Conflict Management in Usage Control Policies

Nada Essaouini

THÈSE / Télécom Bretagne
sous le sceau de l'Université européenne de Bretagne

pour obtenir le grade de Docteur de Télécom Bretagne
En accréditation conjointe avec l'Ecole doctorale Matisse et en co-tutelle avec l'Université Cadi Ayyad – Ecole Nationale des Sciences Appliquées de Marrakech
Mention : Informatique

présentée par
Nada Essaouini

préparée dans le département Logique des usages, sciences sociales & sciences de l'information
Laboratoire Labsticc

Conflict Management in Usage Control Policies

Thèse soutenue 14 janvier 2015
Devant le jury composé de :

Said El Hajji
Professeur, Université Mohamed V Agdal – Rabat - Maroc / président

Olivier Boissier
Professeur, Mines Saint-Etienne / rapporteur

Ludovic Apvrille
Maître de conférences, Télécom ParisTech / rapporteur

Anas Abou El Kalam
Professeur Habilité, ENSAM de Marrakech - Maroc / examinateur

Hanan El Bakkali
Professeur, Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes – Maroc / examinateur

Abdellah Ait Ouahman
Professeur, ENSAM de Marrakech – Maroc / examinateur

Nora Cuppens
Directrice de Recherche, Télécom Bretagne / examinateur

Frédéric Cuppens
Professeur, Télécom Bretagne / directeur de thèse

N° d'ordre : 2015telb0340

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En accréditation conjointe avec l'Ecole Doctorale Matisse

Co-tutelle avec l' Ecole Nationale des Sciences Appliquées de Marrakech

Ecole Doctorale – MATISSE

Conflict Management in Usage Control Policies

Thèse de Doctorat

Mention : Informatique

Présentée par **Nada Essaouini**

Département : LUSSI

Laboratoire : Lab-STICC

Directeur de thèse : Frédéric Cuppens

Soutenue le 14 Janvier 2015

Jury :

M. Olivier Boissier, Professeur, Ecole des Mines de St-Etienne, France (Rapporteur)
M. Ludovic Apvrille, Professeur Assistant, Telecom ParisTech, France (Rapporteur)
Mme. Nora Cuppens, Directrice de Recherche, Telecom Bretagne, France (Examinateur)
M. Abdellah Ait Ouahman, Professeur, ENSA de Marrakech, Maroc (Examinateur)
M. Said El Hajji, Professeur, Université Mohammed-V, Maroc (Examinateur)
M. Anas Abou El Kalam, Professeur Habilité, ENSA de Marrakech, Maroc (Examinateur)
Mme. Hanane El Bakkali, Professeur Habilité, ENSIAS, Maroc (Examinateur)

Abstract

The security policies are commonly specified through permissions, prohibitions and obligations. Permissions and prohibitions are generally used to specify access control policies, while obligations are useful to express usage control policies. Two different types of obligations are generally considered, namely system obligations and user obligations. User obligations are associated with deadlines. When these obligations are activated, these deadlines provide the user with some time to enforce the obligation before violation occurs. Using obligations with deadlines in security policies may cause a new type of conflict. This kind of conflict could happen in presence of overlapping deadlines. In this thesis, we propose a language based on a deontic logic of actions to express permissions and obligations. The semantic of the proposed language is defined using the situation calculus formalism. This allows us to analyze decidability and complexity of several problems such as planning tasks. We then use the planning task to prove the existence of conflictual situations. Once the conflicts are detected, we use delegation to redistribute obligations in order to resolve these conflicts and thus avoid possible violations. Furthermore, we show that obligations and permissions are not sufficient to preserve interests of the system's users. Indeed, for fairness reasons, the possibility of executing actions achieving their interests should be always preserved. Otherwise, violations are triggered. However, these actions are not obligations since users have the choice to execute them or not. Thus, the violation is the consequence of system's failure in preserving the choice of users. Consequently, we enriched our model with right rules to enable this feature. Finally, we show how the use of rights allows the refinement of responsibility when conflicts occur.

