution to the already existing discretionary and mandatory access controls. The RBAC framework borrows its structure from the organization structure. Roles are created for the various job positions in the organization and users

are assigned to roles based on their qualifications and responsibilities. A user acquires authorizations based on the roles assigned to him. RBAC is neither a MAC nor a DAC model. However, it has the capacity to express both models.

The main idea behind its role structure is to allow an easier update of rights independently of the actual identity of the users. That is while users may step in and out of roles, the access control rights will not be affected. This gives a flexible way to update the system in the case of a user being fired for example, or in the case where more permissions are granted for a given position (role). Thus instead of updating the access rights for each user involved as is the case in [50], the update is only done once. For example, in the case a user loses his job, the update of user-role relation by removing the user from the role is enough for the user to lose all access rights associated with the role. Further in the case where more permissions are added to a role, an update of the role-permission relation, by adding the corresponding permission(s) to the role, is sufficient so that all members authorized to that role acquire the additional permissions.

In what follows we give some of the basic features of RBAC.

2.2.1 RBAC features

The RBAC framework was first motivated [36] and formalized in [37, 69]. Its main advantage is its ability to express organizational policies. The RBAC model was defined via the following components:

- U, R, P, and S are finite sets of users, roles, permissions, and sessions, respectively,
- $PA \subseteq P \times R$ is a many-to-many permission to role assignment relation,
- $UA \subseteq U \times R$ is a many-to-many user to role assignment relation,

The users can step in and out of roles without the need to modify the corresponding accesses. This innovation came to solve the heavy procedure of updating access matrices.

As in the case of a MAC model, RBAC allowed the enforcement of the least privilege principle which states that a user must only have access to the minimum set of rights necessary to perform a given task. In fact, the least privilege principle, a property highly recommended in an organization environment can be guaranteed by the addition of sessions to the RBAC structure as follows:

- $user : S \rightarrow U$ represents a function mapping each session s_i to the user $user(s_i)$, and
- $roles : S \rightarrow 2^R$ represents a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r | (user(s_i), r) \in UA\}$ and session s_i has the permissions $\bigcup_{r \in roles(s_i)} \{p | (p, r) \in PA\}$.

The notion of session distinguishes between being member of a given role and activating the role. While the later gives the active subjects all permissions

associated with the role the former only provides the subject with the right to activate the role in a session. The least privilege principle can then be enforced by putting constraints on the set of roles that can be activated by a given user.

Additionally, the activation of roles can be constrained for example to prevent a subject from acquiring the permissions associated with two exclusive roles. For example in a banking organization it is necessary to prevent a user from being a member for both purchase manager and payment manager as this may lead to abuse of authority and cases of fraud. Other types of constraints may be used for binding of duty or cardinality constraints (see Chapter 7 for more details).

Note that sessions contribute to the expression of some kind of dynamic access control, where the dynamic aspect lies in the choice of active roles in a given session.

Another important feature added to the RBAC structure is the *role hierarchy*. For example, in a medical environment, both a *cardiologist* and a *surgeon* have the permissions allocated to *general doctors* who in turn have the permissions allocated to *interns*. In order to prevent redundancy in permission allocation for these roles, the definition of a hierarchy gives the minimal set of permissions for each of these roles and allows the inheritance of permissions for roles in the hierarchy.

- $RH \subseteq R \times R$ is a partial order on R called the role hierarchy relation and written as $\geq$, and
- $roles : S \rightarrow 2^R$ is modified to require $roles(s_i) \subseteq \{r|(\exists r' \geq r)[(user(s_i), r') \in UA]\}$ and the session s_i has the permissions $\bigcup_{r \in roles(s_i)}\{p|(\exists r'' \leq r)[(p, r'') \in PA]\}$

In fact, role hierarchies contribute in the definition of the organization structure in terms of the stratification of roles and responsibilities of the various organization positions. Given a role r, we say a role r' is junior to r if $r' < r$ in the role hierarchy. A user can activate in a session any combination of roles junior to the roles the user is a member of. In that case the user acquires the permissions associated with the activated roles, but also all permissions associated with roles junior to the activated roles.

Finally, RBAC can also express delegation of roles[69], a feature by which a user that is not initially member of a role, acquires the rights associated to that role by delegation. This feature is very important in organizations, as it allows a flexible allocation of rights in cases of emergency, without modifying the initial structure of the organization. For example, this can occur in cases when the main member assigned to the role is on a leave, or if additional subjects are needed in a given role.

RBAC in its general form provides a solid ground to express a variety of organizational access control policies. However as stated in [69], it is not capable of expressing "situations where sequences of operations need to be controlled." Also the notion of role administration and role hierarchy remains ambiguous.

2.2.2 Role administration and management

To formalize role administration, ARBAC [67] provided a structure to manage role-based access control in a decentralized manner by adding to the RBAC structure a supplementary layer for administrative permissions and roles. The management of role hierarchies and permission-role allocation is done by a centralized authority, the management of users and the user-role allocation and revocation can be done by subjects pertaining to administrative roles. User-role and permission-role relations thus become ternary relation including the administrator role, where the allocation condition depends not only on the ordinary role but also on the administrator role. For example, an administrator can assign a role to a user, if the administrator has the adequate administrating role and the user has the authorization for the membership of the assigned role. Note however that ARBAC does not handle the problem of creating new roles and does not offer a mechanism to express delegation.

2.2.3 Role based access control in a workflow system

Another very interesting extension of RBAC is the workflow based RBAC presented in [16]. Workflow management systems (WFMS) are used to coordinate and regulate the business processes of an organization. In large organizations it is often the case that process activities are allocated to roles rather than users in an RBAC-like structure. In [16], the authors present a model supporting both specification of RBAC in WFMS and the enforcement of role-based constraints in the workflow.

A workflow consists of several tasks, each of which can be executed a number of times. Each task is associated with one or more roles that are authorized to execute the task. When a user tries to execute a task, the access control system must check whether there exists a role authorized to execute the task and for which the user is a member. In order to do that, the authors define *workflow role specification* as a set of task role specification. Namely, each task role specification is a 3-tuple $(T_i, (RS_i, \prec_i), act_i)$ where T_i is a task, RS_i is a set of roles authorized for T_i, $\prec_i$ is a local role order relationship and act_i is the number of activations of task T_i. In order to express workflow specification and constraints, five types of predicates are put in place, namely

- *specification predicates* express the task-role, user-task, user-role and role hierarchy relations;
- *execution predicates* keep a trace for the effect of the task execution;
- *planning predicates* express the restrictions imposed on the workflow, each restriction being expressed by a distinct predicate;
- *comparison predicates* and *aggregate predicates*.

The constraint specification is a set of clauses in a general logic program. This model, by introducing the possibility to keep a history of executions as

well as distinguishing between static constraints that can be evaluated in the classic access control policy and dynamic constraints that can only appear at the execution of the system, offers a ground for research that take into account the need for a dynamic framework in addition to the modeling of an access control policy within the organization. However, this implementation is performed by adding ad-hoc predicates in order to evaluate the different constraints. This makes it complicated to extend the model without modifying the semantics of the framework.

2.3 A flexible authorization framework

In [45], the authors present the Flexible Authorization Framework (FAF), a unified logic-based framework that can express and enforce multiple access control policies. FAF specifies rules in the form of Prolog-style rules. The authorization framework consists of the following components:

- A *history table* whose rows describe the executed accesses. A predicate $done(object, subject, role, action, time)$ records in a row each time *action* is executed by *subject* in *role* on *object* at a specific *time*;
- An *authorization table* whose rows are authorizations composed of the triples $(o, s, +a)$ or $(o, s, -a)$ denoting that the right a is permitted, or denied respectively for subject s on object o;
- A *propagation policy* that specifies how to obtain new permissions from the authorization table with respect to subject and objects hierarchies;
- A *conflict resolution policy* that specifies how to eliminate contradictory authorizations that may arise during the propagation stage;
- *decision policies* that allow grant or deny decision based on a history table;
- *integrity constraints* where all authorizations that violate such constraints will be denied.

FAF provides a local stratification for its predicates which leads to a unique stable model and well-formed semantics. Jajodia et al. maintain a materialization structure that associates each instance A of valid predicates with the set S of rules that directly supports its truth. Changes to the materialization structure are realized explicitly by means of special operators that enforce the addition or removal of a pair (A, S) from the materialization structure. Note that any change to the authorization table will lead to changes in the materialization structure.

FAF can express dynamic access control policies through the addition of a history table that contributes to the dynamic evaluation of access control decisions. Also the hierarchical structure of subjects and objects along with the possibility to define propagation rules liberate FAF from the constraints of RBAC. In fact, RBAC structure can be easily implemented in FAF, but rules

that allow the propagation of a permission to a user without passing through a given role are also possible in FAF. However the flexibility of FAF has its repercussions on the syntax of the language. In fact the syntax offered by FAF is rather complicated, the different policy stages and the large number of predicates due to the need for stratification, makes the language difficult to use. Finally, the dynamic aspect of FAF is constrained to the recording of executed actions in the history table which is very useful to express separation of duty constraints for example, but cannot encode the evolution of a business process for example.

2.4 Organization based access control

Organization based access control [2] came from the need for a decentralized model that allows the management of access control between different organizations or between different parts of the same organization on the one hand and the necessity of a dynamic access control model. Unlike [45] which relies on subjects and objects hierarchies, OrBAC defines access control policies around a given organization. To circumvent the need for constraints on elements other than subjects and roles, OrBAC defines in addition to the *role* structure, two other components, namely, *views* and *activities*. Finally, the main interest in OrBAC in terms of dynamicity is the ability to modify the access control policy depending on the situation. As such, the change is influenced by a given *context* rather than by a recording of executed actions.

The main unit in OrBAC is the organization, and the main components are *roles*, *views* and *activities*. To each role is allocated a set of subjects sharing a common interest, to each view is allocated a set of objects satisfying a common property and to each activity is allocated a set of action of the same kind. Also the organization is a parameter in the OrBAC access control rules, allowing thus the expression of organization-specific rules and policies within a unique framework.

In order to express dynamic access control, OrBAC adds the notion of *context* that allows to dynamically change the access control policy with respect to a given context. For example, in a normal context a patient's records can only be accessed by his own treating doctor, while in the case of an emergency, any subject playing the role doctor can access the records. Such policies are difficult to express in a standard RBAC structure but can be flexibly presented in OrBAC given the addition of a context parameter in the access control predicates. The access control policy of an organization is defined with respect to permissions (but also obligations and recommendations) that have as parameters *roles*, *views* and *activities*. The context is used to specify concrete circumstances where organizations grant roles the permission to perform activities on views, the context is defined with respect to concrete cases as a relation between *subjects*, *objects* and *actions*. Finally subject-role, object-view and action-activity are defined with organization-specific predicates.

A concrete permission for a subject s to execute action a on object o is

evaluated with respect to the general permission on the role, view and activity, the context relation for the requesting subject with respect to the action over the object and the allocation relations for the subject, object and action with respect to the roles, views and activities.

OrBAC has been extended in various directions to handle role hierarchies, delegation, workflow management, and negotiation among others.

Negotiation in OrBAC In [40] negotiation is presented as a way to exchange sensitive information between different entities (organizations). In each entity information is classified by sensitivity level. During a negotiation between different entities the level of sensitivity of a resource may change depending on the availability and relevance of negotiated information. Accordingly, at the end of a negotiation session, either a response is reached or an exception module is called to treat unresolved negotiations. The distinction is made between the negotiation module whose role is to establish trust and collect information necessary for access evaluation, and an access control system based in the extended RBAC profile of XACML (see Section 2.5 below.). The choice to classify resources rather than policies comes from the fact that the sensitivity level of a resource may depend on the context. Thus one may choose, for the same resource, different sensitivity levels in different context situations. Note that imposing sensitivity levels on resources is the same as defining a *negotiation policy* over these resources that complements the general negotiation guidelines of the entity.

2.5 The extensible access control markup language (XACML)

XACML [64] is an OASIS standard that describes both a policy language and an access control decision request/response (context) language. The policy language describes general access control requirements, while the context language enables the expression of a query.

To acquire the permission to perform an action on a given resource, a user must make a request to the entity that protects that resource which is called a Policy Enforcement Point (PEP). The PEP will form an XACML context based on the requester's attributes, the resource in question, the action, etc.

It will then send this request to a Policy Decision Point (PDP), which will respond with `Permit` or `Deny` according to the result of the rules pertaining to the request context and to its decision combining algorithm, or with `Not Applicable` if no rule applies, or with `Indeterminate` if an error occurred. The PEP then allows or denies access to the requester (or issue a request to another PDP.)

In addition to providing request/response and policy languages, XACML also provides a mechanism for looking for a policy that applies to a given request and evaluating the request against that policy to come up with an answer to

be sent to the PEP to interpret it and send it to the user. Another important feature of XACML is the use of Obligations which are a set of commands or requests that the PDP can send to the PEP along with the response and that will be interpreted by the PEP.

XACML is widely used due to its flexibility in defining access control policies. The Security Assertion Markup Language (SAML) profile for XACML [63], provides means to write assertions on identities, roles or attributes, and defines a protocol for exchanging these assertions between entities. XACML was also extended towards expressing core and hierarchical RBAC [62].

2.6 Dynamic RBAC logical frameworks

The need to express access control in a distributed environment became more important with the development of web services. In fact, it is often the case that services need to collaborate one with another without having a prior knowledge of each others identity. This situation leads to the need to establish trust between the different actors of such a collaboration.

2.6.1 A role-based trust management framework

Li et al. address in [52] access control and authorization problems in large-scale, decentralized systems. They combine RBAC with trust management in order to define an access control policy that is capable of managing and regulating the collaboration between organizations.

The role-based trust management framework RT is based on roles as means to assign permissions to users. This adds to the RT frameworks all the advantages associated with RBAC features such as the notion of session, role activation, role hierarchy or, delegation, as described in Section 2.2.1.

The trust management aspect comes from the use of credentials to manage distributed authority. The use of attributes in credentials allows for example assertions of one entity about attributes of another entity. In RT a role R is defined with respect to an entity A and denoted by $A.R$. The entity A is thus the owner of the role, it can define its members, its sub-roles or its delegatees and this by issuing role-definition credentials. For example

- $A.R \leftarrow A.R_1$ is a credential denoting that entity A defines role R_1 to higher in the hierarchy to role R
- $A.R \leftarrow B.R_1$ is a credential denoting that A delegates a part R_1 of role R to B. This can be used to decentralize user-role assignment or to define role-mapping across multiple organizations when they collaborate.

Several extensions to RT are presented, namely RT_1 expresses parameterized roles, RT_2 adds logical objects to RT_1, RT^T provides manifold roles and thus the possibility to express thresholds and separation of duty policies and RT^D provides the possibility of having delegation of role activation. This allows possibility to use selective rights as follows;

- $B_1 \xrightarrow{\text{D as A.R}} B_2$ denotes that entity B_1 can delegate to B_2 the ability to act on behalf of D as a member of role $A.R$

RT is based on Constraint Datalog. The credential definitions allow the expression of trust management by the exchange of *role ownership* or user-role assignments between collaborating entities. The roles are viewed as permissions and thus the credential definitions allow trust negotiation over these permissions. The possibility to delegate role activation adds some dynamic aspect to the framework, but the basic strength of the *RT* framework lies more in its ability to define manifold roles and flexible forms of separation of duty constraints than in providing a dynamic framework.

Finally, the *RT* framework cannot be considered as a generic framework as every access control aspect is handled with a specific construct that changes the syntax and evaluation semantics of the roles. In particular, some of the extensions are not conservatives as is the case in the extension to take into account delegation. In fact a security policy that does not use delegation will have a different meaning according to whether it is interpreted in the setting with or without delegation.

2.6.2 The Cassandra access control framework

Cassandra [13, 14] is a logical framework to express role based access control in a distributed environment. It relies on credentials to authenticate users. Furthermore, the Cassandra trust management system allows entities to share resources according to their own access control policy even if they do not know each other in advance. The access control policy is specified in constraint Datalog and is made of access control rules. The access control rules are defined by five predicates:

- $\texttt{canActivate}(e, r)$ and $\texttt{hasActivated}(e, r)$ express respectively, the fact that an entity e can activate the role r or has already activated the role r. The former corresponds to a role membership in RBAC, while the latter designates that e is active in the role.

- $\texttt{canDeactivate}(e_1, e_2, r)$ expresses the fact that entity e_1 can deactivate entity e_2 from role r. If this deactivation is triggered, then the fact $\texttt{isDeactivated}(e_2, r)$ becomes true indicating that the entity e_2 is not active in role r anymore.

- $\texttt{permit}(e, a)$ expresses that entity e is permitted to perform action a,

- $\texttt{canReqCred}(\texttt{e}_1, \texttt{e}_2.\texttt{p}(\overrightarrow{e}))$ expresses the fact that entity e_1 can request a credential issued by entity e_2 asserting information in $p(\overrightarrow{e})$. This last predicate is used to communicate between entities in case of trust negotiation.

The Cassandra framework is based on a Cassandra interface. In order to communicate, each entity runs a copy of the Cassandra service that ensures the

relation between the Cassandra policy and the remote access to other entities. Unlike the *RT* framework, the Cassandra framework is flexible enough to express access control features without the need for ad hoc constructs. Also, the operational semantics of Cassandra operations consolidate the dynamic expressivity of the language especially in treating the activation and deactivation of roles which was not very clearly defined in the initial RBAC structure. However although the Cassandra framework can keep track of executed actions, as in the case of [45], there is no structure for the expression of effects for these actions.

2.7 SecPAL: a decentralized authorization language

In previous sections we presented different access control policies based on roles. However such a structure constrains the expressivity of the language. In [45], the authors circumvented this limitation via the definitions of independent hierarchies (see Section 2.3.) and in [2] more expressivity was added by the definition of the notions of *activity* and *view* (see Section 2.4.) However this did not liberate these models entirely from the rigidity of RBAC. The need for more expressivity led more research to *attribute*-based access control as means to express more human-expressive policies.

In [11] the authors argued that applications depend on complex and changing authorization criteria and the current nature of organizational structure makes it more needed to define trust relations between different applications, each with its own authorization policy. To make it easier to use, policies should be human readable and should be updatable without need to change application code.

SecPAL is a declarative authorization language. It distinguishes between access control requests that are mapped to access control queries and assertions that help in evaluating these queries.

The main features of SecPAL is its clarity. Being based on assertions, SecPAL is human readable and can be translated to Datalog with constraints. Policies are phrased in terms of principal attributes asserted by adequate delegation chains. An assertion in SecAPL is of the form:

$$A \text{ says } fact \text{ if } fact_1, \ldots, fact_n, c$$

where A is a constant denoting an entity (or principal), $fact, fact_1, \ldots fact_n$ are facts and c is a condition (matching a variable against a regular expression, equality between two expressions, etc.). The *if* part is optional and allow to put constraints on the assertion.

The facts are the main elements of the language, we distinguish different forms of facts as follows:

- Arbitrary user-defined sentences of the form

  $$A \text{ action } X$$

  where *action* is a user-defined action and X is a variable.

- Special constructs of the form

$$A \text{ cansay}_0 \ fact$$

or

$$A \text{ cansay}_\infty \ fact$$

that express delegation, meaning that a principal A accepts assertions made by another principle, the indices indicate the delegation depth.

- Special constructs of the form

$$A \text{ can act as } fact$$

express that a principal can act as another principal, meaning that the former has all the rights that the latter has.

As in [52, 13], assertions take into consideration the source of authority or authentication (the entity that "says" what is true) and emphasizes the need for delegation (the ability to perform an action on behalf of another entity) which is essential in the case of entity collaboration. Further as in the case of [13], SecPAL is a tunable language in the sense that the user can add verbs and entities without modifying the system. In SecPAL there is a distinction between assertions, that form the static system of rules and may not contain negations, and the queries that make use of the assertions to make a decision and may contain negation if the negated literal is fully instantiated before the negation can take place. The queries are evaluated against the current database of clauses and, when evaluated, update the database with new assertions. This last feature is very important as it presents an implicit way to define a dynamic framework.

2.8 Other dynamic policies

In [34] is defined a framework to represent the behavior of access control policies in a dynamic environment. The authors argue that the evaluation of a security policy affects the environment where the evaluation occurs. In order to express this, they define a mathematical model that takes into account both the policies and their environment. This model allows the evaluation of a policy in a given environment and the update of that environment with respect to the policy evaluation. Policies are expressed by Datalog rules. The environment is viewed as a transition system where the labels correspond to permitted actions. Each state consists of an instance of the extensional predicates (i.e the predicates only occurring in the body of a rule) referred to by the policy. Evaluating the policy leads to the modification of the access matrix. When a request is granted, a "fact" stating that the *subject* that executed an *action* on a *resource* is added to the access table. When a request is not permitted or denied, the corresponding transitions are eliminated, and eventually all unreachable states are eliminated.

Another logical framework to express dynamic access control policies is the state modifying model (SML) of Becker and Nanz presented in [12]. The state modifying model rules are defined with Datalog extended with effects and a simple form of negation. As in [34] the authors argue that access requests can have effects on the authorization state. The updates are factored out of the resource guard by adding or removing facts from the authorization state, which facilitates the expression of policies that take the history of access requests into account. The state modifying model also introduces the possibility to query the authorization state. Queries are declarative rules with pre-conditions and effects.

2.8.1 Discussion

Access control has evolved rapidly with the development of services and automatization of the organization policies. The need for an efficient access control model that is capable of matching the rapid growth of organizations and their constant modifications (in the case of a merge or a split of a part of the organization for example), led to the emergence of a variety of dynamic models. The *RBAC* model saw in roles an interesting way to improve the policy design of organizations providing a very important flexibility in terms of user-permission assignments. OrBAC, by introducing the notion of context developed a unique dynamic model that changes with respect to different scenarios defined by the language's contexts. The addition of the organization identity as a parameter in the access control policy allowed the expression of distributed access control by considering that different organizations can collaborate. The dynamic access control took a new definition with the *Cassandra* framework that explicitly defined the activation and deactivation of roles as transition relations. SecPAL saw the need to free the access control from the notion of roles to gain more flexibility in the expression of security policies and introduces a notion of dynamicity by updating the security state according to query evaluation. Finally, SML acknowledged the need to express the changes occurring to the system state by adding and removing facts in the access control database and this by extending the Datalog access control rules with the addition of effects.

2.9 Trust negotiation and managing certificates

Nowadays, the distributed nature of organizations has become commonplace. Applications routinely span several administrative domains. This is the case in electronic commerce, electronic banking, or electronic government applications where collaboration between applications, also called services, that do not necessarily know each other, is needed. While the access control policy regulates the access to each entity resources and acts independently of other resources, trust management offers a solution that allows the specification and implementation of a security policy to determine whether the access to a given resource should be granted or not based on a number of conditions. We can find a trust

management aspect in [13] via the distinction in a credential rule between the entity holding the credential and the entity issuing the credential and the definition of rules evaluation semantics for credential retrieval within the Cassandra engine. *RT* [52] also was presented as an access control framework that can regulate trust management by delegation of attributed roles from one entity to another with respect to some conditions. The assertion based language SecPAL [11] also handles trust management through conditions on the issuer entity and the assertion delegation.

The main purpose of the aforementioned models is to express access control policy embedded in a given organization or entity. Their trust management mechanism is thus within the access control policy.

In this section we present a glimpse of existing trust management systems whose main objective is to specify and establish trust between entities. This overview will help us in the sequel to formalize an access control framework that is capable of specifying and enforcing an independently defined trust negotiation policy (see Chapter 4).

Trust management as defined by Blaze et al. in [22] is a unified approach to specify and interpret security policies, credentials and relationships that allow direct authorization of security critical actions. Credentials are signed by public keys, and delegation of trust can be described among public keys. Trust management binds keys directly to authorizations to perform specific tasks.

In earlier approaches, the response to a request from the client amounts to first determine who signed the request, and then query the local policy to decide whether the signer is allowed or not to access the requested resource. This is not feasible in a distributed environment with a large number of requests most often from unknown users.

Delegation is a key feature for trust management in a distributed environment since decision is not centralized in one entity but distributed on several specialized entities. As such it is important to be able to express that an entity A *trusts* the verdict of entity B on the information about entity C. In [1] a logical framework for authentication is defined allowing the specification of trust via predicates. Namely, the predicate $B|A$ is obtained when B speaks on behalf of A not necessarily with a proof, and $B\ for\ A$ is obtained when B speaks on behalf of A with appropriate delegation certificate.

The language presents axioms necessary to model delegation in the presence of cryptographic keys. The access decisions are triggered by a request and evaluated against an access control list with the possibility to define roles and groups. The trust establishment is granted if the request can be evaluated positively against the access control list's entries.

The current trust management approach is based on credentials request and credential disclosure. Credentials can be seen as a digital substitute for paper credentials that we use in real life. They allow user authentication but can also provide additional information (medical record, credit card number, ...). Credential disclosure for a given entity is subject to a local policy. Trust management consists in verifying if in a given entity a set C of credentials prove that the request r complies with the local policy P of the entity.

A trust management engine takes as input a request r, a set of credentials C and the local policy P and outputs a decision with possibly some information on how to proceed if the decision was negative. This is the case of KeyNote [21] and PolicyMaker [23] for example.

In [78] is defined an automated trust negotiation model that takes into account the fact that credentials may be sensitive. The trust negotiation architecture involves a client application and a server application. Each application defines a *credential access policy* for each of its credentials. The credential access policy of a credential usually needs the disclosure of credentials obtained from the opposed application. When the credential access policy of a credential is satisfied, the credential is disclosed.

A trust negotiation session is defined by this flow between the client and the server through an alternating sequence of credential requests and disclosures. Negotiation can be run according to an eager strategy or a parsimonious strategy. The eager strategy consists in sending all the credentials that are satisfied by the credential access policy until the two parties reach an agreement. This strategy may lead to the disclosure of unneeded credentials, however it does not reveal sensitive information. The parsimonious strategy computes a sequence of mutual requests, and once an agreement is reached, sends the adequate credentials. This exchange of requests may however reveal sensitive information that are not permitted by the credential access policy in form of a request.

In [54] Li et al. extend the trust management system of the role-based trust management framework defined in [52]. They consider the case where credentials are not stored in one place. Unlike [78], a trust negotiation session can involve more than two players. The process of making access control decision involves finding a delegation chain from the source of authority to the requester. As such they study the credential problem that consists of determining whether such a chain exists and if it does find it. Credentials have the form $R_1.A \leftarrow R_2.B$ where A and B are attributes, R_1 is the *issuer* and R_2 is the subject. Credentials can thus be stored either with the issuer of the credential or with the subject of the credential. Credential chain discovery consists in finding new credentials according to the location of previously discovered credentials.

For example, suppose the existence in addition to the above presented credential, of credentials $R_2.B \leftarrow R_3.C$ and $R_3.C \leftarrow R_4$. If one knows that the credential $R_1.A \leftarrow R_2.B$ is stored in R_1 and credential $R_3.C \leftarrow R_4$ is stored in R_4, one can directly look for additional credentials in R_3 and R_2 to complete the credential chain discovery. Furthermore, distributed chain discovery can be constructed in a non-linear manner. In fact, a procedure can begin evaluation with an incomplete set of credentials, then suspend the evaluation, issue a credential request, then resume the evaluation if the additional credentials are obtained.

2.10 Conclusion

Access control policies have evolved from simple access control rights of subjects on objects within an access control list (ACL) to complicated policies that take into account the existence and interaction of different access control systems or organizations each with its own set of access rights.

In this chapter we presented different access control models taking into account their evolution in terms of decision making, their capacity to express security state evolution and their ability to define a trust policy to collaborate in a distributed environment.

The variation between access control policies based on access matrices, roles and attributes also led to the clarification of the expressivity of these different access control frameworks. The evolution of access control models from the rigid Lampson access control matrix [50], to the functional RBAC structure [69] , and then more complex roles based models such as the FAF model [45] and the OrBAC model [2] came as a natural answer to a rising need to add more expressivity, but also to add flexibility in case of policy modification.

However, the actual structure of organizations became more complex. In fact access control policies became more specific and are often defined in terms of common characteristics that are more restrictive than roles. As such role-based structure often becomes cumbersome with the addition of a large number of roles in order to take into account such policies. This, and other reasons encouraged researchers to turn into attribute based access control, defined in its basic version by the HRU model [41]. This was the case in SecPAL [11] and SML [12].

The need to take into account the modification of the environment in the decision making was studies only partially. In HRU the dynamic aspect relied in the capacity to modify entries in the access control matrix but unfortunately evaluation turned out to be undecidable in this general form. The notion of session and role activation contributed in adding a dynamic aspect to RBAC. In [45, 13] the capacity to record executed actions allowed a better specification of the access control framework especially in terms of separation of duty constraints or the definition of access control policies with respect to executed action. OrBAC, in addition to the dynamic features inherited from the RBAC structure offered a mean to expressing policy changes according to a change of contexts. SecPAL and SML acknowledged the need for a more expressive language and specified explicitly changes that affect the security state when authorizations are executed, and this by modifying the security state database.

The need to collaborate in a distributive environment makes trust relation between different entities, each with its own authorization policy, necessary. This notion was expressed in access control models such as OrBAC [2] through the definition of policies parameterized by organizations, but more specifically in [52, 13] through the establishment of a trust management mechanism, and in SecPAL through assertions and the definition of queries. This was also the concern of more specialized trust management systems that we presented in Section 2.9.

Chapter 3

A logical approach to dynamic role-based access control

In contrast with the traditional organization of the economy in which a central organization or the State was responsible for the production of one good, today's economy relies increasingly on the integration of processes from different sources, and on an optimal aggregation of these sources. In an electronic banking system, the bank would for example rely on the Credit Bureau, an external partner, to assess the credit profile of a customer in case of a loan request. Furthermore, the loan origination process will be processed by different services within the banking organization in order to evaluate it, make a decision and eventually make an offer to the customer. In these examples, the partners need to share information to ensure the success of a cooperation, but at the same time they want to restrain the diffusion of the released information for privacy or financial reasons.

In Chapter 2, we presented some access control models that take into account, in addition to the access control management, some dynamic aspect. However this dynamic aspect was restricted to recording role activation [13] or action execution [45, 34, 12]. In this chapter we present a language with its semantics in which the *access control policies* governing the diffusion of information can be expressed. The proposed language is rule-based and introduces a dynamic aspect to the expression of access control rules by explicitly specifying effects to authorized actions. Our approach is similar to that presented in [11]. However while SecPAL uses queries that modify the security state of the system when the request is permitted, our approach models effects of an authorized action depending on whether the action is executed or not. As in the case of [13, 11] we encode RBAC extensions such as delegation, separation of duty but also role hierarchy and role activation within our framework.

Finally, as our main objective is to define a language capable of modeling

effects of actions, we chose to model in addition to authorizations the concept of *Obligation*. Obligations refers to those parts of security policies in which the future behavior of a subject is constrained either positively (a subject has to perform an action) or negatively (the subject will not perform an action.) The need for obligations is well recognized for expressing privacy policies [3], but their utilization is tricky. First, one has to rely on an external monitor [19, 43] to ensure that obligations contracted by a subject will be abode by. Part of the work presented in this chapter can be found in [6]. Second, it may happen that one action implies a disjunction of obligations. In [19], this case is handled by assuming a non-deterministic choice of the set of obligations that applies. We believe that this solution is non satisfactory from a language design point of view since non-determinism is a potential source of flaws in a security policy.

The concept underlying our proposal is that an access control system is characterized by decision contexts and an invariant Datalog program. A decision context is defined by a set of permissions and obligations. Within each decision context, an access control decision is based on the computation of whether the requested permission (obligation) is obtainable using the Datalog program from the set of permissions (obligations) defining the current decision context. The access control system evolves from one decision context to another according to actions performed by a user. A drawback of this simplicity is the declaration of every action that can alter the decision context, including *e.g.* the action stating that a client is active in a given role.

In spite of its simplicity, this model permits us to express seamlessly core RBAC policies as well as their different extensions such as role hierarchies, delegation and separation of duty. It is similar in spirit to Cassandra, though we have no built-in role management: being active in a role is an action similar to the invocation of a service. Finally, in contrast with Transaction Logic [25] the side effects are not attached to a rule but to an effective action of the client. The consequence of this choice is the determinism of the system w.r.t. client's actions (instead of client's choice of rules to apply).

In Section 3.1 we present a novel language for expressing access control policies. Then, in Section 3.2 we present how RBAC can be encoded into this language. Section 3.3 is devoted to the definition and complexity analysis of decision problems related to access control in our language.

3.1 Access control policies

In this section, we present our framework for expressing role-based access control policies and their extensions.

3.1.1 Domains

Active processes acting on behalf of users are referred to as subjects whereas passive resources accessible on a computer system are referred to as objects. A key feature of our access control policies is that all actions are done through roles,

i.e. subjects receive permissions to execute actions on objects only through the roles to which they are assigned. In our policies, following the notions considered in RBAC, subjects are organized in groups called roles, and these groups are hierarchically structured.

Let S denote the set of subjects, A the set of actions, O the set of objects and R the set of roles. We assume that:

- $S \subseteq O$ with S and R pairwise disjoint,
- O contains the null character ε.
- A and O are pairwise disjoint.

We define a *domain* in our settings to be a tuple

$$\mathcal{D} = \langle S, A, O, R \rangle$$

3.1.2 Security states

Consider a domain $\mathcal{D} = \langle S, A, O, R \rangle$, a *security state* based on $\mathcal{D}$ is a tuple

$$\mathcal{S} = \langle \Omega, \Pi \rangle$$

whose components are subsets of $S \times A \times O \times R$.

In fact a security state can be seen as an access control list whose members are either permissions or obligations for a subject in S to perform an action in A on an object in O through a role in R.

Elements of a security state

For all s in S, for all a in A, for all o in O and for all r in R, we will write $\Omega(s, a, o, r)$ instead of $(s, a, o, r) \in \Omega$ and $\Pi(s, a, o, r)$ instead of $(s, a, o, r) \in \Pi$.

- $\Omega(s, a, o, r)$ says that "subject s has in $\mathcal{S}$ the obligation to execute access a on object o through role r"
- $\Pi(s, a, o, r)$ says that "subject s has in $\mathcal{S}$ the permission to execute access a on object o through role r".

We consider that $\Omega \subseteq \Pi$ that is all obligations are permitted. A primary relation of interest between security states is that of inclusion, under which the set of security states based on $\mathcal{D}$ forms a complete lattice:

$$\langle \Omega, \Pi \rangle \sqsubseteq \langle \Omega', \Pi' \rangle \text{ iff } \Omega \subseteq \Omega' \text{ and } \Pi \subseteq \Pi'$$

3.1.3 Atomic formulae and conditions

Consider a domain $\mathcal{D} = \langle S, A, O, R \rangle$ and a security state $\mathcal{S} = \langle \Omega, \Pi \rangle$ based on $\mathcal{D}$.

Variables

We assume an alphabet of variable symbols: X, Y, etc, possibly with subscripts.

Terms

We define a term based on $\mathcal{D}$ to be either an element of $S \cup A \cup O \cup R$ or a variable symbol.

Interpretation function

An interpretation function for $\mathcal{D}$ is a function I mapping the variable symbols to elements of $S \cup A \cup O \cup R$. The value $\widetilde{I}(t)$ of a term t is defined as follows:

- if t is an element of $S \cup A \cup O \cup R$ then $\widetilde{I}(t) = t$,
- if t is a variable symbol then $\widetilde{I}(t) = I(t)$.

A 4-tuple (t_1, t_2, t_3, t_4) of terms based on $\mathcal{D}$ is said to be correct iff the following conditions are satisfied:

- t_1 is either a variable symbol or an element of S,
- t_2 is either a variable symbol or an element of A,
- t_3 is either a variable symbol or an element of O,
- t_4 is either a variable symbol or an element of R.

Atomic formula

We define an *atomic formula* based on $\mathcal{D}$ as an expression of the form $\pi(t_1, t_2, t_3, t_4)$ or $\omega(t_1, t_2, t_3, t_4)$ where (t_1, t_2, t_3, t_4) is a correct 4-tuple of terms based on $\mathcal{D}$.

The atomic formula $\pi(X, a, Y, r)$ denotes that "subject X has the permission to execute action a on object Y through role r".

The atomic formula $\omega(X, a, o, r)$ denotes that "subject X has the obligation to execute action a on object o through role r."

Well-formed conditions

We define the well-formed conditions (ϕ, ψ, etc, possibly with subscripts) based on $\mathcal{D}$ by the grammar

$$\phi ::= C \mid \bot \mid \top \mid (\phi_1 \vee \phi_2) \mid (\phi_1 \wedge \phi_2).$$

where C is an atomic formula.

The satisfiability relation $\mathcal{S}, I \models \phi$ between a security state $\mathcal{S} = \langle \Omega, \Pi \rangle$, an interpretation function I and a condition ϕ is defined as follows:

- $\mathcal{S}, I \models \omega(t_1, t_2, t_3, t_4)$ iff $(\widetilde{I}(t_1), \widetilde{I}(t_2), \widetilde{I}(t_3), \widetilde{I}(t_4)) \in \Omega$,
- $\mathcal{S}, I \models \pi(t_1, t_2, t_3, t_4)$ iff $(\widetilde{I}(t_1), \widetilde{I}(t_2), \widetilde{I}(t_3), \widetilde{I}(t_4)) \in \Pi$,
- $\mathcal{S}, I \not\models \bot$,
- $\mathcal{S}, I \models \top$,
- $\mathcal{S}, I \models \phi_1 \vee \phi_2$ iff $\mathcal{S}, I \models \phi_1$ or $\mathcal{S}, I \models \phi_2$,
- $\mathcal{S}, I \models \phi_1 \wedge \phi_2$ iff $\mathcal{S}, I \models \phi_1$ and $\mathcal{S}, I \models \phi_2$.

3.1.4 Static clauses and static policies

Security states are dynamic in nature, i.e. they are likely to change over time in reflection of ever evolving environmental conditions. Our interest is thus to formalize this change. In order to do that we use *rule-based access control policies*. Rule-based access control policies will be built up using static clauses and dynamic clauses. Static clauses will specify the access control conditions in order to authorize access to a resource by a subject in a role, while dynamic clauses specify the effects associated with the permitted accesses depending on whether the actions was or not executed.

In this section, we define the concept of static clauses. The concept of dynamic clauses will be defined in the next section.

Static clause

Consider a domain $\mathcal{D} = \langle S, A, O, R \rangle$ and a security state $\mathcal{S} = \langle \Omega, \Pi \rangle$ based on $\mathcal{D}$. A static clause based on $\mathcal{D}$ is an expression of the form

$$A \leftarrow \phi,$$

where A is an atomic formula and ϕ is a well-formed condition.

Example 3.1.1. In a banking system, the fact that a customer has the right to regulate access on his own information file is expressed by the static clause:

$$\pi(X, \texttt{grant} - \texttt{access}, Y, client) \leftarrow \pi(X, \texttt{own}, Y, client)$$

This clause says that "if X has the permission to execute `own` on Y through role *client* then X has the permission to execute $\texttt{grant} - \texttt{access}$ on Y through role *client*".

Example 3.1.2. In the same banking system, a clerk has the obligation to ask the customer permission when accessing the customer's information file, this is expressed by the static clause:

$$\omega(X, \texttt{request} - \texttt{access}, Y, clerk) \leftarrow \pi(Z, \texttt{own}, Y, client)$$

This clause says that "if Z has the permission to execute `own` on Y through role *client* then X has the permission to execute $\texttt{request} - \texttt{access}$ on Y through role *clerk*".

Static policy

A static policy based on $\mathcal{D}$ is a finite set SP of static clauses based on $\mathcal{D}$. We shall say that $\mathcal{S}$ is a model of SP, in symbols

$$\mathcal{S} \models SP,$$

iff

- for all interpretation functions I for $\mathcal{D}$ and for all static clauses $A \leftarrow \phi$ in SP,
- if $\mathcal{S}, I \models \phi$ then $\mathcal{S}, I \models A$.

The reader may easily verify that the set $\{\mathcal{S} : \ \mathcal{S} \models SP\}$ has a least element under $\sqsubseteq$. Let $l(SP)$ be this least element.

3.1.5 Dynamic clauses and dynamic policies

We have designed static policies to specify access control constraints with first order Horn clauses as in [14, 52]. The innovation in our approach is the addition of dynamic policies. As such, we give to the subject the *choice* to execute or not a permitted action. The subject will then be responsible for the effects of his choice, thus we assume that the security state is modified with respect to this choice. The dynamic policy manages the consequence of executing (or not) an authorized action.

Dynamic clause

Consider a domain $\mathcal{D} = \langle S, A, O, R\rangle$ and security states $\mathcal{S} = \langle \Omega, \Pi\rangle$, $\mathcal{S}' = \langle \Omega', \Pi'\rangle$ based on $\mathcal{D}$. A *dynamic clause* based on $\mathcal{D}$ is an expression of the form

$$\phi \rightarrow (\psi_1, \psi_2)$$

where neither ψ_1 nor ψ_2 contain occurrences of the Boolean connective $\vee$.

It is said to be consistent iff neither ψ_1 nor ψ_2 contain occurrences of the Boolean connective $\perp$.

Example 3.1.3. A basic requirement to preserve the privacy of a client in a banking system is that a *clerk* cannot access the information file of a given customer without having the permission of the customer. This can be expressed by the dynamic clause

$$\pi(X, \texttt{grant} - \texttt{access}, Y, client) \rightarrow (\pi(Z, \texttt{access}, Y, clerk), \top).$$

This clause is consistent. It says that "if X has the permission to execute $\texttt{grant} - \texttt{access}$ on Y through *client* then

- either X executes $\texttt{grant} - \texttt{access}$ on Y through *client* and Z next obtains the permission to execute $\texttt{access}$ on Y through *clerk*, or

- X does not execute $\texttt{grant} - \texttt{access}$ on Y through *client*.

Example 3.1.4. Another requirement to preserve privacy of the client is that the clerk has the obligation to ask for the client's consent before accessing the client's files. In case of violation, the system enters in an inconsistent state. This can be expressed by the dynamic clause

$$\omega(X, \texttt{request} - \texttt{access}, Y, clerk) \rightarrow (\top, \bot)$$

This clause is not consistent. It says that "if X has the obligation to execute $\texttt{request} - \texttt{access}$ on Y through *clerk* then

- either X executes $\texttt{request} - \texttt{access}$ on Y through *clerk*, or
- X does not execute $\texttt{request} - \texttt{access}$ on Y through *clerk* and the system enters in an inconsistent state.

Informally, each dynamic clause $\phi \rightarrow (\psi_1, \psi_2)$ defines a transition relation from a security state $\mathcal{S}$ to a state $\mathcal{S}'$ as follows:

if all permissions/obligations in ϕ are true in $\mathcal{S}$

then **if** all the "actions in ϕ" are executed

then ψ_1 will be true in the next state $\mathcal{S}'$

else ψ_2 will be true in the next state $\mathcal{S}'$

else the rule is not applied.

Dynamic policy

A (consistent) dynamic policy based on $\mathcal{D}$ is a finite set DP of (consistent) dynamic clauses based on $\mathcal{D}$.

The *effects* of the dynamic policy are specified with respect to the set of permitted and/or obligatory actions and differ in the case such actions are executed or not.

The set of executed actions

Given a security state $\mathcal{S} = \langle \Omega, \Pi \rangle$, let $\mathcal{A} \subseteq \mathcal{S}$ be the tuple of sets of permitted and obligatory actions that are actually executed in a state $\mathcal{S}$. $\mathcal{A} = \langle \mathcal{A}^{\Omega}, \mathcal{A}^{\Pi} \rangle$ where $\mathcal{A}^{\Pi} \subseteq \Pi$ and $\mathcal{A}^{\Omega} \subseteq \mathcal{A}^{\Pi}$. In the rest of this chapter, the tuple $\mathcal{A} \subseteq \mathcal{S}$ denotes the set of permitted and obligatory actions that are executed at state $\mathcal{S}$.

Transition relation

Given a security state $\mathcal{S} = \langle \Omega, \Pi \rangle$, a subset $\mathcal{A} \subseteq \mathcal{S}$ of actions that are actually executed and a dynamic policy DP, the security state $\mathcal{S}' = \langle \Omega', \Pi' \rangle$ can be evaluated with respect to $\mathcal{S}$, $\mathcal{A}$ and DP through a transition relation.