Keywords: Computer security, Security policy, Access and usage control, Obligation with deadline, Situation calculus, Conflict management, Planning.

Résumé

Les politiques de sécurité s'expriment en général par des règles de permissions et d'interdictions. Plus récemment, les spécifications et mises en oeuvre des règles d'obligation commencent à voir le jour, notamment pour exprimer des politiques de contrôle d'usage. Dans cette thèse, nous proposons un langage reposant sur les modalités déontiques pour spécifier des politiques d'obligations avec délais. Ce modèle est intégré dans le langage du calcul des situations séquentiel temporel. Le modèle permet de prouver si un ensemble d'obligations actives dans une situation donnée est globalement satisfaisable ou non. La démarche repose sur une recherche de planification des obligations. Le modèle permet aussi d'exprimer les permissions et analyser un autre type de conflit lorsqu'il est impossible de trouver un plan d'actions permises qui permet de remplir les obligations avec deadline. Le modèle permet aussi de spécifier un ensemble de contraintes associées à la politique de sécurité. La démarche permet de prouver que les contraintes seront toujours satisfaites. Finalement, nous avons étendu notre modèle pour définir une politique incluant des règles de droit. La sémantique proposée permet de formaliser la différence entre permission et droit. Cette distinction permet de prouver si un conflit dans une situation donnée provient d'une anomalie dans la politique ou si elle relève de la responsabilité d'un utilisateur. De plus, le modèle formalise une propriété d'équité dans le jugement des responsabilités. Lorsqu'un utilisateur a la possibilité de changer son comportement pour éviter un conflit, il est considéré responsable. Le modèle permet également de formaliser les situations de responsabilité partagée.

Mots Clés : Sécurité informatique, Politique de sécurité, Contrôle d'accès, Contrôle d'usage, Obligations avec délais, Calcul des situations, Gestion des conflits, Planification.

Contents

1 Introduction **1**

1.1 Motivation and background 1

1.2 Contributions 2

1.3 Outline of the dissertation 3

2 Preliminaries and State of the Art **5**

2.1 Introduction 5

2.2 Access control models 5

2.2.1 Identity based access control model 5

2.2.2 Role-based access control model 7

2.2.3 Task based access control model 8

2.2.4 View based access control model 8

2.2.5 Team-based access control model 9

2.2.6 Dynamic and contextual authorization models 9

2.2.7 Organization based access control model 10

2.3 Usage control models 11

2.3.1 Policy rule management including provisions and obligations . . 12

2.3.2 Privacy policies in access control model 13

2.3.3 The $UCON_{ABC}$ model 13

2.3.4 Obligation specification language 14

2.3.5 Enforcement and management of obligation policies 14

2.4 Managing policy conflicts 14
2.4.1 Access control policy analysis techniques 15
2.4.2 Obligation policies analysis techniques 16
2.5 Situation calculus 17
2.5.1 The language 17
2.5.2 fundamental axioms 19
2.6 Conclusions 21

3 Specification of Obligation with Deadline Policies 23
3.1 Introduction 23
3.2 Motivation example 24
3.2.1 Impact of deadlines to complete medical records on the availability of information 24
3.2.2 Rules regarding the completion of the patient's medical record . 25
3.3 Security policy specification 26
3.3.1 Specification using the deontic logic of actions 26
3.3.2 Example of rule specification 26
3.4 Actual norm derivation and violation detection 28
3.4.1 The semantic of actual permission 28
3.4.2 The semantic of active obligation 30
3.4.3 Fulfillment and violation detection 31
3.4.4 How the situation calculus helps us by resolving the frame problem 33
3.5 Conclusion and Contribution 35

4 Specifying and Enforcing Constraints Policy in a Dynamic World 37
4.1 Introduction 37
4.2 Specifying constraints 38
4.2.1 Example 39
4.3 Enforcing constraints 42

4.3.1 Enforcing ahistorical constraints . . . 42
4.3.2 Rewrite historical constraints in the form of simple state constraint 43
4.3.3 Enforcing the historical constraints . . . 44
4.3.4 Defining and characterizing a secure system . . . 44
4.3.5 Example . . . 45
4.3.6 Expressive power . . . 47
4.4 Related work . . . 50
4.5 Conclusions and contribution . . . 51