Formally, we say that the pair $(\mathcal{S}, \mathcal{S}')$ is a transition of DP through $\mathcal{A}$, in symbols

$$\mathcal{S} \Longrightarrow_{DP}^{\mathcal{A}} \mathcal{S}'$$

iff

- for all interpretation functions I for $\mathcal{D}$ and for all dynamic clauses

  $$\phi \rightarrow (\psi_1, \psi_2)$$

  in DP,
- if $\mathcal{S}, I \models \phi$ then
  - either $\mathcal{A}, I \models \phi$ and $\mathcal{S}', I \models \psi_1$ or
  - $\mathcal{A}, I \not\models \phi$ and $\mathcal{S}', I \models \psi_2$

The reader may easily verify that the set $\{\mathcal{S}' : \mathcal{S} \Longrightarrow_{DP}^{\mathcal{A}} \mathcal{S}'\}$ has a least element under $\sqsubseteq$. We denote $L(\mathcal{S}, DP, \mathcal{A})$ this least element.

3.1.6 Rule-based policies

Consider a domain $\mathcal{D} = \langle S, A, O, R \rangle$ and security states $\mathcal{S} = \langle \Omega, \Pi \rangle$, and $\mathcal{S}' = \langle \Omega', \Pi' \rangle$ based on $\mathcal{D}$.

Access control policy

A (consistent) rule-based access control policy based on $\mathcal{D}$ is a tuple

$$\mathcal{P} = \langle SP, DP \rangle$$

whose first component is a static policy based on $\mathcal{D}$ and second component is a (consistent) dynamic policy based on $\mathcal{D}$.

Example 3.1.5. Combining Example 3.1.1 and Example 3.1.3 gives us a rule-based policy that specifies when a client has the right to secure an information file and grant access to a clerk to this file depending on the action chosen by the customer. Let $\mathcal{P}$ be the access control policy defined by $\langle SP, DP \rangle$ such that:

- The static policy SP is:

  $$\pi(X, \texttt{grant} - \texttt{access}, Y, client) \leftarrow \pi(X, \texttt{own}, Y, client) \qquad (3.1)$$

- the dynamic policy DP is:

  $$\pi(X, \texttt{grant} - \texttt{access}, Y, client) \rightarrow (\pi(Z, \texttt{access}, Y, clerk), \top). \qquad (3.2)$$

Let $\mathcal{S}$ be the initial security state of $\mathcal{P}$, and let I be an interpretation function such that $I(X) = bob$, $I(Y) = file - bob.doc$ and $I(Z) = mary$.

- Let $\pi(bob, \texttt{own}, file - bob.doc, client) \in \mathcal{S}$, then from rule 3.1,
$$\pi(bob, grant - access, file - bob.doc, client) \in \mathcal{S}.$$
- Let $\mathcal{A}^{\Pi} \subseteq \Pi$ and suppose $\pi(bob, \texttt{grant} - \texttt{access}, file - bob.doc, client) \in \mathcal{A}^{\Pi}$, then applying the transition relation with respect to rule 3.2 leads to a new security state $\mathcal{S}'$ such that
$$\pi(mary, \texttt{access}, file - bob.doc, clerk) \in \mathcal{S}'.$$

Transition relation with respect to the access control policy

Formally, for all $\mathcal{A} \subseteq \mathcal{S}$, we shall say that the pair $(\mathcal{S}, \mathcal{S}')$ is a transition of $\mathcal{P}$ through $\mathcal{A}$, in symbols
$$\mathcal{S} \Longrightarrow_{\mathcal{P}}^{\mathcal{A}} \mathcal{S}',$$
iff

- for all interpretation functions I for $\mathcal{D}$ and for all dynamic clauses
$$\phi \rightarrow (\psi_1, \psi_2)$$
in DP, if
$$\Pi, I \models \phi$$
then
  - either $\mathcal{A}, I \models \phi$ and $\mathcal{S}', I \models \psi_1$ or,
  - $\mathcal{A}, I \not\models \phi$ and $\mathcal{S}', I \models \psi_2$,
- $\mathcal{S}' \models SP$.

The reader may easily verify that the set $\{\mathcal{S}' : \mathcal{S} \Longrightarrow_{\mathcal{P}}^{\mathcal{A}} \mathcal{S}'\}$ has a least element under $\sqsubseteq$. We denote $L(\mathcal{S}, \mathcal{P}, \mathcal{A})$ this least element.

3.2 RBAC features

In Section 3.2 we presented the logical framework for our rule-based access control language. In this section we present an encoding of RBAC and some of its extensions into our language.

3.2.1 Terminology

In the rest of this section, for the purpose of characterizing RBAC features, we consider special actions. Namely, the special actions `can-play` and `is-active` express role membership and role activation respectively. The special actions `acquire-perm` and `acquire-obl` denote the acquiring of permissions and obligations relative to a role. The special actions `delegate` and `d-play` express delegation of a role and playing the role by delegation respectively. We use the symbol ε in the object argument when the presence of an object is not relevant.

3.2.2 Role activation

The essential notion in RBAC is that permissions are associated with roles and users are assigned to appropriate roles. To this end we define role membership by the predicate $\pi(X, \texttt{can-play}, X, r)$ saying that subject X has the permission to play role r. On the other hand $\pi(X, \texttt{is-active}, \varepsilon, r)$ expresses that X is currently active in role r. The assignment of permissions to roles is expressed with the special action `acquire-perm` in the body of rules as follows:

$$\pi(X, a, o, r) \leftarrow \pi(X, \texttt{acquire-perm}, \varepsilon, r) \tag{3.3}$$

That is, if user X has acquired the permissions associated with role r then X will have the permission to do action a on the object o.

Similarly, the action `acquire-obl` expresses the assignment of obligations to roles as follows:

$$\omega(X, a, o, r) \leftarrow \pi(X, \texttt{acquire-obl}, \varepsilon, r) \tag{3.4}$$

That is, if user X has acquired the obligations associated with role r then X will have the obligation to do action a on the object o.

We assume that there is one rule for each triple (a, o, r) in the permission-role and/or obligation-role assignment relation.

One way for user X to acquire the permissions (obligations) associated with role r is to activate the role r. This is done by executing the action `can-play`, and induces the addition of $\pi(X, \texttt{is-active}, X, r)$ at the next state. This is expressed by the dynamic rule:

$$\pi(X, \texttt{can-play}, \varepsilon, r) \rightarrow (\pi(X, \texttt{is-active}, \varepsilon, r), \top) \tag{3.5}$$

and the two static rules

$$\pi(X, \texttt{acquire-perm}, \varepsilon, r) \leftarrow \pi(X, \texttt{is-active}, \varepsilon, r) \tag{3.6}$$

$$\pi(X, \texttt{acquire-obl}, \varepsilon, r) \leftarrow \pi(X, \texttt{is-active}, \varepsilon, r) \tag{3.7}$$

When X chooses to activate her role membership by executing the action `can-play` on r, then by rule 3.5 X is active in r at the next security state and the two static rules 3.6 are satisfied. This induces the *acquisition* by X of all permissions and obligations associated with role r by rules 3.3 and 3.4.

In our modeling of RBAC, we express the permission-role assignment relation initially defined in [37] in terms of role activation instead of role membership. Our aim being to associate actual permissions to *active* users and not to all *members* of a given role, we associate the permissions (and obligations) to the active occurrence of a role represented by the truth of an instance of $\pi(X, \texttt{acquire-perm}, \varepsilon, r)$ and $\pi(X, \texttt{acquire-obl}, \varepsilon, r)$. Accordingly, the user X can step out of a role r by simply choosing not to activate $\pi(X, \texttt{can-play}, \varepsilon, r)$. In this case she automatically loses all privileges associated with role r in the next state.

Finally we impose that when user X becomes active in the role r, she must acquire the associated permissions and obligations, and this acquisition is modeled by explicit actions. This is guaranteed by the following three dynamic rules:

$$\begin{cases} \pi(X, \texttt{is-active}, \varepsilon, r) \rightarrow (\top, \bot) \\ \pi(X, \texttt{acquire-perm}, \varepsilon, r) \rightarrow (\top, \bot) \\ \pi(X, \texttt{acquire-obl}, \varepsilon, r) \rightarrow (\top, \bot) \end{cases}$$

If the actions `is-active`, `acquire-perm` and `acquire-obl` are not executed the system enters in an inconsistent state. We note that in a real system, these mandatory actions can be performed on a server as a consequence of the explicit actions of a client.

3.2.3 Role hierarchy

Role hierarchy is very useful in structuring roles within a certain organization. In an RBAC model, we say r_1 is junior to role r_2 if the permissions associated with r_1 are inherited by members of r_2. This is not the case in this model where to a role is associated both a set of permissions and a set of obligations. As such we define two distinct hierarchies relative to permissions and obligations.

The rationale for this distinction is that in an RBAC policy, it is likely that a manager will inherit the permissions associated with role clerk, but rather unlikely that he will also inherit the associated obligations. On the other hand, a clerk in the bank organization would inherit the obligations of the general manager without acquiring more permissions.

Role hierarchy with respect to permissions

In its simplest form, role hierarchy with respect to permissions can be expressed by the rule

$$\pi(X, \texttt{acquire-perm}, \varepsilon, r_1) \leftarrow \pi(X, \texttt{acquire-perm}, \varepsilon, r_2) \tag{3.8}$$

Example 3.2.1. In a banking system, a manager can inherit the permissions associated with role clerk. This can be expressed by

$$\pi(X, \texttt{acquire-perm}, \varepsilon, \text{clerk}) \leftarrow \pi(X, \texttt{acquire-perm}, \varepsilon, \text{manager})$$

However, such a representation would necessitate to define a rule for each couple of roles that pertain to the same hierarchy. To simplify the implementation of role hierarchy, we define an order relative to permissions and denoted by $(<_{perm})$. We say r_1 is junior to r_2, and denote it $r_1 <_{perm} r_2$, if r_2 can inherit the permissions associated to the role r_1. In Example 3.2.1 the hierarchy can be defined by (clerk $<_{perm}$ manager).

Accordingly, role hierarchy with respect to permissions expressed as in rule 3.8 can be generalized to be expressed by the single rule:

$$\pi(X, \texttt{acquire-perm}, \varepsilon, r_1) \leftarrow \pi(X, \texttt{acquire-perm}, \varepsilon, r_2) \wedge (r_1 <_{perm} r_2) \tag{3.9}$$

where the truth of the hierarchy is tested by the order relation and the associated permissions are then granted accordingly.

Example 3.2.2. In the banking system a clerk is junior to an assistant-manager who in turn is junior to a manager. This can be specified by the relations clerk $<_{perm}$ assistant-manager and assistant-manager $<_{perm}$ manager in the initial security state $\mathcal{S}$.

Also, $Mary$ is a manager at the bank, (i.e. $\pi(\text{Mary}, \texttt{can-play}, \varepsilon, \text{manager})$ is true in $\mathcal{S}$.). If Mary decides to activate her manager role then:

- by rule 3.5 and 3.6, $\pi(\text{Mary}, \texttt{acquire-perm}, \varepsilon, \text{manager})$ is true at the next state $\mathcal{S}'$
- by the permission order relations in the banking system and rule 3.9, both $\pi(\text{Mary}, \texttt{acquire-perm}, \varepsilon, \text{assistant-manager})$ and $\pi(\text{Mary}, \texttt{acquire-perm}, \varepsilon, \text{clerk})$ will be true in $\mathcal{S}'$.
- thus, by rules 3.6 Mary automatically acquires the permissions for roles assistant-manager and clerk at the security state $\mathcal{S}'$

Role hierarchy with respect to obligations

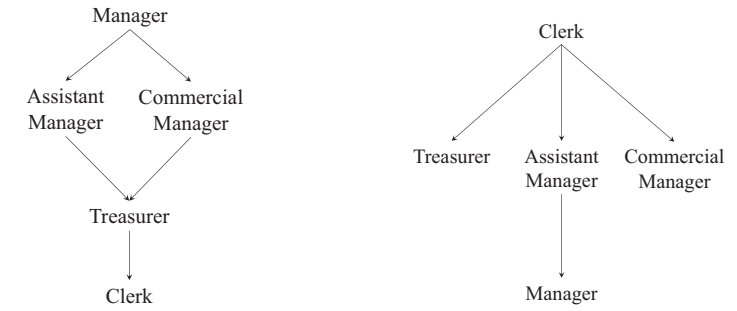
In the same manner, we define another inheritance relation for obligations denoted by $<_{obl}$. Given roles $r_1, r_2 \in R$, the expression $r_1 <_{obl} r_2$ means that r_2 inherits obligations from role r_1. As such the role hierarchy with respect to obligations is defined by a rule of the form:

$$\pi(X, \texttt{acquire-obl}, \varepsilon, r_1) \leftarrow \pi(X, \texttt{acquire-obl}, \varepsilon, r_2) \wedge (r_1 <_{obl} r_2) \qquad (3.10)$$

The role hierarchy rule for obligations is expressed as a permission rule rather than an obligation rule. We made this choice in order to keep the same encoding for all types of hierarchies.

Example 3.2.3. In the banking system, a clerk can inherit the obligations of the assistant manager. Such obligations can include the obligation to verify customer permission to access his personal data, or the obligation to write report in case a loan request was refused. The obligation role hierarchy is defined by the order relation assistant-manager $<_{obl}$ clerk. Thus, if Bob is a clerk, then by activating his role clerk Bob inherits all obligations associated with role assistant-manager by applying rules 3.5, 3.6 and 3.10 .

Note that the obligation role hierarchy does not depend on the permission-role hierarchy. That is for the same role, one can define two distinct hierarchies one that manages the inheritance of permissions and the other that manages the inheritance of obligations as can be seen in Figure 3.1



(a) Permissions hierarchy (b) Obligations hierarchy

Figure 3.1: Inheriting permissions and obligations

Role inheritance has side effects

Note that the consequences (or side-effect) of executing a permitted (or obligatory) action may vary depending on the role of the subject.

Example 3.2.4. Suppose a manager inherits the permission $\pi(X, \texttt{access}, \text{file}, \text{clerk})$ from the role clerk. Then both the manager and the clerk will have the permission to access the file. Additionally the *clerk* will also have the obligation to write a report.

We define a new order associated with the responsibilities of executing an action. Explicitly, $r_1 <_{\text{resp}} r_2$ means that r_2 has at least as much responsibilities as r_1. Acquiring responsibilities, similarly to acquiring permissions and obligations will be denoted by $\pi(X, \texttt{acquire-resp}, \varepsilon, r)$ and will be true when r is activated (see rule 3.6). This is expressed by the rule

$$\pi(X, \texttt{acquire-resp}, \varepsilon, r) \leftarrow \pi(X, \texttt{is-active}, \varepsilon, r) \tag{3.11}$$

Responsibility inheritance is expressed by:

$$\pi(X, \texttt{acquire-resp}, \varepsilon, r_1) \leftarrow \pi(X, \texttt{acquire-resp}, \varepsilon, r_2) \wedge (r_1 <_{resp} r_2) \tag{3.12}$$

As such Example 3.2.4 would be expressed by the order (manager $<_{\text{resp}}$ clerk), since the clerk has at least as much responsibilities as a manager, and the rule

$$\pi(X, \texttt{access}, file, clerk) \wedge \pi(X, \texttt{acquire-resp}, \varepsilon, manager) \rightarrow (\pi(X, \texttt{modify}, file, manager), \top) \tag{3.13}$$

$$\pi(X, \texttt{access}, file, clerk) \wedge \pi(X, \texttt{acquire-resp}, \varepsilon, clerk) \rightarrow (\omega(X, \texttt{write}, report, clerk), \top) \tag{3.14}$$

In fact with the specification of the responsibility order relation we can distinguish between the two following cases:

> If a subject in role manager chooses to execute the permission to access the file in the role clerk (by role inheritance), then only rule 3.13 will be satisfied and the permission to modify the file becomes true.
>
> If a subject in the role clerk chooses to execute the same permission, then rule 3.14 will be satisfied and the obligation to write a report becomes true, also by the responsibility order relation and rule 3.12, rule 3.13 is satisfied and the permission to modify the file also becomes true.

Encoding order relations

The order relations presented above can also be coded in our language by predicates of the form $\pi(r_1, \texttt{inherit}, x, r_2)$ to denote the relation $(r_2 <_x r_1)$ where $x \in \{\text{perm}, \text{obl}, \text{resp}\}$. The transitivity property is guaranteed by the general rule:

$$\pi(r_1, \texttt{inherit}, x, r_3) \leftarrow \pi(r_1, \texttt{inherit}, x, r_2) \wedge \pi(r_2, \texttt{inherit}, x, r_3)$$

Note that as in the case of role activation, role hierarchy is expressed in terms of the special action `acquire-perm` as opposed to `can-play` or `is-active`. In fact, we chose to represent role hierarchy in this manner in order to model the automatic inheritance of permissions, obligations or responsibilities. That is a subject does not need to *choose* to start role inheritance. The permissions (obligations or responsibilities) associated to junior roles are automatically acquired when a senior role is activated without the need to activate the junior roles. However, with the addition of the obligation and responsibilities hierarchies we lose the least privilege principle.

3.2.4 Role Delegation

Delegation is the act of authorizing or requesting someone to act on one's behalf. In order to be able to delegate a role r, an entity should be active in some role r_1 or allowed to the set of permissions and/or obligations associated with that role.

$$\pi(X, \texttt{delegate}, Y, r) \leftarrow (\pi(X, \texttt{acquire-perm}, \varepsilon, r_1) \vee \pi(X, \texttt{acquire-obl}, \varepsilon, r_1)) \wedge \pi(Y, \texttt{can-play}, \varepsilon, r_2)$$

Activating a delegation

We suppose that a delegation is performed in two steps. First, the delegator decides to execute the delegation, then the delegatee decides to accept the delegation. This can be expressed with the following dynamic rules

$$\pi(X, \texttt{delegate}, Y, r) \rightarrow (\pi(Y, \texttt{d-play}, \varepsilon, r), \top) \tag{3.15}$$

$$\pi(X, \texttt{d-play}, \varepsilon, r) \rightarrow (\pi(X, \texttt{acquire-perm}, \varepsilon, r) \wedge \pi(X, \texttt{acquire-obl}, \varepsilon, r)), \top) \tag{3.16}$$

The above rules specify that

- if X chooses delegate role r to Y, then Y gains the right to play r by delegation in the next state
- then, if Y chooses to activate this delegation, Y will acquire the permissions and delegations associated with the role r at the next state.

Example 3.2.5. In the banking system, the manager can delegate his duties to the assistant manager.

$$\pi(X, \texttt{delegate}, Y, manager) \leftarrow (\pi(X, \texttt{acquire-perm}, \varepsilon, manager) \wedge \pi(Y, \texttt{can-play}, \varepsilon, assistant - manager) \tag{3.17}$$

$Mary$ is a manager, she is on vacation, thus she decides to delegate her duties to her assistant manager $John$.

Let $\mathcal{S} = \langle \Omega, \Pi \rangle$ be the security state. Suppose that

$$\{\pi(Mary, \texttt{acquire-perm}, \varepsilon, manager), \pi(John, \texttt{can-play}, \varepsilon, assistant - manager)\} \subseteq \Pi,$$

then by rule 3.17,

$$\pi(Mary, \texttt{delegate}, John, manager) \in \Pi.$$

If DP contains rules 3.15 and 3.16, then:

- If Mary chooses to activate the delegation, (i.e. $\mathcal{A} \subseteq \mathcal{S}$ is such that $\pi(Mary, \texttt{delegate}, John, manager) \in \mathcal{A}^{\Pi}$) then
- by rule 3.15, the next state $\mathcal{S}_1 = \langle \Omega_1, \Pi_1 \rangle$ is such that

$$\pi(John, \texttt{d-play}, \varepsilon, manager) \in \Pi_1.$$

- If John chooses to activate the delegation, (i.e. $\mathcal{A}_1 \subseteq \mathcal{S}_1$ is such that $\pi(John, \texttt{d-play}, \varepsilon, manager) \in \mathcal{A}_1^{\Pi_1}$) then
- by rule 3.16, the next state $\mathcal{S}_2 = \langle \Omega_2, \Pi_2 \rangle$ is such that

$$\{\pi(John, \texttt{acquire-perm}, \varepsilon, manager), \pi(John, \texttt{acquire-obl}, \varepsilon, manager)\} \subset \Pi_2.$$

That is John can fulfill the duties associated with role manager.

Note that when the delegatee accepts the delegated role, the delegatee does not become a member of the role, nor an active member, instead, the delegatee only acquires the permissions and/or obligations associated with the role and eventually additional responsibilities that the delegatee can inherit from other roles in the permissions or obligation hierarchy.

Revoking a delegation

Since delegation is initiated by the *choice* to execute the delegation action, the revocation of the delegation occurs when the choice is *not to execute* the delegation. Namely, if X chooses not to activate the action delegate in rule 3.15, then Y loses his privileges at the next state.

Example 3.2.6. In Example 3.2.5, suppose that Mary is back from vacation. Thus Mary wants to revoke her delegation to John. Suppose the system is in state $\mathcal{S}_{\in}$. Recall that

$$\pi(Mary, \texttt{delegate}, John, manager) \in \Pi_2.$$

Suppose Mary decides not to execute the delegation, then, by rule 3.15 no new permission is added to the security state $\mathcal{S}_2$ and rule 3.16 is no more applicable, thus John loses his privileges at state $\mathcal{S}_3$.

An obligatory delegation

One can express the case when a delegator delegates the *obligation* to play the role. In this case, unlike the standard delegation, the delegatee will find himself obliged to accept the delegation. In order to express this kind of delegation we use the action `o-delegate`. As in the previous case, in order to enforce an obligatory delegation for a role r, an entity should be active in some role r_1 or allowed to the set of permissions and/or obligations associated with that role.

$$\begin{aligned}\pi(X, \texttt{o-delegate}, Y, r) \leftarrow (&\pi(X, \texttt{acquire-perm}, \varepsilon, r_1) \\ &\vee \pi(X, \texttt{acquire-obl}, \varepsilon, r_1)) \wedge \pi(Y, \texttt{can-play}, \varepsilon, r_2)\end{aligned}$$

The obligatory delegation is also expressed in two steps by the following dynamic rules:

$$\pi(X, \texttt{o-delegate}, Y, r) \rightarrow (\omega(Y, \texttt{d-play}, \varepsilon, r), \top) \tag{3.18}$$

$$\begin{aligned}&\omega(X, \texttt{d-play}, \varepsilon, r) \rightarrow \\ &\qquad (\pi(X, \texttt{acquire-perm}, \varepsilon, r) \wedge \pi(X, \texttt{acquire-obl}, \varepsilon, r) \wedge C, C')\end{aligned} \tag{3.19}$$

where C and C' are expressions denoting additional permissions or obligations that are to be added in case the delegation is accepted or refused by the delegatee respectively.

3.2.5 Separation of duties

The separation of duty principle can be seen in both its static and dynamic aspects.

Dynamic separation of duty

In the dynamic separation of duty, a subject may have the permission to play two mutually exclusive roles, but can become active in only one of them.

For example, a subject X can be both a manager and a customer at a bank, however X will not have the right to activate the role manager if X is currently playing the role customer (e.g. applying for a loan as a customer) in the same bank. We express this constraint as follows:

$$\pi(X, \texttt{acquire-perm}, \varepsilon, \text{manager}) \wedge \pi(X, \texttt{acquire-perm}, \varepsilon, \text{customer}) \rightarrow (\bot, \top)$$

Note that $\pi(X, \texttt{acquire-perm}, \varepsilon, r)$ is true only if a role r is activated, inherited or delegated, that is if X is active in the role. Thus the above rule states that if both permissions are executed, then no matter what X does, the system enters into an inconsistent state.

Static separation of duty

In the static separation of duty, a subject having the right to play the role teller in a bank will not be allowed to be a member of the role auditor of the same bank. A subject is considered a member of a role if she has the permission to play the role, or was delegated the role or inherited the permissions associated with the role. Accordingly, we define a special action $\widehat{\text{play}}$ such that

$$\pi(X, \widehat{\text{play}}, \varepsilon, r) \leftarrow \pi(X, a, \varepsilon, r) \text{ for } a \in \{\text{play}, \text{d-play}, \texttt{acquire-perm}\}$$

Then the static separation of duty can be expressed by the rule:

$$\pi(X, \widehat{\text{play}}, \varepsilon, \text{teller}) \wedge \pi(X, \widehat{\text{play}}, \varepsilon, \text{auditor}) \rightarrow (\bot, \bot)$$

If the subject X acquires both permissions at the same state of the dynamic model, then the system will enter into an inconsistent state, no matter what X decides to do.

3.2.6 Binding of duty

By binding of duty, we mean the necessity to execute two actions at the same time. For example, to open the safe in a bank both a clerk and a manager should enter a password otherwise the system would block:

$$\pi(X, \texttt{enter}, pswrd, clerk) \wedge \pi(Y, \texttt{enter}, pswrd, manager) \rightarrow (\top, \bot)$$

In this case we do not take into consideration the difference between not executing any action or executing only one action. However, in real life, executing only one may be considered as an intrusion in the system and should not be accepted. To express this case, the above rule must be replaced by the following set of rules:

$$\pi(X, \texttt{enter}, \text{pswrd}, \text{clerk}) \wedge \pi(Y, \text{enter}, \text{pswrd}, \text{manager}) \rightarrow (\top, p),$$

$$\pi(X, \text{enter}, \text{pswrd}, \text{clerk}) \rightarrow (p_{\text{clerk}}, \top),$$

$$\pi(Y, \text{enter}, \text{pswrd}, \text{manager}) \rightarrow (p_{\text{manager}}, \top),$$

$$p \wedge (p_{\text{clerk}} \vee p_{\text{manager}}) \rightarrow (\bot, \bot).$$

where p, p_{agent} and p_{manager} are ground predicates that will only be used in the last rule above. This will guarantee that if the agent or the manager execute the action alone, then the system will enter into an inconsistent state.

3.3 Assigning permissions

It is worth to note that a study on the computational complexity of assigning permissions and obligations was done in collaboration with Philippe Balbiani. In this section we present these decision problems, the complexity results are given in Table 3.1. The proofs can be found in [6].

Static assignments

The $STATIC(\exists)$ problem is the following decision problem:

- $STATIC(\exists)$:

  Input given a domain $\mathcal{D}$, a static policy SP based on $\mathcal{D}$ and a condition ϕ based on $\mathcal{D}$,

  Output Sat if there exists an interpretation function I for $\mathcal{D}$ such that $l(SP), I \models \phi$

This problem consists in checking whether the expression ϕ is satisfied with respect to the static policy SP.

Dynamic assignments

The $DYNAMIC(\exists, \exists)$ problem and the $DYNAMIC_{con}(\exists, \exists)$ problem are the following decision problems:

- $DYNAMIC(\exists, \exists)$ ($DYNAMIC_{con}(\exists, \exists)$):

  Input given a domain $\mathcal{D}$, a security state $\mathcal{S}$ based on $\mathcal{D}$, a (consistent) dynamic policy DP based on $\mathcal{D}$ and a condition ϕ based on $\mathcal{D}$,

Decision problem	Complexity
$STATIC(\exists)$	NP-complete
$DYNAMIC_{con}(\exists,\exists)$	NP-complete
$DYNAMIC(\exists,\exists)$	in $\Sigma_2 P$
$DYNAMIC^{path}_{con}(\exists,\exists)$	$PSPACE$-complete
$DYNAMIC^{path}(\exists,\exists)$	$PSPACE$-complete
$RULEBASED_{con}(\exists,\exists)$	NP-complete
$RULEBASED(\exists,\exists)$	in $\Sigma_2 P$
$RULEBASED^{path}_{con}(\exists,\exists)$	$PSPACE$-complete
$RULEBASED^{path}(\exists,\exists)$	$PSPACE$-complete

Table 3.1: Complexity results

Output Sat if there exists $\mathcal{A} \sqsubseteq \mathcal{S}$, there exists an interpretation function I for $\mathcal{D}$ such that $L(\mathcal{S}, DP, \mathcal{A}), I \models \phi$.

These problems consist in checking for a given expression ϕ there exists a set of executed actions that satisfies ϕ with respect to the dynamic policy DP.

The $DYNAMIC^{path}(\exists,\exists)$ problem and the $DYNAMIC^{path}_{con}(\exists,\exists)$ problem are the following decision problems:

- $DYNAMIC^{path}(\exists,\exists)$ ($DYNAMIC^{path}_{con}(\exists,\exists)$):

  Input given a domain $\mathcal{D}$, a security state $\mathcal{S}$ based on $\mathcal{D}$, a (consistent) dynamic policy DP based on $\mathcal{D}$ and a condition ϕ based on $\mathcal{D}$,

  Output Sat if there exists an integer $n \geq 0$ and security states $\mathcal{S}_0$, ..., $\mathcal{S}_n$ based on $\mathcal{D}$ such that
  - $\mathcal{S}_0 = \mathcal{S}$,
  - for all integers $i \geq 0$, if $1 \leq i \leq n$ then there exists $\mathcal{A} \sqsubseteq \mathcal{S}_{i-1}$ such that $\mathcal{S}_i = L(\mathcal{S}_{i-1}, DP, \mathcal{A})$,
  - there exists an interpretation function I for $\mathcal{D}$ such that $\mathcal{S}_n, I \models \phi$.

This problem consists in checking whether for a given expression ϕ there exists an initial security state $\mathcal{S}$ and an integer n, such that there exists a path from $\mathcal{S}$ such that after n transitions with respect to the dynamic policy DP and the choice of a set of executed actions, ϕ is true.

Rule-based assignments

The $RULEBASED(\exists,\exists)$ problem and the $RULEBASED_{con}(\exists,\exists)$ problem are the following decision problems:

- $RULEBASED(\exists,\exists)$ ($RULEBASED_{con}(\exists,\exists)$):

  Input given a domain $\mathcal{D}$, a security state $\mathcal{S}$ based on $\mathcal{D}$, a (consistent) rule-based policy $\mathcal{P}$ based on $\mathcal{D}$ and a condition ϕ based on $\mathcal{D}$,

Output Sat if there exists $\mathcal{A} \sqsubseteq \mathcal{S}$, there exists an interpretation function I for $\mathcal{D}$ such that $L(\mathcal{S}, \mathcal{P}, \mathcal{A}), I \models \phi$.

This problem consists in checking whether a given expression ϕ is satisfied with respect to a rule-based access control policy $\mathcal{P}$.

The $RULEBASED^{path}(\exists, \exists)$ problem and the $RULEBASED^{path}_{con}(\exists, \exists)$ problem are the following decision problems:

- $RULEBASED^{path}(\exists, \exists)$ ($RULEBASED^{path}_{con}(\exists, \exists)$):

  Input given a domain $\mathcal{D}$, a security state $\mathcal{S}$ based on $\mathcal{D}$, a (consistent) rule-based policy $\mathcal{P}$ based on $\mathcal{D}$ and a condition ϕ based on $\mathcal{D}$,

  Output Sat if there exists an integer $n \geq 0$ and there exists security states $\mathcal{S}_0, \ldots, \mathcal{S}_n$ based on $\mathcal{D}$ such that
  - $\mathcal{S}_0 = \mathcal{S}$,
  - for all integers $i \geq 0$, if $1 \leq i \leq n$ then there exists $\mathcal{A} \sqsubseteq \mathcal{S}_{i-1}$ such that $\mathcal{S}_i = L(\mathcal{S}_{i-1}, \mathcal{P}, \mathcal{A})$,
  - there exists an interpretation function I for $\mathcal{D}$ such that $\mathcal{S}_n, I \models \phi$.

This problem consists in checking whether for a given expression ϕ there exists an initial security state $\mathcal{S}$ and an integer n, such that there exists a path from $\mathcal{S}$ such that after n transitions with respect to the policy $\mathcal{P}$, ϕ is true.

3.4 Conclusion

In this chapter, we defined an access control logical framework based on rules. In this framework a policy consists in a set of static clauses and a set of dynamic clauses defined in terms of permissions and obligations. Static clauses characterize what remains true during the life of a system whereas dynamic clauses characterize the different ways according to which the system can change. We have provided examples on how to express RBAC features using static clauses and dynamic clauses and we have addressed the complexity issue of some decision problems related to the assignment of permissions and obligations with respect to this kind of policies.

Discussion. The main feature of the language proposed in this chapter is its capacity to express dynamic access control policies. It allows the specification of actual effects of an action execution. This feature encouraged us to direct our interest into the specification of business processes. A business process is a collection of structured activities or tasks that produce a specific service or product (serve a particular goal) for a particular customer.

To this end we validated our language in the expression and specification of business processes. In a car registration process presented in the context of the European project AVANTSSAR and described in [73] and presented in Chapter

5, the case study describes an e-government application consisting in registering a car purchase electronically via the interaction of different services.

The first obstacle encountered in our language is that it was too restrictive with respect to some specific requirements. In fact the role based structure of the permission and obligation predicates makes it complicated to specify some requirements. For example, the requirement that a given document is not used by more than one clerk at a time cannot be specified in the following framework. The difficulty here lies in the fact that the restriction is on one instance of an object. The syntax of our language is not capable of handling such constraints. Another difficulty came from the need to specify the order of the tasks in the business process. In fact, in the language presented in this chapter, it is relatively simple to express the effects for a given action, but we cannot provide a general specification for the business process execution.

These two main obstacles lead us to turn towards attribute based access control as a mean to express a large variety of constraints. Also the need to specify a business process took us in the direction of defining a more elaborate transition system to express in addition to action effects, an order over these actions that we call a workflow. In the next chapter we define an extension of our framework that takes into account these characteristics.

Chapter 4

Reasoning about policies with trust negotiation and workflows in a distributed environment

4.1 Introduction

There is an increasingly widespread acceptance of Service-Oriented Architecture as a paradigm for integrating software applications within and across organizational boundaries. In this paradigm, independently developed and operated applications and resources are exposed as (Web) services. These services communicate one with another by passing messages over HTTP, SOAP, etc. A fundamental advantage of this paradigm is the possibility to orchestrate existing services in order to create new business services adapted to a given task. Several languages (WS-CDL [48], WSBPEL [47], BPMN [77], ...) have been proposed to describe the workflow of an orchestrating service. These languages can be given an operational semantics in terms of (extension of) π-calculus [56] or Petri nets [44].

For business, security and legal reasons, it is necessary to control within a workflow and on the workflow interface in which contexts an action can be executed. This implies that, together with the workflow defining the orchestrating service one has to provide an application-level security policy describing the roles, separation of duty and other constraints to be enforced in the workflow. In order to foster agility (*i.e.* to specify the process so that it can be employed in a variety of environment) one usually adds a trust negotiation layer so that principals can get the chance to prove that they are legitimate users of the service.

Given the skills required to implement these aspects, they are usually sep-

arated into a security token server, an XACML firewall, a Business Process management system, plus additional ones for aspects abstracted in this chapter. We have chosen to describe services with logical *entities* that gather all the aspects pertaining to one application or resource.

Related works There already exists some works aiming at adding an access control aspect to workflows. In [17, 61] the access control is specified with roles that can execute activities, users that have attributes allowing them to enter roles, and ordering on activities. We believe that the RBAC-WS-BPEL language is significantly less expressive than our proposal. In particular it does not provide for dynamic separation of duty constraints, or other complex constraints based on the documents exchanged and the environment of execution. In [49] is proposed a framework in which even messages are interpreted as mobile processes, and in which processes communicate one with another to exchange credentials. The trust negotiation rules and their evaluation is similar to what we propose, but the workflow description is absent and thus we believe it to be much harder to express fine access control policies that depend on the current state of a process. Moreover the overall architecture is completely different. In [24] is presented a uniform framework for regulating the access to services and information disclosure in a distributed environment. The idea is based on using *accessibility rules* that set constraints on the access to the service and *disclosure rules* that specify constraints on the disclosure of a service information. In [80] the authors present a framework to reason about authorizations in a distributed system through knowledge bases for each service and the capacity to push information through sending messages and to pull information through queries. The trust negotiation mechanisms in [24, 80] are mainly based on Horn clauses as is the case in our trust negotiation rules, also the knowledge base used in [80] can be expressed via certificates and the access control policy in our framework. However in both cases, the notion of state is not explicit and their respective frameworks do not offer the ability to express the task execution or the dynamic evolution of the service. In [39, 11, 34] the workflow is embedded within the access control system, *i.e.* the possible evolutions of a process are embedded in the access control rules. Another point is that there is no notion of local state, which is replaced by the proof of reachability of a state. This approach implies that one cannot follow exactly how many times a given task is executed.

Contribution The main originality of this work is in two parts. On the one hand, the interplay between workflow execution and access control which is permitted by this unified framework. It permits us to express naturally the constraints that are encountered when dealing with real-life business processes. On the other hand, the addition of a trust negotiation layer allows us to express the interaction between different services in the environment. The strength of our frameworks lies thus in the collaboration of these three structures, namely, the access control policy, trust negotiation policy and the workflow in order

to express the evolution of a security state subject to interaction with other services in a distributed environment. These results were published in [7]. In Section 4.2 we present a sketch of our model, before presenting the syntax of our language in Section 4.3 that shall be used to define the security rules in Section 4.4. In Section 4.5 we give the evaluation semantics for the security rules. In Section 4.6 we present the different elements of the workflow and present the transition relations for the evolution of the workflow in Section 4.7. We conclude this chapter with an application in Section 4.8.

4.2 The model

We aim to present a logical framework that takes into account access control properties for entities in a distributed environment, but also the evolution of such entities depending on the execution of their functionalities. In fact we consider that an *entity* is responsible for the execution of certain functionalities. A *functionality* is a task provided by the entity such as *store, write, pay etc.* The access to these functionalities is protected by an access control policy in each entity. If allowed, the access to a functionality leads to a change in the *state* of the entity.

As we can see in Figure 4.1 an entity is viewed as a logical unit made up of a set of access control rules and a set of trust negotiation rules. Access control and trust negotiation rules are Horn clauses in a first-order language (see details in Section 4.4). *Access control rules* set the conditions to be satisfied in order for a task to be permitted within an entity whereas *trust negotiation rules* determine the conditions needed in order to send information (modeled as objects) to other entities. We assume that access control decisions may be constrained by the acquisition of such information.

To an entity is associated a local state that models the dynamic aspect in our framework. It consists of a repository and a substitution. The *repository* constitutes the database of the entity, i.e. it stores the objects holding information about users and objects used in the entity, and the *substitution* is a function that maps the local identifiers to their values. The local state of the entity changes in accordance with the tasks executed.

Finally, a *workflow* is also associated to the entity. It manages the evolution of the local state of each entity, and consists of a process that defines an ordering between the tasks executed in the entity.

4.3 Syntax

In this section we define the syntax that shall be used in the representation of the framework.

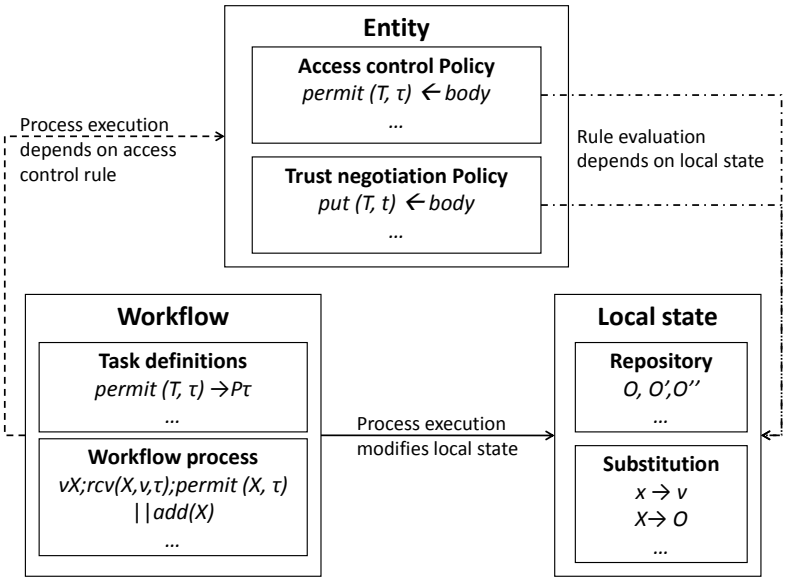


Figure 4.1: Entity model

4.3.1 Objects and values

We use the notions of *objects* and *values* to represent the elements of our framework.

Values

Let Val be a countable set of values (with typical members denoted $v,\ v',\ \ldots$). Values can be numbers, strings, Booleans, ... Let $Act \subseteq Val$ denote the set of actions (with typical members denoted $\mathtt{act}, \mathtt{act}',\ \ldots$).

Attributes and task names

Let Att be a finite set of attributes (with typical members denoted $\mathbf{att},\ \mathbf{att}',\ \ldots$) and $\mathcal{T}$ a countable set of task names (with typical members $\tau,\ \tau',\ \ldots$). We assume that $Att \cap \mathcal{T} = \emptyset$.

Object

We define an object as a partial function:

$$O : Att \to Val$$

and we denote by $\mathcal{O}$ the set of all objects. For each object O, let $dom(O)$ be the domain of O. We write $\perp$ to denote the unique object O such that $dom(O) = \emptyset$.

Moreover, we write $O.\mathbf{att}$ to denote $O(\mathbf{att})$ and we use the notation $O.\mathbf{att} = \perp$ for $\mathbf{att} \notin dom(O)$, i.e. to express that the object O is undefined for the attribute **att**.

Example 4.3.1. The information stating that a subject u_1 can play the role *clerk*, certified by a *central authority* (CA) is expressed by the object O

$$O : \begin{cases} \mathbf{subject} & \mapsto & u_1 \\ \mathbf{action} & \mapsto & \texttt{can}-\texttt{play} \\ \mathbf{role} & \mapsto & clerk \\ \mathbf{certifier} & \mapsto & ca \end{cases}$$

Object notation. In order to ease the readability of our security rules and thus for a better understanding of the expression of security properties in our framework, we use a shorthand notation by presenting objects in a human readable notation. We write a non-empty object

$$O : \ \mathbf{att_i} \ \mapsto \ v_i, \quad \text{for } i \in \{1, \ldots, k\}, \ att_i \in Att, \ v_i \in Val.$$

in the form

$$\{\mathbf{a_i} \ v_i\}_{i \in \{1,\ldots,k\}}$$

where $\{\mathbf{a_i} \ v_i\}_{i \in \{1,\ldots,k\}}$ denotes a sequence of attributes $\mathbf{att_i}$ followed by their corresponding value v_i.

In particular, if the object has at least an attribute **subject** and an attribute **action**:

$$O : \begin{cases} \mathbf{subject} & \mapsto & u, & u \in Val \\ \mathbf{action} & \mapsto & \texttt{act}, & \texttt{act} \in Act \\ \mathbf{att_i} & \mapsto & v_i, & \text{for } i \in \{1, \ldots, k\}, \ att_i \in Att, \ v_i \in Val. \end{cases}$$

then we write it in the form

$$\mathbf{subject} \ u \ \texttt{act} \ \{\mathbf{a_i} \ v_i\}_{i \in \{1,\ldots,k\}}$$

where $\{\mathbf{a_i} \ v_i\}_{i \in \{1,\ldots,k\}}$ denotes a sequence of attributes $\mathbf{att_i}$ followed by their corresponding value v_i.

Example 4.3.2. The object in Example 4.3.1 can thus be written as

$$\mathbf{subject} \ u \ \texttt{can}-\texttt{play} \ \mathbf{role} \ clerk \ \mathbf{certifier} \ ca$$

Repositories

We define a repository to be a finite set of objects. Repositories are typically denoted by *Rep* and decorations thereof.

4.3.2 Variables and terms

Variables

Let $VarObj$ be a countably infinite set of variables for objects (with typical members denoted $X,\ Y,\ \ldots$) and $VarVal$ be a countable infinite set of variables for values (with typical members denoted $x,\ y,\ \ldots$).

Terms

We define a term for objects (with typical members denoted $T, U, \ldots$) to be either a variable for objects or an object whereas a term for values (with typical members denoted $t, u, \ldots$) is either a variable for values, a value or an expression $X.a$.

Substitutions

A substitution σ is an idempotent partial mapping that associates to a variable a term of the same kind. For all terms T for objects and for all terms t for values, we define the terms $[\![T]\!]_\sigma$ and $[\![t]\!]_\sigma$ as follows:

$$[\![T]\!]_\sigma = \begin{cases} T, & \text{if } T \text{ is an object;} \\ \sigma(T), & \text{if } T \text{ is in } VarObj. \end{cases}$$

$$[\![t]\!]_\sigma = \begin{cases} t, & \text{if } t \text{ is a value;} \\ \sigma(t), & \text{if } t \text{ is in } VarVal; \\ \sigma(X).a, & \text{if } t \text{ is } X.a. \end{cases}$$

4.3.3 Entities and states

An entity

We define an entity e_t by means of a set of *security rules*, constituted of a set of *access control rules* controlling its functionalities and a set of *trust negotiation rules* that specify conditions on the disclosure of certificates to other entities in the environment (defined in Section 4.4).

Identifier

To each entity e_t we associate a value $t \in Val$ which is its unique identifier.

Local state

To each entity e_t we associate a *local state* s_t which is made of a repository and a substitution and is expressed by a pair $s_t = (Rep_t, \sigma_t)$.

Local substitution

Let σ_t be the substitution local to entity e_t at state s_t and let $dom(\sigma_i)$ denote the domain of σ_i. Given a global substitution σ, for all $X \in VarObj$, $x \in VarVal$, we define $[\![X]\!]_\sigma^{\sigma_i}$ and $[\![x]\!]_\sigma^{\sigma_i}$ as follows:

$$[\![X]\!]_\sigma^{\sigma_i} = \begin{cases} [\![X]\!]_{\sigma_i}, & X \in dom(\sigma_i), X \neq \bot; \\ [\![X]\!]_\sigma, & \text{otherwise.} \end{cases}$$

and

$$[\![x]\!]_\sigma^{\sigma_i} = \begin{cases} [\![x]\!]_{\sigma_i}, & x \in dom(\sigma_i), x \neq \bot; \\ [\![x]\!]_\sigma, & \text{otherwise.} \end{cases}$$

The rationale behind this is that each entity has its own local substitution that can be modified at a given security state (see Section 4.7). Also we suppose in the system the existence of a global substitution. Thus, the definition $[\![-]\!]_\sigma^{\sigma_i}$ ensures that the local variables of the entity are not overwritten by the global substitution σ.

4.4 Access control and trust negotiation

The security rules are Horn clauses written in a first order language. We present the body of security rules, then we give the syntax for the access control rules and trust negotiation rules.

4.4.1 Body of rules

The rules in our framework can specify conditions on the availability of an object, but can also define conditions on the content of an object. Accordingly, the body of the security rules is defined by:

$body := \top \mid Test \mid body \wedge body \mid body \vee body$

$Test := has(T) \mid not(has(T)) \mid get(T, t) \mid t_1 = t_2 \mid t_1 \neq t_2$ where T is a term for objects and t, t_1, t_2 are term for values.

The intended meaning for these expressions is as follows:

- $has(T)$ is true if the object denoted by T is in the repository of the entity;
- $not(has(T))$ is true if the object denoted by T is not in the repository of the entity;
- $get(T, t)$ is true if the object denoted by T can be received through a trust negotiation with the entity denoted by e_t.
- $t_1 = t_2$ is true if the values denoted by t_1 and t_2 are equal;

4.4.2 Security rules

Trust negotiation rules

The trust negotiation rules give the conditions to be satisfied for an object to be sent during a trust negotiation session. We express trust negotiation rules by:

$$put(T, t) \leftarrow body$$

where T is a term for objects and t a term for values. The predicate $put(T, t)$ allows the disclosure of an object T to an entity e_t whenever the conditions in the body of the rule are satisfied.

Example 4.4.1. The *certifying authority* only sends objects to entities that she trusts. To this end, the certifying authority represented by the entity e_{ca} has the following trust negotiation rule:

$$put(X, y) \leftarrow has(X) \wedge has(Y) \wedge Y.\textbf{certifier} = ca \\ \wedge Y.\textbf{subject} = y \wedge Y.\textbf{action} = \texttt{is-trusted} \quad (4.1)$$

This rule says that an object X can be sent to an entity e_y if

- the object X is in the repository of the entity
- there exist an object Y in the repository of the entity such that the values for the attributes **certifier**, **subject** and **action** are ca, the value of y and $\texttt{is} - \texttt{trusted}$ respectively.

Trust negotiation policy. We define a trust negotiation policy $\mathcal{TN}_t$ for an entity e_t to be the set of trust negotiation rules contained in e_t.

Access control rules

The access control rules give the conditions to be satisfied for the task τ to be *permitted*. Access control rules are of the form:

$$permit(T, \tau) \leftarrow body$$

where T is a term for objects and $\tau \in \mathcal{T}$ is a task name. If *body* holds in the rule above, then the predicate $permit(T, \tau)$ says that the task τ is permitted with respect to the object denoted by T.

Example 4.4.2. In a car registration process, a subject has the permission to store an object in the *central repository* if he can provide a proof from a *certifying authority* certifying that he is a clerk. This requirement can be expressed as an access control rule in the entity *central repository* as follows:

$$permit(X, store) \leftarrow get(Y, ca) \wedge Y.\mathbf{certifier} = ca \wedge Y.\mathbf{action} = \texttt{can} - \texttt{play} \\ \wedge Y.\mathbf{subject} = X.\mathbf{user} \wedge Y.\mathbf{role} = clerk \quad (4.2)$$

This rule says that the task *store* can be executed on an object denoted by X if a trust negotiation with the entity e_{ca} provides an object y such that:

- the values for the attributes **certifier**, **action** and **role** of object Y are ca, $\texttt{can} - \texttt{play}$ and $clerk$ respectively,
- the value for the attribute **subject** of object Y is equal to the value of the attribute **user** of object X.

Access control policy. We define an access control policy $\mathcal{AC}_t$ for an entity e_t by the set of access control rules of e_t.

Using the object notation

In Example 4.4.2 and Example 4.4.1 we used conjunctions over predicates of the form $X.a = v$ in the body of the rule to describe the relevant attributes in the variable for objects X. However, this notation can become cumbersome and complicates the readability of the rule when the number of objects in a rule exceeds two for example. In order to improve the readability of the security rules, we use the notation for objects presented in Section 4.3.1.

- A trust negotiation rule of the form

$$put(X, t) \leftarrow Cond \wedge (\bigwedge_{i \in 1,\ldots,k} X.\mathbf{att_i} = v_i)$$

  is equivalent to

$$X := \{\mathbf{att_i}\ v_i\}_{i \in \{i,\ldots,k\}} \\ put(X, t) \leftarrow Cond$$

- A trust negotiation rule of the form

$$put(X, t) \leftarrow Cond \wedge (\bigwedge_{i \in 1,\ldots,k} X.\mathbf{att_i} = v_i) \wedge [\bigwedge_{j \in \{1,\ldots,l\}} (\bigwedge_{i \in 1,\ldots,k} Y_j.\mathbf{att_{j_i}} = v_{j_i})]$$

  where Y_j are arguments of predicates in $Cond$ is equivalent to

$$X := \{\mathbf{att_i}\ v_i\}_{i \in \{i,\ldots,k\}} \\ Y_j = \left\{\mathbf{att_{j_i}}\ v_{j_i}\right\}_{i \in \{i,\ldots,k\}}, \quad j \in \{1, \ldots, l\} \\ put(X, t) \leftarrow Cond$$

- An access control rule of the form

$$permit(X, \tau) \leftarrow Cond \wedge (\bigwedge_{i \in 1,\ldots,k} X.\mathbf{att_i} = v_i)$$

is equivalent to

$$X := \{\mathbf{att_i}\ v_i\}_{i \in \{i,\ldots,k\}}$$
$$permit(X, \tau) \leftarrow Cond$$

- An access control rule of the form

$$permit(X, \tau) \leftarrow Cond \wedge (\bigwedge_{i \in 1,\ldots,k} X.\mathbf{att_i} = v_i) \wedge [\bigwedge_{j \in \{1,\ldots,l\}} (\bigwedge_{i \in 1,\ldots,k} Y_j.\mathbf{att_{j_i}} = v_{j_i})]$$

where Y_j are arguments of predicates in $Cond$ is equivalent to

$$X := \{\mathbf{att_i}\ v_i\}_{i \in \{i,\ldots,k\}}$$
$$Y_j = \left\{\mathbf{att_{j_i}}\ v_{j_i}\right\}_{i \in \{i,\ldots,k\}}, \quad j \in \{1, \ldots, l\}$$
$$permit(X, \tau) \leftarrow Cond$$

Example 4.4.3. Using the above notation, rule 4.2 of Example 4.4.2 can be written as:

$$X := \mathbf{user}\ u$$
$$Y := \mathbf{subject}\ u\ \texttt{can-play}\ \mathbf{role}\ clerk\ \mathbf{certifier}\ ca$$
$$permit(X, store) \leftarrow get(Y, ca)$$

and rule 4.1 of Example 4.4.1 can be written as:

$$Y := \mathbf{subject}\ y\ \texttt{is-trusted}\ \mathbf{certifier}\ ca$$
$$put(X, y) \leftarrow has(X) \wedge has(Y)$$

4.5 Semantics

In this section we provide the evaluation semantics for the security rules. Let $\mathcal{E} = (e_1, \ldots, e_n)$ be an n-tuple of entities, where $e_i = \langle \mathcal{AC}_i, \mathcal{TN}_i \rangle$, and let $\mathcal{S} = (s_1, \ldots, s_n)$ be an n-tuple of associated local states, as described in Section 4.3.3.

The use of the predicate $get()$ in the body of access control rules induces the fact that the rule evaluation of the access control policy for a given entity e_t depends on the result of a trust negotiation policy of other entities. Also, the rule evaluation semantics of trust negotiation rules is computed with respect to the trust negotiation policies of all the entities in $\mathcal{E}$. In Section 4.5.1 we define the computation of the result of a trust negotiation session that will be used in order to evaluate access control rules in Section 4.5.2.

4.5.1 Trust negotiation semantics

The trust negotiation mechanism involves communication of objects between different entities according to their respective trust negotiation policies.

Available objects

We define an *available object* to be a pair (O, i) where $O \in \mathcal{O}$ is an object and $i \in Val$ is a value denoting the identifier of an entity.

Informally, we say an object O is available to an entity e_j if there exists an entity e_i that can, according to its trust negotiation policy, send (put) the object O the entity e_j.

Trust negotiation round

We define a trust negotiation round as n tuple of sets of available objects.

$$\Lambda_k = (\Sigma^k_{\mathcal{E},\mathcal{S},e_1}, \ldots, \Sigma^k_{\mathcal{E},\mathcal{S},e_n})$$

where $\Sigma^k_{\mathcal{E},\mathcal{S},e_j}$ denotes the set of available objects for entity e_j at the beginning of round k.

Trust negotiation session

Let us consider a sequence

$$(\Lambda_0, \Lambda_1, \ldots)$$

of n-tuples of sets of *available objects* inductively defined as follows:

base case: $\Sigma^0_{\mathcal{E},\mathcal{S},e_1} = \emptyset, \ldots, \Sigma^0_{\mathcal{E},\mathcal{S},e_n} = \emptyset$,

induction: - Let e_i with $i \in \{1, \ldots, n\}$ be an entity. Suppose $\Sigma^k_{\mathcal{E},\mathcal{S},e_i}$ has already been defined for some non-negative integer $k \geq 0$,

- Define $\Sigma^{k+1}_{\mathcal{E},\mathcal{S},e_i}$ inductively with respect to $\Sigma^k_{\mathcal{E},\mathcal{S},e_i}$ as follows:

$$\Sigma^{k+1}_{\mathcal{E},\mathcal{S},e_i} = \Sigma^k_{\mathcal{E},\mathcal{S},e_i} \cup \Omega^k_{\mathcal{E},\mathcal{S},e_i}$$

where $\Omega^k_{\mathcal{E},\mathcal{S},e_i} = \{(O, j) : \mathcal{E}, \mathcal{S}, e_j \models_k put(O, i)\}$ is the set of available objects acquired by e_i during the k^{th} trust negotiation round.

In the rest of this subsection we present how $\Omega^k_{\mathcal{E},\mathcal{S},e_i}$ is computed with respect to Λ_k.

Truth conditions:

Given $\Sigma^k_{\mathcal{E},\mathcal{S},e_j}$ the set of available objects for entity e_j at the beginning of round k, and a substitution σ, we say

$$\mathcal{E}, \mathcal{S}, e_j, \sigma \models body$$

iff

- *body* is $\top$
- *body* is $has(T)$ and $[\![T]\!]_\sigma^{\sigma_j} \in Rep_j$
- *body* is $not(has(T))$ and $[\![T]\!]_\sigma^{\sigma_j} \notin Rep_j$
- *body* is $get(T,t)$ and $([\![T]\!]_\sigma^{\sigma_j}, [\![t]\!]_\sigma^{\sigma_j}) \in \Sigma^k_{\mathcal{E},\mathcal{S},e_j}$
- *body* is $t_1 = t_2$ and $[\![t_1]\!]_\sigma^{\sigma_j} = [\![t_2]\!]_\sigma^{\sigma_j}$
- *body* is $t_1 \neq t_2$ and $[\![t_1]\!]_\sigma^{\sigma_j} \neq [\![t_2]\!]_\sigma^{\sigma_j}$
- *body* is $body_1 \wedge body_2$ and $\mathcal{E},\mathcal{S},e_j,\sigma \models_k body_1$ and $\mathcal{E},\mathcal{S},e_j,\sigma \models_k body_2$
- *body* is $body_1 \vee body_2$ and $\mathcal{E},\mathcal{S},e_j,\sigma \models_k body_1$ or $\mathcal{E},\mathcal{S},e_j,\sigma \models_k body_2$

We say

$$\mathcal{E},\mathcal{S},e_j \models_k put(O,i)$$

if there exists in $\mathcal{TN}_j$ a rule

$$put(T,t) \leftarrow body$$

and there exists a substitution σ such that

- $[\![T]\!]_\sigma^{\sigma_j} = O, [\![t]\!]_\sigma^{\sigma_j} = i$
- $\mathcal{E},\mathcal{S},e_j,\sigma \models_k body$

The result of a trust negotiation session

For all $i = 1,\ldots,n$, *the result of a trust negotiation session* for e_i is denoted by $\Sigma^\omega_{\mathcal{E},\mathcal{S},e_i}$ and is defined as follows:

$$\Sigma^\omega_{\mathcal{E},\mathcal{S},e_i} = \bigcup_{k\geq 0} \Sigma^k_{\mathcal{E},\mathcal{S},e_i}$$

4.5.2 Access control evaluation semantics

Consider an entity $e_i \in \mathcal{E}$. The truth of bodies with respect to $\mathcal{E},\mathcal{S}$, e_i and a substitution σ is defined with respect to the trust negotiation result by induction as follows:

- $\mathcal{E},\mathcal{S},e_i,\sigma \models \top$
- $\mathcal{E},\mathcal{S},e_i,\sigma \models has(T)$ iff $[\![T]\!]_\sigma^{\sigma_i} \in Rep_i$
- $\mathcal{E},\mathcal{S},e_i,\sigma \models not(has(T))$ iff $[\![T]\!]_\sigma^{\sigma_i} \notin Rep_i$
- $\mathcal{E},\mathcal{S},e_i,\sigma \models get(T,t)$ iff $([\![T]\!])_\sigma^{\sigma_i}, [\![t]\!]_\sigma^{\sigma_i}) \in \Sigma^\omega_{\mathcal{E},\mathcal{S},e_i}$
- $\mathcal{E},\mathcal{S},e_i \models t_1 = t_2$ iff $[\![t_1]\!]_\sigma^{\sigma_i} = [\![t_2]\!]_\sigma^{\sigma_i}$

- $\mathcal{E}, \mathcal{S}, e_i, \sigma \models t_1 \neq t_2$ iff $[\![t_1]\!]^{\sigma_i}_{\sigma} \neq [\![t_2]\!]^{\sigma_i}_{\sigma}$
- $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body \wedge body'$ iff $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body$ and $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body'$
- $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body \vee body'$ iff $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body$ or $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body'$

A task is permitted

We will say that a task $\tau \in \mathcal{T}$ "is permitted" with respect to an object $O : Att \to Val$ for $\mathcal{E}, \mathcal{S}, e_i$ and denote it by

$$\mathcal{E}, \mathcal{S}, e_i \models permit(O, \tau)$$

iff $\mathcal{AC}_i$ contains an access control rule of the form

$$permit(T, \tau) \leftarrow body$$

and there exists a substitution σ such that the following conditions are satisfied:

- $[\![T]\!]^{\sigma_i}_{\sigma} = O$
- $\mathcal{E}, \mathcal{S}, e_i, \sigma \models body$

Example 4.5.1. We revisit Example 4.4.2 and Example 4.4.1 and suppose that rule 4.2 is in the access control policy of entity e_{cr} denoting a *central repository* and the rule 4.1 is in the trust negotiation policy of entity e_{ca} denoting a *central authority.*
We consider the set of attributes

$$Att = \{\textbf{certifier}, \textbf{subject}, \textbf{role}, \textbf{user}, \textbf{action}, \textbf{status}\}$$

and the set of values

$$Val = \{cr, ca, john, clerk, \texttt{is}-\texttt{trusted}, \texttt{can}-\texttt{play}, inuse\}.$$

We consider the set of objects

$$\{Role, Trust, Doc_0\},$$

where

$Role :=$ **subject** $john$ **action** $\texttt{can}-\texttt{play}$ **certifier** ca **role** $clerk$
$Trust :=$ **subject** cr **action** $\texttt{is}-\texttt{trusted}$ **certifier** ca
$Doc_0 :=$ **user** $john$ **status** $= inuse$

And we consider the entities e_{cr} and e_{ca} defined as follows:

Entity e_{ca}

$X :=$ **subject** u $\texttt{act}$ $\{\textbf{att}_\textbf{i}\ v_i\}_{i \in \{i,\ldots,k\}}$
$permit(X, \tau) \leftarrow Cond$

Entity e_{cr}

$$X := \textbf{user } u$$
$$Y := \textbf{subject } u \texttt{ can} - \texttt{play } \textbf{role } clerk \textbf{ certifier } ca$$
$$permit(X, store) \leftarrow get(Y, ca)$$

Their associated local states are defined as follows:

Local state s_{ca}: The local state s_{ca} associated with the entity e_{ca} is given by the pair (Rep_{ca}, σ_{ca}) such that

The repository Rep_{ca} associated with the entity e_{ca} is $\{Role, Trust\}$ and,

the substitution σ_{ca} associated with e_{ca} is the empty substitution.

Local state s_{cr}: The local state s_{cr} associated with entity e_{cr} is given by the pair (Rep_{cr}, σ_{cr}) such that:

The repository Rep_{cr} associated with the entity e_{cr} is the empty repository and,

the substitution σ_{cr} associated with e_{cr} is the empty substitution.

Let $\mathcal{E} = (e_{ca}, e_{cr})$ and $\mathcal{S} = (s_{ca}, s_{cr})$.

In e_{ca} Computing the result of the trust negotiation session with respect to the substitution $\sigma = \{X \mapsto Role, Y \mapsto Trust, y \mapsto cr\}$

- $has(Role)$ and $has(Trust)$ are true with respect to $\mathcal{E}, \mathcal{S}, e_{ca}$ since $Role$ and $Trust$ are in the repository of ca in the local state s_{ca}
- The equalities $Trust.\textbf{certifier} = ca$, $Trust.\textbf{subject} = cr$ and $Trust.\textbf{action} = \texttt{is} - \texttt{trusted}$ all hold.

Thus, the body of the trust negotiation rule in e_{ca} is satisfied with respect to $\mathcal{E}, \mathcal{S}, e_{ca}, \sigma$ and

$$\mathcal{E}, \mathcal{S}, e_{ca} \models_0 put(Role, cr).$$

In this case we say that the entity ca can send the object $Role$ to the entity cr (or $Role$ is an available object for cr) via a trust negotiation session and $(Role, ca)$ is added to the set of *available objects* $\Sigma^0_{\mathcal{E},\mathcal{S},e_{cr}}$. Note however that in this example, the result of the trust negotiation session with respect to $\mathcal{E}, \mathcal{S}$, e_{cr} and σ is given by

$$\Sigma^{\omega}_{\mathcal{E},\mathcal{S},e_{cr}} = \Sigma^{1}_{\mathcal{E},\mathcal{S},e_{cr}} = \{(Role, ca)\}$$

In e_{cr} Evaluating the access control rule with respect to the substitution $\sigma = \{X \mapsto Doc_0, Y \mapsto Role\}$:

- $get(Role, ca)$ is true with respect to $\mathcal{E}, \mathcal{S}, e_{cr}$ since $(Role, ca) \in \Sigma^{\omega}_{\mathcal{E},\mathcal{S},e_{cr}}$

- The equalities $Role.\mathbf{certifier} = ca$, $Role.\mathbf{action} = \texttt{can} - \texttt{play}$, $Role.\mathbf{subject} = Doc_0.\mathbf{user}$, $Role.\mathbf{role} = clerk$ all hold.

Thus the body of the access control rule in e_{cr} is satisfied with respect to $\mathcal{E}, \mathcal{S}, e_{cr}$ and σ.

Hence we can say that the task *store* is permitted with respect to the object Doc_0, that is

$$\mathcal{E}, \mathcal{S}, e_{cr} \models permit(Doc_0, store).$$

In Section 4.4 and this section we presented the security aspect of our model by defining the rules managing the access to a given entity on the one hand, and rules regulating the exchange of objects between entities on the other hand. The interaction between these two sets of rules allows us to define the access control policy of one entity with respect to the negotiation policies of the other entities as is the case with the task *store* in Example 4.5.1. However, in order to model a business process, one needs to add a dynamic aspect. The dynamic policy should on one hand define the effects of a permitted task, and on the other hand be able to express an order over such tasks. In the next section we define a workflow structure whose elements will englobe the above mentioned properties.

4.6 Workflow: Syntax

When writing a Business Process, one usually differentiates between atomic actions, tasks which according to [38] are partial orderings on atomic actions, and business roles which are entities to which a set of tasks is assigned. We have chosen instead to consider only the *notion of task as a named process that encompasses the notions of activity, task and role.* As mentioned in Section 4.2, an entity is responsible for the execution of a number of such tasks. Moreover, we assume that entities can communicate by means of sending and receiving messages. A message is usually sent by one entity to a given task in another entity in the environment. Consequently, to each entity we associate a workflow. The workflow is responsible for the update of the local state of the entity by means of execution of atomic actions. The possibility to execute these actions is subject to the security rules of the entity, the local state associated with the entity and the messages received from other entities.

We define processes in a language whose syntax is borrowed from existing process algebra languages and define *actions* as the atomic components in the process constructs. An action is *possible* in a process if there exists a reduction rule that consumes this action. We first present these atomic actions, then we define processes and the workflow.

4.6.1 Atomic actions

The atomic actions are responsible for the update of the local state associated with an entity e_i. Each action affects a component of its local state (the repository or the substitution) but also models the exchange of messages with other entities. We distinguish between different types of actions that we define as follows:

$$\begin{aligned} action := \quad & snd(T,t,\tau) \mid rcv(T,t,\tau) \\ & \mid add(T) \mid rmv(T) \\ & \mid \nu X \mid \nu x \mid X := T \mid X.\mathbf{att} := t \mid x := t \\ & \mid permit(T,\tau) \end{aligned}$$

where T is a term for objects, t a term for values, X and x variables for objects and values respectively, τ a task name, and **att** an attribute. Let us now describe these different actions.

- $snd(T,t,\tau)$ and $rcv(T,t,\tau)$ model the communication between the entities in the environment. $snd(T,t,\tau)$ sends an object T to an entity e_t requesting the permission to execute the task τ in e_i whereas $rcv(T,t,\tau)$ receives an object T from the entity e_t as a request for the permission to execute the task τ in e_i.

- $add(T)$ and $rmv(T)$ update the repository of the entity e_i by adding the object T to the repository of e_i, and removing T from the repository of e_i respectively.

- νX and νx extend the substitution of the entity e_i with new variable X for object and new variable x for values respectively and maps them to the object $\perp$.

- $X := T, x := t$ and $X.\mathbf{att} := t$ modify the substitution of the entity e_i.

  $X := T$ and $x := t$ express the assignment of a new term for objects T to the variable X and a new term for values t for the variable x respectively.

  $X.\mathbf{att} := t$ modifies the object denoted by X by assigning a term for values to the attribute **att** in X; if t is $\perp$ it undefines the attribute **att** in X and if the attribute **att** is not defined in X, it extends X by adding **att** to the domain of X and assigns the term for values t to the freshly added attribute.

- $permit(T,\tau)$ consists in

  (i) checking if the task τ is permitted with respect to the object T for e_i

  (ii) in which case $permit(T,\tau)$ is replaced by a named process associated to it as described in Section 4.6.2.

4.6.2 Processes and workflow

Processes

The set of all processes (with typical members P, P', ...) is defined by the following grammar:

$$P := skip \mid action \mid action; P \mid P! \mid P||P \mid P + P$$

where *skip* denotes the empty process and ;, !, || and + stand respectively for the sequence, iteration, parallel composition and non-deterministic choice of processes.

Tasks

A task definition is an expression of the form:

$$Task := permit(T, \tau) \rightarrow P$$

where T is a term for objects, τ a task name and P is a process.

Workflows

A workflow associated to an entity is specified by a set $\mathcal{T}$ of task definitions of the form $permit(X, \tau) \rightarrow P_\tau$ and by a workflow process P. Thus, it is represented as a pair $(\mathcal{T}, P)$.

Example 4.6.1. In this example we define a workflow W_{cr} associated with the entity e_{cr} (see Example 4.4.2). We suppose that X and Y_{Doc} are variables for objects and u is a variable for values. We write $W_{cr} = (\mathcal{T}_{cr}, P_{cr})$ where:

$$\mathcal{T}_{cr} = \{permit(X, store) \rightarrow X.\mathbf{status} := \bot; add(X)\}$$

and

$$P_{cr} = \nu u; \nu Y_{Doc}; recv(Y_{Doc}, u, store); permit(Y_{Doc}, store)$$

The above expression consists of

- a task definition for $permit(X, store)$ by the process $X.\mathbf{status} := \bot; add(X)$ that undefines the attribute **status** for the object denoted by X and then adds the object X to the repository of e_i and,
- a process that generates new variables u and Y_{Doc}, receives the object Y_{Doc} from the entity u in the task *store* and then executes $permit(Y_{Doc}, store)$. Note that executing $permit(Y_{Doc}, store)$ consists of
  1- checking if *store* is permitted with respect to Y_{Doc} for e_i and in which case,
  2- replacing $permit(Y_{Doc}, store)$ by the associated process

$$X.\mathbf{status} := \bot; add(X).$$

4.7 Workflow: operational semantics

The operational semantics for the workflow is given by the definition of reduction rules for processes.

The set of messages

Let $\mathcal{M}$ be the multi-set of messages that are sent but not yet received. Members of the set $\mathcal{M}$ are quadruplets of the form (O, s, r, τ) where O is an object, s and r are values denoting the sending and receiving entities in the environment respectively and τ is the intended task in entity e_r. We assume that a reception is executable in entity e_r only if there exists a matching message sent by some entity e_s that is waiting to be received.

The model

Let $\mathcal{E} = (e_1, \ldots, e_n)$ be an $n-$tuple of entities, $\mathcal{S} = (s_1, \ldots, s_n)$ an $n-$tuple of associated local states where $s_i = (Rep_i, \sigma_i)$ and $\mathcal{W} = (W_1, \ldots, W_n)$ an $n-$tuple of associated workflows where $W_i = (\mathcal{T}_i, P_i)$.

In Section 4.7.1, we give the operational semantics for a given workflow process P_i in the workflow W_i associated with the entity e_i by defining the reduction rules for the atomic action defined in Section 4.6.1 with respect to $\mathcal{M}, \mathcal{S}, \mathcal{W}$ restricted to the given entity e_i. That is, we consider the model restricted to $\mathcal{M}$, s_i the i^{th} component of $\mathcal{S}$ and P_i the i^{th} component of the workflow process in $\mathcal{W}$. We give the semantics for the *global transition* with respect to $\mathcal{M}, \mathcal{S}$ and $\mathcal{W}$ in Section 4.7.2.

4.7.1 Local transitions

Consider an entity $e_i \in \mathcal{E}$, and let $W_i = (\mathcal{T}_i, P_i)$ be the workflow associated to e_i. We assume that P_i can be one of the following:

- *skip* or an atomic action as described in Section 4.6.1;
- a composed process: $action; P_i^1 \mid P_i^1 || P_i^2 \mid P_i^1 + P_i^2 \mid P_i^1!$.

Skip and atomic actions semantics

We present in this subsection the operational semantics for atomic actions by means of transition relations denoted by describing the local state transition from $\mathcal{M}, s_i, P_i$, (where $s_i = (Rep_i, \sigma_i)$ and P_i is the workflow process in W_i) to a new state $\mathcal{M}', s_i', P_i'$ where the update of $\mathcal{M}'$, s_i', and P_i' is defined with respect to the atomic actions as follows:

Send action: If P_i is $send(T, t, \tau)$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M} \cup (\llbracket T \rrbracket_{\sigma_i}, i, \llbracket t \rrbracket_{\sigma_i}, \tau), s_i' = s_i, P_i' = skip$$

Receive action: If P_i is $rcv(T, t, \tau)$ and there exists an object O, a value s and a substitution σ such that $(O, s, i, \tau) \in \mathcal{M}$ and $\llbracket T \rrbracket_\sigma = O, \llbracket t \rrbracket_\sigma = s$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M} \setminus (O, i, s, \tau), s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows:

For all variables for values y:

$$\sigma_i'(y) = \begin{cases} s, & \text{if } t \text{ is the variable } y \\ \sigma_i(y) & \text{otherwise.} \end{cases}$$

For all variables for objects Y:

$$\sigma_i'(Y) = \begin{cases} O, & \text{if } Y \text{ is the variable } T\ ; \\ O', & \text{if } Y.\mathbf{att} \text{ is } t \text{ and } Y \text{ is not } T, \\ \sigma_i(Y) & \text{otherwise,} \end{cases}$$

where O' is an object whose domain consists of the attribute **att** and such that $O'.\mathbf{att} = s$.

Add action: If P_i is $add(T)$ and either T is an object or T is a variable for objects s.t. $T \in dom(\sigma_i)$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i \cup \left\{ \llbracket T \rrbracket_{\sigma_i} \right\}, \sigma_i), P_i' = skip$$

Remove action: If P_i is $rmv(T)$ and either T is an object or T is a variable for objects in $dom(\sigma_i)$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i \setminus \left\{ \llbracket T \rrbracket_{\sigma_i} \right\}, \sigma_i), P_i' = skip$$

Variable creation(objects): If P_i is νX and $X \notin dom(\sigma_i)$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows: For all variables y for values $\sigma_i'(y) = \sigma_i(y)$, and for all variables Y for objects such that $Y \in dom(\sigma_i) \cup X$,

$$\sigma_i'(Y) = \begin{cases} \perp & \text{if } Y \text{ is } X; \\ \sigma_i(Y), & \text{otherwise.} \end{cases}$$

Variable creation(values): If P_i is νx and $x \notin dom(\sigma_i)$ then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows:

For all variables Y for objects $\sigma_i'(Y) = \sigma_i(Y)$, and for all variables y for values such that $y \in dom(\sigma_i) \cup x$,

$$\sigma_i'(y) = \begin{cases} \perp & \text{if } y \text{ is } x; \\ \sigma_i(y), & \text{otherwise.} \end{cases}$$

Variable assignment(objects): If P_i is $X := T$ and either T is an object or T is a variable for objects and $T \in dom(\sigma_i)$ then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows:

For all variables y for values $\sigma_i'(y) = \sigma_i(y)$, and for all variables Y for objects

$$\sigma_i'(Y) = \begin{cases} [\![T]\!]_{\sigma_i} & \text{if } Y \text{ is } X; \\ \sigma_i(Y) & \text{otherwise.} \end{cases}$$

Variable assignment(values): If P_i is $x := t$ and t is a value, or t is a variable for values s.t $t \in dom(\sigma_i)$ or t is $X.\mathbf{att}$ s.t $X \in dom(\sigma_i)$, then we write

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows:

For all variables Y for objects $\sigma_i'(Y) = \sigma_i(Y)$, and for all variables y for values

$$\sigma_i'(y) = \begin{cases} [\![t]\!]_{\sigma_i} & \text{if } y \text{ is } x; \\ \sigma_i(y) & \text{otherwise.} \end{cases}$$

Attribute creation: If P_i is $X.\mathbf{att} := t$, $X \in dom(\sigma_i)$ then we write

$$\mathcal{M}, s_i, P_i \to \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = skip$$

where σ_i' is defined with respect to σ_i as follows: For all variables y for values $\sigma_i'(y) = \sigma_i(y)$, and for all variables Y for objects such that $Y \neq X$, $\sigma_i'(Y) = \sigma_i(Y)$ and $\sigma_i'(X)$ is defined as follows:

$$\sigma_i'(X).\mathbf{a} = \begin{cases} [\![t]\!]_{\sigma_i}, & \mathbf{a} = \mathbf{att}; \\ \sigma_i(X).\mathbf{a}, & \forall \mathbf{a} \in dom(\sigma_i(X)),\ \mathbf{a} \neq \mathbf{att}. \end{cases}$$

Task invocation If P_i is $permit(T, \tau)$ and $\mathcal{E}, \mathcal{S}, e_i, \sigma \models permit([\![T]\!]_\sigma^{\sigma_i}, \tau)$ and there exists a task definition $permit(T', \tau) \to P_\tau$ in $\mathcal{T}_i$ (the set of task definition in W_i) and a substitution δ of domain the variable T' such that $\delta(T') = T$, then we write

$$\mathcal{M}, s_i, P_i \to \mathcal{M}', s_i', P_i'$$

if

$$\mathcal{M}' = \mathcal{M}, s_i' = (Rep_i, \sigma_i'), P_i' = \delta(P_\tau)$$

where σ_i' defined with respect to σ_i as follows:

For all variables y for values $\sigma_i'(y) = \sigma_i(y)$, and for all variables Y for objects

$$\sigma_i'(X) = \begin{cases} [\![T]\!]_\sigma^{\sigma_i}, & \text{if } X \text{ is } T \\ \sigma_i(X) & \text{otherwise} \end{cases}$$

Composed processes

We define transition relations for composed processes as follows:

Sequence: P_i is $action; P_i^1$ where $action$ is an atomic action distinct from $permit(T, \tau)$:

$$\frac{\mathcal{M}, s_i, action \to \mathcal{M}', s_i', skip}{\mathcal{M}, s_i, action; P_i^1 \to \mathcal{M}', s_i', P_i^1}$$

P_i is $permit(T, \tau); P_i^1$:

$$\frac{\mathcal{M}, s_i, permit(T, \tau) \to \mathcal{M}', s_i', P_\tau}{\mathcal{M}, s_i, permit(T, \tau); P_i^1 \to \mathcal{M}', s_i', P_\tau; P_i^1}$$

Parallel Composition P_i is $P_i^1||P_i^2$:

$$\frac{\mathcal{M}, s_i, P_i^1 \rightarrow \mathcal{M}', s_i', P_i^{1\prime}}{\mathcal{M}, s_i, P_i^1||P_i^2 \rightarrow \mathcal{M}', s_i', P_i^{1\prime}||P_i^2}$$

and,

$$\frac{\mathcal{M}, s_i, P_i^2 \rightarrow \mathcal{M}', s_i', P_i^{2\prime}}{\mathcal{M}, s_i, P_i^1||P_i^2 \rightarrow \mathcal{M}', s_i', P_i^1||P_i^{2\prime}}$$

We identify $P||skip$ and $skip||P$ with the process P.

Non-deterministic choice P_i is $P_i^1 + P_i^2$:

$$\frac{\mathcal{M}, s_i, P_i^1 \rightarrow \mathcal{M}', s_i', P_i^{1\prime}}{\mathcal{M}, s_i, P_i^1 + P_i^2 \rightarrow \mathcal{M}', s_i', P_i^{1\prime}}$$

and

$$\frac{\mathcal{M}, s_i, P_i^2 \rightarrow \mathcal{M}', s_i', P_i^{2\prime}}{\mathcal{M}, s_i, P_i^1 + P_i^2 \rightarrow \mathcal{M}', s_i', P_i^{2\prime}}$$

Iteration P_i is $P_i^1!$:

$$\frac{\mathcal{M}, s_i, P_i^1 \rightarrow \mathcal{M}', s_i', P_i^{1\prime}}{\mathcal{M}, s_i, P_i^1! \rightarrow \mathcal{M}', s_i', P_i^{1\prime}||P_i^1!}$$

4.7.2 Global state and global transition

In Section 4.7.1 we presented the semantics for the transitions associated to a specific entity. We now give a formal description of a transition occurring in the general model. Let $\mathcal{E} = (e_1, \ldots, e_n)$ be an $n-$tuple of entities, we define a *global state* for $\mathcal{E}$ by $\mathcal{M}, \mathcal{S}, \mathcal{W}$ where $\mathcal{M}$ is the set of messages sent but not yet received, $\mathcal{S} = (s_1, \ldots, s_n)$ is an $n-$tuple of associated local states and $\mathcal{W} = (W_1, \ldots, W_n)$ is an $n-$tuple of associated workflows.

We say a transition from state $\mathcal{M}, \mathcal{S}, \mathcal{W}$ to a state $\mathcal{M}', \mathcal{S}', \mathcal{W}'$ is possible and denote it

$$\mathcal{M}, \mathcal{S}, \mathcal{W} \rightarrow \mathcal{M}', \mathcal{S}', \mathcal{W}'$$

iff the following conditions are satisfied:

- there exists $i \in \{1, \ldots, n\}$ and
- there exists local states s_i and s_i' associated to e_i such that:

$$\mathcal{M}, s_i, P_i \rightarrow \mathcal{M}', s_i', P_i' \text{ (as defined in Section 4.7.1)}$$

and, for all $j \in \{1, \ldots, n\}$, if $j \neq i$ then $s'_j = s_j$ and $P'_j = P_j$.

Example 4.7.1. We refer to the workflow given in Example 8.1.3, and the access control policy of the entity e_{cr} given in Example 4.4.2, and suppose that the workflow is associated with entity e_{cr}. We define a local state s_{cr} associated with e_{cr} by (Rep_{cr}, σ_{cr}), where Rep_{cr} is the empty repository and σ_{cr} is the empty substitution. Also let $\mathcal{M} = \{(Doc_0, ca, i, store)\}$. Recall that X and Y_{Doc} are variables for objects, Doc_0 is an object such that $Doc_0.\mathbf{user} = John$ and $Doc_0.\mathbf{status} = inuse$, Doc'_0 is an object, u is a variable for values, and ca, i are values. We shall present the evolution of the workflow process P_{cr} with respect to the local state of the entity e_{cr}.

The first executable action in the workflow process P_{cr} is νu that creates a new variable for values u, and results in the local state where Rep_{cr} is the empty repository and σ_{cr} is such that $\sigma_{cr}(u) = \perp$. The workflow process state is

$$\nu Y_{Doc}; recv(Y_{Doc}, u, store); permit(Y_{Doc}, store)$$

The action νY_{Doc} is now executable and creates a new variable for objects Y_{Doc}, and results in the local state where Rep_{cr} is the empty repository and σ_{cr} is such that $\sigma_{cr}(u) = \perp$ and $\sigma_{cr}(Y_{Doc}) = \perp$ and the workflow process state is

$$recv(Y_{Doc}, u, store); permit(Y_{Doc}, store)$$

The action $recv(Y_{Doc}, u, store)$ is now executable. This action can be executed since the message $(Doc_0, ca, i, store) \in \mathcal{M}$ is waiting to be received, and it will result in the local state where Rep_{cr} is the empty repository and σ_{cr} is such that $\sigma_{cr}(u) = ca$, $\sigma_{cr}(Y_{Doc}) = Doc_0$ and the workflow process state is

$$permit(Y_{Doc}, store)$$

This action is then replaced by the definition of $permit(X, store)$ by substituting X with $\sigma_{cr}(Y_{Doc})$. This replacement is permitted since $\mathcal{E}, \mathcal{S}, e_{cr} \models permit(Doc_0, store)$ is true (as shown in Section 4.5.1), and will result in the local state where Rep_{cr} is the empty repository and σ_{cr} is such that $\sigma_{cr}(u) = ca$ and $\sigma_{cr}(Y_{Doc}) = Doc_0$ and the workflow process state is

$$Y_{Doc}.\mathbf{status} := \perp; add(Y_{Doc})$$

The action $Y_{Doc}.status := \perp$ is executable for Doc_0 and result in the local state where Rep_{cr} is the empty repository and σ_{cr} is such that $\sigma_{cr}(u) = ca$ and $\sigma_{cr}(Y_{Doc}) = Doc'_0$ where $Doc'_0.\mathbf{user} = John, Doc'_0.\mathbf{status} = \perp$ and the workflow process state is

$$add(Y_{Doc})$$

Finally the action $add(Y_{Doc})$ is executable for the object Doc'_0 and results in the final state where Rep_{cr} is $\{Doc'_0\}$ and σ_{cr} is such that $\sigma_{cr}(u) = ca$, $\sigma_{cr}(Y_{Doc}) = Doc'_0$ and the workflow process state is

$$skip.$$

4.8 Application to access control problems

The framework presented in this chapter allows the expression of complex access control policies that may be difficult to express with role based access control($RBAC$) [69] or $RBAC$-based extensions. More specifically, we mentioned, at the end of Chapter 3, that access control decisions may be related to information on the *object* and not only specifications related to the *user*. An example for such a requirement is the constraint that an object can be used by only one *user* at a given time. It is clear that such a requirement cannot be easily expressed within a standard $RBAC$ framework since the restriction is on one instance of the object rather than on the user.

To illustrate the expressivity of our framework in this respect we extend the access control policy for the entity e_{cr} in our running example by adding the new access control rule A_2 and we extend the associated workflow with the corresponding task definition. The access control policy for e_{cr} is presented by:

$$\begin{array}{ll} \text{(rule1)} & Y := \textbf{certifier}\ ca\ \textbf{subject}\ u\ \textbf{role}\ clerk \\ & permit(X, store) \leftarrow get(Y, ca) \wedge Y.\textbf{subject} = X.\textbf{user} \end{array}$$

$$\text{(rule2)} \quad permit(X, use) \leftarrow X.\textbf{status} = \perp$$

and the workflow associated with it is given by

$$W_{cr} = (\mathcal{T}_{cr}, P_{cr})$$

where

$$\mathcal{T}_{cr} = \begin{array}{l} (\{permit(X, store) \rightarrow X.\textbf{status} := \perp; add(X), \\ permit(X, use) \rightarrow X.\textbf{status} := inuse; add(X)\} \end{array}$$

and

$$P_{cr} = \begin{array}{l} ((\nu u; \nu Y_{Doc}; recv(Y_{Doc}, u, use); permit(Y_{Doc}, use)) \\ ||(\nu u'; \nu Y'_{Doc}; recv(Y'_{Doc}, u', store); \\ permit(Y'_{Doc}, store)))! \end{array}$$

We argue that the constraint preventing multiple uses of the same object can be simply modeled by adding an attribute **status** to the object representation.

As such, in rule 2, the task *use* is permitted with respect to an object denoted by X if $X.\textbf{status} = \perp$ which essentially means that the object is not in use.

The effect of this task presented by the task definition

$$permit(X, use) \rightarrow X.\textbf{status} := inuse; add(X)$$

modifies the object, (which is now in use), by extending it with the value *inuse* associated to the attribute status.

Note that the resulting object does not satisfy the body of rule 2 anymore. Note also that after the execution of task *store* with respect to an object X, if rule 1 is satisfied for X, the object X is no more seen as being in use. This is expressed by the effect of *store* in the task definition

$$permit(X, store) \rightarrow X.\textbf{status} := \perp; add(X)$$

where the attribute **status** is again undefined.

4.9 Conclusion

We presented a logical framework to express the dynamic evolution of entities in a distributed environment with respect to their security policies. We assumed that an entity can execute some functionalities. These functionalities, that we expressed as tasks, are protected by a set of access control rules. However the constraints in access control rules can depend on information (modeled as objects) provided through a negotiation with other entities. Consequently, we described an entity in our framework as a set of access control rules protecting its functionalities but also a set of trust negotiation rules that manage the exchange of information with the other entities in the environment. We presented these rules by means of Horn clauses in a first order language and we provided their evaluation semantics in order to check whether *a task is permitted* with respect to an object in a given entity, and if *an object is available* for a given entity. Also, we associated to each entity a local state constituted of a repository and a substitution, and we argued that the state evolution is performed with respect to a workflow process associated with it. The workflow gives an order on the *actions* and *tasks* to be performed within the given entity and updates the local state accordingly. The actions in the workflow are invoked on demand and are verified with respect to the security rules of the entity before their execution. In this chapter we presented the operational semantics for the execution of the workflow subprocess.