5 Conflict Detection in Obligation with Deadline Policies 53
5.1 Introduction . . . 53
5.2 Motivation example . . . 54
5.3 Conflict detection . . . 55
5.3.1 Conflict in feasibility of obligations . . . 57
5.3.2 Conflict between permission and obligation rules . . . 58
5.3.3 Conflict detection algorithm . . . 58
5.4 Implementation . . . 59
5.5 Related work . . . 65
5.6 Conclusion and Contribution . . . 67

6 Enriching Obligation with Deadline Policies with Rights 69
6.1 Introduction . . . 69
6.2 Motivation example . . . 71
6.3 Specification of right rules . . . 72
6.4 Derivation of effective rights and detection of rights violation . . . 73
6.4.1 Derivation of effective rights . . . 73
6.4.2 Violation detection . . . 74
6.5 Conflicting situations . . . 76

6.5.1 Conflict in exemplarity of feasibility of obligations 76
6.5.2 A conflict in feasibility of rights 78
6.5.3 A conflict between obligations and rights 81
6.6 Related work and discussion 82
6.7 Conclusion and contributions 84

7 Conflict resolution based on delegation and renunciation of right 87
7.1 Introduction 87
7.2 Detected conflict and responsibility 88
7.2.1 Situations where the conflict could have been avoided 89
7.2.2 Shared and multiple responsibilities 92
7.3 Possible conflict resolution 93
7.3.1 Obligation delegation 93
7.3.2 Renunciation of its own rights 95
7.3.3 Solvable conflict 96
7.4 Conclusion and contributions 97

8 Conclusion and perspectives 99
8.1 Perspectives 100
8.1.1 Delegation based on workload assigned to users 101
8.1.2 Delegation based on user reputation 102

A Actual norm derivation of the case study described in chapter 3 105

B Prolog file: ConflictDetection.pl 107

C Gestion Des Conflits dans les Politiques de Contrôle d' Usage 113

List of Publications 119

List of Figures 131

CHAPTER 1

Introduction

1.1 Motivation and background

In Information systems used in many domains like health care information systems, it is important that information is accessible only to authorized users (confidentiality property) and must protect the accuracy and integrity of information and treatment methods (integrity property). It is also important that authorized users can access to information when they need it (availability property). Security of an information system is characterized by enforcement of these three properties which are guaranteed through a set of rules defined in the security policy of this system.
The security policies are often defined as permissions, prohibitions and obligations. Permissions and prohibitions are generally used to specify access control policies. As suggested in many works before, obligations are useful to express usage control policies [Cuppens et al. 2005, Elrakaiby et al. 2012a]. The application of these rules to the same object may lead to conflicting situations. Preliminary work on the classification of conflicts are reported in [Moffett and Sloman 1993], where several types of conflicts have been defined (see also [Bertino et al. 1996, Dinolt et al. 1994]). [Benferhat et al. 2003] presents an approach based on possibilistic logic to deal with conflicts in prioritized security policies. However, there is another type of conflict which is not managed yet, namely the conflict between obligations with deadlines. This kind of conflict could happen in the case of overlapping deadlines. Thus, one of the objectives of this thesis is to provide a model for expressing the rules of obligations with deadline, detect violations and conflicting situations.
Furthermore, obligation and permission rules are not sufficient to express certain interests. Indeed, there are actions representing the interests of users in the information system which should always be possible to be executed in order to protect those interests, otherwise violations are triggered. These actions are not obligatory, insofar as there are no violations due to non-execution of these actions. However, the failure to ensure their execution induces a violation. This need could be formulated using

the rules of right. For example, CNIL (Commission National de l'informatique et des libertés) [1] gives a list of rights and made provisions of steps to follow if these rights are violated[2]. However, the use of right rules in the security policies causes a new type of conflict which is not yet treated in the literature. It is a conflict that arises when the execution of a right prevents an obligation from being fulfilled within its deadline or conversely, fulfilling an obligation within its deadline necessarily leads to the violation of a right. Thus, another goal of this thesis consists in providing a formal framework which first gives a semantics for rights, then enables detection of right violations and conflicts between rights and obligations.
Nevertheless, a conflict in a given situation does not come necessarily from security policies. Sometimes it could be avoided if users had adopted a different behavior within an interval of time before that conflict occurs. Another goal of this thesis is to specify when it is possible to challenge the actions executed by users in order to determine responsibilities concerning conflictual situations.