Discussion The attribute based feature of this framework guarantees more flexibility in order to take into account non-standard requirements. In fact we do not make an explicit distinction between users and objects, but rather extract from them the relevant information that shall be used as a base for a decision (ex. *user, role, object name, object type, object content, user age etc.*) and express conditions in terms of restrictions (equality, inequality checking) on the attribute values for these *object structures.* Moreover, the dynamic feature provided by the workflow models in a natural way the modifications that occur after the execution of an action and thus allows a new evaluation of permissions based on actions already performed.

Chapter 5

A Case study: The Car Registration process

In this chapter we present the modeling of a complete case study provided by the Avantssar Project deliverable D5.1 [73]. We illustrate the construction step by step by defining the entities, and adding the security policy and the workflow structure to each of them.

5.1 Car Registration process

Mike is looking for a new car. After consultation of several car dealers, he chooses a two-year old mid-size car and pays using his credit card. Of course, before he is allowed to drive the car, he has to register his new car at the car registration office, which of course is closed during weekend. Thus, Mike takes advantage of the citizens portal where he can lodge a car registration anytime.

The different entities

We first present the main actors of the scene.

Mike is the customer of the car dealer and the one who wants to register his new car.

CentrRep is a central repository where car registration documents are stored among other things. Additionally, empty forms for several purposes are stored and available to everyone. The central repository is not purely passive: it checks digital signatures of documents and authorization of users retrieving and storing documents. The documents themselves are not inspected by the central repository.

CarRegOffice is an office where requests for car-registration have to be submitted. There might be several CarRegOffices, e.g., one per district. All

CarRegOffices use the same CentrRep to access and store documents. In this example, CarRegOffice stands for the office which is in charge for Mike.

RegOffEmpl is an employee of the car registration office. He is processing car registration requests. Employees of the car registration office are allowed to read documents from the local repository, add any number of comments to them and store documents in the local repository. Additionally, a RegOffEmpl is allowed to store fully processed car registration requests to CentrRep if his or her head of the car registration office legitimates him/her to do so. In our example the RegOffEmpl Peter is processing Mike's request. The head is responsible for leading a car registration office and is the only person which is primarily permitted to write documents to the central repository. Of course, s/he can delegate this privilege to trustworthy employees to support him/her in his/her hard work. Please note, that the head is always also a RegOffEmpl. Melinda is the head of the CarRegOffice in charge for Mike.

RegOffCA is the certificate authority of the car registration office. Its task is to generate trustworthy certificates on employees. A "certificate" in this context is an authentic assertion, such that the consumer of the assertion is sure that the author of the assertion is the "signer", in this case the RegOffCA.

The car registration process is shown in Figure 5.1.

For Mike, the process is completely transparent, he does not know about this process running in the background. He logs into the citizens portal, navigates to the car registration office and initiates the car registration process by fetching an empty form (1). He fills in data about his new car (2). After that, he adds any required and optional certificates to the form, e.g. a proof that he possesses a valid car insurance, a certificate that the car is roadworthy, etc (3). After that, he signs the document and sends it to the CarRegOffice (4).

Thus, the document is stored in the local repository of the car registration office. The role of Mike is over, he only waits for the response of the car registration office.

The car registration process:

On Monday, Peter, a user of the RegOffEmpl, fetches Mike's document, reads it (5) and adds some comments to the document (6). These steps may be repeated by any number of RegOffEmpl. After that, Peter performs the following actions in an atomic way:

7. checks Mike's signature,

8. checks the authorization of Mike, based on the certificates provided by Mike (e.g. car insurance, etc),

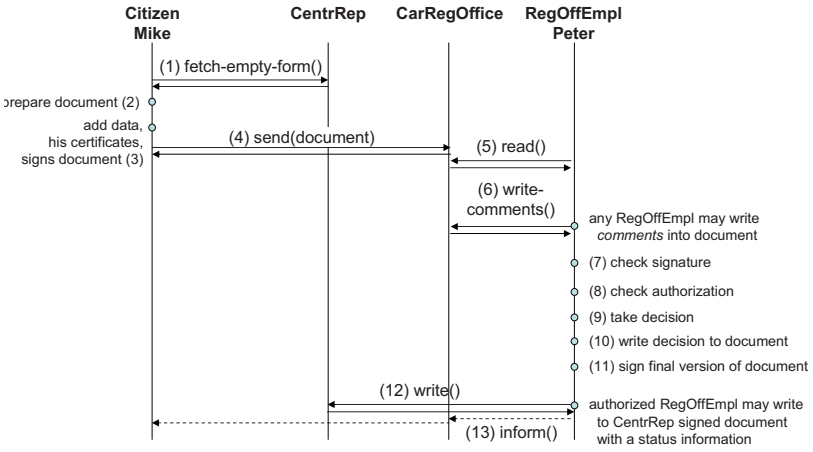


Figure 5.1: The Car Registration Process

9. takes a decision,
10. writes the decision in the same document,
11. signs the final version of the document, and
12. writes it into CentrRep. Note that RegOffEmpl needs special permissions for this step.
13. Finally, Peter informs the CarRegOffice that he has completed his task.

Thereupon the CarRegOffice stored the document in the CentralRep and informs Mike about the decision taken upon his request. With this last action, the car registration process terminates.

The system constraints

The system constraints of this scene are given below:

- Peter holds three certificates:
  - The RegOffCA confirms, that Peter is a RegOffEmpl and
  - that Melinda is RegOffHead.
  - Melinda permits Peter to write documents to CentrRep.

- The access control list (ACL) of CentrRep states, that
  - Anybody can get empty forms
  - RegOffEmpl can read documents from the repository, but can only write documents if his/her RegOffHead permits it.
  - RegOffHead can write documents.
- The local (trust) policy states that RegOffCA can say
  - who is RegOffHead and
  - who is RegOffEmpl.

With these system constraints in mind, it is easy to see that Peter actually is authorized to process Mike's request and is also authorized to write the final document back to the CentrRep.

5.2 Modeling the car registration process

As presented in Section 5.1, we distinguish five distinct entities, namely the *central repository* that we denote by e_{rep}, the *car registration office* denoted by e_{car}, the *registration office employee* denoted by e_{empl}, the *central authority* denoted by e_{ca} and the customer denoted by e_c. Let $\mathcal{E} = \{e_{rep}, e_{car}, e_{empl}, e_{ca}, e_c\}$ denote the set of entities in the model. Our aim is to express in the best possible manner the different requirements given in Section 5.1 in a set of logical rules as defined in Chapter 4. In this section we present some general guidelines to the encoding of the model.

5.2.1 The car registration form

The car registration process models the different steps that are to be performed on a given car registration form before a decision can be made. To do this, it is necessary to have a communication between the different elements of the model (i.e. the different entities). Throughout the modeling of the car registration process, we do not consider the car registration form as it is, but we suppose that relevant information were extracted form the initial document and are expressed in the form of attributes with corresponding values. In the next section we present the modifications and the evolution of the car registration form between the different entities. This evolution will be represented in the form of the addition or modification of attributes associated with the defined object. Throughout this example we use the attribute **objectId** to denote the reference id of the document, **requester** denotes the name of the customer requesting the registration, **signature** and **list** are attributes concerning the acceptability of the document, **comment** contains the comments added by the employee, **decider** contains the name of the employee that made the decision and **decision** contains the decision made on the document.

5.2.2 Requests and other elements of the model

Apart from the object representing a car registration form, we use objects to denote certificates for role membership or delegation as will be seen in Section 5.3.2. For example, an object of the form

$$\textbf{subject}\ v\ \texttt{is}-\texttt{member}\ \textbf{role}\ \mathit{head}$$

denotes that a subject u is a member of role *head*. In order to acquire permissions, we also define request objects. In this case study most requests concern actions to be performed on the registration form (*store*, *access*, *get*, *decide*, etc.). Thus we shall prefix the object representing the car registration form with the attribute **subject** to denote the user requesting the permission and the attribute **action** that will have a s value `request` to denote that the object is a request.

5.2.3 Modeling the workflow

In addition to the security rules, one needs to express the state evolution of the entities. In fact, to model the car registration process, one needs to define a sequence of message exchange that provide the link between the different entities in the model. The expression of the dynamic aspect of the entities is done in two times. First we provide the process definition of the entity's functionalities then we present the workflow associated to each entity.

5.3 The encoding of the car registration process

In this section we give the encoding of the car registration process by defining the different entities presented in Section 5.2 and their associated workflow.

5.3.1 The central repository

The entity e_{rep} consists of two access control rules that regulate the access to the entity's functionalities, namely *store* and *access*. The task *store* is authorized on a given object if the object contain a decision and the user requesting the store functionality can store the document. The task *access* is authorized for any member of the role employee. The entity e_{rep} is thus defined as follows:

Access control rules

$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{store}-\texttt{doc}\ \textbf{certifier}\ v$
$Z := \textbf{subject}\ v\ \texttt{is}-\texttt{member}\ \textbf{role}\ \mathit{head}$
$\mathit{permit}(X, \mathit{store}) \leftarrow \mathit{get}(Y, ca) \wedge \mathit{get}(Z, ca) \wedge X.\textbf{subject} = u \wedge X.\textbf{decision} \neq \perp$

$X := \textbf{subject}\ u\ \texttt{request}\ \textbf{objectId}\ y$
$Y := \textbf{subject}\ u\ \texttt{is}-\texttt{member}\ \textbf{role}\ \mathit{employee}$
$\mathit{permit}(X, \mathit{access}) \leftarrow \mathit{get}(Y, ca)$

To the entity e_{rep} is associated a workflow. The workflow consists of the set of process definitions on one hand and the actual process that defines the behavior of the entity on the other hand.

Process definitions

$$permit(X, store) \rightarrow X.\mathbf{subject} := \bot; X.\mathbf{action} := \bot; add(X)$$

$$permit(X, access) \rightarrow \nu Y; Y.\mathbf{subject} := X.\mathbf{subject}; Y.\mathbf{objectId} := X.\mathbf{objectId}; \\ send(Y, empl, \tau)$$

Workflow process

$$(\nu X; recv(X, car, store); permit(X, store) \\ ||\nu y; recv(Y, empl, access); permit(Y, access))!$$

5.3.2 The central authority

The entity e_{ca} is responsible for the delivery of objects certifying who is employee and who is head. It also manages the delegation certificate in the case where the head of the car registration office decides to delegate the right to store to one of the employees of the car registration office. The entity e_{ca} consists of one access control rule that regulates the access to the entity's functionality, namely $delegate - store$. In addition, e_{ca} also contains two trust negotiation rules to regulate role inheritance, but also to regulate the diffusion of objects to other entities. In general, the entity e_{ca} can send an object to another entity in $\mathcal{E}$ if it has it or can get it by trust negotiation. The entity e_{ca} is thus defined as follows:

Access control rules

$$X := \mathbf{subject}\ v\ \texttt{request}\ \mathbf{delegatee}\ u \\ Y := \mathbf{subject}\ v\ \texttt{is}-\texttt{member}\ \mathbf{role}\ head \\ Z := \mathbf{subject}\ u\ \texttt{is}-\texttt{member}\ \mathbf{role}\ employee \\ W := \mathbf{subject}\ v\ \texttt{is}-\texttt{head}\ \mathbf{delegatee}\ u \\ permit(X, delegate - store) \leftarrow has(Y) \wedge has(Z) \wedge has(W)$$

Trust negotiation rules

$$X := \mathbf{subject}\ v\ \texttt{is}-\texttt{member}\ \mathbf{role}\ employee \\ Y := \mathbf{subject}\ v\ \texttt{is}-\texttt{member}\ \mathbf{role}\ head \\ put(X, self) \leftarrow has(Y)$$

$$put(X, x) \leftarrow has(X) \vee get(X, self)$$

To the entity e_{ca} is associated a workflow. The workflow consists of the set of process definitions on one hand and the actual process that defines the behavior of the entity on the other hand.

Process definition

$$permit(X, delegate - store) \rightarrow \nu Y; Y.\mathbf{subject} := X.\mathbf{delegatee};$$
$$Y.\mathbf{action} := \texttt{can} - \texttt{store} - \texttt{doc}; Y.\mathbf{certifier} := X.\mathbf{subject}; add(Y)$$

The workflow process

$$(\nu X; recv(X, empl, delegate - store); permit(X, delegate - store))!$$

5.3.3 The car registration office

The entity e_{car} is responsible for the delivery of documents to the customer but is also the connection point between the request by the customer and the processing done by the employee. It is responsible for the functionality *getAvailableDoc* that fetches car registration documents to be evaluated by the Car registration employees, the functionalities *storeInLocRep* and *getInLocRep* that allow the storage and access to the documents in the local repository of the entity, *storeInCr* that sends the document to be stored in the central repository once a decision is made and $notifyCustomer$ that sends a notification message to the customer about the decision. The entity e_{car} is thus defined as follows:

$X := \mathbf{subject}\ u\ \texttt{request}\ \mathbf{objectId}\ y$
$Y := \mathbf{subject}\ u\ \texttt{is} - \texttt{member}\ \mathbf{role}\ employee$
$permit(X, getAvailableDoc) \leftarrow get(Y, ca) \wedge has(Z) \wedge Z.\mathbf{status} = \bot$
$\wedge Z.\mathbf{objectId} = X.\mathbf{objectId}$

$Y := \mathbf{subject}\ u\ \texttt{is} - \texttt{member}\ \mathbf{role}\ employee$
$permit(X, storeInLocRep) \leftarrow get(Y, ca) \wedge X.\mathbf{subject} = u \wedge X.\mathbf{status} = inUse$

$X := \mathbf{subject}\ u\ \texttt{request}\ \texttt{objectId}\ y$
$Z := \mathbf{subject}\ u\ \texttt{is} - \texttt{member}\ \mathbf{role}\ employee$
$permit(X, getInLocRep) \leftarrow has(Y) \wedge X.\mathbf{objectId} = Y.\mathbf{objectId}$
$\wedge(X.\mathbf{subject} = Y.\mathbf{requester} \vee (get(Z, ca) \wedge Y.\mathbf{status} = \bot))$

$X := \mathbf{subject}\ u\ \texttt{request}\ \mathbf{objectId}\ y\ \mathbf{decision}\ z$
$permit(X, storeInCr) \leftarrow has(Y) \wedge Y.\mathbf{decision} \neq \bot \wedge Y.\mathbf{objectId} = u \wedge Y.\mathbf{sent} = \bot$

$X := \mathbf{subject}\ car\ \texttt{request}\ \mathbf{objectId}\ y\ \mathbf{requester}\ z\ \mathbf{decision}\ w$
$permit(X, notifyCustomer) \leftarrow has(Y) \wedge Y.\mathbf{notify} = \bot \wedge Y.\mathbf{objectId} = X.\mathbf{objectId}$
$\wedge Y.\mathbf{decision} \neq \bot$

To the entity e_{car} is associated a workflow. The workflow consists of the set of process definitions on one hand and the actual process that defines the behavior of the entity on the other hand.

Process definitions

$$permit(X, getInLocRep) \rightarrow X.\mathbf{status} := inUse; send(X, empl, check)$$

$$permit(X, storeInLocRep) \rightarrow \nu Y; Y := X; Y.\mathbf{subject} := \bot; Y.\mathbf{action} := \bot; \\ Y.status := \bot; add(Y)$$

$$permit(X, storeInCr) \rightarrow X.\mathbf{sent} := ok; send(X, cr, store)$$

$$permit(X, notifyCustomer) \rightarrow \nu Y; Y.\mathbf{objectId} := X.\mathbf{objectId}; \\ Y.\mathbf{decision} := X.\mathbf{decision}; send(Y, X.requester, \tau)$$

The workflow process

$$((\nu(X; recv(X, empl, getInLocRep); permit(X, getInLocRep)) \\ ||(\nu Y; recv(Y, empl, storeInLocRep); permit(Y, storeInLocRep)) \\ ||(\nu Z; recv(Z, empl, storeinCr); permit(Z, storeInCr); permit(Z, notifyCustomer)))!$$

5.3.4 The employee

The entity e_{empl} is responsible for the processing of the car registration forms and for the decision making. An employee looks for available documents in the car registration office, gets the corresponding document, checks for the signature validity and if the document list is complete, then write comments on the document. The document can be viewed by several employees, one at a time, as long as a decision has not been taken. Once a decision is written in the document, the document is sent back to the car registration office in order to notify the customer. It is responsible for the functionalities *check* that performs an automatic check to verify the list of documents and the validity of the signature, $refuse$ and *accept* that decide if the car registration document can be processed, *comment* that allows employees to write comments on the car registration document and *decide* that adds a decision to the document and closes the process in the entity. The entity e_{empl} is thus defined as follows:

$X :=$ **subject** u `request` **objectId** x
$permit(X, check) \leftarrow \top$

$X :=$ **subject** u `request` **objectId** x
$permit(X, refuse) \leftarrow has(Y) \wedge Y.\mathbf{objectId} = x \wedge Y.\mathbf{decision} = \bot$
$\wedge(Y.\mathbf{signature} = notvalid \vee Y.\mathbf{list} = incomplete)$

$X :=$ **subject** u `request` **objectId** x
$Y :=$ **objectId** x **signature** z **list** w
$permit(X, accept) \leftarrow has(Y) \wedge (Y.\mathbf{signature} = valid \vee Y.\mathbf{list} = complete)$
$\wedge Y.\mathbf{decision} = \bot$

$X :=$ **subject** u `request` **objectId** x **comment** z
$Y :=$ **objectId** x
$permit(X, comment) \leftarrow has(Y) \wedge Y.\mathbf{decision} = \bot$

$X :=$ **subject** u `request` **objectId** x **decision** z
$Y :=$ **objectId** x
$permit(X, decide) \leftarrow has(Y) \wedge Y.\mathbf{decision} = \bot \wedge (z = accept \vee z = reject)$

To the entity e_{empl} is associated a workflow. The workflow consists of the set of process definitions on one hand and the actual process that defines the behavior of the entity on the other hand.

Process definitions

$permit(X, check) \rightarrow X.\mathbf{signature} := v_1; X.\mathbf{list} := v_2; add(X);$
$(permit(X, refuse) + permit(X, accept))$

$permit(X, refuse) \rightarrow X.\mathbf{decision} := reject; X.\mathbf{decider} := X.\mathbf{subject}; add(X)$

$permit(X, accept) \rightarrow skip$

$permit(X, comment) \rightarrow X.\mathbf{comment} := z; add(X);$
$send(X; car, storeInLocRep)$

$permit(X, decide) \rightarrow X.\mathbf{decision} := z; X.\mathbf{decider} := X.\mathbf{subject};$
$add(X); send(X; car, storeInCr)$

The workflow

$$(\nu U; send(U, ca, delegate - store)$$
$$||\nu X; send(X, cr, access)$$
$$||\nu Y; send(Y, car, getInLocRep); \nu Z; recv(Z, empl, check); permit(Z, check)$$
$$||\nu V(permit(V, comment) + permit(V, decide)))!$$

5.4 Conclusion

we have presented in this chapter a modeling for the car registration process. This process involves different actors that we presented as entities. Each entity consists of a set of access control rules and possibly trust negotiation rules that regulate the access to the entity's functionalities by authorized users. In order to model the internal behavior of the entities we defined for each entity the direct effects of the entity's functionalities. Then we defined a workflow that put an order on the execution of the functionalities and ensures the communication between the different entities. In fact as we can see in this model, the communications between the different entities are done by sending and receiving requests to access the corresponding functionalities.

Part II

Expressing delegation and separation constraints using access control and workflow

Introduction

In the first part of this thesis we presented an attribute based logical framework to express access control policies with trust negotiation and workflow execution. Our main concern was to allow the expression of policies beyond the scope of the role based access control and to provide a formal model for sharing information between entities in a distributed environment. In this part we present security *features and specifications* to reason about the safety in the framework.

In the access control framework defined in Chapter 4 we distinguish between a static policy and a dynamic policy. The static policy evaluates the current state of the different entities with respect to the trust negotiation rules, whereas the dynamic policy is expressed by the interaction between the access control rules and the workflow and modifies the security state accordingly. The explicit definition of workflow processes and transition rules allows the update of an entity's security state in order to reevaluate new decisions with respect to the evolution of the state. The access control rules act as guards on the execution of these processes in the workflow. Moreover, the trust negotiation rules provide a formal model for the communication between the entities in the environment in order to reach mutual trust. Finally, the advantage of working with attributes allows the expression of security properties via the definition of new objects without modifying the syntax of our framework.

In this second part we shall be concerned with the expression and specification of security properties both on the level of access control security and trust negotiation communication.

On the one hand, an access control framework must be able to express various access control properties implemented in the structure of an organization. For example, it should offer the possibility for the delegation of tasks, while preserving separation or binding of duty constraints and possibly guaranteeing the least privilege property. As discussed in [14], it is often the case that different variants of the same security property, such as delegation for example, are needed in one organization. We present an encoding of delegation and separation of duty properties in a distributed environment. To do that we make use of the interplay between the access control policy of a given entity and the trust negotiation policies of the other entities in the model. This offers an interesting way to express *centralized* access control properties in a *decentralized* manner due to the possibility of negotiating certificates among the different entities. However in the presence of access control features such as delegation or role inheritance for example, additional authorizations that were not initially taken into account in the policy design may take place. Due to the dynamicity of the framework, it is not always easy to predict such additional authorizations, since access rights are modified in each security state. To this end we present in this part some decision problems to check for violation of security constraints.

On the other hand, the distributed nature of the environment necessitates communication between entities in order to exchange objects during a trust negotiation session. In Chapter 4 we assumed that these communications take place in a secured environment, i.e. all entities abide by their trust negotiation

rules. However, this is not usually the case in real world applications. Thus it is necessary to consider the security offered by the communication channels between the different entities in order to assess the security of the trust negotiation infrastructure. To this end we present in this part an extension of our framework that is concerned in securing the trust negotiation sessions and give security specifications as decision problems to check for the authenticity and confidentiality properties during a trust negotiation session.

This part is structured as follows. In Chapter 6, we give a specification of our understanding of delegation and revocation in a distributed environment. Then in Chapter 7 we give an overview of security constraints, namely separation and binding of duty constraint and express the modeling and verification of such constraints within our attribute based framework. In Chapter 8 we give a representation of the RBAC model in our framework along with the role membership, role activation and role inheritance properties. In Chapter 9 we explore a different security problem. We assume the presence of a malicious entity in the environment and specify an intruder model in a trust negotiation session in order to test some authentication and confidentiality properties with the presence of this malicious entity. To do so we extend the syntax and evaluation semantics of our security rules to take into account the notion of channels and encryption keys.

Chapter 6

On specifying delegation and revocation in a dynamic framework

Most organization have business rules and regulations that govern their security policy. Among these are separation of duty, least privilege and delegation. Delegation is the act of assigning rights and responsibilities to another person (e.g. from a manager to a subordinate) to carry out specific activities. We can distinguish between delegating a right, delegating a set of rights or delegating a role. Nowadays, since more and more organizations work in a decentralized environment, it is not always evident to assign the best task to the best person. That is, the tasks may be allocated to a given service that in turn should delegate subtasks to qualified personnel. For example the general manager of a company may delegate a task to the regional manager who in turn delegates it in form of subtasks to various employees.

In our representation of access control properties in a distributed environment we define *access control objects* to be tasks, roles or activation certificates within a given entity. We suppose that each entity is responsible for a collection of access control objects. For example, the entity representing the accounting department will be responsible for the roles **accountant** and **account − supervisor** and tasks `compute` and `check`. Thus this entity will be responsible for all operations (delegation, role inheritance, role activation, etc..) concerning these objects. The decentralization of the information is then performed via trust negotiation. That is, in order to receive information about a given access control object, a trust negotiation session is to be launched with the entity holding this object.

Consequently, it is not necessary to have a centralized authority to handle delegation of tasks or assigning of roles. Rather, a decentralized expression of the requirements to use an access control object can be identified in the entity responsible for it, and the trust negotiation policy then regulates the diffusion

of such authorizations to other entities in the environment.

This chapter is divided into two independent parts. The first part provides an overview of delegation. In Section 6.1 we present different aspects of delegation and revocation in the literature and we give a short survey on the expression of delegation and revocation in access control frameworks. The second part of this chapter is devoted to our modeling of the concept of delegation and revocation. In Section 6.2 we present the delegation context by providing the elements needed to define delegation and revocation rules. In Section 6.3 we give a generic form of delegation and revocation rules as access control rules in our attribute based framework, then we define in Section 6.4 the process definition for the different delegation and revocation tasks. In Section 6.5 we complement our model with the needed rules in order to express the acquisition of the delegated right and the propagation of delegation. Finally, Section 6.6 provides an illustrative example for the use of delegation in a distributed system.

6.1 Related Works

The delegation is one of the most important business aspects in an organization as it allows more flexibility and efficiency in the division of tasks between authorized users. One of the first manifestations of delegation is the *grant* action in the take-grant model [46], where a user u having the right for object o can grant the right to read the object o to a user v. As a result, v acquires the right to use o by delegation from u. The development of the RBAC framework enhanced the expression of access control policies as discussed in Chapter 2 and thus the expression of delegation was then directed towards delegating a role, part of a role or even a role hierarchy. This was subject to many interesting research ([9, 8, 81, 82, 68, 29, 31, 75, 76]).

The default user-permission assignment may not always be feasible for the given user. For example, a manager that is assigned to more than one project may choose to delegate them to an assistant manager in order not to be late for the delivery date. Delegation can also be useful in case of sickness, leave or lack of sufficient qualifications to perform the permitted task. Delegation thus provides a possibility to define new user-permission assignment that are decided by the users and not imposed by the access control administrator. In order to enhance the efficiency of user-permission assignments, different flavors of delegation are studied in the literature. For example, an administrative delegation where the delegator does not necessarily have the right or the qualification to perform the delegated task, or a user-delegation where the delegator has the qualifications and thus the right to execute the task but may have other constraints making him delegate it to someone else etc.

6.1.1 Role delegation

One of the first works that dealt systematically with delegation between users are by Barka and Sandhu [9, 8]. They define delegation in the RBAC framework

as a role delegation, i.e. a user belonging to some role can delegate his role to another user acting in another role. The authors identified an exhaustive list of the characteristics that are related to delegation. They also tried to characterize the use of revocation and state some of the difficulties that appear when explicitly modeling revocation. In this subsection we present the different aspects of delegation defined in [9], and then state the problems and some solutions for the implementation of revocation in a delegation model. The short presentation of delegation and revocation features helps us in characterizing the development of research with respect to these notions and to eventually propose a flexible model that can solve some of these difficulties.

Delegation

When talking about delegation, different characteristics are to be taken into consideration, Barka and Sandhu [9] list these different aspects as follows:

Permanence: A *permanent* delegation allows the delegatee to replace the delegating user in the delegating role (the delegator will not be able to regain his role unless an administrator assigns him again to that role.) while a *temporary* delegation keeps the right for the delegator to revoke the delegating role at any time.

Monotonicity: A delegation can be *monotonic* (often referred to as a *grant* delegation.) in which case the delegator maintains power on his role even after the delegation or *non-monotonic* (often referred to as a *transfer* delegation.) when the delegator loses control over the delegated role for the duration of the delegation. The delegator can however revoke the delegation at anytime.

Totality: This is a notion proper to role delegation. A *total* delegation delegates all the permissions assigned to a role whereas in *partial* delegation only a subset of the role is delegated (this amounts to delegating a part of a role hierarchy.).

Levels of delegation: A *single-step* delegation cannot be further delegated whereas a *multi-step* delegation can be delegated as many times as indicated.

Multiple delegation: refers to the number of users to whom a delegator can delegate the same role.

Agreements: We talk about *bilateral agreement* when both the delegator and the delegatee need to accept the delegation for it to take place, and *unilateral* agreement where the delegator alone can decide on delegating a role and the delegatee has no choice but to accept it.

A permanent delegation is often viewed as a form of permission assignment performed by a user rather that an administrator. We are more interested in the expression of temporary delegation as this type of delegation can be revoked

and thus offers a more general notion of delegation. We express both monotonic and non-monotonic delegation and distinguish between single and multi-step delegation. We will not encode all the possible types of delegation, but instead we hope to convince the reader that any of the above considerations can be taken into account. For the sake of simplicity, we assume that a delegator can only delegate a given right on the same object once. Multiple delegation can be treated similarly. Finally we do not present the totality feature explicitly as we work on an attribute based framework as opposed to a role-based framework. Thus this property can be encoded by the modeler in the specification of the model if needed.

Revocation

In the case of a temporary delegation, the delegation usually expires after a period of time, specified by the delegator. However this assumption is not always sufficient. In many cases, the delegator may wish to terminate the delegation prior to the end of validity. Barka and Sandhu enumerate two main characteristics concerning revocation, namely:

Grant-dependency: A revocation can be *grant-dependant* if only the direct delegator can revoke membership or *grant-independent* in which case any original member of the delegated role can revoke membership of the delegated role.

Cascading revocation: A delegatee is revoked from a delegated role if the delegatee is revoked from his *supporting role* i.e the role to which he belonged prior to the delegation. If a delegator is revoked from the sponsoring role (i.e the role responsible for bringing in delegated members) then all memberships delegated by that delegator are also revoked.

Depending on the context, several assumptions can be made regarding the enforcement of revocation.

One of the problems of revocation is the case where the delegator loses his sponsoring role, that is the role in which the delegation was possible. In that case two choices are possible: either the delegatee(s) delegated by that user through the sponsoring role keep the delegated role or the delegatee loses the delegation through a cascading revocation.

Another problem concerns the person performing the revocation. In fact if we assume that revocation is grant-dependent then it may sometimes be complicate to revoke a delegatee from its delegated role especially if the concerned delegatee is at the end of a delegation chain. Also if one allows multiple delegations for the same delegatee, then if one of the delegators revokes the delegation, one has to define what should happen to the delegated rights of the delegatee.

In our representation of delegation, the permission to revoke a delegation is stated as an access control rule that acts as a guard for the revocation process. As such, it is possible to define both grant-dependent and grant-independent

revocation without modifying the actual definition of the delegation and revocation processes. As for the cascading revocation, it is sufficient for one delegation to be revoked to automatically revoke the latter delegations in a delegation chain in the case of a multi-step delegation. These features are presented in Sections 6.4 and 6.5.

6.1.2 Delegation Models

In RBDM [8], a user can delegate his/her role to another user. The delegation is a total one step delegation and concerns flat roles in RBAC. Zhang et al [81] defined RDM2000 a rule based framework to model delegation and revocation in RBAC. Unlike RBDM, RDM2000 considers delegation in the presence of a role hierarchy and deal with multi-step delegation as well as different types of revocation. They use a rule-based declarative language to define and enforce constraints on the delegation and revocation taking into account role hierarchy and delegation paths. However the delegation unit in RBDM and RDM2000 is the role, making delegation a user to user role assignment. In the case of partial delegation, the partiality is dealt w.r.t. role hierarchy, i.e. partial delegation refers to delegating a lower role in the role hierarchy for the delegator. In the same logic, total delegation means delegating the strongest role in the hierarchy.

PBDM [82] is a flexible delegation model that supports multi-step delegation and revocation in role and permission level. In order to distinguish between role delegation and permission delegation, PBDM defines auxiliary roles called delegatable and delegation roles, that allow flexibility in assigning delegated member to *copies* of the original roles. In PBDM0, a user can delegate all or part of his/her permissions to delegatees. PBDM0 allows the delegation of a role without delegating the associated hierarchy through the creation of copies of the original roles, called *delegation roles* and associating delegatees to the latter rather than to the original roles. In PBDM1 and PBDM2, the permission flow is managed by a security administrator with delegatable role. In PBDM2, role-to-role delegation (i.e the delegation of a part or a whole of a role to other roles without passing through a specific user) is done by adding more artificial roles, namely *fixed delegatable roles* (that represent the delegator) and *temporal delegator roles* (that consist of a set of permissions that can be delegated). Thus when a role is delegated the associated permissions are assigned to the corresponding delegation role. The difficulty of the model lies in the fact that it relies on the role-based structure, that is the main unit for access control is the role, and thus to add and remove permissions one needs to define auxiliary roles given that in RBAC one cannot distinguish between the delegation of a permission (or a set of permissions) and the delegation of a role.

In these models keeping the user-role and permission role relations as defined in RBAC complicates the expression of policies rather than simplifying it. This structure makes expressing partial delegation, especially permission delegation rather cumbersome as is the case in [82].

6.1.3 Delegation in access control frameworks

Delegation and revocation were also expressed in more general access control models with dynamic properties. In this section we present different aspects of delegation presented in some access control logical frameworks.

In Cassandra [13] delegation is expressed as a form of chained appointment on role activations.

For example a subject *mgr* can delegate the role *employee* to a subject *emp* if *mgr* can activate a special role *appointEmployee*. The delegation is expressed via the predicate *canActivate*() that gives to the delegatee the permission to activate the role but does not force him to actually use the delegated role in a bilateral agreement.

$$\texttt{canActivate}(mgr, AppointEmployee(emp)) \leftarrow \texttt{hasActivated}(mgr, Manager()) \quad (6.1)$$

$$\texttt{canActivate}(emp, Employee(appointer)) \leftarrow \texttt{hasActivated}(appointer, AppointEmployee(emp)) \quad (6.2)$$

The role *appointEmployee* represents the activation of the *right to delegate*. In this example a delegation of the role is done in two steps, *Employee* can be activated if the delegator is active in the role manager (rule 6.1), and willingly activated the right to delegate (rule 6.2). Revocation is expressed by a deactivation of roles (that is the removal of credentials associated with role membership). As such constraints can be imposed on who can deactivate the role *appointEmployee* i.e the revocation can be either grant-dependent or grant-independent. A cascading chain of revocation is also expressed in the same manner. The employee role is automatically revoked for *emp* when *appointEmployee*(*emp*) is deactivated.

Although the Cassandra framework is expressive enough to model many the delegation characteristics, the fact that it is based on roles necessitates the creation of new roles in order to define partial delegation. Also Cassandra does not support non-monotonic (transfer) delegation.

In SecPAL [11] there is a possibility to define attribute based delegation. The assertions about some given attributes (needed to access an entity) are delegated to another entity provided this latter is trusted (assertions can be provided concerning its identity for example).

For example:

$$Shop\ says\ x\ is\ entitled\ to\ discount\ if\ x\ is\ student$$

$$Shop\ says\ univ\ can\ say\ x\ is\ student\ if\ univ\ is\ university$$

$$Shop\ says\ BoardOfEducation\ can\ say\ univ\ is\ university$$

SecPal also allows width bounded delegation, that is a subject may delegate a task (or a fact) to someone satisfying certain criteria. An example in SecPAL is as follows

$$Alice\ says\ x\ can\ say\ y\ is\ friend\ if\ x\ is\ delegator$$

$$Alice\ says\ Bob\ is\ delegator$$

$$Alice\ says\ x\ can\ say\ y\ is\ a\ delegator\ if\ x\ is\ a\ delegator,\ y\ possesses\ Email\ email$$

In fact in this example, is presented an example of a multi-step delegation, however, a constraint is imposed in the delegation propagation (only subjects with an email address can become delegators). SecPAL is an assertion based language, and thus is more expressive in terms of delegation specification. However, the encoding of revocation is delicate. In fact in SecPAL only the issuer of the delegation assertion can revoke it, and the revocation is done by the deletion of the assertion that can be retrieved through a unique identifier added to the assertion. Also SecPAL cannot express non-monotonic delegation.

The study of non-monotonic (transfer) delegation is not very present in research on delegation. One of the first works that explored *transfer* delegation in addition to the grant delegation was [29]. This work distinguishes between role-based and permission-based delegations. In the former, role delegation takes into account hierarchy associated with the delegated role. In the latter, only specified permissions are delegated. An access control policy specifies whether a delegator can delegate a role but also whether a delegatee can be authorized to the delegated role.

This authorization mechanism is defined through two relations $\texttt{can}-\texttt{delegate}$ and $\texttt{can}-\texttt{receive}$ that specify the set of roles that can be delegated by the delegator and the set of roles that can be received by a delegatee. The distinction between constraints on the delegator and the delegatee allows a flexible representation of conditions for the different actors of a delegation. For example, when a role is revoked, it is deleted form the $\texttt{can}-\texttt{delegate}$ relation but the $\texttt{can}-\texttt{receive}$ relation remains intact.

The enforcement or update of delegation is done through a *delegation history* relation DH. This latter records successful delegations and temporary delegation relations. Temporary delegation relations record temporary user-role assignments `tempUA` and temporary user-permission assignments `tempPA` that arise from delegation operations. For example, after a successful role transfer delegation of role r from user u to user v the relation (u, r) is deleted from `tempUA` and (v, r) is added to `tempUA`. In the case of revocation, (v, r) is deleted form`tempUA` and (u, r) is added to `tempUA`. The role-based reference monitor uses these relations to make access control decisions based on the different delegation and revocation actions.

This delegation model is extended in [31] to *task delegation* in workflow systems. A workflow defines a partial order on a collection of activities called tasks. The execution of a task by a user is authorized with respect to some user-task relations that can either be defined directly or acquired through role-based

structure. A task is executable in a workflow structure if the user assigned to this task is authorized to perform the task. The workflow management system maintains a list of the concrete tasks and generates a *task list* that assigns tasks to users during a workflow execution. As in [29], delegation is controlled by DH and a temporary task-user authorization ($TempTA$).

We chose a delegation model close to this model first because of the clear distinction between the specification of the delegator right to delegate and the delegatee right to receive the delegation, and second because of the delegation history that facilitates the expression of the delegation and revocation effects both in the case of a monotonic and non-monotonic delegation.

In [75] the authors propose an expression of *user to user* delegation. In their model a *grantor* wishes to delegate the *object of delegation* to a *delegatee*. In order to do that, the grantor must have the *right to delegate* the object. This can be a *direct right* inherited by the RBAC structure (through role activation or role inheritance for example.), or a right granted by delegation. They also offer the possibility to delegate delegation rights in a flexible manner. The possibility of defining generic constraints ensures the security of the organization against violation of separation of duty constraints for example. They allow multiple delegation, i.e. the same delegatee may receive the same delegated right from more than one delegator restrictions on the delegation depth, this creates cycles (e.g. one subject delegates a right to another subject who in turn delegate the same right to the former).

To model revocation, they define, a chain delegation recognizable via a path between the delegator and the delegatee. A delegator can revoke a right from a delegatee along a given path if there is no other path that leads to the same right (through another delegation). Generic constraints are defined to preserve the security properties of the system, and thus the only restriction on the act of delegating is to not violate these generic constraints.

In [76] the authors argue that delegation is a dynamic aspect that alters the local security state. The security state is made up mainly of the user-permission relations that users can acquire either by delegation or by role activation and inheritance. The act of delegating a right changes the permission relation of the delegatee (adds extra rights to the delegatee permission relation.). Conditions on the delegator and the delegatee are specified by predicates. u_1 can grant role r to u_2 if u_1 is a member of r, u_1 satisfies the conditions to grant the role r and u_2 satisfies the conditions to receive role r.

$$grant(u_1, u_2, r) \leftrightarrow (u_1, r) \in UR \wedge can-grant(c_1, r) \in RL \wedge (u_1\ satisfies\ c_1) \\ \wedge can-receive(c_2, r) \in RL \wedge (u_2\ satisfies\ c_2)$$

A workflow is represented as a tuple $< S, \prec, C >$ where S is a set of tasks, $\prec$ defines a partial order among tasks and C is a set of constraints. In an access control state, the permissions to perform tasks in the workflows are assigned to roles. The state evolution in the workflow is based on the set of permissions allocated to each user either by role membership, inheritance, or by delegation.

They present the distinction between the grant and transfer delegations through the definition of the transition operations that modify the local state of the workflow. However they assume that users can only delegate roles to which they belong. Further, delegation is a single-step delegation and does not support multi-step delegation. Finally the delegated right can only be revoked by the user who performed the delegation.

Discussion The models in [29, 31, 76] offer similar solutions as our representation of delegation, as far as it is possible to express both grant and transfer delegation by changing the state of the system. However in our proposed model the state changes are expressed via workflow processes that add and remove data from the state of the system. Unlike [76], our model allows multi-step delegation. Also, we have the possibility to model both grant-dependent and grant-independent revocation. The extension to workflow systems in [31] is also very interesting in the fact that delegation history is a guard against violation of security constraints. In our framework, as in the case of [13, 11], the delegation model is an extension of an access control framework, and thus can express delegation in the access control context, and attributes allow the expression of delegation beyond the scope of RBAC as is the case in [11].

6.2 Defining the delegation context

As we mentioned in Section 6.1, research was mainly concentrated on delegation within the RBAC framework. This gave more flexibility and enhanced the usability of the RBAC model but constrained the expressivity of delegation. In fact in RBAC constraints are only about roles or role specifications, and thus cannot take into account a user specification. For example a requirement stating that a user in the role *manager* can delegate this role to an assistant with five years experience maybe difficult to express in the RBAC framework. As such our attribute based framework can express the RBAC structure and adds more flexibility in expressing additional constraints that may not be related to the specification of roles. In the rest of this chapter we give our representation of delegation and revocation.

In general, delegation may be either a *transfer* where the delegator loses the right over the delegated objects after the delegation, or a *grant* where the delegator keeps the delegated right. Moreover the delegator may choose to only delegate the authorization over the object or to allow the propagation of the delegation by also giving the right to delegate such an object.

We start by defining the objects that we shall use to express the delegation properties and effects.

6.2.1 The right to delegate

We will not worry about how a subject acquires the right to delegate or to receive the delegation as we assume that such rights are guaranteed by the

access control policy of the responsible entity. We shall however focus on the various steps involved in the delegation and revocation of such an object. Thus, we suppose that, in order to be authorized to perform the act of delegation on an object, the delegator must provide a proof certifying that he has the right to delegate the object. Further, in order to be authorized to receive the delegation for an object, the delegatee must provide a proof certifying that he can receive the delegation for the object. To this end we present special objects intended to certify these properties. Note that these objects will be used later to define the general rules for delegation and revocation.

Certifying the right to delegate

The certification that a user u has the right to delegate a task depends on two criteria, the type of the delegation (i.e. grant or transfer) and the depth of the delegation (i.e. single-step or multi-step), and is given by a special object that we call the *delegator object*. The type of the delegation is expressed by values $grant$ and $transfer$ associated with the attribute **type** in the delegator object to denote that the delegation is a grant or transfer delegation. The depth of delegation is expressed by the action $\texttt{can}-\texttt{1}-\texttt{delegate}$ for a single-step delegation and $\texttt{can}-\texttt{delegate}$ for a multi step delegation.

A delegator object has the form:

$$\textbf{subject}\ u\ \texttt{action}\ \textbf{object}\ \tau\ \textbf{type}\ type$$

where u is the identity of the delegator, τ is the object to be delegated, $\texttt{action} \in \{\texttt{can}-\texttt{1}-\texttt{delegate}, \texttt{can}-\texttt{delegate}\}$, and $type \in \{grant, transfer\}$.

Note that one can add additional attributes in order to further constrain delegation.

Example 6.2.1. The object

$$\textbf{subject}\ mary\ \texttt{can}-\texttt{delegate}\ \textbf{object}\ report\ \textbf{right}\ read\ \textbf{type}\ grant$$

denotes that *mary* has the right to delegate the right *read* over the object *report* in a grant multi-step delegation.

Delegator objects can be either assigned to the user at the initial state or acquired by the security policy of the entity.

Certifying the right to receive the delegation

The certification that a user u has the right to receive the delegation for an object is given by a special object that we call the *delegatee object*. The right to receive the delegation is independent of the type of depth of the delegation and is expressed by the action $\texttt{can}-\texttt{rcv}-\texttt{delegation}$ The delegatee object is of the form:

$$\textbf{subject}\ u\ \texttt{can}-\texttt{rcv}-\texttt{delegation}\ \textbf{object}\ \tau$$

Note that one can add additional attributes in order to specify the criteria over the delegated object

Example 6.2.2. The object

$$\textbf{subject } \textit{bob } \texttt{can}-\texttt{rcv}-\texttt{delegation } \textbf{object } \textit{report } \textbf{right } \textit{read}$$

denotes that *bob* has the right to receive the delegation for the right *read* over the object *report*.

Delegatee objects can be either assigned to the user at the initial state or acquired by the security policy of the entity.