1.2 Contributions

First, we provide a language based on deontic modalities to specify obligation policies with deadline, and we present a model based on temporal sequential situations calculus to give semantics to our language. Our model enables us to prove if active obligations in a given situation are globally enforceable or not using the planning task. This allows us to detect a conflict between obligations with deadline in usage control policies. This work was published in [Essaouini et al. 2013]. Our model is then extended in [Essaouini et al. 2014a] to express permissions and system obligations. This induces a new type of conflict which occurs when it is impossible to find a plan of permitted actions to achieve a situation where obligations are fulfilled in their deadlines. We provide an algorithm for searching situations where this kind of conflict arises. In the plan search, the choice of the execution time of the elected action obeys to equations and inequalities over the reals which needs to be solved. For this purpose, we use a component allowing temporal reasoning and SIMPLEX resolution. To illustrate our approach, we take an example inspired from existing laws in hospitals regulating deadlines for completing patient medical records. The example is formally specified in our language and implemented in ECRC Common Logic Programming System ECLIPSE

[1] An independent French administrative authority. It is responsible for ensuring that information technology is at the service of citizens and it does not affect human identity, nor the rights or privacy, or individual and public freedoms. It carries out its tasks in accordance with Law No. 78-17 of 6 January 1978 amended August 6, 2004

[2] http://www.cnil.fr/vos-droits/plainte-en-ligne/

3.5.2, which is equipped with Simplex algorithm for solving linear equations and inequalities over the reals. In the implementation, we show how the plan search can be optimized through the use of some heuristics and make some evaluation tests.
Furthermore, constraints in access control models are used to organize access privileges in order to avoid fraudulent situations. Ensuring that the constraints are satisfied during the evolution of the system is an important issue. In our work [Essaouini et al. 2014b], we extend our model to express the constraints policy, this allows us to have a formal reasoning language in order to prove that the constraints are always satisfied. The proposed language can be used to specify various constraints mentioned in the literature. In addition, we formally specify the condition to prove that the system specification is secure with respect to the access control requirements. Finally, our model is enriched by rules of right. We give semantics to the right through the difference that we make in our language between permitted actions and possible actions. In our model, we can prove if a conflict in a given situation comes from the policy or from the users responsibility. The behavior of users that make them responsible for conflict can be due to intervals of time exploited wrongly. It can also be the result of execution of some actions which prevent the users from fulfilling some of active obligations for example. This leads to a situation where several obligations have been accumulated and become impossible to fulfill within their deadlines. However not all actions may be revoked. We cannot revoke an action for which there is a right to execute it. This is because we consider it is unfair to revoke a right, which has been exercised and also because a right which is not preserved leads to penalties as well as failure to fulfill obligations within their deadlines. This is another interest for enriching a security policy by a new modal operator of right that our model can express. In addition our model ensures a property of fairness in the judgment of responsibilities. To the extent that every time there is a user who could change its behavior to avoid conflict, he is considered responsible. Our model also allows us to formally express a shared responsibility.