6.2.2 After the delegation

When talking about delegation, one also needs to take into consideration the act of revoking the delegation. In most cases in the literature, it is assumed that the revocation is made solely by the delegator. This is not always true. In fact, one can consider a situation where a higher authority monitors the delegations and has the power to revoke a delegation in the place of the delegator if a conflict or abuse of power is reached. On the other hand, before performing a revocation, one needs to have a proof that the delegation actually took place. In order to remedy to these situations, we define two special objects.

Certifying the right to revoke a delegation

We assume that a revocation can be performed by a subject that has the right to revoke the delegation and express this by the action $\texttt{can}-\texttt{revoke}$. The certification for the right to revoke is given by a *revocation object* of the form:

$$\textbf{subject } u \ \texttt{can}-\texttt{revoke } \textbf{object } \tau$$

Note that this right can be granted by an administrator or can be acquired through the access control policy.

The result of performing a delegation

When the act of delegation is executed, it has to be recorded in order to keep track of the delegation and eventually revocations. To this end we define the *delegation history object* records the executed actions as an object of the form:

$$\textbf{subject } u \ \texttt{delegated } \textbf{delegatee } v \ \textbf{object } \tau \ \textbf{type } \textit{type}$$

where u denotes the identity of the delegator, `delegated` is the action referring to the recorded delegation v denotes the identity of the delegatee, τ is the delegated object and $type \in \{grant, transfer\}$ refers to the type of the delegation.

Note that in the history record we do not need to specify the delegation depth as this will be taken into account in the definition of delegation rules in Section 6.3.

Remark: In the above presented objects, and in the rest of this chapter we treat delegation in general. In fact we talk of a delegation *object* as a value for the attribute **object**. However, this value can express a permission, a task, a role or any other attribute of our choice. In order to add to the clarity of the delegated object, we may add an optional attribute **nature** that would have for value the nature of the delegation (i.e role, task, etc...) when this is needed. For example

subject u `delegated` **delegatee** v **object** τ **nature** *role* **type** *type*

records a delegation for a role τ.

6.3 Expressing the rights to delegate and revoke

In Section 6.2 we presented the objects that will be used in order to specify delegation. Recall from Chapter 4 that access control rules are defined by the predicate *permit* that takes as argument an object representing the request, and a task. In this section we use the access control rules to specify the constraints concerning delegation and revocation. The tasks denote the different types of delegations and their corresponding revocations. We define four different types of delegation, namely single-step grant delegation, single-step transfer delegation, multi-step grant delegation and multi-step transfer delegation. Their corresponding tasks will be denoted $grant$, $transfer$, $pgrant$ and $ptransfer$ respectively. Further, we define four corresponding types of revocation which tasks will be denoted $rvk - grant$, $rvk - trans$,$rvk - pgrant$, $rvk - ptrans$ respectively.

Delegation task

We assume that a delegation of an object from a delegator to a delegatee is authorized if the delegator holds the right to delegate and the delegatee the right to receive the delegation. Further we assume that a delegation can always be revoked if the revocation is executable. We assume that the delegator can only delegate the same object once. As such we suppose that once the delegation is performed, the delegator loses the right to delegate. One can choose to make the assumption of multiple delegation by keeping the object certifying the right to delegate with the delegator. The single step grant delegation is expressed by the following access control rule:

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$Y :=$ **subject** u `can` $-$ `1` $-$ `delegate` **object** τ **type** $grant$
$Z :=$ **subject** v `can` $-$ `rcv` $-$ `delegation` **object** τ
$W :=$ **subject** u `delegated` **object** τ **type** $grant$
$permit(X, grant) \leftarrow get(Y, self) \wedge get(Z, self) \wedge not(has(W))$

Discussion: Recall from Section 6.2 that W is an object that records the execution of a delegation of a specific type on a specific object. Thus, the presence of such an object in the repository of the entity denotes that a delegation is performed and thus the delegator loses the right to perform a new delegation. We shall see in Section 6.4 that this object is removed when a revocation is performed, in which case the delegator gains back the right to delegate.

We define the access control rules for authorizing a delegation corresponding to the three remaining types of delegation in a similar manner as follows:

Single-step transfer delegation

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$Y :=$ **subject** u `can` $-$ `1` $-$ `delegate` **object** τ **type** $transfer$
$Z :=$ **subject** v `can` $-$ `rcv` $-$ `delegation` **object** τ
$W :=$ **subject** u `delegated` **object** τ **type** $transfer$
$permit(X, transfer) \leftarrow get(Y, self) \wedge get(Z, self) \wedge not(has(W))$

Multi-step grant delegation

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$Y :=$ **subject** u `can` $-$ `delegate` **object** τ **type** $grant$
$Z :=$ **subject** v `can` $-$ `rcv` $-$ `delegation` **object** τ
$W :=$ **subject** u `delegated` **object** τ **type** $grant$
$permit(X, pgrant) \leftarrow get(Y, self) \wedge get(Z, self) \wedge not(has(W))$

Multi-step transfer delegation

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$Y :=$ **subject** u `can` $-$ `delegate` **object** τ **type** $transfer$
$Z :=$ **subject** v `can` $-$ `rcv` $-$ `delegation` **object** τ
$W :=$ **subject** u `delegated` **object** τ **type** $transfer$
$permit(X, ptransfer) \leftarrow get(Y, self) \wedge get(Z, self) \wedge not(has(W))$

Revocation task

We assume that a delegation by a delegator u to a delegatee v of an object τ can be revoked by a subject w, if the w has the right to revoke τ and the delegation was recorded. The revocation is regulated by the following access control rules. As in the case of delegation, we define four access control rules for revocation, and for the sake of simplicity assume that they have the same body. Note however that the policy modeler can add more restriction in the body of the rule if needed. The access control rules for revocation are defined as follows:

$X :=$ **subject** w `rvk` $-$ `del` **delegator** u **delegatee** v **object** τ
$Y :=$ **subject** u `delegated` **delegatee** v **object** τ
$Z :=$ **subject** w `can` $-$ `revoke` **object** τ
$permit(X, task) \leftarrow get(Y, self) \wedge (get(z, self) \vee w = u)$

where

$$task \in \{rvk - grant, rvk - trans, rvk - pgrant, rvk - ptrans\}$$

As we shall see in Section 6.4 a revocation is always performed on a request, that is we do not automatically allow cascading revocation.

6.4 The effect of delegation and revocation

In Section 6.3 we defined the access control rules for both delegation and revocation. In this section we present the effects for the execution of a delegation. The effect of the delegation and revocation is implemented by defining corresponding processes. Note that the process definition for delegation is different depending on whether the delegator chooses single-step or multi-step delegation. In this section we present these different aspects of delegation as well as revocation. In all cases when a delegation is executed a *delegation history object* that records this delegation is added to the repository of the entity (see Section 6.2). In the case of a multi-step delegation, the delegatee also receives the right to delegate expressed by the addition of an object to the repository.

In this section we are only interested in the propagation feature of delegation. This is why we consider the same effects for the tasks $grant$ and $transfer$ on one hand, and $pgrant$ and $ptransfer$ on the other hand.

Process delegation for single-step grant delegation

The process definition of delegation includes both the effect of delegation and revocation. We express the process definition for the single-step grant delegation as follows:

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$W :=$ **subject** w `rvk` $-$ `del` **delegator** u **delegatee** v **object** τ
$permit(X, grant) \rightarrow$
$(\nu Y; Y.$**subject** $:= u; Y.$**action** $:=$ `delegated`$; Y.$**delegatee** $:= v;$
$Y.$**object** $:= \tau; Y.$**type** $:= grant; add(Y);$
$\nu W; rcv(W, x, rvk - grant); permit(W, rvk - grant); rmv(Y))$
$||\nu X'; permit(X', grant)$

Discussion: This process creates a new variable Y, records in Y the execution of the delegation, adds Y to the repository then waits for a request message in W to revoke the delegation. The subprocess $permit(W, rvk - grant); rmv(Y)$ is added at the end of the delegation process definition to make sure that the revocation action is performed on the corresponding delegation. As such, the revocation process denoted by $permit(W, rvk - grant)$ acts as a simple guard on the execution of the revocation. The process delegation for revocation is thus defined by:

$$permit(W, rvk - grant) \rightarrow skip$$

In fact we made the choice of adding the effects of revocation explicitly in the process definition of the delegation in order to ensure that the changes (here removing the object Y) are made for the corresponding delegation instance. The revocation process thus becomes a mere guard that in case the access control requirements are satisfies, unblocks the execution of the actions in question.

Finally, the presence of a delegation process in parallel at the end of the process definition ensures the possibility of executing additional delegations (by other users, on other objects) without blocking the process execution.

Process definition of the other delegation processes

The process definitions for $transfer$, $pgrant$ and $ptransfer$ are defined in the same manner as follows:

The process definition for $transfer$ (single-step delegation)

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$W :=$ **subject** w `rvk` $-$ `del` **delegator** u **delegatee** v **object** τ
$permit(X, transfer) \rightarrow$
$(\nu Y; Y.\textbf{subject} := u; Y.\textbf{action} :=$ `delegated`$; Y.\textbf{delegatee} := v;$
$Y.\textbf{object} := \tau; Y.\textbf{type} := transfer; add(Y);$
$\nu W; rcv(W, x, rvk - transfer); permit(W, rvk - transfer); rmv(Y))$
$||\nu X'; permit(X', transfer)$

The process definition for $pgrant$ (multi-step delegation)

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$W :=$ **subject** w `rvk` $-$ `del` **delegator** u **delegatee** v **object** τ
$permit(X, pgrant) \rightarrow$
$(\nu Y; Y.\textbf{subject} := u; Y.\textbf{action} :=$ `delegated`$; Y.\textbf{delegatee} := v;$
$Y.\textbf{object} := \tau; Y.\textbf{type} := grant; add(Y);$
$\nu V; V.\textbf{subject} := v; V.\textbf{action} :=$ `can` $-$ `delegate`$; V.\textbf{object} := \tau;$
$V.\textbf{type} := grant; add(V); \nu W; rcv(W, x, rvk - pgrant);$
$permit(W, rvk - pgrant); rmv(Y); rmv(V))$
$||\nu X'; permit(X', pgrant)$

The process definition for $ptransfer$ (multi-step delegation)

$X :=$ **subject** u `delegates` **delegatee** v **object** τ
$W :=$ **subject** w `rvk` $-$ `del` **delegator** u **delegatee** v **object** τ
$permit(X, ptransfer) \rightarrow$
$(\nu Y; Y.\textbf{subject} := u; Y.\textbf{action} :=$ `delegated`$; Y.\textbf{delegatee} := v;$
$Y.\textbf{object} := \tau; Y.\textbf{type} := transfer; add(Y);$
$\nu V; V.\textbf{subject} := v; V.\textbf{action} :=$ `can` $-$ `delegate`$; V.\textbf{object} := \tau;$
$V.\textbf{type} := transfer; add(V); \nu W; rcv(W, x, rvk - ptransfer);$
$permit(W, rvk - ptransfer); rmv(Y); rmv(V))$
$||\nu X'; permit(X', ptransfer)$

where

$$permit(X, \tau) \rightarrow skip$$

for $\tau \in \{rvk - transfer, rvk - pgrant, rvk - ptransfer\}$.

Note that in the case of multi-step delegation, the revocation leads to the removal of both the record object and the object giving the right to delegate to the delegatee.

In the above process definitions we did not take into account the type (grant or transfer) of the delegation, nor did we specify how the delegatee acquires the right to use the delegation. These two notions will be presented in Section 6.5.

6.5 Using a delegation

The main purpose of performing a delegation is to give a user that does not normally have the right to execute a task, this right. In Section 6.4 we defined the effects of a delegation process. In this section we present how one can express the delegation right acquired by a delegatee on the one hand and how to express delegation chains in the case of multi-step delegation on the other hand.

6.5.1 Acquiring the right to use an object

In our model we assume that delegation can refer to any attribute. In this section, we suppose that the object of delegation is a task. This assumption does not undermine the generality of the model as the same logic applies to any other object of delegation. We make this choice to show the flexibility of the model.

We say a user u has the right to use task τ if one of the following three conditions holds:

- u has initially the right to use τ
- u acquired the right to use τ by a grant delegation
- u acquired the right to use τ by a transfer delegation and u did not delegate this right.

In order to take into account these different cases, we define a trust negotiation rule that delivers an object certifying that a user can use the task if at least one of the above conditions is satisfied. Note that, in order to be faithful to the syntax given in Chapter 4 we suppose that objects certifying that a user can use a task are expressed by an object of the form

$$\textbf{subject}\ u\ \texttt{can} - \texttt{use}\ \textbf{task}\ \tau$$

and not

$$\textbf{subject}\ u\ \texttt{can} - \texttt{use}\ \textbf{object}\ \tau.$$

Thus one may impose in these cases the addition of the attribute **nature** and associate it with the corresponding value $task$

$X :=$ **subject** u `can − use` **task** τ
$Y :=$ **subject** z `delegates` **delegatee** u **object** τ **nature** $task$
$Z :=$ **subject** z `can − use` **task** τ
$W :=$ **subject** u `delegates` **delegatee** y `object` τ **nature** $task$ **type** $transfer$
$put(X, x) \leftarrow get(X, self) \vee (get(Y, self) \wedge get(Z, self) \wedge not(has(W)))$

6.5.2 Constructing a delegation chain

In the above trust negotiation rule we used the object

subject z `delegates` **delegatee** u **object** τ

to express the fact that a delegation from z to u took place. However, such an object can either be the product of the actual execution of the delegation by z to the delegatee u, or can be acquired through a multi-step delegation.

In the case of multi-step delegation, one needs to construct the delegation chain (from the initial delegator to the last delegatee.). To do this, one need to define a transitive relation between delegations. We express this by a trust negotiation rule of the form:

$X :=$ **subject** u `delegated` **delegatee** w **object** τ **nature** $task$ **type** t
$Y :=$ **subject** u `delegated` **delegatee** v **object** τ **nature** $task$ **type** t
$Z :=$ **subject** v `delegated` **delegatee** w **object** τ **nature** $task$ **type** t
$put(X, self) \leftarrow get(Y, self) \wedge get(Z, self)$

where $t \in \{grant, transfer\}$.

This rule allows to keep track of a delegation chain by providing certificates that link two users only if no revocation occurred between them. The assumption that the object

subject u `delegated` **delegatee** w **object** τ **nature** $task$ **type** t

is removed when a revocation occurs guarantees that one can only deliver certificates in the presence of non-broken delegation chain.

6.5.3 Delegation as a graph representation

One can represent a delegation chain for a given delegation object τ as a graph $G_\tau = (V_\tau, E_\tau)$, where V_τ is the set of vertices containing the users and the set $E_\tau \subseteq V_\tau \times V_\tau$ represents the delegation relation, we say $(u, v) \in E_\tau$ if there exist an object

$X :=$ **subject** u `delegated` **delegatee** v **object** τ

such that

$$\mathcal{E}, \mathcal{S}, e_i \models put(X, i)$$

where e_i is the entity responsible for the delegation object τ, $\mathcal{E}$ and $\mathcal{S}$ represent the set of entities and their corresponding local states as described in Chapter 4. When a delegation is revoked, i.e. when the object is removed from the repository of the entity, the corresponding edge is removed from E_τ.

Let E_τ^+ denote the transitive closure for the delegation relation in G_τ and denote by u_i the initial delegator[1]. We say a user w has the authorization for the delegated object τ if $(u_i, w) \in E_\tau^+$ that is there exists a path from the initial delegator to the user w with respect to the delegation of τ. Moreover, if an edge in this path is removed, (u_i, w) may no longer belongs to the transitive closure of the graph, in which case user w would lose authorization over the delegation object. In this model we suppose a cascading revocation. Also the assumption that a delegator can only delegate once allow the construction of simple graphs avoiding loops and branching.

However, allowing multiple delegation would authorize one user to gain authorization for a given object through different delegations. In this case, when a delegation is revoked, the delegatee may still have the authorization to use the object gained through another delegation (if there exists another path for the same delegation.). In [75] this case is considered and revocation takes place only if there is no other path in the delegation graph that leads to the same authorization. Note that in our framework we can consider the same assumption. In fact, multiple delegation amounts to the addition of multiple objects referring to these delegations, as such when revoking a given delegation, other objects remain in the repository and thus the authorization for the delegated object remains possible.

Example Suppose a, b, c, d and e are users, a is the initial delegator. Suppose that a delegates the right τ to b who in turn delegates it to c who delegates it to d who delegates it to e as follows:

$$a \xrightarrow{\tau} b \xrightarrow{\tau} c \xrightarrow{\tau} d \xrightarrow{\tau} e$$

In order to compute the transitive closure for the delegation chain, suppose that:

$$a \xrightarrow{\tau} b \tag{6.3}$$
$$b \xrightarrow{\tau} c \tag{6.4}$$
$$c \xrightarrow{\tau} d \tag{6.5}$$
$$d \xrightarrow{\tau} e \tag{6.6}$$

[1]The initial delegator has an object certifying that he can use the delegation object

Then,

$$(6.3)(6.4) \Rightarrow a \xrightarrow{\tau} c \tag{6.7}$$
$$(6.4)(6.5) \Rightarrow b \xrightarrow{\tau} d \tag{6.8}$$
$$(6.5)(6.6) \Rightarrow c \xrightarrow{\tau} e \tag{6.9}$$
$$(6.7)(6.5) \Rightarrow a \xrightarrow{\tau} d \tag{6.10}$$
$$(6.8)(6.6) \Rightarrow b \xrightarrow{\tau} e \tag{6.11}$$
$$(6.10)(6.6) \Rightarrow a \xrightarrow{\tau} e \tag{6.12}$$

Thus there exist in the transitive closure of the above chain an edge from the initial delegator a to e labeled with the delegation τ. This provides a certificate that a delegates to e.

Suppose now that b revokes the delegation object from c, then the delegation chain becomes:

$$a \xrightarrow{\tau} b \qquad\qquad c \xrightarrow{\tau} d \xrightarrow{\tau} e$$

That is, we have the following transition relations

$$a \xrightarrow{\tau} b \tag{6.13}$$
$$c \xrightarrow{\tau} d \tag{6.14}$$
$$d \xrightarrow{\tau} e \tag{6.15}$$

Thus we can compute,

$$(6.14)(6.15) \Rightarrow c \xrightarrow{\tau} e \tag{6.16}$$

That is there does not exist in the broken chain an edge from the initial delegator a to e labeled with the delegation τ, that is no object stating that a can delegate to e can be provided and e's delegation is automatically revoked. The same applies to c and d.

Note that the condition that the *initial delegator* has an object certifying that he can use the delegation object makes it possible to discard the chain that occurs after the chain break (i.e. the revocation).

6.6 An illustrative example

In this section we present a simple access control system consider two different scenarios to illustrate the expression of delegation. We suppose the existence of two entities:

The central authority denoted by e_{ca} that is responsible for issuing certificates related to user roles and permissions associated to roles. The central authority can send a certificate to an entity if it has it in its repository.

A central repository denoted by e_{cr} that is responsible for storing documents.

Scenario 1: Only a user in role manager can store and read documents in the central repository.

- The central authority e_{ca}

  Security policy The security policy of e_{ca} consists of the following trust negotiation rules:

$$put(X, x) \leftarrow has(X)$$

$$X := \textbf{subject}\ u\ \texttt{is}-\texttt{member}\ \textbf{role}\ r$$
$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{role}\ r$$
$$put(Y, x) \leftarrow has(Y)$$

- The central authority e_{cr}

  Security policy The security policy of e_{cr} consists of the following access control rules:

$$X := \textbf{subject}\ u\ \texttt{request}\ \textbf{object}\ o\ \textbf{owner}\ z$$
$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{role}\ manager$$
$$permit(X, store) \leftarrow get(Y, ca)$$

  Workflow

$$permit(X, store) \rightarrow \nu Y; Y.\textbf{object} := X.\textbf{object}; Y.\textbf{owner} := X.\textbf{owner}; add(Y)$$

$$W := (\nu X; rcv(X, x, store); permit(X, store))!$$

Scenario 2: A user delegated by the manager can also store and read documents in the central repository.

In this scenario, we suppose that the system supports single-step delegation. Also we suppose that to delegate the right to store a document, a user should be in the role manager. Mary is a manager, she delegates the permission to store an object in the central repository to Claire.

These assumptions modify the security policy of e_{ca} by adding an access control rule for delegation and its corresponding process definition in the workflow. Further, we need to add a trust negotiation rule to take into account the constraint on who can delegate. The new rules for the system are defined below.

- The central authority e_{ca}

  Security policy The security policy of e_{ca} consists of the following access control rule:

$$X := \textbf{subject}\ u\ \texttt{delegates}\ \textbf{delegatee}\ v\ \textbf{object}\ store\ \textbf{nature}\ task$$
$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{1}-\texttt{delegate}\ \textbf{object}\ store\ \textbf{type}\ grant$$
$$Z := \textbf{subject}\ v\ \texttt{can}-\texttt{rcv}-\texttt{delegation}\ \textbf{object}\ store$$
$$W := \textbf{subject}\ u\ \texttt{delegated object}\ store\ \textbf{type}\ grant$$
$$permit(X, grant) \leftarrow get(Y, self) \wedge get(Z, self) \wedge not(has(W))$$

and the following trust negotiation rules:

$put(X, x) \leftarrow has(X)$

$X :=$ **subject** u `is − member` **role** r
$Y :=$ **subject** u `can − use` **role** r
$put(Y, x) \leftarrow has(Y)$

$X :=$ **subject** u `can − 1 − delegate` **object** $store$ **type** $grant$
$Y :=$ **subject** u `can − use` **role** $manager$
$put(X, self) \leftarrow get(Y, self)$

$X :=$ **subject** u `can − use` **task** $store$
$Y :=$ **subject** z `delegated` **delegatee** u **object** $store$ **nature** $task$
$Z :=$ **subject** z `can − use` **task** $store$
$put(X, x) \leftarrow get(X, self) \vee (get(Y, self) \wedge get(Z, self) \wedge not(has(Z)))$

Workflow

$permit(X, grant) \rightarrow$
$(\nu Y; Y.\textbf{subject} := u; Y.\textbf{action} :=$ `delegated`$; Y.\textbf{delegatee} := v;$
$Y.\textbf{object} := store; Y.\textbf{type} := grant; add(Y);$
$\nu W; rcv(W, x, rvk - grant); permit(W, rvk - grant); rmv(Y))$
$||\nu X'; permit(X', grant)$

$permit(X, rvk - grant) \rightarrow skip$

$\mathcal{W} := \nu X; rcv(X, x, grant); permit(X, grant)!$

- The central authority e_{cr}

Security policy The security policy of e_{cr} consists of the following access control rules:

$X :=$ **subject** u `request` **object** o **owner** z
$Y :=$ **subject** u `can − use` **role** $manager$
$Z :=$ **subject** u `can − use` **task** $store$
$permit(X, store) \leftarrow get(Y, ca) \vee get(Z, ca)$

Workflow

$permit(X, store) \rightarrow \nu Y; Y.\textbf{object} := X.\textbf{object}; Y.\textbf{owner} := X.\textbf{owner};$
$add(Y)$

$W := (\nu X; rcv(X, x, store); permit(X, store))!$

In addition to e_{ca} and e_{cr} we add two entities for Mary and Claire, namely e_m and e_c.

- e_m

  Workflow:

  $$\nu X; send(X, ca, grant)!||\nu Y; send(Y, cr, store)!$$

- e_c

  Workflow:

  $$\nu X; send(X, cr, store)!$$

6.7 Conclusion

In this chapter we presented an overview of delegation models presented in the literature and we proposed a model to express flexible delegation and revocation in our attribute-based access control framework. Our objective was to be faithful to the syntax presented in Chapter 4 while presenting a structure that is flexible enough to represent different kinds of delegation. We make use of the interdependency between the access control rules and the trust negotiation rules to base delegation authorization and delegation rights of certificates.

The delegation act is evaluated with respect to the disclosure of objects certifying that the delegator has the right to delegate and the delegatee has the right to receive the delegation. This generality allows more flexibility. Instead of taking into account different delegation constraints in the delegation rule, one can specify constraints on the delivery of these certificates.

Further, expressing the right to use the delegation or the propagation of the delegation is also treated without any extra mechanism by trust negotiation rules. This distinction leads to more flexibility in the management of constraints on the different steps of the delegation process.

Finally, the workflow structure in our framework provides an efficient tool to update the state of the system with respect to the execution of delegations or their corresponding revocations.

Chapter 7

Separation of duty constraints

In Chapter 6, we presented different aspects of delegation in a dynamic context. However, the existence of delegation, along with the possibility to activate and deactivate users in roles, increases the chances of having security breaches or collaboration between users working on sensitive tasks, leading to fraud situations. When expressing an access control policy in an organization, one has to take into account *sensitive tasks*. By *sensitive tasks* we mean tasks that need security constraints in order to be executed such as those involved in a banking process or a military procedure. Security constraints involve mainly separation of duty constraints, where two different tasks cannot be completed by the same user; or binding of duty constraint in which case two different tasks need to be performed by the same user; or least privilege property by which a user only activates the minimum set of permissions that he needs in order to perform a task.

In Section 7.1 we present an overview of the different aspects of separation of duty constraints and their enforcement in different logical frameworks. In Section 7.2 we give our modeling of the different aspects of separation of duty constraints, and we define general separation of duty constraints properties. We show the difference between modeling high-level security constraints and expressing low-level policy enforcement in the workflow within a given entity. In Section 7.3, we also give some security properties that are to be enforced on the global level.

7.1 Related Works

7.1.1 On the history of separation of duty

Separation of duty is a very important concept in computer security. It was first introduced in 1975 by Saltzer and Schroeder [66] by the name of "separation

of privilege" as one of the design principles for the protection of information in computer systems.

> Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information.

Static separation of duty

In its simplest incarnation, separation of duty ensures that one user cannot perform two mutually exclusive operations. For example, when processing an invoice, the operations *record* and *authorize payment* for the invoice cannot be authorized to the same user since this may lead to a fraud situation. To enforce this condition, one can allocate each authorization to a different user. In this case the separation of duty is static, since users cannot change roles for the treatment of two different invoices. However such constraint is not always feasible in real world applications. In fact in an organization it is often the case that the same person can perform these two operations. A more relaxed version of separation of duty states that one user cannot perform two mutually exclusive operations *at the same time*. For example, if two users have both the right to *record invoice* and *authorize payment*, then dynamic separation of duty mandates that if one user actually performs the operation *record invoice* then he will not be authorized to perform the operation *authorize payment*.

Dynamic separation of duty

Several other *dynamic* separation of duty properties came to light when dealing with real life examples. In [28] Clark and Wilson argue that separation of duty can be ensured by separating an operation into several subparts and requiring that each subpart be executed by a different person. They give the example of *purchasing an item* involving the following operations:

- authorizing the purchase order,
- recording the arrival of the invoice,
- recording the arrival of the item, and
- authorizing the payment

and state that by allowing the same person to perform all operations, fraud may occur, whereas if each operation is performed by a different user, the likelihood of fraud is diminished.

Object-based separation of duty

In 1990, Nash and Poland [60] gave a more flexible constraint called *object based separation of duty*. They argue that to preserve the integrity of the purchase transaction for a given data item (object) a user is permitted to perform one of the above stated operations if:

- The user is authorized to perform the given operation on the data item,
- The user did not execute any other operation on that data item

In their example, the authors put the constraint on the object rather that on the user. That is they add to the specification of the objects (i) which operations are permitted, and (ii) the identity of the user that performed the operation when executed. Moreover, to each user they add the list of operations permitted to that user. Thus, checking the constraint amounts to checking (i) if the user has the authorization to perform the operation and (ii) if he did not execute another operation on the object.

Operational separation of duty

In 1995 Ferraiolo, Cugini and Kuhn [36] claimed that static and dynamic separation of duty are sometimes very strong constraints. They stated that to preserve the purchase transaction from fraud, it is enough to assume that no user can perform all the operations associated with the transaction. This is the *operational separation of duty* property and is expressed in RBAC as a constraint that has to be checked before assigning operations to a role.

History-based separation of duty

In 1997 Simon and Zurko [71] stated that *object-based separation of duty* does not allow the same user to perform more that one operation on the same object, whereas *operational separation of duty* does not allow one user to perform all the operations for a given task on different objects even if this is authorized by the policy. They proposed the addition of *history-based separation of duty* thus defining separation of duty constraint based on the actual execution trace of the users.

In fact, taking the same example by Clark and Wilson, the authors claimed that usually the same persons that are responsible for authorizing a purchase can also authorize the payment, and thus the only restriction to be enforced is that a user may not authorize payment for the same invoice he authorized. Also one can add that two users must authorize the invoice before it can be treated. This restriction cannot be taken into account by simple dynamic separation of duty which only states restrictions on operations done at the same time. It cannot be taken into account in operational separation of duty since it puts constraints on the set of operations but does not take into account the different objects. In order to take into account such constraints, the authors proposed to keep a log for executed and non-executed actions for each user and checking

separation of duty constraints with respect to prior executions of these users. As such they proposed a rule based language for the specification of separation of duty constraints, and a history based implementation that guarantees the enforcement of such constraints.

We note that separation of duty constraints can be imposed on the policy level as is the case of static and dynamic separation of duty constraints, but also on the run-time level as in the case of object or operation-based separation of duty. As such, when talking about separation of duty constraints, we distinguish between three different aspects, namely the expression of security constraints on the policy specification level, the enforcement of constraints on the run-time level and the model checking for a security breach in the presence of the security constraints.

7.1.2 Specifying separation of duty constraints

With the rising need to protect the resources of organizations from internal threats in addition to the protection from external threats, many researches have focused on the study of separation of duty constraints in the three different areas, namely the specification of the constraints on the policy level and the enforcement of such constraints on the implementation level such as in [36, 71, 53, 14, 11], and on the checking of the security policy against possible breaches of these constraints [70]. In this subsection we present some of the work that has been done in these areas.

While the early research work in the separation of duty area defined constraints with respect to users and operations as seen in Section 7.1.1, later work and most recent studies are done with respect to the Role based access control framework (RBAC).

In [36], Ferraiolo, Cugini and Kuhn present the main features of RBAC and adapt the separation of duty constraints to the role structure. They argue that administrators can place constraints on *role authorization*, *role activation* and *operation execution*. Moreover, constraints may affect the authorization of an operation to a role and the authorization of performing an operation on objects. Separation of duty constraints are imposed on roles rather than on operations. Namely, mutually exclusive operations are associated to mutually exclusive roles, and the separation of duty constraints are set on the user-role relation. *Static separation of duty* is a constraint added in the preconditions of *authorizing a role* to a user. For example, if the roles *auditor* and *teller* are mutually exclusive, then a user authorized for the role *teller* will not be authorized for the role *auditor*. The *dynamic separation of duty* is a constraint added in the preconditions of *activating a role*. For example, the same user may be authorized for both *auditor* and *teller* but will not be authorized to activate the role *teller* if he is already active in the role *auditor*.

Note that *static separation of duty* is set on the *authorization level* while *dynamic separation of duty* is checked on the *execution level*.

In [14] the specification of separation of duty constraints is also expressed with respect to roles. Cassandra uses Datalog with constraints to express access

control rules. The user-defined functions and the parameters in the predicates allow the definition of dynamic separation of duty and object based separation of duty.

For example, the constraint that a user u *can activate* the role r on object o if he has not activated the role r' on the same object is done by the definition of an *aggregate rule* that counts the number of activations of role r', and a rule stating that the activation of role r is permitted if the activation count for role r' is zero.

$$canActivate(u, r(o)) \leftarrow countinitiators(n, u, o), n = 0$$

$$countinitiators(count < z >, u, o) \leftarrow hasActivated(z, r'(o)), z = u$$

An object based separation of duty can also be defined allowing a user to activate the role r for object o if he did not already activate it for another object o'. To do so a function $conflict$ is defined, that returns the set of conflicting roles. This function is used to express constraints on the activation of the role with respect to conflicting objects as follows:

$$canActivate(u, r(o)) \leftarrow \neg hasActivated(u, r(o')), \{o, o'\} \subseteq conflict()$$

The separation of duty constraints in the Cassandra framework relies on the non-occurrence of an activation. In order to express this negative condition the authors define a counting function for the activation of a role with respect to a given user on a given object. This allows the expression of history based separation of duty with respect to roles, but it is rather difficult to express operation based separation of duty as the activation only takes into account roles. Also the definition of a counting function for each constraint expression can be sometimes complex.

In SecPAL [11], separation of duty constraints are expressed by means of query methods. A method can check for the validity of some given conditions and adds an assertion to the assertion context in case the conditions are satisfied with respect to the assertion context.

For example, to express that a payment transaction must be *initiated* and *authorized* by two distinct *managers*:

- a method `can-initiate(r, p)` checks if the requester is a manager and that no other user has initiated the payment, and if true adds the assertion "`Bank says r has initiated p`" to the local state

- a method `can-authorize(r,p)` checks if the requester is a manager and that there exits an assertion "`Bank says x has initiated p`" where $x \neq r$

SecPAL does not keep an explicit log for activation or execution of facts. However, it is possible to keep a trace of the state change of the system through the definition of queries and methods. Further, the use of assertions makes it

more flexible and intuitive to define action based or object based separation of duty in SecPAL.

Crampton [30] argues that the specification of separation of duty constraints is rather complicated and it is enough to enforce such constraints without a high level specification in order to secure the safety of an organization. In his enforcement model, Crampton relies on the specification of a family of *bad sets*. For instance, to express that a user cannot perform both permissions p_1 and p_2, a constraint $c = (U, \{p_1, p_2\}, x)$ defines a family of constrained sets $Q_c = \bigcup_{u \in U}(\{u\}, \{p_1, p_2\})$. An authorization constraint c is *enforced* if for all $Q \in Q_c$ it is not possible for all requests $q \in Q$ to be granted.

Crampton assumes that the configuration or state of a system is modified through the interaction of the user with the system and this through requests to invoke a permission, activate a role, etc. In order to implement the enforcement model to history based constraints, he constructs a dynamic *blacklist* that needs to be checked before a decision on the request is made.

Instead of checking at each time if the constraint is preserved, by looking at the trace of already executed actions, Crampton explicitly enumerates the list of all *bad states* and compares each request to this list (i.e. an action is authorized if it does not lead to a bad state.).Crampton's model relies on a *constraint monitor* for the enforcement of these constraints. However this model can only express constraints where the constraint set has at most two elements, that is a constraint stating that one user cannot perform all of p_1, p_2 and p_3 cannot be enforced as some ambiguity will arise in the construction of the corresponding black list. Such a model may be practical for small organizations where constraints and bad states can be enumerated but it will be rather difficult to implement it in a distributed environment or large organizations as enumerating all bad states may be complicated.

In [76] the authors offer methods to guarantee security by making verification checks for every workflow step depending on generic constraints in order to prevent security breaches in the presence of delegation. The authors discuss two kinds of attacks that may occur when allowing delegations.

Example A: Suppose a task t is to be completed by one user by activating roles r_1 and r_2 successively, and that Alice is a member of role r_1 and Bob is member of role r_2. By allowing delegation, Alice can delegate her role r_1 to Bob who can then execute both subtasks that constitute t, a privilege that he wouldn't have acquired without the delegation.

Example B: Suppose now that two subtasks s_1 and s_2 need to be performed by two distinct users in the same role r, that Alice is a member of a role r, and can delegate r to Bob who is member of the role r_1. Then Alice transfers her role r to Bob who can now perform task s_1, then Alice revokes the role r and performs s_2 herself.

In both cases the attack occurs when a user who cannot perform the task without the delegation gains more power after the delegation.

The authors propose an algorithm to check the security of the system in the presence of delegation. Their algorithm consists of checking, after the execution of the subtasks, whether the sources of the privileges satisfy the constraints of the workflow.

One of the few works to define a specification to express and verify security constraints is by Li and Wang [53]. They distinguish between a high level policy design and a low level enforcement design to secure sensitive tasks. Sensitive tasks can be made of several steps, in the high level security policy only general specifications on the task as a whole are defined, while specific enforcement details on each single step of the task are guaranteed in the enforcement level.They define an algebra to express and specify security constraints. Their algebra is based on terms and can express both quantitative and qualitative constraints. An atomic term can be a user, a role or the keyword *All*. Terms are constructed by means of operators, for example

$$((Manager \odot Accountant) \otimes Treasurer)$$

is a term that requires a manager, an accountant and a treasurer such that the first two can be satisfied by a single user.

$$\{Alice, Bob, Carl\} \otimes \{Alice, Bob, Carl\}$$

is a term that requires any two distinct users of a list of three. Their algebra is based on RBAC but can be used in an attribute based framework. They define a configuration (state of the system) by a pair $< U, UR >$ where U is a set of users and UR denotes user role membership.

A security policy is defined by a couple $< task, \phi >$ where ϕ is a term in the algebra and denotes that only users that satisfy ϕ can perform the task $task$. Enforcing the policy amounts to defining a term for each sensitive task, such that every set of users that together can perform the task must satisfy the term. In the case of the enforcement of static separation of duty, this can be done by specifying all the permissions needed to perform the task (the task is usually divided into several steps) and check that any user set in which all users together have all the permissions for the given task must satisfy ϕ. In the case of a dynamic separation of duty, one needs to maintain a history about these steps and who has executed them. For any task instance, one can compute the set of users who have performed at least one step in the instance U_{past}. Before a user performs a step, the system checks to ensure whether there exists a superset of $U_{past} \cup \{u\}$ that can satisfy ϕ upon finishing all steps of the task.

With this algebra one can verify whether the term definition is coherent with a given configuration and user set.

This specification has been extended by Basin et al. [10] to take into account dynamic policies. To do this the authors generalize this algebra to take into account multisets of users and interpret the algebra's terms over workflow traces allowing changes in role assignments. A workflow is defined in a process algebra. They present a map of the term algebra into processes that can act

with access control policies and workflow and are able to make use of the process operational semantics in order to enforce security checks. This approach however only supports user-role based constraints and does not support action specific constraints.

In [18], Bertino, Ferrari and Alturi use first order logic to check the consistency of constraints defined over workflows. They use predicate symbols to express workflow specifications, effects of workflow execution and restrictions on the set of roles/users that can execute a task. The separation of duty constraints are then defined in first order logic with respect to these predicate symbols. A workflow consists of several tasks to be executed sequentially. A planner generates a set of possible assignments, so that all constraints stated as part of the authorization specification are satisfied. The planner is activated before the workflow execution starts to perform an initial plan. This plan can, however, be dynamically modified during workflow execution to account for specific situations, such as aborting a task.

This framework permits one to reason on the execution trace of the workflow and to differentiate between checking static constraints on access control level and dynamic constraints on the workflow execution level. The model is however rigid in the sense that each predicate specifies a specific condition or constraint. As such, adding new constraints, or modifying the constraints to deal with user-action specification rather than user-role assignments amounts to a change in the syntax of the language.

Schaad, Lotz and Sohr [70] extend separation of duty constraints to workflow. They describe the workflow and the access control policy (including the delegation and revocation steps) as steps in a finite state machine. They define separation of duty constraints in Linear Temporal Logic and use a model checker to check for policy violation. They only focus on the model checking aspect and do not offer an enforcement specification for the security constraints.

7.2 On the expression of security properties

Security constraints need to be defined at the policy level, in order to formally specify the model, but also on the policy enforcement level (i.e. in the execution of the workflow) in order to enforce such constraints.

We distinguish between the *security properties*, that specify the objectives to be satisfied in the security policy, and *the security constraints* that are to be enforced by the workflow. We define a task to be an ordering over a set of steps or subtasks. In our framework a task can be seen as a subworkflow. In the high level policy language only security constraints on the tasks are specified, whereas in the workflow execution constraints may be enforced on the individual steps constituting the task.

In [53], the authors present a scenario consisting of a sensitive task denoted by $\texttt{share} - \texttt{classified} - \texttt{doc}$ that involves the subtasks `request`, `retrieve`, `prepare`, `send`. On the one hand, a high level policy may enforce that at least two managers must be involved in the task. On the workflow level, on the other

hand, this is translated into the allocation of users to the different steps in the task. Namely, one supposes that `request` and `send` need to be performed by the same user in the role **coordinator** whereas two instances of the `approve` need to be performed by two different **managers** before a **docAdmin** performs the step `retrieve` document.

We make use of the interplay between the security policy and the workflow evolution of our unified framework defined in Chapter 4 to express different aspects of security constraint.

7.2.1 On activation

When talking about separation of duty, one needs to define the notion of activity. In fact, except for static separation of duty, all types of separation of duty constraints depend on an activity, be it activation of a role, execution of an action or use of an object, etc. In this section we present the notion of activation with respect to roles since this is the most common notion of activation due to the development of the separation of duty constraints in the presence of the RBAC framework.

In our model a session is an asynchronous execution of workflows of a collection of entities in an environment. In Chapter 8, we present a modeling for role activation. In this section we situate ourselves in a more flexible framework that does not necessarily depend on roles. We say that a user is *active for a given task* when a record of the task execution can be acquired from the repository of the entity responsible of the task.

Executing a task Activation for *tasks* is given in the notion of execution. We assume that to execute a task a user u needs to get an object specifying that u can use the given task as described in Chapter 6. We express the recording of an *execution object* for a given task τ in the process definition of the tasks. An execution object is of the form:

$$\textbf{subject}\ u\ \texttt{executed}\ \textbf{task}\ \tau\ \textbf{object}\ o$$

The process definition for a task τ is given by:

$$\begin{aligned} &permit(X,\tau) \rightarrow \nu Y; Y.\textbf{subject} := X.\textbf{subject}; Y.\textbf{action} := \texttt{executed}; Y.\textbf{task} := \tau; \\ &\quad Y.\textbf{object} := X.\textbf{object}; add(Y); P \end{aligned}$$

where P is a process.

Using the execution objects In order to specify security constraints in the following subsection, we suppose that each entity has in its trust negotiation policy rules to regulate the access to these certificates. To this end, we assume that each entity has in its trust negotiation policy a rule of the form:

$$put(X, self) \leftarrow has(X).$$

Such a rule permits the retrieval of the objects stored in the repository of the entity when computing the least fixed point. This makes it possible to know the state of the entity with respect to the execution of the workflow and thus allows a form of *centralization* of information. This information can then be used by the trust negotiation policy to verify security properties in a distributed environment.

7.2.2 Static and dynamic separation of duty

As we presented in Section 7.1.1, different aspects of separation of duty exist in the literature. Although static and dynamic separation of duty are generally defined with respect to roles, in this section we express separation of duty constraints in general. The specific case of role-based separation of duty will be presented in Chapter 8.

Given a set $\mathcal{T}$ of mutually exclusive tasks, we defined the following separation of duty constraints on $\mathcal{T}$.

Static separation of duty (SSOD)

SSOD requires that one user does not have the authorization to use more than one task in $\mathcal{T}$.

This is expressed by the trust negotiation rule:

$$X := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{task}\ \tau$$
$$Y_{\tau'} := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{task}\ \tau'$$
$$put(X, x) \leftarrow get(X, self) \bigwedge_{\tau' \in \mathcal{T}, \tau' \neq \tau} not(has(Y_{\tau'}))$$

The above defined rule assumes that SSOD is preserved in the initial states and checks for violations throughout the execution of the system.

Dynamic Separation of duty (DSOD)

In its original definition, DSOD requires that a user cannot be active in more than one role from a set of mutually exclusive roles. This notion will be expressed in Chapter 8. Note however that if one translates this definition to take into account mutually exclusive tasks, then DSOD requires that a user *active* in a given task, is not authorized to activate (i.e. execute) another task from a set of mutually exclusive tasks. This is the operational separation of duty constraint that we present in the next subsection.

7.2.3 More dynamic separation of duty constraints

As presented in Section 7.1.1, operational separation of duty and object-based separation of duty are to be enforced at runtime. We present the specification of these constraints in this section.

Operational separation of duty (OpSOD)

OpSOD states that one user cannot execute more than one task from a set of mutually exclusive tasks $\mathcal{T}$ in a given entity. This amounts to adding a trust negotiation rule of the form:

$X :=$ **subject** u `can − use` **task** τ
$Y :=$ **subject** u `executed` **task** τ
$Z_{\tau'} :=$ **subject** u `executed` **task** τ'
$put(X, x) \leftarrow get(X, self) \wedge not(has(Y)) \wedge \bigwedge_{\tau' \in \mathcal{T}, \tau' \neq \tau} not(has(Z_{\tau'}))$

In the general definition of operational separation of duty, we suppose that a user can only execute a given task once in the execution of a workflow instance. That is the task execution, unlike in the case of roles, cannot be deactivated in the execution of the workflow instance. One can express this constraint by taking into account the object on which the task is executed. This is the object-based separation of duty.

Object-based separation of duty (ObjSOD)

ObjSOD requires that for a given object o, a user can execute a task from a mutually exclusive set of tasks $\mathcal{T}$ on the object o if he did not already execute another task from $\mathcal{T}$ on the object o. This is expressed by the following trust negotiation rule:

$X :=$ **subject** u `can − use` **task** τ **object** o
$Y :=$ **subject** u `executed` **task** τ **object** o
$Z_{\tau'} :=$ **subject** u `executed` **task** τ' **object** o
$put(X, x) \leftarrow get(X, self) \wedge not(has(Y)) \bigwedge_{\tau' \in \mathcal{T}, \tau' \neq \tau} not(has(Z_{\tau'}))$

7.2.4 Binding of duty constraint

In addition to the separation of duty constraints we presented in this section, we define the binding of duty constraint. A binding of duty constraint requires that a given user must execute all tasks in a set of critical tasks $\mathcal{T}$. This differs from the previously defined constraints. In fact to be able to express binding of duty constraint one needs to check before the execution of each tasks in $\mathcal{T}$ whether the same user is executing the task. To this end we make use of the interplay between the workflow and the access control policy of the entity. Suppose τ is the first task to be executed from the set of tasks $\mathcal{T}$ then, in the process definition of τ we add an object certifying that the first task in $\mathcal{T}$ was executed.

This can be expressed by the process definition of τ as follows:

$X :=$ **subject** u `request` **object** o
$permit(X, \tau) \rightarrow$
$\nu Y; Y.$**task** $:= \tau; Y.$**action** $:=$ `is − executed`$; Y.$**order** $:= start; add(Y);$
$\nu Z; Z.$**subject** $:= u; Z.$**action** $:=$ `executes`$; Z.$**task** $:= \mathcal{T}; add(Z); P$

where P is a process defining the effect of the task τ. Note that here $\mathcal{T}$ denotes the identifier of the set of the sensitive tasks.

Then, the right to execute the sensitive tasks in $\mathcal{T}$ will be regulated by the following access control rule

$$
\begin{array}{l}
X := \textbf{subject } u \texttt{ request } \textbf{object } o \\
Y := \textbf{task } \tau \texttt{ is} - \texttt{executed } \textbf{order } start \\
Z := \textbf{subject } u \texttt{ executes } \textbf{task } \mathcal{T} \\
permit(X, \tau) \leftarrow get(X, self) \wedge (not(has(Y)) \vee has(Z))
\end{array}
$$

This rule states that a given user u can execute a task in $\mathcal{T}$ either

- if no one has already executed the first task for $\mathcal{T}$, in which case, after the execution of the task he will be constrained to execute the rest of the tasks in $\mathcal{T}$.

- Or, if the user u is the one who executed the first task and thus has the object certifying that he can execute the rest of the tasks.

7.2.5 History-Based separation of duty

Throughout this section we expressed the specification of security constraints that must be enforced in a given entity. However, this specification is not sufficient to secure an entity. We also need to specify the enforcement of such properties within the execution of entity's workflow. As presented by Schaad et al. in [70] there is a difference between defining security properties in the definition of a task for a business process at the modeling time and securing such properties in the task instance at run-time.

Suppose for example that a security rule specifies that tasks τ_1, τ_2 and τ_3 need to be executed by at least two users. Such a rule cannot be the object of a general specification alone since it depends on a specific run of the workflow (i.e. on a specific instance). In fact, in order to model such constraints, it is sufficient to add a *security task* that has as parameters all the information extracted from the workflow relevant to the execution of the tasks in question. This security task will act as a guard whose aim is to verify security properties on the fly.

For example, to take into account the constraint that one user cannot execute all the tasks τ_1, τ_2 and τ_3, we define a security task $check - sod$. Since the enforcement of security properties can only be done within a given entity, we add the security task $check - sod$ before the execution of the last task (here τ_3) in the workflow. The object of $check - sod$ is to check whether the user supposed to execute τ_3 did not already executed the two previous tasks.

We define the task $check - sod$ which takes as parameters the values of the subject attribute for the three tasks. Suppose that the workflow definition is

given by the following expression:

$X_1 :=$ **subject** u_1 `request` **object** o_1
$X_2 :=$ **subject** u_2 `request` **object** o_2
$X_3 :=$ **subject** u_3 `request` **object** o_3
$W = \nu X_1; \nu X_2; \nu X_3; rcv(X_1, x_1, \tau_1); permit(X_1, \tau_1) || rcv(X_2, x_2, \tau_2);$
$permit(X_2, \tau_2); rcv(X_3, x_3, \tau_3); \nu Y; Y.\mathbf{user} := X_3.\mathbf{subject};$
$Y.\mathbf{action1} := \tau_1; Y.\mathbf{action2} := \tau_2; Y.\mathbf{action3} := \tau_3; permit(Y, check - sod);$
$permit(X_3, \tau_3).$

Then the task definition of the task $check - sod$ will be given by an access control rule that checks if the same user executed the first two tasks as follows:

$X :=$ **action1** τ_1 **action2** τ_2 **action3** τ_3 **user** u_3
$Y_1 :=$ **subject** u_3 `executed` **task** τ_1
$Y_2 :=$ **subject** u_3 `executed` **task** τ_2
$permit(X, check - sod) \leftarrow not(has(Y_1)) \vee not(has(Y_2))$

Remark: Note that the notation for the objects X_1, X_2, X_3 and Y is not part of the syntax of the workflow. We added it here for illustration purposes as we did not express the access control rules related to the tasks in question.

Local enforcement versus distributed verification of constraints

Throughout this section we assumed that the specification of security properties is done in the entity responsible for the sensitive properties, then shared with other entities.

In fact the specification of security constraints requires the verification of *not* having an object. While this verification is straightforward in a given entity, since one can check for the non-existence of the object in a finite set of objects in the repository of the entity, it is not feasible in case the tasks are not in the same entity. To this end we define the general security properties in Section 7.3 that are to be checked on the level of the system.

7.3 Verifying the security properties

In this section we define the security properties that need to be satisfied on the system level. In fact, separation of duty and binding of duty constraints are security constraints that mainly affect the execution of the system. To this end we need to be able to verify if in a given model (a set of entities with security policy and local state) such security properties can be satisfied or violated.

7.3.1 Specifying security constraints

Let $\mathcal{M}$ be the set of messages sent but not yet received, $\mathcal{E}$ the set of entities and $\mathcal{S}$ the set of associated local states (see Chapter 4). Let us describe how one

can verify security constraints for the entire process by verifying the security properties after each workflow execution. To this end we define different decision problems to check for the enforcement of the security constraints presented in Section 7.2. A decision problem permits to check whether

$$\mathcal{M}, \mathcal{E}, \mathcal{S} \models \varphi$$

where φ corresponds to the specification of the security property in question.

Static separation of duty problem

Given a set of entities $\mathcal{E}$, an associated set of local states $\mathcal{S}$ (containing the status of the repository and the position of the workflow for each entity) and a set of messages sent but not yet received $\mathcal{M}$, and a set of mutually exclusive tasks $\mathcal{T}$.

We say that the static separation of duty constraint is preserved with respect to $\mathcal{M}, \mathcal{E}, \mathcal{S}$ if there exist a value u and entities e_i in $\mathcal{E}$ such that:

$$X_\tau := \textbf{subject } u \texttt{ can} - \texttt{use } \textbf{task } \tau$$
$$\mathcal{M}, \mathcal{E}, \mathcal{S} \not\models \bigwedge_{\tau \in \mathcal{T}} (\bigvee_{e_i \in \mathcal{E}} put_{e_i}(X_\tau, self))$$

Operational separation of duty

Given a set of entities $\mathcal{E}$, an associated set of local states $\mathcal{S}$ (containing the status of the repository and the position of the workflow for each entity) and a set of messages sent but not yet received $\mathcal{M}$, and a set of mutually exclusive tasks $\mathcal{T}$.

We say that the operational separation of duty constraint is preserved with respect to $\mathcal{M}, \mathcal{E}, \mathcal{S}$ if there exist a value u and entities e_i in $\mathcal{E}$ such that:

$$X_\tau := \textbf{subject } u \texttt{ executed } \textbf{task } \tau$$
$$\mathcal{M}, \mathcal{E}, \mathcal{S} \not\models \bigwedge_{\tau \in \mathcal{T}} (\bigvee_{e_i \in \mathcal{E}} has_{e_i}(X_\tau, self))$$

Object-based separation of duty

Given a set of entities $\mathcal{E}$, an associated set of local states $\mathcal{S}$ (containing the status of the repository and the position of the workflow for each entity) and a set of messages sent but not yet received $\mathcal{M}$, and a set of mutually exclusive tasks $\mathcal{T}$.

We say that the object-based separation of duty constraint is preserved with respect to $\mathcal{M}, \mathcal{E}, \mathcal{S}$ if there exist a value u and entities e_i in $\mathcal{E}$ such that:

$$X_\tau := \textbf{subject } u \texttt{ executed } \textbf{task } \tau \textbf{ object } o$$
$$\mathcal{M}, \mathcal{E}, \mathcal{S} \not\models \bigwedge_{\tau \in \mathcal{T}} (\bigvee_{e_i \in \mathcal{E}} has_{e_i}(X_\tau, self))$$

Binding of duty

Given a set of entities $\mathcal{E}$, an associated set of local states $\mathcal{S}$ (containing the status of the repository and the position of the workflow for each entity) and a set of messages sent but not yet received $\mathcal{M}$, and a set of mutually exclusive tasks $\mathcal{T}$.

We say that the binding of duty constraint is preserved with respect to $\mathcal{M}, \mathcal{E}, \mathcal{S}$ if there exists a value u and entities e_i in $\mathcal{E}$ such that:

$$X_\tau := \textbf{subject } u \texttt{ executed } \textbf{task } \tau$$
$$\mathcal{M}, \mathcal{E}, \mathcal{S} \models \bigwedge_{\tau \in \mathcal{T}} (\bigvee_{e_i \in \mathcal{E}} has_{e_i}(X_\tau, self))$$

7.3.2 Monitoring security properties

The separation of duty constraints are defined in terms of the executed tasks and can concern more than one entity in the model. As we argued in Section 7.2, in our framework we cannot enforce constraints that depend on the execution of workflows in more than one entity without centralizing the model. In Section 7.3.1, we gave the specification of various decision problems in order to check for the enforcement of security constraints. In order to guarantee the good functioning of the system, we consider that the verification of these security constraints is done by a *monitor* who, after each execution of the workflow, checks the overall model for security breaches.

In order to reason on the level of the system rather than on the level of each entity, the monitor performs a step-by-step model checking. In the case of constraint violation, the monitor should be able to intervene to stop the execution of the system. This necessitates the collaboration of all the entities in the model that should allow the disclosure of their information to the monitor on the one hand, and authorize the monitor to have access to the respective workflows on the other hand.

7.4 Conclusion

In this chapter, we defined the different aspects of separation of duty constraints. We showed that to specify such constraints, it is necessary to have a record for the execution of tasks. To this end we defined the general framework to record execution logs for the tasks, and gave the specification of static and dynamic separation of duty in the form of trust negotiation rules within a given entity. In order to enforce security constraints, we defined security properties on the level of the system (i.e. taking into account the collection of entities defining the model) and argued for the existence of a monitor that would play the role of a model checker to verify the good functioning of the overall system. However, this modeling is not entirely satisfactory. In fact, so far, the monitor is a mere observant of the system and cannot intervene in case of violation (unless to halt the execution of the workflow). It would be interesting to allow the monitor, in case of a violation, to trace back the source of the violation and thus allow the

policy modeler to modify the security policy of the entity in question in order to remedy to the problem.

Chapter 8

Encoding RBAC

In our framework we choose not to have an explicit RBAC structure. This allows us to have more flexibility in the expression of access control authorizations with respect to a collection of attributes rather than a predefined role structure. We believe that in doing so we are able to express a diversity of models and constraints without the need to extend the language or modify the model. It is however possible to integrate RBAC and role structures in our framework. In this chapter we provide a formalization of the RBAC structure and some of its properties. We give a modeling for the role structure in a distributed environment. We take into consideration the presence of role hierarchy and express role activation as well as delegation and separation of duty constraints.

8.1 Expressing a role-based access control framework

In role-based access control, access control authorizations are associated to roles and users are made members of roles. Let R be the set of roles in the model and U the set of users.

8.1.1 Expressing role-based access control structure

To express role-based access control in our framework we need to define *permission-role* and *user-role* as special objects as follows:

- permission-role relations are objects of the form:

  $$\textbf{role}\ r\ \texttt{is}-\texttt{assigned}\ \textbf{task}\ \tau$$

  where $r \in R$. Note that permissions in our framework correspond to tasks that can be executed in a given entity. These objects can be present in the repository at the initial state or can be modified dynamically by an administrator during the execution of the workflow.

- user-role relation are objects of the form:

$$\textbf{subject } u \texttt{ is} - \texttt{member } \textbf{role } r$$

where $u \in U$ and $r \in R$. This kind of object defines user role membership. As in the case of $permission-role$ relation objects, such objects can exists at the initial state or can be acquired by the security policy of the entity.

Acquiring permissions:

If we take into account the initial RBAC model without the presence of sessions, we can suppose that a user acquires permissions associated with a given role if the user is member of the role. In general this is expressed by a trust negotiation rule of the form:

$$\begin{array}{l} X := \textbf{subject } u \texttt{ can} - \texttt{use } \textbf{task } \tau \\ Y := \textbf{subject } u \texttt{ is} - \texttt{member } \textbf{role } r \\ Z := \textbf{role } r \texttt{ is} - \texttt{assigned } \textbf{task } \tau \\ put(X, x) \leftarrow has(Y) \wedge has(Z) \end{array}$$

Example 8.1.1. The role *clerk* has permissions *read* and *store* associated to it. *Bob* is a member of the role *clerk*. Suppose e_{rbac} is the entity responsible for the delivery of permission certificate for the role *clerk*. Then in the repository of e_{rbac} we have the following three objects:

$$\begin{array}{l} \textbf{role } clerk \texttt{ is} - \texttt{assigned } \textbf{task } read \\ \textbf{role } clerk \texttt{ is} - \texttt{assigned } \textbf{task } store \\ \textbf{subject } bob \texttt{ is} - \texttt{member } \textbf{role } clerk \end{array}$$

Then according to the trust negotiation rule

$$\begin{array}{l} X := \textbf{subject } u \texttt{ can} - \texttt{use } \textbf{task } \tau \\ Y := \textbf{subject } u \texttt{ is} - \texttt{member } \textbf{role } r \\ Z := \textbf{role } r \texttt{ is} - \texttt{assigned } \textbf{task } \tau \\ put(X, x) \leftarrow has(Y) \wedge has(Z) \end{array}$$

the entity e_{rbac} can deliver (put) objects of the form

$$\begin{array}{l} \textbf{subject } bob \texttt{ can} - \texttt{use } \textbf{task } read \\ \textbf{subject } bob \texttt{ can} - \texttt{use } \textbf{task } store \end{array}$$

The case presented in this example supposes the simplest case of RBAC, taking into account flat roles and the absence of activation sessions. In the next subsection we present the RBAC structure in the presence of role hierarchy.

8.1.2 Role hierarchy

When we talk about role hierarchy, we suppose the presence of a partial order among roles. We say a role r_1 is *senior* to role r_2 when the permissions associated

to r_2 are inherited by the role r_1. For example in a medical context the role *cardiologist* is senior to the role *doctor*. To express partial order on roles we take the same model defined for delegation chains (see Section 6.5). Thus we define a *seniority* object of the form:

$$\textbf{subject } r \texttt{ is} - \texttt{senior } \textbf{role } r'$$

Remark In the case of a branching roles we may have two objects

$$\begin{cases} \textbf{subject } r \texttt{ is} - \texttt{senior } \textbf{role } r' \\ \textbf{subject } r \texttt{ is} - \texttt{senior } \textbf{role } r'' \end{cases}$$

As in the case of delegation chains we define the role hierarchy chain as a trust negotiation rule of the form:

$$\begin{array}{l} X := \textbf{subject } r_1 \texttt{ is} - \texttt{senior } \textbf{role } r_3 \\ Y := \textbf{subject } r_1 \texttt{ is} - \texttt{senior } \textbf{role } r_2 \\ Z := \textbf{subject } r_2 \texttt{ is} - \texttt{senior } \textbf{role } r_3 \\ put(X, x) \leftarrow get(Y, self) \wedge get(Z, self) \end{array}$$

The trust negotiation rules allow the expression of a transitivity relation in the case of role hierarchy. In fact the least fixed point evaluation semantics of trust negotiation rules helps to regulate the relations between the different roles. If a role is revoked, the hierarchy is automatically modified. As such, as in the case of delegation, the role hierarchy relation can be viewed as a transition graph labeled with the inherited role (see Section 6.5.3).

Remark It is also possible to define role hierarchy with respect to junior roles rather than senior roles. to this end one defines the object

$$\textbf{subject } r' \texttt{ is} - \texttt{junior } \textbf{role } r$$

and a trust negotiation rule of the form:

$$\begin{array}{l} X := \textbf{subject } r' \texttt{ is} - \texttt{junior } \textbf{role } r \\ Y := \textbf{subject } r \texttt{ is} - \texttt{senior } \textbf{role } r' \\ put(X, self) \leftarrow get(Y, self) \end{array}$$

8.1.3 Role inheritance

The importance of the role hierarchy lies in the role inheritance property. When member of a role r, a user inherits membership for all roles junior to r. This is expressed by a trust negotiation rule of the form:

$$\begin{array}{l} X := \textbf{subject } u \texttt{ is} - \texttt{member } \textbf{role } r' \\ Y := \textbf{subject } u \texttt{ is} - \texttt{member } \textbf{role } r \\ Z := \textbf{subject } r \texttt{ is} - \texttt{senior } \textbf{role } r' \\ put(X, self) \leftarrow get(Y, self) \wedge get(Z, self) \end{array}$$

Example 8.1.2. Let us go back to Example 8.1.1 and assume that Mary is a manager, and that a manager is senior in the role hierarchy to the role clerk. This amounts to the addition in the repository of the entity e_{rbac} of the object

$$\textbf{subject}\ manager\ \texttt{is}-\texttt{senior}\ \textbf{role}\ clerk$$

that expresses the seniority property and the object

$$\textbf{subject}\ mary\ \texttt{is}-\texttt{member}\ \textbf{role}\ manager$$

that expresses Mary's role membership. In addition, we add to the security policy of e_{rbac} the inheritance rule:

$$\begin{array}{l} X := \textbf{subject}\ u\ \texttt{is}-\texttt{member}\ \textbf{role}\ r' \\ Y := \textbf{subject}\ u\ \texttt{is}-\texttt{member}\ \textbf{role}\ r \\ Z := \textbf{subject}\ r\ \texttt{is}-\texttt{senior}\ \textbf{role}\ r' \\ put(X, self) \leftarrow get(Y, self) \wedge get(Z, self) \end{array}$$

With this new setting, the entity e_{rbac} can deliver (put) additional objects concerning Mary. Namely, the object

$$\textbf{subject}\ mary\ \texttt{is}-\texttt{member}\ \textbf{role}\ clerk$$

certifies that Mary is a member of the role clerk by the inheritance rule, and thus can acquire the objects

$$\begin{array}{l} \textbf{subject}\ mary\ \texttt{can}-\texttt{use}\ \textbf{task}\ read \\ \textbf{subject}\ mary\ \texttt{can}-\texttt{use}\ \textbf{task}\ store \end{array}$$

by the permission acquisition rule presented in Example 8.1.1.

8.1.4 Activation of roles

In another extension of RBAC, [69], a subject can use a permission associated with a role if the subject is active in the role. This extension adds to RBAC the dynamic aspect.

An RBAC *active role* is a role which is *authorized* for a given subject and which was *set* to be used at the beginning of a session. As such a session specifies on the one hand the actual agent(s) (human users) that shall act as subject(s) and on the other hand the set of active roles (i.e. authorized roles that are used in the given session). Note that in this setting the activation is centralized, that is active users are defined with respect to a given service, and predetermined in the initial state.

In our framework, we assume that users can act on a collection of services and that they can choose to intervene (become active) at anytime in the course of one session. We argue in favor of such modeling for several reasons.

- First we assume that services are in a distributed environment, i.e. they do not have access to the entire security policy of the model and thus cannot decide alone on the activation of users.

- Second, we suppose that users can "step into" the session at any time, and thus activation permission should be available independently of the local state of a given entity but rather depend on the state of the system as a whole.

The user's certificates may be used in several services within the same session according to the access control policy of the entities.

We say that a user is active in a role, within an RBAC structure if an activation request (sent by the user to the entity holding the given role) is accepted and an activation certificate can thus be delivered. To this end we present in this section special objects to express role activation.

The activation and deactivation objects

We suppose that role activation can be done in a decentralized manner by sending a request to the entity responsible for the role to be activated. The right to activate is given by an object of the form:

$$\textbf{subject}\ u\ \texttt{can}-\texttt{activate}\ \textbf{role}\ r$$

Similarly, the right to deactivate a role is given by an object of the form:

$$\textbf{subject}\ u\ \texttt{can}-\texttt{deactivate}\ \textbf{role}\ r$$

As in the case of delegation and task execution, when a role is activated, an activation object records the activation. It has the form

$$\textbf{subject}\ u\ \texttt{activated}\ \textbf{role}\ r$$

Giving the right to activate a role

We assume that a user can activate a role if he is a member of that role and has not activated it yet. This is expressed by the following trust negotiation rule:

$$X := \textbf{subject}\ u\ \texttt{can}-\texttt{activate}\ \textbf{role}\ r$$
$$Y := \textbf{subject}\ u\ \texttt{is}-\texttt{member}\ \textbf{role}\ r$$
$$Z := \textbf{subject}\ u\ \texttt{activated}\ \textbf{role}\ r$$
$$put(X, self) \leftarrow get(Y, self) \wedge not(has(Z))$$

Activating and deactivating a role

A request for a user u to activate a role r is authorized if the following access control policy is satisfied:

$$X := \textbf{subject}\ u\ \texttt{request}\ \textbf{role}\ r$$
$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{activate}\ \textbf{role}\ r$$
$$permit(X, activate) \leftarrow get(Y, self)$$

We assume that a user can be deactivated from a role by another user as in [14]. A user v can deactivate a user u from role r if v has an object certifying

he can deactivate u from r. This is expressed by an access control rule of the form:

$$X := \textbf{revoker}\ v\ \texttt{request}\ \textbf{subject}\ u\ \textbf{role}\ r$$
$$Y := \textbf{subject}\ v\ \texttt{can}-\texttt{deactivate}\ \textbf{role}\ r$$
$$Z := \textbf{subject}\ u\ \texttt{activated}\ \textbf{role}\ r$$
$$permit(X, deactivate) \leftarrow get(Y, self) \wedge get(Z, self)$$

Effects of activation and deactivation

The process definition for the task *activate* is given as follows:

$$permit(X, activate) \rightarrow (\nu Y; Y.\textbf{subject} := u; Y.\textbf{action} := \texttt{activated};$$
$$Y.\textbf{role} := r; add(Y); rcv(Z, x, deactivate); permit(Z, deactivate); rmv(Y))$$
$$||\nu X'; permit(X', activate)$$

As in the case of delegation (see Chapter 6), we add the deactivation process in the definition of the activation process in order to make sure that the same activation instance will be deactivated. A user can always deactivate a role by sending a deactivation request and if authorized, the activation certificate will be removed. The deactivation process is a simple guard and is defined as follows:

$$permit(X, deactivate) \rightarrow skip$$

User-permission relation

If we take into account role activation, we assume that the acquisition of new permissions is related to role activation rather than role membership. To this end, we modify the rule presented in Section 8.1.1 to take into account role activation rather than role membership.

We say a user u is active in a role r and denote it by the action $\texttt{can}-\texttt{use}$ if u activated r. This is expressed by the trust negotiation rule:

$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{role}\ r$$
$$Z := \textbf{role}\ u\ \texttt{activated}\ \textbf{role}\ r$$
$$put(X, x) \leftarrow get(Y, self) \wedge has(Z)$$

Namely, a user has the permission for a given task τ if he is an active member of a role assigned to τ. This can be expressed by the following trust negotiation rule

$$X := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{permission}\ \tau$$
$$Y := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{role}\ r$$
$$Z := \textbf{role}\ r\ \texttt{is}-\texttt{assigned}\ \textbf{task}\ \tau$$
$$put(X, x) \leftarrow get(Y, self) \wedge has(Z)$$

Example 8.1.3. Reconsider Example 8.1.2 and suppose that Bob is not active in role clerk, and that Mary decides to activate her role clerk.

role *clerk* `is − assigned` **task** *read*
role *clerk* `is − assigned` **task** *store*
subject *bob* `is − member` **role** *clerk*
subject *manager* `is − senior` **role** *clerk*
subject *mary* `is − member` **role** *manager*

The trust negotiation policy of e_{rbac} contains the following rules:

(Active) $X :=$ **subject** u `can − use` **role** r
$Y :=$ **subject** u `activated` **role** r
$put(X, self) \leftarrow get(Y, self)$

(CanUse) $X :=$ **subject** u `can − use` **task** τ
$Y :=$ **subject** u `can − use` **role** r
$Z :=$ **role** r `is − assigned` **task** τ
$put(X, x) \leftarrow has(Y) \wedge has(Z)$

(Inherit) $X :=$ **subject** u `is − member` **role** r'
$Y :=$ **subject** u `is − member` **role** r
$Z :=$ **subject** r `is − senior` **role** r'
$put(X, self) \leftarrow get(Y, self) \wedge get(Z, self)$

(CanAct) $X :=$ **subject** u `can − activate` **role** r
$Y :=$ **subject** u `is − member` **role** r
$Z :=$ **subject** u `activated` **role** r
$put(X, self) \leftarrow get(Y, self) \wedge not(has(Z))$

The access control policy of e_{rbac} contains the following rules:

(pActivate) $X :=$ **subject** u `request` **role** r
$Y :=$ **subject** u `can − activate` **role** r
$permit(X, activate) \leftarrow get(Y, self)$

(pRevoke) $X :=$ **revoker** v `request` **subject** u **role** r
$Y :=$ **subject** v `can − deactivate` **role** r
$Z :=$ **subject** u `activated` **role** r
$permit(X, deactivate) \leftarrow get(Y, self) \wedge get(Z, self)$

and the workflow of e_{rbac} contains the process definitions:

$permit(X, activate) \rightarrow (\nu Y; Y.\mathbf{subject} := u; Y.\mathbf{action} :=$ `activated`$;$
$Y.\mathbf{role} := r; add(Y); rcv(Z, x, deactivate); permit(Z, deactivate); rmv(Y)$
$||\nu X'; permit(X', activate)$

$permit(X, deactivate) \rightarrow skip$

and the workflow

$$(\nu X; permit(X, activate))!$$

With these new settings we can have the following observations:

- By rule (Inherit) the entity e_{rbac} can *put* the object

$$\textbf{subject}\ mary\ \texttt{is-member}\ \textbf{role}\ clerk$$

- By rule (CanAct) the entity e_{rbac} can *put* the object

$$\textbf{subject}\ mary\ \texttt{can-activate}\ \textbf{role}\ clerk$$

- thus rule (pActivate) is satisfied and *activate* is executable

- By process definition of task *activate* the object

$$\textbf{subject}\ mary\ \texttt{activated}\ \textbf{role}\ clerk$$

is added to the repository of e_{rbac} (and the process waits for the reception of a request message to deactivate the role).

- By rule (Active) the entity e_{rbac} can *put* the object

$$\textbf{subject}\ mary\ \texttt{can-use}\ \textbf{role}\ clerk$$

- By rule (canUse) the entity e_{rbac} can *put* the objects

$$\textbf{subject}\ mary\ \texttt{can-use}\ \textbf{task}\ read$$
$$\textbf{subject}\ mary\ \texttt{can-use}\ \textbf{task}\ store$$

8.2 Delegation in RBAC

In the previous section we provided a representation of role features in our attribute based access control framework. In this section we express delegation in the presence of a role hierarchy. Note that if we suppose that roles are flat[1] then the same rules defined in Section 6.5 apply in the case of a role delegation. However in the presence of role hierarchy, the delegation is propagated to the junior roles in the hierarchy. That is if a delegatee acquires a delegation on a role r, then he also inherits the roles junior to r in the hierarchy. We say a subject u can use a role r if one of the following three conditions holds:

- he is a member of r (directly or through role inheritance.);

- he received the delegation for role r;

- he received the delegation for a role r' senior to the role r.

[1] Flat roles are roles that are not associated to a role hierarchy

We suppose that one can get an object certifying a role by the same trust negotiation rule defined in Section 6.5. In addition, we take into account the role hierarchy and role memberships by adding the following trust negotiation rule:

$X :=$ **subject** u `can − use` **role** r
$Y :=$ **subject** u `is − member` **role** r
$Z :=$ **subject** u `can − use` **role** r'
$W :=$ **subject** r' `is − senior` **role** r'
$put(X, x) \leftarrow get(X, self) \vee get(Y, x) \vee (get(Z, x') \wedge get(W, x')$

In this case if a user loses the right to use a role, then he loses the rights for all associated sub-roles.

Partial delegation for roles In Section 6.1 we presented some delegation models that take into account partial delegation. In this section we give our proposition to express partial delegation for roles by delegating permissions associated to a given role individually, without delegating the role membership. This is expressed by a trust negotiation rule as follows:

$X :=$ **subject** u `can − use` **task** τ
$Y :=$ **subject** v `delegated` **delegatee** u **object** r **nature** $partial - role$
$Z :=$ **subject** r `is − assigned` **task** τ
$put(X, x) \leftarrow get(Y, x) \wedge get(Z, x)$

8.3 Static and dynamic separation of duty

As presented in Section 7.1.1, different aspects of separation of duty exist in the literature. Static separation of duty guarantees that no user can be authorized for more than one role from a set of mutually exclusive roles whereas dynamic separation of duty constraints ensures that even though one user may be authorized for more than one mutually exclusive roles, he cannot be active in more than one role in the same session. In our framework, given that users are acting in different entities, it is important to guarantee and verify that the access control policies are safe with respect to the separation of duty constraints specified by the organization.

Static separation of duty (SSOD) requires the setting of mutually exclusive roles. This is not possible in a distributed environment since we do not have control on the users credentials prior to an execution of the model. We can however model such a property by centralizing the use-role membership. Namely, let $\mathcal{R}$ be a set of mutually exclusive roles, then we set a rule of the form:

$X :=$ **subject** u `is − member` **role** r
$Y :=$ **subject** u `is − member` **role** r'
$put(X, x) \leftarrow get(X, self) \bigwedge_{r' \in \mathcal{R}, r' \neq r} not(has(Y))$

Dynamic separation of duty (DSOD) is more interesting in our setting. In fact DSOD says that given a set of mutually exclusive roles $\mathcal{R}$, a user can have membership for more than one of these roles but cannot be active in more than one role at a time. Again this property can be expressed in a decentralized manner through the use of execution records. Suppose, role r and role r' are mutually exclusive, and suppose that entity e_i is responsible for role r whereas entity e_j is responsible for role r'. In order to ensure that the dynamic separation of duty constraint is preserved in both entity, we suppose the existence of a central authority to which we give the power to activate users in roles and add a restriction on role activation as follows:

$$
\begin{array}{l}
X := \textbf{subject}\ u\ \texttt{can}-\texttt{use}\ \textbf{role}\ r \\
Y := \textbf{subject}\ u\ \texttt{activated}\ \textbf{role}\ r \\
Z := \textbf{subject}\ u\ \texttt{activated}\ \textbf{role}\ r' \\
put(X, x) \leftarrow get(X, j) \wedge not(has(Y)) \wedge \bigwedge_{r' \in \mathcal{R}, r' \neq r} not(has(Z))
\end{array}
$$

Note that this presupposes the collaboration of all entities with this central authority by sending the activation objects and requesting activation authorization.

8.4 Conclusion

In this chapter we presented our modeling of the RBAC structure with its different extensions. We also presented the expression of delegation and separation of duty in the presence of role hierarchy. We are aware that the RBAC structure can and was modeled more efficiently with optimized tools. However, our aim in this chapter was to show that with our attribute based framework it is possible to give a distributed representation of the RBAC structure without modifying the syntax of the language. This chapter can be viewed as an example of the expressivity of the language. Though we have provided one possible modeling of the different RBAC properties, we believe that other understandings of role activation, role inheritance or even role delegation, can be encoded as well. The exact adaptation is left to the discretion of the policy modeler.

Chapter 9

An intruder model for trust negotiation

9.1 Introduction

Security has been thoroughly studied in the literature, be it on the level of defining security problems, providing solutions and protocol implementation or model checking techniques for testing the accuracy of such protocols. The atomic elements in communication are *communication channels*. A channel is the mean of transportation of a message from the sender (the source of the channel) to the receiver (the destination of the channel). In order to preserve the security of communications *secure communication channels* are used along with cryptographic techniques, such as public/secret key system, digital signature,... In general, we say a channel satisfies confidentiality if its output is exclusively accessible to the specified receiver and a channel satisfies authenticity if its input is exclusively accessible to a specified sender. Usually principals communicating on a given channel are aware of the type of security it offers. Securing communication channels can be done via *cryptographic primitives* as in the case of the Transport Layer Security protocol (TLS) [32], or via the communication of assertions according to a security policy as in the case of the Security Assertion Markup Language (SAML) [65].

A *security protocol* is an algorithm or a series of operations that has as objective to secure the communication between the agents in the presence of an intruder. The aim of such protocols is to achieve certain security properties, such as authenticity or secrecy. That is at the end of the protocol execution, the initiator of the protocol should becomes aware of the true identity of the responder or that at the end of the protocol, no secret has been learned by an unauthorized principal.

The TLS protocol is a communication-oriented protocol that provides a secure channel between a client and a server. TLS supports confidentiality, data integrity and client/server authentication. The authentication is provided by a

handshake protocol which is a sequence of alternating request-response messages between the client and the server. During a handshake protocol, the server provides a certificate in order to be authenticated with respect to the client in a typical end-user communication. However, it is also possible to ensure mutual authentication, in which case the server also requests a digital certificate from the client. The confidentiality is ensured through a *record protocol* that specifies how data should be transmitted.

SAML on the other hand is an XML standard for exchanging authentication and authorization data between security domains. SAML provides a framework to define how a security identity can be obtained and transferred from one entity to another. SAML is made up of three constituent components; the assertions that provide information on the identity of the user, but also on different attributes concerning the user; the protocol consisting of a sequence of send/receive of assertions; and the binding that defines a mapping from the SAML message exchange to the Simple Object access protocol (SOAP) [58]. Note however that SAML does not directly provide message integrity or confidentiality; it relies on XML Signature to protect integrity and on SSL/TLS for confidentiality. The SAML Single Sign On (SSO) can assert authorization across multiple services allowing thus a user to log on once for these affiliated but separate services. Several forms of security properties exist in literature. In [55] Lowe provides a hierarchy for the specification of authentication properties and give examples of protocols satisfying (or not) such properties. For example, one of the weakest form of authentication is *aliveness* where whenever a principal A acting as an initiator completes a run of a protocol, apparently with responder B, then B has previously been running the protocol. Note that this property is weak since it only guarantees that both A and B are active as initiator and responder but does not guarantee the interference of the intruder between different runs of the protocol. In one of the strongest forms of authentication, *agreement*, the initiator A agrees with the responder B on a set of data items ds if whenever A acting as an initiator completes a run of the protocol apparently with responder B, then B was acting as a responder in his run and the two principals agreed on the set of data items ds, and each run of A corresponds to a unique run of B. In this stronger version the principals A and B are aware of the presence of one another and agree on the essential features of a protocol run.

In [57] the authors describe the process of establishing security in a distributed system as a two-phase process. First, agreement on the shared secret keys and public keys through a secure communication is done during the *initialization phase*. Then during the *communication phase* communication can be done over insecure channels (that shall be made secure via the use of cryptographic techniques). They give a classification of security properties for channels and interpret cryptographic primitives as transformations for channel security properties. In their view a protocol will then be used to transform a set of secure channels established during the *initialization phase* together with the set of insecure channels available during the operation of the system into the set of secure channels specified by the security requirements.

In fact, the discovery of serious attacks on well known security protocols made it necessary to provide formal ways of reasoning in order to check wether a given protocol meets its security goals. Burrows, Abadi and Needham [26] provide a logic for authentication to express the beliefs of principals during an authentication protocol and how to derive them. Their logic allows to reason on an abstract level to check whether the protocol does exactly what it has to do. They focus on the beliefs of the trustworthy parties involved in the protocols and on the evolution of these beliefs as a consequence of the communication.

In [4] the authors present a general model for security protocols based on a set rewriting formalism and the Linear Temporal Logic (LTL). They consider a model M as a labeled transition system modeling the behavior of honest agents and the intruder, and place constraints in LTL specifying C_I the allowed behavior of the intruder on the one hand and C_H the allowed behavior of honest entities on the other hand. Also they specify security properties, that is the goals to be satisfied by the protocols as LTL formulas G. The model checking problem amounts to testing

$$M \models (C_I \wedge C_H) \Rightarrow G.$$

This framework allowed the discovery of an attack on Google Single Sign On (SSO) application [5] by modeling the SAML specification of the single sign on application using the set of rewriting rules and the LTL constraints in order to specify the behavior of the honest principals and the intruder and tested their model against the security goals stated by the application protocol.

Contribution

We have presented so far a logical framework for reasoning about policies in a distributed environment by means of negotiation of certificates. We assume in the environment the existence of entities, each of which has a set of objects. A trust negotiation policy regulates the access of other entities to these objects. Given the set of trust negotiation policies of the entities in the environment, the set of objects that can be sent to some entity e is found through the computation of a least fixed point. While one usually considers only entities abiding by their trust negotiation policies, we take into account in this chapter the presence of a malicious entity.

In [33] Dolev and Yao have introduced a notion of symbolic intruder to represent the capacities of a malicious agent trying to attack a cryptographically secured communication protocol. We present in this chapter an adaptation of that intruder that retains the same deductive capabilities but is specialized for the analysis of the exchanges during a trust negotiation session. In particular this permits us to analyze the security of a distributed access control policy *w.r.t.* a malicious insider.

We extend the usual setting with channels between entities. Certificates are sent over these channels, which are specified by their properties with respect to the intruder. The malicious entity acts like a Dolev-Yao intruder [33]: it can intercept any object sent on a public channel, and can also replace it with

an object constructed from its initial knowledge and parts of objects sent by other entities in the same or in other sessions. A malicious entity can also block access to some objects for other entities depending on the nature of the communication. Also the malicious entity can read information in objects and create new objects. We assume that to construct a new object, the intruder can compose, decompose, encrypt and decrypt messages in case he knows the keys.

As presented in Chapter 4, the framework we consider is based on *Automated Trust Negotiation* [78], from which we have borrowed the eager strategy. Considering secured transport protocols or cryptographic primitives is not new [42, 79, 51] but these works do not consider an active malicious insider. Our notion of *authenticity* originates from [55], while the notion of confidentiality is the classic *weak secrecy* one.

In our model, a protocol is defined by the set of trust negotiation rules defined in the security policy of the entities. We model the intruder by defining a set of trust negotiation rules giving him a maximal control over the communications during a trust negotiation session. We define the actions of the intruder both on the level of his ability to send and receive negotiated objects on behalf of others, and his ability to read, sign and create objects based on intercepted ones.

Our goal is to take into account the properties (such as confidentiality, authenticity, etc...) of these channels as well as the cryptographic primitives employed to secure the message, and to assess a trust negotiation infrastructure.

We present the basics of the language in Section 9.2, and our modeling of secure channels in Section 9.4. We introduce our intruder model in Section 9.5, and the adaptation of the trust negotiation algorithm to take it into account in Section 9.6. We present in Section 9.7 the security properties we analyze and in Section 9.8 we give our modeling of Google's SAML Single-Sign On implementation.

9.2 Syntax

In this section we define the syntax that shall be used to represent a trust negotiation infrastructure. This is an extension of the syntax presented in Chapter 4 to take into account cryptographic primitives. For the sake of clarity of this chapter, we recall some definitions presented in Section 4.3.

Attributes, values and objects

As presented in Section 4.3, Val is a countable set of values (with typical members denoted v, v', ...), and Att is a finite set of attributes (with typical members denoted $\mathbf{att}$, $\mathbf{att'}$, ...). An object is defined as a partial function:

$$O : Att \rightarrow Val$$

and the set of all objects is denoted by $\mathcal{O}$. For each object O, let $dom(O)$ be the domain of O. We define the *empty object* ε such that $dom(\varepsilon) = \emptyset$.

Keys and signed objects

Let $\mathcal{K}$ be a finite set of public and private keys (with typical members denoted $k,\ k', k^{-1}, k'^{-1},\ \ldots$). We assume that $Att \cap \mathcal{K} = \emptyset$.

In order to simplify notations we also consider in this chapter only the case of cryptographic operations with public and private keys. We model the encryption and signature operations with a single binary symbol f.

We let $VarObj$ be a countably infinite set of variables called *variables for objects* (with typical members denoted $X,\ Y,\ \ldots$). The set of *signed objects* is denoted $\varphi(\mathcal{O})$ and is the smallest algebra that contains $\mathcal{O} \cup VarObj$ and such that, if $O \in \varphi(\mathcal{O})$ and $k, k^{-1} \in \mathcal{K}$ then:

- $f(O,k)$ is in $\varphi(\mathcal{O})$ and denotes either the encryption of O with the encryption key k or the verification of the signature O with the validation key k;
- $f(O,k^{-1})$ is in $\varphi(\mathcal{O})$ and denotes either the signature of O with the signature key k^{-1} or the decryption of O with decryption key k^{-1}.

For soundness of analysis we assume that in the model we consider there are disjoint sets of key pairs for encryption and signature, and that the nature of an operation (*e.g.* encryption or validation) is clear given the key employed. The distinction between these two kinds of key pairs is however irrelevant in terms of possible symbolic operations, hence our unique set $\mathcal{K}$ of keys and unique symbol f.

In order to model the relative properties of decryption *w.r.t.* encryption and validation *w.r.t.* signature we consider the algebra $\varphi(\mathcal{O})$ *modulo* the following two equations:

$$\begin{cases} f(f(O,k),k^{-1}) = O \\ f(f(O,k^{-1}),k) = O \end{cases}$$

It is straightforward to see that these equations define a convergent rewriting systems on the algebra $\varphi(\mathcal{O})$, and thus that signed objects can be put in a *normal form* in which none of the above equations can be applied from left to right. In the rest of this chapter we consider that all signed objects without variables are in normal form. A signed object without variables is *clear* iff it is not of the form $f(T,k)$ or $f(T,k^{-1})$ for some signed object without variables T. The notation $O.a$ is defined only on clear objects.

Operations on objects

Object update Given two objects O_1 and O_2 we call the *update of* O_1 *by* O_2, and denote $O_1 \leftarrow\!\!+\, O_2$ the object O of domain $dom(O_1) \cup dom(O_2)$ such that for $x \in dom(O)$ we have $O(x) = O_2(x)$ if $x \in dom(O_2)$ and $O(x) = O_1(x)$ otherwise.

Object domain restriction We define the domain restriction operation $\setminus$ that, given an object O and a set of attributes A, computes $O \setminus A$, the restriction of O to $dom(O) \setminus A$.

These operations are mainly specific to the intruder, and express the different operations that an entity can perform on a given object.

Interpretations

An interpretation function for variables is a function I that associates to every variable X for objects an element $I(X)$ in $\varphi(\mathcal{O})$ and that associates to every variable x for values an element $I(x)$ in VAL.

Also, we define the interpretation on signed objects by: $I(O) = O, I(f(T,k)) = f(I(T),k)$, $I(f(T,k^{-1})) = f(I(T),k^{-1})$, and the interpretation on terms for values by: $I(v) = v$ and $I(X.a) = I(X).a$[1].

Entities and states

As presented in Section 4.3, an *entity* e_t is a set of trust negotiation rules (defined below). To each entity e_t we associate a value $t \in VAL$ which is its unique identifier. To each entity e_t we associate a *security state* s_t consisting of a *repository*, *i.e.* a set of objects, and an interpretation function for variables. We denote by $\mathcal{E}$ a finite set of entities. The set of security states for entities in $\mathcal{E}$ is denoted by $\mathcal{S}$.

9.3 Trust negotiation policy

A trust negotiation policy consists of a set of trust negotiation rules. A trust negotiation rule gives the conditions to be satisfied in order to send (*put*) an object during a trust negotiation session. We extend the definition of trust negotiation predicate *put* in Section 4.4 to take into account an additional argument. A trust negotiation rule is thus defined as follows:

$$put(T, t_1, t_2) \leftarrow body$$

where T is a signed object, t_1 is a term for values representing the apparent sender and t_2 is a term for values representing the intended receiver. In the case where the sender is a honest entity, the apparent sender is the entity's own identifer. However, this is not always the case when one considers a malicious entity.

The atom $put(T, t_1, t_2)$ models the disclosure of the signed object T to an entity t_2 (as being sent by t_1) whenever the conditions in the body of the rule are satisfied.

[1]This expression is defined only if $I(X)$ is a clear object

Body of rules

The body of the security rules is defined by the grammar:

$$body := \top \mid Test \mid ObjOp \mid body \wedge body \mid body \vee body$$

$$Test := has(T) \mid get(T, t) \mid t_1 = t_2 \mid t_1 \neq t_2$$

where T is a signed objects and t, t_1, t_2 are terms for values.

The intended meaning of these expressions is presented in Section 4.4.

Trust policy of honest entities vs trust policy of malicious entities

In a perfect world, entities only send messages using their own identity, that is the apparent sender is the same as the real sender. However the presence of a malicious entity, the intruder, induces forged trust negotiation objects sent by the intruder impersonating a honest entity. Let e_i, $i \in \{1, ldots, n\}$ be honest entities and let e_{int} be the intruder.

Trust negotiation rules for honest entities e_i, $i \in \{1, ldots, n\}$ are of the form:

$$put(T, i, j) \leftarrow body$$

for $j \in \{1, \ldots, n, int\}$.

Trust negotiation rules for the intruder e_{int} are of the form

$$put(T, i, j) \leftarrow body$$

for $i \in \{1, \ldots, n, int\}$ and $j \in \{1, \ldots, n, int\}$.

That is, the intruder sends messages using his own identity (when $i = int$), or using the fake identity of a honest entity (when $i \neq int$). This latter case is only possible if we assume that the communication with entity e_j is not done on an authentic channel. The restriction according to the different types of communication will be discussed in Section 9.4.

A running example

In [5] the authors describe an attack on Google's implementation of SAML Single Sign-On protocol that allows a dishonest service provider to impersonate a user at another service provider in the case of Google Applications. In this scenario we assume the presence of a client BOB, an identity provider IDP and an intruder acting as a service provider SP.

Throughout this chapter we extract examples from this scenario in order to describe the communication of "messages" between these different entities. We start with a usual trust negotiation modeling and then add new elements to our framework in order to be able to express the communication using different types of channels along with giving an intruder model to test the safety conditions of the access control policy. We present the complete modeling in Section 9.8.

Example 9.3.1. In our running example there are three honest entities e_{user}, e_{sp} and e_{idp} denoting respectively a client, a service provider and an identity provider. A fourth entity e_{int} is malicious.

The service provider (e_{sp}) sends a request to identify the client (e_{user}) to the identity provider (e_{idp}) via the client. The entity e_{idp}, upon reception of the request from e_{user}, will send a response containing the authentication of the client to e_{user} who in turn forwards it to e_{sp} in order to be granted access. This is expressed by the following rules:

In e_{user} we have the following rules:

$$put(X, user, sp) \leftarrow get(X, idp) \wedge X.respondent = idp$$

$$put(X, user, idp) \leftarrow get(X, sp) \wedge X.respondent = sp$$

In e_{sp} we have the following rules:

$X :=$ **resource** y **access** $granted$
$RES :=$ **certifier** idp `says` **auth** $-$ **status** $is - authentic$ **resource** y
$put(X, sp, user) \leftarrow get(RES, user)$

In e_{idp} we have the following rules:

$REQ :=$ **subject** sp **requestId** id_{sp} **respondent** idp **resource** y
$RES :=$ **subject** $user$ **auth** $-$ **status** $is - authentic$ **certifier** idp
resource y **respondent** sp
$put(RES, idp, user) \leftarrow get(REQ, user)$

Note that in the case of a non-confidential communication for e_{user}, an intruder can *get* the object X intended to e_{user} sent by the e_{idp} and in the case of a non-authentic communication from the intruder to the service provider e_{sp}, the intruder can fake being e_{user} by sending a request to access e_{sp}. In the next section we present a modeling of the properties provided by such channels in order to be able to model the behavior of the intruder in such different cases.

9.4 The intruder in the presence of secured communication

We assume that the exchange of objects within a trust negotiation session is done via communication channels between the communicating entities. These channels satisfy a certain level of security provided by the transport protocol such as authenticity or confidentiality. In this chapter we abstract away the actual protocols and consider only the security properties they guarantee, among authenticity, confidentiality, and unblockability. In this section we model these aspects with respect to the objects exchange during the trust negotiation mechanism and to the intruder.

9.4.1 Security properties of channels

Suppose two entities e_s and e_r want to exchange objects in a trust negotiation session. We wish to specify security properties for such a communication depending on the communication channels used by these entities in order to send/receive an object O. A channel is confidential if its output is exclusively accessible to a specific receiver, it is authentic if its input is exclusively accessible to a specific sender, and it is *unblockable* if the intruder cannot prevent a message sent on this channel to reach its destination. Otherwise the channel is said to be *blockable*.

For example, if an entity e_r receives the object O on an *authentic* channel then e_r will know who is the real sender of the object O. On the other hand if e_s sends O on a *confidential* channel then e_s will know that only the intended receiver will receive the object. Also two entities living on the same computer system will employ unblockable channels if the intruder does not have access to that computer system.

9.4.2 Extending the syntax

We extend the definition of the *put* and *get* predicates to take into account the different kinds of communication by indexing it with a set of tags $A \subseteq \{auth, conf, block\}$.

- If $auth \in A$, then $put_A(T, t_1, t_2)$ denotes that the object T is sent on behalf of the apparent sender e_{t_1} to the entity e_{t_2} through a communication channel satisfying the authenticity property for the sending entity. Note that in such a case e_{t_1} is also the real sender of T.

- If $conf \in A$ then $put_A(T, t_1, t_2)$ denotes that the object T is sent on behalf of the apparent sender e_{t_1} to the entity e_{t_2} through a communication channel satisfying the confidentiality property for e_{t_2}. In such a case e_{t_2} is the unique receiver of that object.

- If $unblock \in A$ then $put_A(T, t_1, t_2)$ denotes that the object T is sent on behalf of the apparent sender e_{t_1} to the entity e_{t_2} through a communication channel on which objects cannot be blocked by the intruder.

$get_A(T, t)$ is defined in the same manner. Note that the *indexing* of the *put* and *get* predicates by the set A of tags allows the specification of the communication types in our framework, as will be explained in Section 9.6.

Available objects

We define an *available object* to be a quadruplet $(O, rs, s, r)_A$ where $O \in \varphi(\mathcal{O})$ is an object and rs, s and $r \in VAL$ are values denoting the real sender, the apparent sender, and the intended receiver respectively and A is a set of tags from $\{auth, conf, unblock\}$. For example, if $auth \in A$ then $(M, s, s, r)_A$ denotes

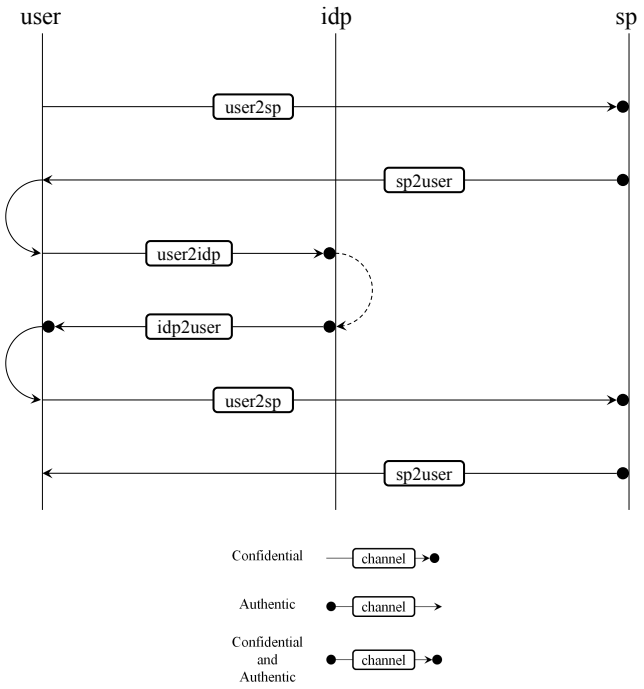


Figure 9.1: Communication of messages on secured channels for SAML SSO

an available object sent on an authentic communication channel between e_s and e_r.

Example 9.4.1. In Google's implementation of SAML SSO, the certificates are sent by the client over a confidential channel to the service provider, and they are accepted by the client only on a confidential and authentic channel from the identity provider. The certificates are sent by the identity provider only on confidential and authentic channels, and they are accepted by the identity provider on confidential channels. Finally, certificates are sent by the service provider on authentic channels and accepted on confidential channels (see Figure 9.1). Also we suppose that messages partially encrypted are split into two object messages, one signed and the other clear.

To reflect this policy the rule of Example 9.3.1 is now written:

In e_{user} we have the following rules:

$X :=$ **subject** $user$ **respondent** sp **resource** y
$put_{\{conf\}}(X, user, sp) \leftarrow has(X)$

$put_{\{conf\}}(X, user, sp) \leftarrow get_{\{auth,conf\}}(X, idp) \wedge X.\textbf{respondent} = idp$

$put_{\{conf\}}(X, user, idp) \leftarrow get_{\{auth\}}(X, sp) \wedge X.\textbf{respondent} = sp$

In e_{sp} we have the following rules:

$REQ :=$ **subject** sp **requestId** id_{sp} **respondent** idp **resource** y
$X :=$ **subject** $user$ **respondent** sp **resource** y
$put(REQ, sp, user) \leftarrow get(X, user)$

$X :=$ **resource** y **access** $granted$
$RES :=$ **certifier** idp says **auth** $-$ **status** $is - authentic$
$RES_1 :=$ **resource** y
$put_{\{auth\}}(X, sp, user) \leftarrow get_{\{conf\}}(f(RES, k_{idp}^{-1}), user) \wedge get_{\{conf\}}(RES1, user)$

In e_{idp} we have the following rules:

$RES :=$ **certifier** idp says **auth** $-$ **status** $is - authentic$ **subject** $user$
respondent sp
$REQ :=$ **subject** sp **requestId** id_{sp} **respondent** idp **resource** y
$put_{\{auth,conf\}}(RES, idp, idp) \leftarrow get_{\{conf\}}(REQ, user)$

$put_{\{auth,conf\}}(f(RES, k_{idp}^{-1}), idp, user) \leftarrow get_{\{auth,conf\}}(RES, idp)$
$\wedge RES.\textbf{subject} = user$

$RES1 :=$ **respondent** sp **resource** $yREQ :=$ **subject** sp **requestId** id_{sp}
respondent idp **resource** y
$put_{\{auth,conf\}}(RES1, idp, user) \leftarrow get_{\{conf\}}(REQ, user)$

In e_{int}

$$put_{\{conf\}}(X, i, j) \leftarrow get_A(X, k)$$

where $i, j, k \in \{user, sp, idp\}$ and $auth \notin A$.

It is easy to see that the intruder can intercept the messages sent by e_{user} and use them to construct new messages and communicate with the service provider in the name of the honest client e_{user}. In the following section we present the actions that can be done by the intruder on objects that he knows or gained access of on non-confidential channels.

9.5 Formalizing the intruder

In this section we describe the behavior of the intruder with respect to secure objects. That is, we assume that the intruder can encrypt, decrypt, sign or create objects depending on his knowledge. Note that the *knowledge* of the intruder consists of objects that the intruder either has in his repository at the beginning of the trust negotiation session, or has acquired (legitimately or by listening to the network) during the previous rounds of negotiation.

9.5.1 Entities and keys

As noted in Section 9.2, we suppose that each entity $e_i \in \mathcal{E}$ possesses key pair for encryption and a key pair for signature that we denote by k_i and k_i^{-1} respectively.

Also, we assume the existence in the repository of the intruder of objects that we call *knowledge objects*. A knowledge object is an object K whose domain contains a characteristic attribute kn such that $K.kn = v$ for some $v \in VAL$. The rationale behind defining such an object is to keep a trace of the attributes or values that the intruder gained knowledge of (through the reading of available objects he knows) and that he can use in order to create new objects.

In the rest of this section we model the behavior of the intruder with respect to the security properties of objects.

9.5.2 Trust negotiation policy of the intruder

In this section we present the actions that can be made by the intruder on an object available to him depending on whether the object is signed, encrypted or non-secured through the definition of trust negotiation rules.

Sending objects

The intruder e_{int} can send an object to an entity e_j in $\mathcal{E}$ pretending to be an entity e_i if he has the object in his repository or if he "gained" access to the object during the trust negotiation session. This is expressed by the following rule:

$$put_A(X, i, x_j) \leftarrow get_{A'}(X, x_l) \vee has(X)$$

for $x_j, x_l \in \{1, \ldots, n, int\}$ and with the assumption that either $auth \notin A$ or $i = int$.

In particular, we assume that the intruder *sends to himself* all the objects that he can acquire from other entities. The rules modeling the objects "gained" by the intruder can thus be expressed by:

$$put_A(X, int, int) \leftarrow get_{A'}(X, x_i)$$

for $x_i \in \{1, \ldots, n, int\}$

The intruder can always use the empty object ε to construct new objects:

$$put_A(\varepsilon, int, int) \leftarrow \top$$

Use of cryptographic operations

The intruder e_{int} can use cryptographic operation on an object he already gained access of. Let $\mathcal{K}_{int} = \{k_i : i \in \{1, \ldots n, int\}\} \cup \{k_{int}^{-1}\}$ be the set of keys available to the intruder. The intruder can use the public keys of all entities as well as his own private key. We express the behavior of the intruder as follows:

$$put_A(f(X,k), int, int) \leftarrow get_{A'}(X, int) \vee has(X)$$

for $k \in \mathcal{K}_{int}$.

The intruder can create a new object from objects that he can read, or from information that he *gained*.

Extracting information

The fact that the intruder can extract information from objects he knows is expressed by the rule:

$$put_\emptyset(X, int, int) \leftarrow get_\emptyset(Y, int) \wedge Y.a = x \wedge X = \epsilon \leftrightarrow (kn, x)$$

Object creation

The intruder can create new objects by adding or removing attributes from objects.

Adding attributes: The intruder can create a new object by adding an attribute that he already "knows" to an object he already gained access of, as follows:

$$put_\emptyset(X, int, int) \leftarrow get_\emptyset(Y, int) \wedge get_\emptyset(Z, int) \wedge \ Z.kn = x \wedge \ X = Y \leftrightarrow (a, x)$$

for $a \in ATT$

Removing attributes The intruder can create a new object by removing an attribute from objects he already gained access of, as follows:

$$put_A(X, i, j) \leftarrow get_{A'}(Y, l) \wedge X = Y \setminus \{a\}$$

for $a \in Att$, $i, j, l \in \{1, \ldots, n, int\}$ and with the assumption that either $auth \notin A$ or $i = int$.

9.6 Trust negotiation semantics

In this section we give the trust negotiation semantics for both honest entities and the intruder. Note that while honest entities are assumed to abide by their trust negotiation policy, the intruder can receive objects destined to others in a non-confidential communication and can send objects to others pretending to be someone else in a non-authentic communication.

9.6.1 How entities receive available objects

With the presence of an intruder, the real sender and real receiver of a negotiated object may differ from the intended one. We illustrate this by the evaluation semantics for the $put(T, t_1, t_2)$ and $get(T, t)$ predicates in the trust negotiation policy of the honest entities on the one hand and the intruder on the other hand.

Trust negotiation session

We define a trust negotiation session to be a sequence of trust negotiation rounds. A trust negotiation session terminates when there are no more objects that can be negotiated, in which case the least fixed point is reached. We model the eager trust negotiation. That is we assume honest entities send all the objects that satisfy their trust negotiation policy during a trust negotiation session. However we assume that the intruder can perform a non-deterministic choice on the objects that he actually sends from the set of objects that satisfy his trust negotiation policy.

Adequate available objects

We have assumed that honest entities did not impersonate other entities nor listened to communications directed to others. Informally we say that an available object abiding by these rules is adequate. Formally let $\mathcal{A} = (O, x, y, z, A)$ be an available object and assume $i \in \{1, \ldots, n, int\}$. We say that $\mathcal{A}$ is *adequate* for i and denote $ad(\mathcal{A}, i)$ iff:

$$\begin{cases} x \in \{y, int\} \;\; \& \;\; (auth \in A \Rightarrow x = y) & \text{(send restrictions)} \\ i \in \{z, int\} \;\; \& \;\; (conf \in A \Rightarrow z = i) & \text{(receive restrictions)} \end{cases}$$

Let us explain the role of the conditions:

- $x \in \{y, int\}$ and $i \in \{z, int\}$ enforce respectively the facts that only the intruder sends objects as someone else or listens to objects sent to others;

- $auth \in A \Rightarrow x = y$ enforces the fact that the real and apparent senders must be equal for an available object transported on an authentic channel;

- $conf \in A \Rightarrow y = i$ enforces the fact that an available object transported on a confidential channel can only be listened by the intended receiver.

Trust negotiation round

A trust negotiation round among n honest entities and the intruder is an $(n+1)$-tuple of sets of *available objects*

$$\Sigma_k = (\Sigma^k_{\mathcal{E},\mathcal{S},e_1}, \ldots, \Sigma^k_{\mathcal{E},\mathcal{S},e_n}, \Sigma^k_{\mathcal{E},\mathcal{S},e_{int}})$$

where $\Sigma^i_{\mathcal{E},\mathcal{S},e_j}$ denotes the set of *available objects* for the entity e_j at the beginning of round i. The tuples $(\Sigma_k)_{k\geq 0}$ are defined inductively as follows:

base case: $\Sigma^0_{\mathcal{E},\mathcal{S},e_1} = \cdots = \Sigma^0_{\mathcal{E},\mathcal{S},e_n} = \Sigma^0_{\mathcal{E},\mathcal{S},e_{int}} = \emptyset$;

induction: Let $i \in \{1, \ldots, n, int\}$ be an entity, $k \geq 0$. Suppose by induction that $\Sigma^k_{\mathcal{E},\mathcal{S},e_i}$ is defined. Furthermore assume that the set $\Omega^k_{\mathcal{E},\mathcal{S},e_i}$ of available objects acquired by i during the k-th trust negotiation step is defined.

We define $\Sigma^{k+1}_{\mathcal{E},\mathcal{S},e_i} = \Sigma^k_{\mathcal{E},\mathcal{S},e_i} \cup \Omega^k_{\mathcal{E},\mathcal{S},e_i}$.

In the remaining of this section we present how $\Omega^k_{\mathcal{E},\mathcal{S},e_i}$ is computed given Σ_k.

9.6.2 What entities can send

Let $\Sigma^k_{\mathcal{E},\mathcal{S},e_i}$ denote the set of objects available for an entity e_i at the beginning of round k. We say that a signed object $O \in \varphi(\mathcal{O})$ can be sent by entity e_i to entity e_j at step k, and write $\mathcal{S}, e_i \models_k put_A(O, i, j)$, iff $put_A(O, i, j) \leftarrow body$ is a ground instance of a rule in the trust negotiation policy of an entity e_l such that:

$$\mathcal{S}, e_l \models_k body$$

To evaluate this formula we say $\mathcal{S}, e_i \models_k body$ iff in the rule instance:

- *body* is $has(O')$ and $O' \in Rep_i$
- *body* is $t = t$
- *body* is $t \neq t'$ and t and t' are different
- *body* is $get_{A'}(O', j)$ and there exists $y, z \in \{1, \ldots, n, int\}$ and A'' such that $(O', y, j, z, A'') \in \Sigma^k_{\mathcal{E},\mathcal{S},e_i}$ and $A' \subseteq A''$ and $ad((O', y, j, z, A''), i)$.
- *body* is $body_1 \wedge body_2$ and $\mathcal{S}, e_i \models_k body_1$ and $\mathcal{S}, e_i \models_k body_2$
- *body* is $body_1 \vee body_2$ and $\mathcal{S}, e_i \models_k body_1$ or $\mathcal{S}, e_i \models_k body_2$.

The adequacy criterion (see Section 9.6.1) in the fourth point implies in particular that honest entities only receive objects sent to them ($i = z$) and send objects as being themselves ($y = j$). Also, if i is the intruder and $i \neq z$ then the communication channel must be non-confidential and if $y \neq j$ the communication channel must be non-authentic.

9.6.3 Computing the set of available objects

The set of available objects for entities computed at round k depends on the set of objects sent by honest entities, the set of objects sent by the intruder as himself or as another entity and the set of objects blocked by the intruder.

Let $\Pi^k_{\mathcal{E},\mathcal{S},e_i}$ be the maximal set of objects that are available to e_i taking the assumption that the intruder sends everything he can and blocks nothing. By definition we have:

$$\begin{aligned}\Pi^k_{\mathcal{E},\mathcal{S},e_i} \quad = \quad & \{(O,x,j,y,A) : \mathcal{S}, e_x \models_k put_A(O,j,y) \text{ with } x,j,y \in \{1,\ldots,n,int\}\,, \\ & \text{and } ad((O,x,j,y,A),i)\}\end{aligned}$$

Let $\Lambda^k_{\mathcal{E},\mathcal{S},e_i}$ be the minimal set of objects that are available to e_i taking the assumption that the intruder does not send anything and blocks all objects that can be blocked. By definition we have:

$$\Lambda^k_{\mathcal{E},\mathcal{S},e_i} = \{(O,j,j,i,A) \quad : \quad \mathcal{S}, e_x \models_k put_A(O,j,y), j \neq int \text{ and } unblock \in A\}$$

We define the set of available objects for an entity e_i at step k to be a set $\Omega^k_{\mathcal{E},\mathcal{S},e_i}$ chosen non-deterministically by the intruder and such that:

$$\Lambda^k_{\mathcal{E},\mathcal{S},e_i} \subseteq \Omega^k_{\mathcal{E},\mathcal{S},e_i} \subseteq \Pi^k_{\mathcal{E},\mathcal{S},e_i}$$

We note that a complete strategy (*w.r.t.* the search of an attack) for the intruder is to always choose *at least* all the messages intended to himself.

9.7 Security requirements

In this section we define some security requirements that should be satisfied by the model during a trust negotiation session. Recall that a trust negotiation session is the process of computing a least fixed point with respect to (i) the set of trust negotiation rules for each entity in the model and (ii) the non-deterministic choice of actions performed by the intruder, as presented in Section 9.6. In this section we present different aspects of confidentiality and authenticity properties.

9.7.1 Confidentiality

We distinguish two different kinds of confidentiality, the usual one for which a fixed group of entities shall not be able to obtain a clear object, and a *containment* one that expresses that an entity may possess an object O only if it has been sent to this entity by another honest entity.

Confidentiality. A first notion of confidentiality consists in imposing that an object O must be unknown to a coalition G of entities. This means that, when considering the union $\mathcal{K}_G$ of the sets of keys known by entities in G, none of the entities in G can gain access to a signed object T that can, by performing certain encryption or decryption operations with keys in $\mathcal{K}_G$, retrieve the object O. In order to formalize this notion, let $\mathcal{K}_i$ be the set of available keys for entity e_i. We say that an object O is confidential for the group G if and only if

$$\forall T, \forall k_1, \ldots, k_n \in \mathcal{K}_G, \\ f(f(\ldots f(T, k_1), \ldots), k_n) = O \Rightarrow \forall A, \forall i \in G, \forall m > 0, (\mathcal{E}, \mathcal{S}, e_j \not\models_m get_A(T, i))$$

Containment. An object O is *contained* if no entity can gain access to this object unless it is the exclusive respondent. Note that while this is always the case for a honest entity, such a property is necessary to check whether an intruder can intercept such a message or construct it using his own knowledge. We say an object O preserves containment if and only if

$$\forall A, \forall i, \forall m > 0, \\ (\mathcal{E}, \mathcal{S}, e_{int} \models_m get_A(O, i) \Rightarrow \exists m' \leq m, \exists j \neq int, (\mathcal{E}, \mathcal{S}, e_j \models_{m'} put_A(O, j, int)))$$

9.7.2 Authenticity

Among the notions proposed in [55] we consider two types of authenticity.

Strong authenticity for an object O is expressed by the fact that at no time can the intruder send the object O as being someone else. This property is formally expressed as follows:

$$\forall A, \forall i, j, l \in \{1, \ldots, n, int\}, l \neq i, \forall m > 0, (\mathcal{E}, \mathcal{S}, e_l \not\models_m put_A(O, i, j))$$

Weak authenticity for an object O is expressed by the fact that if an entity e_i receives the object O apparently from entity e_j then the entity e_j previously sent the object O to the entity e_i.

We say an object O preserves weak authenticity if and only if

$$\forall A, \forall i, j \in \{1, \ldots, n, int\}, k > 0, \\ (\mathcal{E}, \mathcal{S}, e_i \models_k get_A(O, j)) \Rightarrow \exists m < k, (\mathcal{E}, \mathcal{S}, e_j \models_m put_A(O, j, i))$$

9.8 Attack on Google's implementation of SAML SSO

In this section we give an instance of the model defined throughout this chapter that presents a confidentiality attack on the Single Sign-on protocol for Google applications found by Armando et al. [5]. The communication of messages in the presence of the intruder are presented in Figure 9.2. Let $\mathcal{E} = \{e_{bob}, e_{google}, e_{idp}, e_i\}$ where e_{bob} is the client entity, and e_{google} is a service provider, then:

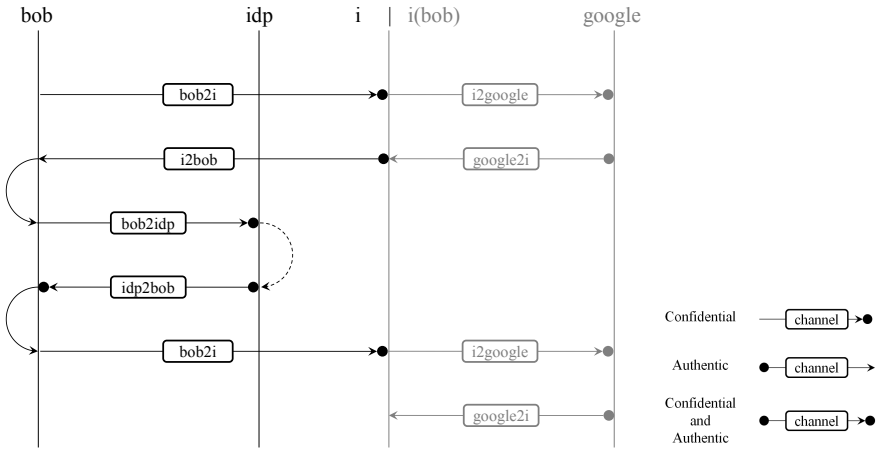


Figure 9.2: The communication of messages in the attack on SAML SSO for Google application

In e_{bob}:

$X := \textbf{subject}\ bob\ \textbf{respondent}\ i\ \textbf{resource}\ uri$
$put_{\{conf\}}(X, bob, i) \leftarrow has(X)$

$put_{\{conf\}}(X, bob, i) \leftarrow get_{\{auth,conf\}}(X, idp) \wedge X.\textbf{respondent} = idp$
$put_{\{conf\}}(X, bob, idp) \leftarrow get_{\{auth\}}(X, i) \wedge X.\textbf{respondent} = i$

In e_{google}:

$X := \textbf{subject}\ bob\ \textbf{resource}\ calendar\ \textbf{respondent}\ google$
$REQ := \textbf{subject}\ google\ \textbf{requestId}\ id_{google}\ \textbf{respondent}\ idp$
$put(REQ, google, bob) \leftarrow get(X, bob)$

$RES := \textbf{certifier}\ idp\ \textbf{auth} - \textbf{status}\ is\text{-}authentic$
$put_{\{auth\}}(X, google, bob) \leftarrow get_{\{conf\}}(f(RES, k_{idp}^{-1}), bob)$
$\wedge get_{\{conf\}}(RES1, bob) \wedge RES1.\textbf{resource} = calendar$
$\wedge X.\textbf{data} = secret$

In e_{idp}:

$REQ :=$ **subject** i **requestId** id_i **respondent** idp
$RES :=$ **subject** bob **auth** $-$ **status** *is-authentic* **certifier** idp
respondent $= i$
$put_{\{auth,conf\}}(RES, idp, idp) \leftarrow get_{\{conf\}}(REQ, bob)$

$put_{\{auth,conf\}}(f(RES, k_{idp}^{-1}), idp, bob) \leftarrow get_{\{auth,conf\}}(RES, idp)$
$\wedge RES.$**subject** $= bob$

$REQ :=$ **subject** i **requestId** id_i **respondent** $= idp$ **resource** *calendar*
$RES1 :=$ **resource** *calendar* **respondent** i
$put_{\{auth,conf\}}(RES1, idp, bob) \leftarrow get_{\{conf\}}(REQ, bob)$

The rules of the intruder e_i are as described in Section 9.5. We assume that the intruder has in his repository the certificates:

$$\{\mathbf{kn} \mapsto i\}, \{\mathbf{kn} \mapsto c\}, \{\mathbf{kn} \mapsto sp\}, \{\mathbf{kn} \mapsto idp\}, \{\mathbf{kn} \mapsto bob\},$$
$$\{\mathbf{kn} \mapsto google\}, \{\mathbf{kn} \mapsto calendar\}, \{\mathbf{kn} \mapsto uri\}$$

Confidentiality attack The property saying that the object

$$O := \{\mathbf{kn} \mapsto secret\}$$

is confidential for the intruder i is violated. In fact e_{bob} communicates with the intruder as being a service provider, however, the intruder acts as e_{bob} to access the service provider e_{google}. It is easy to see that with the rule instances presented in the above scenario, the rule $put_{\{auth\}}(O, google, bob)$ is satisfied due to the fact that the intruder sends authentication response RES and $RES1$ as being e_{bob}. As such, the intruder computed $get_{\{auth\}}(O, google)$ by intercepting a message sent on a non-confidential channel intended to e_{bob}. This message is clear and contains the value *secret*. The intruder can then extract this value and compute the secret object.

9.9 Conclusion

In this chapter we defined an intruder as an entity in our attribute based logical framework whose behavior mimics that of a Dolev-Yao intruder. The model is constructed by extending the syntax and evaluation semantics of the logical framework, defined in Chapter 4, to take into account the presence of various types of communication channels and basic cryptographic primitives to express encryption and decryption of messages. We note that in this setting there is *a priori* an unbounded number of possible different available objects because of the cryptographic operations. However we conjecture that we can apply the result of [35] to obtain a decision procedure for the security properties we consider.

The representation of the intruder in such a manner allows the study of security properties, namely confidentiality and authentication properties. The utility of this work lies in the fact that during the trust negotiation session, certificates are exchanged among entities depending on a security policy (a set of trust negotiation rules) which is defined locally by each corresponding entity. Thus it is interesting to test, in the presence of different entities with different sets of trust negotiation rules, whether the interaction among them satisfies global security properties expressed by authentication or confidentiality constraints on the whole process.

Chapter 10

Conclusion and perspectives

Conclusion

Standard access control features only regulate access to a service functionalities, and thus cannot express access control decisions that depend on an execution context. The first objective of this thesis was to develop a logical language to express complex access control policies in a distributed environment that can model dynamic access control and ensures communications between different entities both on the level of message exchange but also on the level of trust establishment. The second objective was to express access control features and specify security properties that contribute to the reasoning about the safety of security problems be it on the level of access control (such as separation of duty constraints) or on the level of communication (authentication and confidentiality for a trust negotiation session).

We presented in Chapter 3 a logical framework that uses the RBAC structure to define access control rules, and a transition system that defines the effects of access control decisions as a collection of permissions or obligations depending on the choice of the user to activate or not the authorized action. Thus in this first approach we assumed that the security state evolves with respect to the access control decisions and the choice made on the activation of these authorizations. However, when trying to express case studies concerning the modeling of business processes, we found some difficulties in adapting the RBAC structure to requirements that concern other elements of security such as characteristics related to the object rather than the subject of an access control decision. Also, this first approach does not ensure in an explicit manner the communication between different entities in a distributed environment.

In Chapter 4 we turned towards an attribute based logical framework regrouping an access control policy, a trust negotiation policy and a workflow. The interplay between the static policy, expressed by the trust negotiation policy, and the dynamic policy, expressed by the access control policy and the workflow, offers an efficient tool to express the state evolution of one entity

with respect to its interaction with the rest of the environment. The least fixed point evaluation semantics of trust negotiation rules keeps track of the current security state of each entity, whereas the access control rules evaluation depends on the result of the trust negotiation session and thus offers access control decisions depending on the security state. Moreover, the access control rules can be viewed as guards on the execution of the workflow, that is an action can be executed if it is both authorized by the access control policy and executable in the workflow. In order to show the expressivity of this framework and to illustrate the use of the language and the interaction between different entities to simulate a business process, we presented in Chapter 5 a modeling of a car registration process.

The second part of this thesis presented access control features and properties both on the level of securing the access control framework of each entity and on the level of securing the communication between the different entities in the environment on the trust negotiation level. As such, Chapters 6 and 7 were devoted to the study of delegation and separation of duty properties. We presented different perspectives of expressing these properties and the limitations found in the previous works. We showed that, unlike most existing works in the literature, we are able to encode most of the aspects of delegation that exist in the literature as well as to specify separation of duty constraints both on the level of policy design and policy enforcement without modifying the syntax of our language. The information about the use of delegation or the use of a task in general in the presence of security constraints, is evaluated locally in the entity responsible for the delegation or the task execution. To this end, the history of delegation or task execution was recorded by means of adding an object to the repository of the entity. The access control decisions in terms of authorizing delegated rights or enforcing separation of duty constraints depend on the result of queries for the existence (or the non-existence) in the repository of the entity of delegation records of task execution records. The case of delegation is simple. In fact, we assume that each entity decides on the delegations concerning its own functionalities or elements. The trust negotiation policy of the entity can then manage the diffusion of the delegation information in form of objects to other concerned entities. The case of security constraints is more complicated. The difficulty lies in the fact that queries are for the *non-existence of objects* that can only be performed locally in a given entity. Nonetheless, even though the definition of security constraints on the level of the entire framework is not yet feasible, we defined security properties in the form of decision problems in order to check for the safety of the model. An encoding for RBAC and several of its extensions to role hierarchy, role activation and delegation was presented in Chapter 8.

The final chapter of this thesis tackled a different aspect of security. In fact, as the main aim of this thesis was to define a framework to model access control policies in a *distributed* environment, it was important to study the problem of securing the communication between the different entities. To this end, we provided an extension of the language to take into account the security properties of communication channels and assumed the existence in the environment

of a malicious entity that do not necessarily abide to its trust negotiation policy. To this end we defined an intruder model as a logical entity with a specific set of rules, and gave the specifications for authentication and confidentiality properties for the system in the presence of this intruder.

Perspectives

Future works can be directed in three main research directions. First towards the orchestration of services, second towards model checking within a distributed environment and third towards the implementation of this framework in a tool to model and validate access control policies.

On orchestration

In Chapter 4 we defined a unified framework to express access control in a distributed environment, in Chapter 5 we gave an illustration of a car registration business process as a collection of entities each of which with its own set of access control rules, trust negotiation rules and workflow. The different entities communicated by means of negotiating objects, but also by means of sending and receiving requests in order to coordinate the execution of the local workflows and thus complete the business process. However, in order to do that we explicitly model the send/receive process in the communicating entities. It is interesting to abstract away this encoding by assuming the presence of an orchestrator whose role is to regulate the interaction between the different entities by managing the exchange of request messages. The orchestrator can also take into account the security constraints specified for the entire system (through the definition of a business process). This can be done by delegating the control over the sensitive tasks (that may not be controlled by the same entity) to the orchestrator. The orchestrator can thus be seen as a logical entity with a specific set of rules and a workflow that on one hand is responsible for the definition of the general access control policy of the business process through its security policy, and on the other hand regulates the exchange of message requests between the different entities through it workflow definition.

On model checking

In Chapter 7 we gave the specification for the enforcement of security constraints in the environment through the establishment of a monitor that after each execution of the workflow, checks for security violations. However, in the actual framework, such a monitor can only act as guard on the safety of the model. It would be interesting to allow the monitor, in case of a violation, to trace back the source of the violation and thus allow the policy modeler to modify the security policy of the entity in question in order to remediate to the problem. This however is not very simple to achieve. In fact, we need to redefine the framework in order to be able to trace back the source of the violation. To do

that one need to keep track of the history of workflow executions on the one hand, and monitor the access control authorizations that can be acquired in a decentralized manner through the negotiation of objects with other entities.

On implementation

The definition of a formal framework for security policies is necessary to reason about the security properties of policies. However it is important to implement this framework in a tool to express and validate access control policies. A formal language to specify security policies ASLan, is defined through the AVANTSSAR Project. It is based on rewrite rules to express workflows, Horn clauses to express the security policy requirements, and Linear Temporal Logic to express security properties. Tools were developed in order to implement it. In our framework security rules are expressed in first order logic and the workflow is defined as a transition system through the process definitions. It would be interesting to reduce the security rules in our model to Horn clauses in order to make use of these existing tools.

Bibliography

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[2] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[3] A. Anderson. Sun position paper. Technical report.

[4] Alessandro Armando, Roberto Carbone, and Luca Compagna. Ltl model checking for security protocols. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 385–396, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *FMSE '08: Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10, New York, NY, USA, 2008. ACM.

[6] Philippe Balbiani, Yannick Chevalier, and Marwa El Houri. A logical approach to dynamic role-based access control in a distributed environment. In *International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA),*, pages 194–208. Springer-Verlag, 2008.

[7] Philippe Balbiani, Yannick Chevalier, and Marwa El-Houri. A Logical Framework for Reasoning about Policies with Trust Negotiations and Workflows in a Distributed Environment. In *Proceedings of the 4th International Conference on Risks and Security of Internet and Systems*, pages 3–11, Toulouse, France, 2009. IEEE.

[8] E. Barka and R. Sandhu. A role-based delegation model and some extensions.

[9] E. Barka and R. Sandhu. Framework for role-based delegation models. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 168, Washington, DC, USA, 2000. IEEE Computer Society.

[10] David A. Basin, Samuel J. Burri, and Günter Karjoth. Dynamic enforcement of abstract separation of duty constraints. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[11] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. Technical Report MSR-TR-2006-120, Microsoft Research, September 2006.

[12] Moritz Y. Becker and Sebastian Nanz. S.: A logic for state-modifying authorization policies. Technical report, In: European Symposium on Research in Computer Security, 2007.

[13] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, page 159, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *In Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 139–154. IEEE Computer Society Press, 2004.

[15] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report Technical Report M74-244, MITRE Corporation, Bedford, MA,.

[16] E. Bertino, E. Ferrari, and V. Atluri. A flexible model supporting the specification and enforcement of role-based authorizations in workflow management systems. In *2nd ACM Workshop on Role-Based Access Control (RBAC)*. Fairfax, VA, 1997.

[17] Elisa Bertino, Jason Crampton, and Federica Paci. Access control and authorization constraints for ws-bpel. In *ICWS*, pages 275–284. IEEE Computer Society, 2006.

[18] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, 1999.

[19] Jajodia S. Wang X. Wijesekera D. Bettini, C. Provisions and obligations in policy management and security applications. In *Proceedings of 28th International Conference on Very Large Data Bases.*, pages 502–513. Morgan Kaufmann, 2002.

[20] K. Biba. Integrity considerations for computer systems. In *Technical Report ESD-TR-76-372, MITRE, Bedford, Mass*, 1976.

[21] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The keynote trust-management system version 2, 1999.

[22] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. pages 185–210, 1999.

[23] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, pages 254–274, London, UK, 1998. Springer-Verlag.

[24] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2000. ACM.

[25] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[26] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[27] D. Clark and W. Wilson. Evolution of a model for computer integrity. In *Report of the Invitational Workshop on Data Integrity*, Gaithersburg, 1989. In Z.Ruthberg and W.Polk, editors.

[28] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.

[29] Crampton, Jason, Khambhammettu, and Hemanth. Delegation in role-based access control. *International Journal of Information Security (IJIS)*, 7(2):123–136, April 2008.

[30] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 43–50, New York, NY, USA, 2003. ACM.

[31] Jason Crampton and Hemanth Khambhammettu. On delegation and workflow execution models. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2137–2144, New York, NY, USA, 2008. ACM.

[32] T. Dierks and C. Allen. The tls protocol version 1.0, 1999.

[33] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[34] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *of Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[35] Shimon Even and Oded Goldreich. On the security of multi-party ping-pong protocols. In *FOCS*, pages 34–39. IEEE, 1983.

[36] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn. Role-based access control(rbac): Features and motivations. In *Proceedings 11th Annual Computer Security Applications Conference*, pages 241–248, 1995.

[37] D.F. Ferraiolo, R Sandhu, S. Gavrila, D.R. Kuhn, and R. . Chandramouli. Proposed nist standard for role-based access control. In *ACM Transactions on Information and System Security*, pages 4(3):222–274, August 2001.

[38] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[39] Yuri Gurevich and Itay Neeman. Dkal: Distributed-knowledge authorization language. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society.

[40] Diala Abi Haidar, Nora Cuppens-Boulahia, Frédéric Cuppens, and Hervé Debar. XeNA: an access negotiation framework using XACML. *Annales des télécommunications- Annals of telecommunications*, 64(1-2):155 – 169, january 2009.

[41] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[42] Adam Hess, Jared Jacobson, Hyrum Mills, Ryan Wamsley, Kent E. Seamons, and Bryan Smith. Advanced client/server authentication in tls. In *NDSS*. The Internet Society, 2002.

[43] Manuel Hilty, David A. Basin, and Alexander Pretschner. On obligations. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 98–117. Springer, 2005.

[44] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Business Process Management*, pages 220–235, 2005.

[45] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[46] A. Jones, R. Lipton, and L. Snyder. A linear time algorithm for deciding subject security. In *Proceedings of the 17th Annual Symposium on the foundations of Computer Science*, pages 33–41, 1976.

[47] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. OASIS Standard, April 2007.

[48] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web services choreography description language version 1.0. `http://www.w3.org/TR/ws-cdl-10/`, 2005.

[49] Hristo Koshutanski and Fabio Massacci. An access control framework for business processes for web services. In Sushil Jajodia and Michiharu Kudo, editors, *XML Security*, pages 15–24. ACM, 2003.

[50] B. W. Lampson. Dynamic protection structures. In *AFIPS '69 (Fall): Proceedings of the November 18-20, 1969, fall joint computer conference*, pages 27–38, New York, NY, USA, 1969. ACM.

[51] Jiangtao Li, Ninghui Li, and William H. Winsborough. Automated trust negotiation using cryptographic credentials. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *ACM Conference on Computer and Communications Security*, pages 46–57. ACM, 2005.

[52] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.

[53] Ninghui Li and Qihua Wang. Beyond separation of duty: an algebra for specifying high-level security policies. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 356–369, New York, NY, USA, 2006. ACM.

[54] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. *J. Comput. Secur.*, 11(1):35–86, 2003.

[55] Gavin Lowe. A hierarchy of authentication specifications. In *CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[56] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.

[57] Ueli M. Maurer and Pierre E. Schmid. A calculus for secure channel establishment in open networks. In *ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security*, pages 175–192, London, UK, 1994. Springer-Verlag.

[58] Nilo Mitra and Yves Lafon. Soap version 1.2 part 0: Primer (second edition). W3C Recommendation, April 2007.

[59] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: Soap message security 1.1. OASIS Standard Specification, February 2006.

[60] Dr. Michael J. Nash and Dr. Keith R. Poland. Some conundrums concerning separation of duty. *Security and Privacy, IEEE Symposium on*, 0:201, 1990.

[61] Federica Paci, Elisa Bertino, and Jason Crampton. An access-control framework for ws-bpel. *Int. J. Web Service Res.*, 5(3):20–43, 2008.

[62] Lockhart H. Anderson A. Mishra P. Parducci, B. Core and hierarchical role based access control (RBAC) profile of XACML v2.0. Oasis open TC on XACML, 2005.

[63] Lockhart H. Anderson A. Mishra P. Parducci, B. SAML 2.0 profile of XACML v2.0. Oasis open TC on XACML, 2005.

[64] Lockhart H. Anderson A. Mishra P. Parducci, B. XACML 2.0 core. Oasis open TC on XACML, 2005.

[65] Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo. Security assertion markup language (saml) v2.0 technical overview. Technical report, OASIS Security Services, March 2008.

[66] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of IEEE, 63(9)*, pages 1278–1308, 1975.

[67] Ravi Sandhu, Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. In *ACM Transactions on Information and System Security*, February 1999.

[68] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[69] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[70] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In David F. Ferraiolo and Indrakshi Ray, editors, *SACMAT*, pages 139–149. ACM, 2006.

[71] Richard Simon and Mary Ellen Zurko. Separation of duty in role-based environments. *Computer Security Foundations Workshop, IEEE*, 0:183, 1997.

[72] TCSEC. Trusted computer system evaluation criteria. Technical Report DoD Standard, DoD 5200.28-STD, Department of Defense (DoD), 1985.

[73] The Avantssar Project. Problem cases and their trust and security requirements. Deliverable D5.1, Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures (AVANTSSAR), `http://www.avantssar.eu/`, 2008.

[74] Asir S Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçinalp. Web services policy (wspolicy) 1.5 - framework. W3C Recommendation, September 2007.

[75] Jacques Wainer and Akhil Kumar. A fine-grained, controllable, user-to-user delegation method in rbac. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 59–66, New York, NY, USA, 2005. ACM.

[76] Qihua Wang, Ninghui Li, and Hong Chen. On the security of delegation in access control systems. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 317–332, Berlin, Heidelberg, 2008. Springer-Verlag.

[77] Stephen A. White and Derek Miers. *BPMN Modeling and Reference Guide*. Future Strategies Inc, 2008.

[78] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume I, pages 88–102. IEEE Press, Jan 2000.

[79] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, 2002.

[80] Marianne Winslett, Charles C. Zhang, and Piero A. Bonatti. Peeraccess: a logic for distributed authorization. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 168–179, New York, NY, USA, 2005. ACM.

[81] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role-based delegation and revocation. *ACM Trans. Inf. Syst. Secur.*, 6(3):404–441, 2003.

[82] Xinwen Zhang, Sejong Oh, and Ravi Sandhu. Pbdm: a flexible delegation model in rbac. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 149–157, New York, NY, USA, 2003. ACM.

Un modèle formel pour exprimer des politiques dynamiques pour contrôle d'accès et négociation dans un environnement distribué

Résumé en français

Résumé

Le développement de l'internet et l'acceptation de l'architecture orientée service comme moyen d'intégration des applications inter et intra organisationelles ont permis l'apparition de nouvelles formes de strucutres distribuées. Dans ce contexte des applications et des ressources développées independemment les unes des autres sont mis à disposition sous forme de services. Ces services communiquent les uns avec les autres en s'échageant des messages sur HTTP, SOAP, etc. Ce contexte offre ainsi la possibilité d'orchestrer des services existants afin de créer de nouveaux services adaptés à une tâche donnée. Pour des raisons de sécurité ou des raisons légales il est nécessaire de contrôler l'accès à ces services ainsi que la collaboration et l'échange d'information entre eux lors de l'exécution d'une tâche commune.

L'objectif principal de cette thèse est de définir un langage logique de haut niveau qui permet l'expression de politiques de sécurité complexes au sein d'un modèle de contrôle d'accès.

Le développement de ce langage se fait en trois temps. Dans un premier temps nous présentons un modèle dynamique basé sur les rôles. Ainsi, nous considérons que l'évolution de l'état de sécurité d'un service dépend de l'exécution de ses fonctionnalités. Dans un deuxième temps nous définissons un formalisme basé sur les attributs qui offre plus de flexibilité en termes de spécifications des conditions de contrôle d'accès, et ajoutons la notion de *workflow* qui permet de définir un ordre sur l'exécution des fonctionnalités et donc permet de modéliser le comportement d'un service. Dans un dernier temps, afin de prendre en compte la collaboration entre différents services, nous ajoutons un mécanisme de négociation qui permet à chaque service de définir sa propre politique d'échange avec les autres services dans l'environnement.

La conception d'un tel cadre logique unifié facilite les analyses de sûreté des politiques de sécurité puisque tous les facteurs qui influencent les décisions de contrôle d'accès sont pris en compte dans le même cadre. Ainsi le second objectif de cette thèse est d'étudier d'une part les principales propriétés de contrôle d'accès telles la délégation et la séparation des tâches et d'autre part les propriétés de sécurité pour la communication entre les différents services au niveau de la négociation. En effet, afin de montrer l'expressivité du modèle nous présentons un codage génériques pour les différentes notions de délégation et de révocation, nous présentons également une spécifications pour les différents aspects de séparation de tâches ainsi qu'une modélisation du contrôle d'accès

basé sur les rôles et de certaines de ses extensions. Enfin, et pour prendre en compte un réseau de communication réel, nous présentons une extension du langage qui permet de définir un modèle d'intrus actif qui agit au niveau d'une session de négociation en interceptant des messages et en construisanr de nouveaux messages.

Chapitre 1 : Introduction

Contexte

Le développement du réseau Internet et la diffusion de l'information dans des structures distribuées rend la collaboration entre différentes entités nécessaire. Un service est une entité responsable d'un ensemble de fonctionnalités. Il peut être utilisé par un utilisateur humain ou par un autre service. Par exemple, un service connecté à une librairie virtuelle permet aux utilisateurs de faire une recherche sur les livres disponibles. Un second service relié à un système bancaire sécurisé offre des possibilités de paiement protégé. Un troisième service fournit des avantages de livraison. Ainsi, si un utilisateur décide d'acheter un livre en ligne, une collaboration entre ces trois services est nécessaire afin de pouvoir réaliser la tâche de l'utilisateur. Un scénario similaire peut exister au sein d'une organisation contenant différents départements décentralisés.

Afin de pouvoir collaborer, les services doivent être capable de communiquer, échanger des informations et atteindre des accords mutuels. Il existe des langages standards qui garantissent la sécurité des messages comme SOAP [58] qui définit la forme des messages à travers une enveloppe et contrôle la transmission de messages entre les services. De plus, ces services ont besoin d'une politique de sécurité qui spécifie qui a droit d'accéder aux fonctionnalités du service. Des langages standards comme WS-SecurityPolicy [59] et WS-Policy [74] peuvent définir une politique de sécurité au niveau des messages, plus explicitement, ces langages sont capables d'exprimer, pour un service donné, les critères à satisfaire pour accepter un message d'un autre service. Les fonctionnalités d'un service doivent aussi être exécutées suivant un certain ordre afin de pouvoir contribuer à atteindre une tâche globale. Cette propriété peut être garantie par BPEL [47], un langage standard qui spécifie l'interaction entre les services et donc permet l'expression d'un *business process* à partir de l'orchestration du comportement des différents services.

Cependant, de tels langages standards sont conçus essentiellement pour l'implémentation de politiques de sécurité et la conception de *business process* et donc n'offre pas les moyens nécessaires pour raisonner sur les politiques.

Objectifs et contributions de cette thèse

L'objectif principal de cette thèse est de présenter un langage logique de haut niveau capable d'exprimer des politiques de sécurité complexes dans le cadre d'un modèle de contrôle d'accès. Un modèle de contrôle d'accès doit garantir la sûreté du système par rapport aux critères de sécurité prédéfinis, mais aussi permettre la possibilité de procéder à la vérification de la sécurité du système. En particulier, un modèle doit être capable de fournir une solution au problème de sûreté qui, étant donné un éetqt initial et une politique de sécurité, vérifie s'il est possible d'arriver dans un état non sûr lorsqu'on applique la politique de contrôle d'accès. Les critères de sûreté sont souvent définis en terme de confidentialité ou integrité de l'information, mais peuvent aussi dépendre de contraintes opérationelles, disponibilité de l'information, etc. Afin de prendre en compte ces problèmes, il est essentiel de définir un cadre mathématique capable d'exprimer les règles de contrôles d'accès mais aussi les objectifs de sécurités, et dans lequel il serait possible de valider les propriétés du système.

En général, les politiques de contrôle d'accès sont appliquées dans un environnement spécifique et les décisions de contrôle d'accès mènent à la permission ou l'interdiction de certaines actions. Cela a un effet sur l'environnement de la politique. Par conséquent, il est aussi important de pouvoir spécifier les effets des autorisations de contrôle d'accès dans le cadre du modèle de contrôle d'accès. Plus précisement, en plus de la possibilité de fournir des réponses de type "oui ou non" pour les requêtes d'accès, le langage de sécurité doit aussi exprimer le "changement" effectif qui a lieu lorsqu'une requête est autorisée (ou interdite).

De plus, comme désormais l'information devient de plus en plus décentralisée, de nouveaux aspects des contrôle d'accès sont nécessaires. Un modèle de contrôle d'accès doit prendre en compte la nature distribuée de l'environnement et la possibilité de collaboration et de communication entre les différents systèmes de contrôle d'accès qui ne se connaissent pas nécessairement. Ainsi, il est important d'avoir un langage de sécurité qui est capable d'exprimer la négociation de confiance entre les différents acteurs d'une collaboration afin d'établir une confiance mutuelle.

Enfin, les organisations actuelles définissent souvent une politique de contôle d'accès par rapport à des *business processes*. En d'autes termes, la collaboration entres les différents services est ordonnancée d'une manière spécifique afin d'accomplir un but final. Ainsi, pour pouvoir exprimer les *business processes*, le langage de sécurité doit être capable de définir en plus des effets directs des actions autorisées, la possibilité de spécifier un ordre sur ces actions.

Le premier objectif de cette thèse est de formaliser un cadre unifié qui prend en compte le contrôle d'accès dans un environnement où les entités communiquent pour accomplir un ensemble de tâches par rapport à leur politiques de sécurité respectives. De telles entités peuvent être vues comme des services mais aussi comme des utilisateurs ou des organisations.

Il est important de mentionner qu'il existe plusieurs modèles qui répondent à certains des problèmes présentés si dessus ([45, 13, 52, 18, 12, 11]). Cependant,

la contribution de cette thèse est de présenter un cadre unifié qui prend en compte la totalité de ces caractéristiques. En effet, avoir un cadre unifé a pour but de faciliter les analyses de sûreté pour les politiques de sécurité puisque tous les différents facteurs qui influencent les décisions de contrôle d'accès sont pris en compte au sein du même cadre.

Le second objectif de cette thèse est d'étudier les principales propriétés de contrôle d'accès comme la délégation et le séparation des tâches d'une part et les propriétées de sécurités concernant la communication entre les entités au niveau de la négociation d'autre part. L'expression de ces propriétées permettrait la spécification des critères de sûreté pour le cadre unifié en général. En particullier, nous spécifions des contraintes de sécurité qui peuvent être vérifiées au niveau des règles de sécurité, exécutées au niveau du *workflow* au niveau de l'exécution. Nous spécifions aussi les propriéetes d'authneticité et de confidentialité qui doivent être satisfaites lors d'une session de négociation en présence d'une entité malhonnête.

Plan de la thèse

Cette thèse est constituée de deux parties. La première partie présente une étude des modèles de contrôle d'accèes déjà existants ainsi que la conception et la formalisation d'un langage logique pour exprimer des politiques de contrôle d'accès complexes qui modélisent les propriétés dynamiques dans un environnement distribué. La seconde partie présente certaines propriétés de contrôle d'accès qui peuvent être exprimées dans ce nouveau modèle et spécifie des propriétés de sécurité qui contribuent au raisonnement en terme de sûreté des problèmes de sécurité.

Dans le chapitre 2, nous présentons une vue d'ensemble des modèles de contrôle d'accès existants en s'intéressant à trois principaux critères :

- la flexibilité du langage pour exprimer des politiques de contrôle d'accès complexes, c'est à dire la possibilité d'exprimer les conditions requises qui sortent du paradigme standard sujet, objet et action,
- la capacité d'exprimer le contrôle d'accès dynamique, c'est à dire la possibilité de définir des effets aux accès autorisés et de modéliser l'évolution de l'état de sécurité en termes d'un ordre d'exécution bien défini,
- la nécessité de définir une interaction entre les différents acteurs dans l'environnement en définissant un mécanisme de négociation qui permet l'établissement de la confiance entre ces acteurs qui ne se connaissent pas nécessairement.

Nous montrons les solutions offertes par plusieurs modèles de contrôle d'accès à certains de ces critères et expliquons leurs limitations dans le but de motiver le besoin pour un cadre unifié qui prend en compte les trois critères ci dessus. Dans le chaptire 3 nous présentons la première approche qui définit un modèle de contrôle d'accès dynamique basé sur les rôles. L'importance principale de

ce modèle réside dans la définition des politiques dynamiques pour exprimer les effets des autorisations selon que les actions autorisées sont exécutées ou non. Dans le chapitre 4 nous étendons cette modélisation à un cadre logique basé sur les attributs et capable de satisfaire les trois critères présentés ci dessus. Le développement de ce nouveau modèle était partiellement influencé par l'analyse de plusieurs études de cas que nous avons étudiées dans le cadre de l'expression des politiques de sécurité pour les *business processes*. Dans le chapitre 5, nous présentons une de ces études de cas pour modéliser un processus d'enregistrement de voiture afin d'illustrer l'expressivité de notre cadre formel.

Dans le chapitre 6 nous présentons une vue d'ensemble des différents concepts de délégation et de revocation. Nous présentons ensuite un codage pour exprimer les différents types de délégation et de révocation dans notre modèle. Le chapitre 7 présente les propriétés de séparation des tâches qui montre l'expressivité du modèle au niveau de la spécification des contraintes de sécurités mais aussi ou niveau de la vérification de ces contraintes au niveau de l'exécution. Dans le chapitre 8 nous présentons une modélisation du contrôle d'accès basé sur les rôles et quelques unes de ses extensions et illustrons les propriétées de contrôle d'accès dans ce modèle. Le chapitre 9 présente un différent aspect de la sécurité en considérant la présence d'une entité malhonnête au niveau de la négociation. Ainsi, nous présentons une extension du modèle qui prend en considération les différents types de canaux de communication et donnons une formallisation des spécifications pour les propriétées d'authenticité et de confidentialité dans ce cadre.

Enfin, il est nécessaire de mentionner que cette thèse a été partiellement financée par FP7-ICT-2007-1 Project no. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures[1]".

[1] www.avantssar.eu

Partie I : Modèles de contrôle d'accès dynamiques

Chapitre 2 : Travaux voisins

La sécurité a toujours été considérée comme une nécéssité pour les êtres humains, et les décisions de contrôle d'accès ou autorisations ont toujours existé dans la vie réelle à chaque fois que l'homme a eu besoin de protéger ses propriétés. Avec le développement de la technologie de l'information, la prise de décisions s'est fortement liés aux ordinateurs. Ainsi, de plus en plus d'efforts ont été investit dans des recherches concernant le contrôle d'accès afin de contrôler l'accès aux informations sensibles et de protéger les ressources du système. En général, les décisions de contrôle d'accès refèrent à des décisions de type "oui ou non" sur les questions concernant qui peut faire quoi.

Un modèle de contrôle d'accès gère l'accès aux fonctionnalités (tel les web services, les processus ou les organisations) par des utilisateurs autorisés à travers une politique de contrôle d'accès. Dans sa forme la plus simple cette relation est représenté par une liste qui associe à des utilisateurs autorisés des ressources. Lorsqu'un utilisateur présente une preuve sur son identité, une décision d'autorisation concernant la relation utilisateur ressource peut être prise. Le contrôle d'accès est défini à travers les notions d'utilisateur, sujet, objet, et action. Un utilisateur est un agent humain qui interagit avec une machine, un sujet est une entité active, qui agit possiblement au nom d'un utilisateur ou d'une autre machine. Un objet est une entité passive qui fait référence à une ressource, une base de données ou tout autre information et peut être utilisé par des sujets, et une action est un processus actif qui peut être exécuter par un sujet sur un objet.

Dans ce chapitre nous donnons une vue d'ensemble des premiers modèles de contrôle d'accès dans la section 2.1, ensuite, nous présentons le contrôle d'accès basé sur les rôles (RBAC) dans la section 2.2. Dans les sections 2.3 et 2.4 nous présentons deux extensions de RBAC ; FAF et OrBAC qui offrent plus de flexibilité que la structure RBAC. Dans la section 2.5 nous présentons XACML, un langage standard pour exprimer des politiques de sécurité, et dans la section 2.6 nous présentons des modèles basés sur les rôles plus récents qui sont capables d'exprimer des politiques de sécurité dans un environnement distribué.

La section 2.7 présente SecPAL un langage logique basé sur les attributs, et dans la section 2.8 nous présentons des langages pour exprimer les politiques de sécurité dynamiques. Dans la section 2.9 nous présentons quelques notions sur des modèles qui gèrent la négociation de confiance et nous concluons dans la section 2.10.

Chapitre 3 : Une approche logique pour le contrôle d'accès dynamique basé sur les rôles

À la différence de l'organisation traditionnelle de l'économie dans laquelle une organisation centrale ou l'État est responsable pour la production d'un bien, aujourd'hui l'économie est basée de plus en plus sur l'intégration de processus qui proviennent de différentes sources et sur une aggrégation optimale de ces ressources. Dans un système de banque électronique par exemple, la banque va demander au Bureau de Credit, un partenaire externe, d'analyser le profil de crédit d'un client dans le cas d'une demande de prêt. De plus, le processus de demande de prêt va être traité par différents services appartenant à l'organisme de la banque afin d'évaluer le prêt, faire une décision, et éventuellement faire une offre au client. Dans ces exemples, les différents partenaires doivent s'échanger des informations pour s'assurer du succés de la coopération, mais ils doivent aussi restreindre la diffusion de l'information.

Dans ce chapitre nous présentons un langage capable d'exprimer des politiques de contrôles d'accés dynamiques. Ce langage basé sur des règles logiques, ajoute un aspect dynamique à l'expression des règles de contrôle d'accès par la spécification explicite des effets des actions autorisées. Notre approche ressemble à celle présentée dans [11]. SecPAL utilise des requêtes pour modifier l'état de sécurité du système lorsque la requête est permise, cependant dans notre approche nous modélisons les effets d'une action autorisée suivant si elle a été exécutée ou non. Comme dans le cas de [13, 11] nous présentons un codage pour les extensions de RBAC comme la délégation, la séparation des tâches ou la hierarchie de rôle dans notre modèle. Enfin, comme notre objectif principal est de définir un langage capable de modéliser les effets des actions, nous avons choisi de modéliser en plus des permissions, la notion d'obligation.

L'idée de ce modèle réside dans le fait que nous considérons qu'un système de contrôle d'accès est caractérisé par des contextes de décisions et un programme Datalog. Un contexte de décision est défini par un ensemble de permissions et d'obligations. Dans chaque contexte de décision, les décisions de contrôle d'accès sont basés sur le calcul de nouvelles autorisations pour les requêtes de premissions ou d'obligations à partir du programme Datalog et de l'ensemble des permissions et obligations qui définissent le contexte de décision courant. Le système de contrôle d'accès évolue d'un contexte de décision à un autre en fonction des actions exécutées par l'utilisateur. Un désavantage de cette simplicité est qu'il est nécessaire de déclarer toutes les actions qui modifient le contexte de décision à chaque transition.

Cependant, ce modèle nous permet d'exprimer les politiques RBAC ainsi que leurs diverses extensions. Dans la section 3.1 nous présentons le langage, dans la section 3.2 nous présentons comment encoder RBAC dans ce langage, et dans la section 3.3 nous présentons des résultats de complexité concernant des problèmes de décisions sur le contrôle d'accès dans notre langage.

Chapitre 4 : Politiques de sécurité avec négociation et *workflows* dans un environnement distribué

L'architecture orientée service est de plus en plus acceptée comme un moyen d'intégration d'applications inter et intra organisationelles. Dans ce contexte des applications et des ressources développées independemment les unes des autres sont mis à disposition sous forme de services. Ces services communiquent les uns avec les autres en s'échageant des messages sur HTTP, SOAP, etc. Un avantage fondamental dans ce contexte est la possibilité d'orchestrer des services existants afin de créer de nouveaux services adaptés à une tâche donnée. De nombreux langages (WS-CDL [48], WSBPEL [47], BPMN [77], ...) ont été proposés pour décrire le *workflow* d'un service orchestrateur. Ces langages peuvent définir une sémantique opérationnelle en term de π-calculus [56] ou réseaux de Petri [44].

Pour des raisons de sécurité, il est nécessaire de contrôler au sein du *workflow* et sur l'interface du *workflow*, dans quels contextes une action peut être exécutée. Cela implique qu'en plus de la définition d'un *workflow* pour un service orchestrateur, il faut aussi définir une politique de sécurité au niveau de l'application qui peut décrire les rôles, et les contraintes de séparation de tâches ou autres qui doivent être respecter dans le *workflow*. Pour que le processus ainsi défini puisse être employé dans différents environnements, on ajoute une couche de négociation pour les services; afin qu'ils puissent interagir et prouver si nécessaire leur légitimité envers les autres services.

Dans ce chaptire et dans le reste de cette thèse, nous avons choisi de décrire ces services par des entités logiques qui regroupent tous les aspects propres à une application ou une ressource donnée. L'originalité de cette approche consiste en deux parties. D'une part l'interaction entre l'exécution du *workflow* et la politique de contrôle d'accès possible dans le cadre du modèle unifié permet l'expression des contraintes réelles qui existent lorsqu'on considère des instances réelles de *business processes* d'une façon naturelle. D'autre part, l'ajout d'une couche de négociation permet l'expression de l'échange entre les différents services dans l'environnement. Ansi, la force de ce modèle réside dans la collaboration entre ces trois strucutres, la politique de contrôle d'accès, la politique de négociation et le *workkflow*, afin de pouvoir exprimer l'évolution d'un état de sécurité en fonction de son interaction avec d'autres services dans l'environnement distribué.

Dans la section 4.2 nous présentons une vue d'ensemble de notre modèle. Dans la section 4.3 nous présentons la syntaxe utilisée pour définir les règles de sécurité qui seront présentées dans la section 4.4. Dans la section 4.5 nous

donnons la sémantique d'évaluation des règles de sécurité. Dans la section 4.6 nous présentons les différents éléments du *workflow* et dans la section 4.7 nous présentons les relations de transition pour l'évolution du *workflow*. Nous présentons enfin dans la section 4.8 une application dans ce langage.

Chapitre 5 : Une étude de cas : Le processus d'enregistrement de voitures

Dans ce chapitre nous présentons une modélisation complète d'une étude de cas fournie par le projet AVANTSSAR dans le deliverable D 5.1 [73]. Nous présentons cette étude de cas pas à pas en définissant les différentes entités puis en ajoutant successivement leurs politiques de sécurité et leurs *workflows*.

Partie II : Expression des propriétés de sécurité

Dans la première partie de cette thèse nous avons présenté un cadre logigue pour exprimer les politiques de contrôle d'accès avec la possibilité de négociation et d'exécution du *workflow*. Notre objectif principal était de permettre l'expression de politiques de sécurité au delà du cadre du contrôle d'accès basé sur les rôles, et de fournir un modèle formel pour l'échange d'informations entre entités dans un environnement distribué. Dans cette partie nous présentons des propriétés et spécifications de sécurité pour raisonner sur la sûreté du modèle.

Dans le modèle de contrôle d'accès défini dans le chapitre 4 nous distinguons entre une politique statique et une politique dynamique. La politique statique permet d'évaluer l'état courant des différentes entités par rapport aux règles de négociation, tandis que la politique dynamique est exprimée par l'interaction entre les règles de contrôle d'accès et le *workflow* et permet de modifier l'état de sécurité en conséquence.

La définition explicite des processus du *workflow* et des règles de transition permettent la mise à jour de l'état de sécurité d'une entité afin de réévaluer de nouvelles décisions par rapport à l'évolution de cet état. Les règles de contrôle d'accès jouent le rôle de gardes sur l'exécution de ces processus au sein du *workflow*.

De plus, les règles de négociation offrent un modèle formel pour la communication entre les entités dans l'environnement qui permet d'établir une confiance mutuelle. Enfin, l'avantage d'utiliser les attributs réside dans la possibilité d'exprimer des propriétés de sécurité à travers la définition de nouveaux objets sans modifier la syntax du modèle.

Dans cette seconde partie, nous nous intéressons à l'expression et à la spécification des propriétés de sécurité au niveau du contrôle d'accès et au niveau de la communication par négociation.

Un modèle de contrôle d'accès doit être capable d'exprimer différentes propriétés de contrôle d'accès implémentées dans la structure d'une organisation. Par exemple, il doit avoir la capacité de modéliser la délégation tout en préservant les contraintes de séparation de tâches. En effet, il est souvent le cas que différentes variantes de la même propriété de sécurité soient utilisées dans une même organisation. C'est le cas de la délégation dans [14] par exemple. Ainsi,

nous présentons une modélisation de la délégation et de la séparation des tâches dans un environnement distribué. Pour cela nous utilisons l'interaction entre la politique de contrôle d'accès d'une entité donnée et des politiques de négociation des autres entités dans le modèle. Cette interaction présente une façon intéressante pour exprimer des propriétés de contrôle d'accès centralisées d'une manière décentralisée à cause du fait que les entités ont la possibilité de négocier des certificats entre eux. Cependant, en la présence de propriétés de contrôle d'accès, tel la délégation ou l'héritage de rôle par exemple, des autorisations supplémentaires qui n'étaient pas nécessairement prises en considération lors de la conception de la politique de sécurité peuvent avoir lieu. En raison de la dynamicité du modèle, il n'est pas toujours évident de prédire ces nouvelles autorisations comme les droits d'accès sont modifiés à chaque état de sécurité. Pour remédier à ce problème, nous présentons dans cette partie des problèmes de décisions pour vérifier les éventuelles violation des contraintes de sécurité qui peuvent avoir lieu à cause de ces nouvelles autorisations.

De plus, la nature distribuée de l'environnement nécessite la communication entre les différentes entités afin d'échanger des objets durant une session de négociation. Dans le chapitre 4 nous avons considérer que cette communication se fait dans un environnement sécurisé, c'est à dire, nous considérons que toutes les entités respectent leur politique de négociation. Cependant, cela n'est pas toujours le cas dans le monde réel. Ainsi, il est nécessaire de considérer les propriétés de sécurité que garantissent les canaux de communication entre les différentes entités afin d'évaluer la sécurité de l'infrastructure de négociation. Pour cela nous présentons dans cette partie une extension de notre modèle qui prend en considération la sécurité de sessions de négociation et permet de spécifier des problèmes de décision pour vérifier les propriétés d'authenticité et de confidentialité durant une session de négociation.

Cette partie est répartie comme suit ; dans le chapitre 6, nous présentons une spécification de notre approche pour modéliser la délégation et la révocation dans un environnement distribué. Dans le chapitre 7 nous présentons une vue d'ensemble des contraintes de sécurité et donnons une modélisation pour l'expression et la vérification de ces contraintes dans notre modèle. Dans le chapitre 8 nous présentons une modélisation du modèle RBAC dans notre formalisme en prenant en considération les notions d'héritage de rôle, activation de rôle et délégation de rôle. Dans le chapitre 9 nous nous intéressons à un problème de sécurité différent ; nous considérons la présence d'une entité malhonnête dans l'environnement et donnons une spécification pour un modèle d'intrus qui agit durant une session de négociation afin de définir et de vérifier des propriétés d'authenticité et de confidentialité pour le modèle en la présence de cette entité malhonnête.

Chapitre 6 : Sur la spécification de la délégation et de la révocation dans un modèle de contrôle d'accès dynamique

La majorité des organisations possèdent des règles et des régularisations qui gouvernent leur politique de sécurité telles la séparation des tâches ou la délégation. La délégation est l'act de donner des droits et des responsabilités à un autre afin de faire des activités spécifiques. Une délégation peut concerner un droit, un ensemble de droits ou un rôle.

Dans notre représentation des propriétés de contrôle d'accès dans un environnement distribué, nous définissons des objets de contrôle d'accès comme étant des certificats pour représenter des tâches, des rôles ou des historiques d'activation au sein d'une entité donnée. Nous considérons que chaque entité est responsable d'une collection de tels objets de contrôle d'accès. Par exemple, une entité qui représente le département de comptabilité est responsable des rôles *comptable* et *chef comptable*, et des tâches *calculer* et *vérifier*. Ainsi, cette entité sera responsable de toutes les opérations (délégation, héritage de rôles, activation de rôles, etc.) concernant ces objets. La décentralisation de l'information sera ensuite gérée par la politque de négociation. En d'autres termes, pourqu'une entité puissse recevoir des informations sur un objet de contrôle d'accès donné, cette entité doit initier une session de négociation avec l'entité responsable de cet objet. En conséquence, il n'est pas nécessaire de définir une entité centrale pour gérer la délégation ou l'assignation de rôles, en effet il suffit de définir les conditions d'utilisation d'un objet dans l'entité responsable de cet objet et donc de gérer la diffusion de cette information à travers la politique de négociation.

Ce chapitre est divisé en deux parties indépendantes. La première partie présente une vue d'ensemble de la délégation. Dans la section 6.1 nous présentons les différents aspects de la délégation et de la révocation dans la litérature et nous offrons une petite étude sur l'expression de la délégation et de la révocation dans différents modèles de contrôle d'accès.

Le deuxième partie est consacrée à la présentation de notre approche pour modéliser les conceptes de délégation et da révocation. Dans la section 6.2 nous présentons le contexte de la délégation en donnant les éléments nécessaires pour la définition des règles pour la délégation et la révocation. Dans la section 6.3 nous donnons la forme générale des règles de délégation et de révocation sous forme de règles de contrôle d'accès exprimées dans notre modèle. Dans la section 6.4 nous présentons la définition des processus pour les différentes tâches de délégation et de révocation. Dans la section 6.5 nous ajoutons au modèle les règles nécessaires pour exprimer l'acquisition des droits délégués et la propagation de la délégation. Enfin, la section 6.6 offre un exemple qui illustre l'utilisation de la délégation dans un système distribué.

Chapitre 7 : Contraintes de séparation de tâches

Dans le chapitre 6 nous avons présenté différents aspects de délégation dans un contexte dynamique. Cependant, la délégation, ainsi que la possibilité d'activer et de déactiver des sujets dans des rôles, augmentent les chances d'avoir des failles de sécurité ou de collaboration entre des utilisateurs travaillant sur des tâches sensibles, et qui peuvent entrainer à des situations de fraude. L'expression d'une politique de contrôle d'accès d'une organisation donnée doit donc prendre en compte les contraintes sur de telles tâches sensibles. De telles contraintes de sécurité concernent principalement les propriétés de séparation de tâches où des tâches distinctes ne peuvent pas être exécutées par le même utilisateur.

Dans la section 7.1 nous présentons une vue d'ensemble des différents aspects de contraintes de séparation de tâches et leur exécution des différents modèles de contrôle d'accès. Dans la section 7.2 nous présentons notre approche pour la modélisation des différentes contraintes de séparation de tâches, et dans la section 7.3, nous présentons des propriétés de sécurité qui doivent être respecté au niveau global du système.

Chapitre 8 : Coder RBAC

Dans notre modèle nous avons pris le choix de ne pas avoir une structure RBAC explicite. Cela nous a permis de gagner plus de flexibilité dans l'expression de décisions de contrôle d'accès en se basant sur la notion d'attribut au lieu d'une structure prédéfinie basée sur les rôles. Ce choix nous permet d'exprimer une diversité de modèles et de contraintes sans étendre ou modifier notre langage. Cependant, il est possible d'intégrer le modèle RBAC et les structures de rôles dans notre cadre formel. Dans ce chapitre nous présentons une formalisation de la structure RBAC et de ses propriétés. Nous donnons une modélisation de RBAC dans un environnment distribué en prenant en considération la présence de l'hiérarchie de rôles et en exprimant les notions d'activation de rôles, ainsi que la délégation et les contraintes de séparation de tâches concernant les rôles.

Chapitre 9 : Un modèle d'intrus pour la négociation

Nous avons présenté un modèle logique pour définir et étudier des politiques de sécurité dans un environnement distribué en utilisant le principe de négociation de certificats. Nous considérons l'existence dans l'environnement d'un nombre d'entites chacune ayant un ensemble d'objets. Une politique de négociation régularise l'accès à ces objets par d'autres entités. Ainsi, étant donné un ensemble de politiques de négociation des entités, l'ensemble d'objets disponibles à une entité donnée est généré à partir du calcul du plus petit point fixe. Tout au long de cette thèse nous avonc considéré le cas où toutes les entités sont honnêtes et suivent leur politique de négociation, dans ce chapitre nous étudions le cas où une entité malhonnête peut intervenir dans une session de négociation.

Dans [33] Dolev et Yao définissent la notion d'intrus symbolique pour représenter les capacités d'un agent malhonnête essayant d'attaquer un protocol de communication sécurisé cryptographiquement. Dans ce chapitre nous présentons une adaptation de cet intrus qui garde les mêmes capacités de déduction mais qui est spécialisé dans l'analyse des échanges durant une session de négociation. Cela nous permet en particulier d'analyser la sécurité d'une politique de sécurité distribuée en fonction de cet agent malhonnête.

Nous étendons la syntaxe de notre langage pour prendre en considération la présence de canaux de communication entre les entités. Les certificats sont envoyés sur ces canaux spécifiés par leur propriétés par rapport à l'intrus. De plus, l'intrus agit comme un intrus Dolev-Yao [33] : il peut intercepter les objets sur un canal public, le remplacer par un objet qu'il construit à partir de ses connaissances et des objets qui lui sont envoyés par d'autres entités durant la même session ou une session différente. Un intrus peut aussi bloquer l'accès à des objets pour des entités en fonction de la nature de la communication. Un intrus peut aussi lire des informations dans des objets et créer de nouveaux objets. Enfin, nous considérons qu'un intrus peut composer, décomposer, crypter et décrypter des messages pour construire de nouveaux objets.

Dans notre modèle un protocol est défini par un ensemble de règles de négociation. Nous modélisons l'intrus par la définition d'un ensemble de règles de négociation qui lui permettent d'avoir le maximum de contrôle sur la comunication durant une session de négociation. Nous définissons les actions de l'intrus au niveau de sa capacité d'envoyer et de recevoir des objets au nom des autres et sa capacité de lire, signer et créer des objets à partir des objets interceptés. Notre objectif est de prendre en compte les propriétés des cannaux de communications ainsi que les primitives cryptographiques utilisées pour la sécurisation des messages afin d'analyser la structure du mécanisme de négociation.

Nous présentons les bases du langage dans la section 9.2, et notre modélisation des canaux sécurisés dans la section 9.4. Le modèle d'intrus est présenté dans la section 9.5, et l'adaptation de l'algorithme de négociation pour le prendre en compte est présenté dans la section 9.6. Dans la section 9.7 nous présentons les propriétés de sécurités et dans la section 9.8 nous donnons notre modélisation de l'implémentation de *SAML Single-Sign On* de Google.

Chapitre 10 : Conclusion et perspectives

Conclusion

Les propriétés de contrôle d'accès standards régularisent uniquement l'accès aux fonctionnalités d'un service et donc ne sont pas capable d'exprimer des décisions de contrôle d'accès qui dépendent du contexte d'exécution. Le premier objectif de cette thèse était de développer un langage logique pour exprimer des politiques de contrôle d'accès complexes dans un environnement distribué et qui est capable de modéliser le contrôle d'accès dynamique et de garantir la communication entre les différentes entités au niveau d'échange de messages et au niveau de l'établissement de confiance. Le second objectif de cette thèse était d'exprimer les propriétés de contrôle d'accès et de spécifier les propriétés de sécurité qui contribuent au raisonnement sur la sûreté des problèmes de sécurité au niveau du contrôle d'accès ou au niveau de la communication.

Nous avons présenter dans le chapitre 3 un cadre logique qui utilise la structure RBAC pour définir des règles de contrôle d'accès, et un système de transition qui définit les effets des décisions de contrôle d'accès comme une collection des permissions et d'obligations qui depend du choix de l'utilisateur concernant l'activation des actions autorisées. Ainsi dans cette première approche nous avons considéré que l'état de sécurité évolue par rapport aux décisions de contrôle d'accès et du choix effectué sur l'activation de ces autorisations. Cependant, lorsque nous avons essayé d'exprimer des études de cas qui concernent notamment la modélisation de *business processes*, nous avons trouvé des difficultés pour adapter la structure RBAC aux critères qui prennent en compte d'autres éléments de sécurité comme les caractéristiques propres à l'objet et non pas au sujet de la décision de contrôle d'accès. De plus, cette première approche ne garantie pas explicitement la communication entre différentes entités dans un environnement distribué.

Dans le chapitre 4, nous nous sommes intéressés à un modèle logique basé sur les attributs regroupant une politique de contrôle d'accès, une politique de négociation et un *workflow*. L'interaction entre la politique statique, exprimée par la politique de négociation, et la politique dynamique, exprimée par la politque de contrôle d'accès et le *workflow*, offre un outil efficace pour modéliser

l'évolution de l'état d'une entité en fonction de son interaction avec le reste de l'environnement. La sémantique d'évaluation des règles de négociation basé sur l'évaluation d'un plus petit point fixe permet d'évaluer l'état de sécurité courrant de chaque entité. L'évaluation des règles de contrôle d'accès dépend du résultat de la session de négociation et donc fournit des décisions de contrôle d'accès en fonction de l'état de sécurité. Enfin, les règles de contrôle d'accès peuvent être vues comme des gardes sur l'exécution du *workflow*, en effet une action peut être exécutée si elle est autorisée par les règles de contrôle d'accès et exécutable dans le *workflow*. Pour démontrer l'expressivité de ce formalisme et pour montrer l'utilisation du langage et l'interaction entre les différentes entités qui simule un *business process*, nous avons présenté dans le chapitre 5 une modélisation du processus d'enregistrement de voiture.

La seconde partie de cette thèse présente les propriétés de contrôle d'accès au niveau de la sécurité des accès pour chaque entité et au niveau de la sécurité des communications entre les différentes entités dans l'environnementau niveau de la négociation. Ainsi, les chapitres 6 et 7 étaient consacrés à l'étude de la délégation et des contraintes de séparation de tâches. Nous avons présenté différentes perspectives pour l'expression de ces propriétés ainsi que les limitations trouvées dans les travaux existants. Nous avons montré qu'à la différence de la majorité des travaux existants, nous somme capable de coder la plus grande partie des aspects de la délégation qui existent dans la litérature, nous avons aussi montré que notre langage peut exprimer les spécifications pour les contraintes de séparation de tâches au niveau de la concepetion de la politique de sécurité mais aussi au niveau de l'execution sans modifier la syntaxe du langage.

En effet, l'information sur l'utilisation d'une délégation ou l'utilisation d'une tâche en général est évaluée locallement dans l'entité responsable pour cette délégation ou l'exécution de la tâche. Cette information est nécéssaire dans le contexte de l'évaluation et de la vérification des contraintes de sécurité. Pour cela, l'historique de la délégation ou de l'exécurtion des tâches est suvegardée par l'ajout d'un objet dans le *repository* de l'entité. Les décisions de contrôle d'accès concernant l'autorisation des droits délégués ou l'exécution des contraintes de séparation de tâches dépendent du résultat de requêtes dans le *repository* de l'entité sur l'existence (ou non) de records de délégation ou d'exécution de tâches.

Le cas de la délégation est simple. En effet, nous considérons que chaque entité prend les décisions pour les délégations concernant ses propres fonctionalités. La politique de négociation de cette entité peut ainsi gérer la diffusion des informations concernant ces délégations sous forme d'objets aux autres entités. Le cas des contraintes de séparation des tâches est plus compliqué. La difficulté dans ce cas réside dans le fait que les requêtes concernent plutôt la non-existence d'objets qui ne peut être faite que localement dans une entité donnée. Cependant, bien que la définition des contraintes de sécurité au niveau du modèle global n'est pas encore faisable, nous avons défini les propriétés de sécurité sous forme de problèmes de décision afin de pouvoir vérifier la sûreté du modèle. Un codage de RBAC et différentes extensions concernant la hierarchie de rôles, l'activation de rôle et la délégation ont été présentés dans le chapitre

8.

Dans le dernier chapitre de cette thèse nous nous sommes intéressés à un différent aspect de sécurité. En effet, comme l'objectif principal de cette thèse était de définir un modèle pour l'expression de politiques de contôle d'accès dans un environnement distribué, il était important d'étudier le problème de la sécurité des communications entre les différentes entités. Pour cela nous avons présenté une extension du langage qui prend en compte les propriétés de sécurité des canaux de communication et nous avons considéré l'existence d'une entité malhonnête dans l'environnement qui agit au niveau de la négociation. Ainsi nous avons défini un modèle d'intrus comme étant une entité logique avec des règles spécifiques, et nous avons donné les spécifications pour les propriétés d'authenticité et de confidentialité du système en la présence de cet intrus.

Perspectives

Les travaux futures peuvent être explorés dans trois principales directions ; l'orchestration des services, le *model checking* dans un environnement distribué, et l'implémentation de ce modèle dans un outil pour la modélisation et la validation des politiques de contrôle d'accès.

Sur l'orchestration

Dans le chapitre 4 nous avons défini un modèle pour exprimer le contrôle d'accès dans un environnement distribué et dans le chapitre 5 nous avons présenté un *business process* pour illustrer le processus d'enregistrement de voitrue. Ce *business process* est représenté comme une collection d'entités, chacune avec ses propres règles de contrôles d'accès, ses règles de négociation et son *workflow*. Les entités communiquent les unes avec les autres à travers la négociation d'objets mais aussi à travers l'échange de requêtes nécessaires pour la coordination de l'exécution des *workflows* locaux et donc du *business process* complet. Cependant, dans cet exemple, pour pouvoir simuler un *business process* complet, nous avons présenté une modélisation explicite du processus d'envoi et de réception de requêtes entre les différentes entités.

Il serait intéressant de faire une abstraction de cet encodage explicite en considérant la présence d'un orchestrateur dont le rôle est de gérer l'interaction entre les différentes entités en contrôlant l'échange de requêtes. L'orchestrateur pourrait aussi prendre en considération les contraintes de sécurité spécifiées par le système (à travers la définition d'un *business process*). Cela peut être possible en délégant le contrôle sur les tâches sensibles (qui ne peuvent pas être contrôler par la même entité) à l'orchestrateur. Ainsi l'orchestrateur peut être vu comme une entité logique avec un ensemble de règles de sécurite et un *workflow*, et qui est responsable de la définition de la politique de contrôle d'accès globale du *business process* d'une part, et gère l'échange de requêtes entre les différentes entités à travers la définiiton de son *workflow* d'autre part.

Sur le *model checking*

Dans le chapitre 7 nous avons donné les spécifications pour la vérification des contraintes de sécurité dans l'environnement en définissant un moniteur qui après chaque exécution du *workflow*, vérifie s'il y a eu des violations de sécurité. Cependant dans le modèle existant, un tel moniteur peut être vue en temps que garde sur le bon fonctionnement du système uniquement. Il serait intéressant de permettre à ce moniteur dans le cas où il y aurait une violation, de pouvoir retrouver la trace de cette violation et donc de permettre à l'auteur de la politique de modifier la politique de sécurité de l'entité en question afin de remédier au problème. Cela n'est désormais pas simple à accomplir, en effet, nous devons redéfinir le modèle afin de pouvoir prendre en compte l'historique pour retracer la source de la violation. Pour cela, il est nécéssaire de garder une historique des exécutions du *workflow* d'une part, et de contrôler les autorisations de contrôle d'accès qui peuvent être aquises d'une manière décentralisée à travers la négociation d'objets avec d'autres entités d'autre part.

Sur l'implémentation

La définition d'un cadre formel pour exprimer des politiques de sécurité est nécessaire pour analyser les propriétés de sécurité de ces politiques. Cependant, il est important d'implémenter ce modèle dans un outil pour exprimer et valider les politiques de contrôle d'accès. Il existe déjà dans le cadre du projet européen AVANTSSAR, un langage formel ASLan qui permet de spécifier des politiques de contrôle d'accès. Il est basé sur des règles de réécriture pour exprimer les *workflows*, des clauses de Horn pour l'expression des politiques de sécurité et la logique linéaire temporelle pour spécifier les propriétés de sécurité. De plus, des outils existent déjà qui permettent d'implémenter et de valider les modèles de sécurité exprimés en ASLan.

Dans notre modèle, les règles de sécurité sont exprimées dans une logique de premier ordre et le *workflow* est défini comme un système de transition à l'aide de la définition de processus. Ainsi, il serait intéressant de réduire les règles de sécurité de notre modèle en clauses de Horn afin de pouvoir utiliser ces outils existants.