1.3 Outline of the dissertation

In this thesis, we first present in chapter 2 the situation calculus and state of the art related to access control models and usage control policy languages. In chapter 3, we explain how to define security policies that include obligations with deadline. This model is based on deontic logic, and a security policy is viewed as a set of deontic norms. In this chapter we extend situation calculus to formally derive which actual norms apply in a given situation. We also formally define when an obligation with

deadline is violated. In chapter 4, we extend our model to support constraints in access control policies. We show how this language is adequate to express well-known constraints in the literature. Then, we give a procedure based on the regression concept to enforce these constraints. In chapter 5, we show how to detect the presence of conflicting norms in the policy. We give the specification of a motivation example and show how our implementation, using the programming language GOLOG, detects conflicting obligations in a given situation. In chapter 6, we show how right rules can enrich security policies and allow the formulation of interests which can not be formulated using permission and obligation rules. In this chapter, we also formally define when a right is violated and show how to detect the presence of another kind of conflict that occurs when right rules are included in access control policies. We propose in chapter 7 an approach which provides means to determine if there are users responsible for conflictual situations. In this chapter we propose a possible solution to solve conflict based on delegation of obligations and renunciation of rights. Finally, discussions and perspectives are presented in chapter 8.

CHAPTER 2

Preliminaries and State of the Art

2.1 Introduction

A security policy specifies security requirements through permissions, prohibitions and obligations. A logic language is used to express in a clear and unambiguous way these rules in order to reduce the complexity of the specification and facilitate its understanding. A security model provides the elements necessary to analyze, validate and enforce security policies. In particular, a security model is useful to verify that a policy is complete and consistent. Furthermore, it can be used to verify that the implementation by system complies with the expected properties. In this chapter, we give an overview of current security models and policy languages while discussing their limitations. We start by analysing access control models in section 2.2. In section 2.3, we discuss obligation and usage control policy languages and models. Some policy analysing techniques are presented in section 2.4. Finally, we present in section 2.5 an overview of situation calculus, the formalism language that we use in the remainder of this thesis.

2.2 Access control models

2.2.1 Identity based access control model

We call IBAC (Identity Based Access Control) the first access control model proposed in the literature [Lampson 1971]. This model introduces the basic concepts of subject, object and action:

- The subject is the active entity of an information system (IS). Usually, it denotes a user or an application which runs on behalf of a user.

- The object is the passive entity of IS. It denotes an information or a resource to which a subject can access by performing an action.
- The action denotes the desired effect when the subject accesses an object. One can for example read or update the information in an object or copy the content of an object into another object.

The objective of IBAC model is to control the access of subjects to objects through the use of actions. This control is based on the identity of a subject and the object identifier.
The IBAC model introduces the concept of authorization policy. In IBAC, an authorization policy is the set of positive authorizations (or permissions) with the following format: a subject has the permission to perform an action on an object. The specification of negative authorizations (or prohibitions) are introduced in more recent access control models.
In fact, in IBAC, it is implicitly assumed that the access control policy is closed. This means by default all access are prohibited. The authorization policy specifies permissions and an access is denied if the authorization policy does not enable to derive that the access is explicitly permitted.
To represent an authorization policy, the proposed model in [Lampson 1971] introduces another important concept: access control matrix, which will then be included in subsequent models (see for example [Harrison et al. 1976]). In an access control matrix, the rows and columns of the matrix correspond, respectively, to all subjects and objects of the IS. The elements of the matrix represent all the permissions that a given subject has on a given object.
The model of type IBAC is implemented in the most current operating systems such as Windows, Unix or Linux. To implement such a model in IS, the access control matrix is not directly implemented. In fact, there are two main approaches depending if the implementation is based on a decomposition in rows or columns of the matrix.
The decomposition in columns associates with each object, a descriptor called an access control list, which represents all subjects having access to the considered object. And for each subject, all the actions that this subject can perform on that object.
The decomposition in rows associates with each subject, a set of capabilities, representing the set of objects which can be accessed by this subject. And for each object, all the actions that the subject can perform on that object.
The IBAC model has an important limitation: the authorization policy could become quickly complex to express and manage. In fact, it is necessary to list the permissions for each subject, action and object. In particular, when a new subject or an object is created, it is necessary to update the authorization policy in order to define new

permissions associated with this subject or this object.
To overcome this problem, other models have been defined. All these models have in common to provide a more structured authorization policy expression. We present in the following sections, models offering respectively a structuration of subjects, actions and objects.

2.2.2 Role-based access control model

The RBAC model (Role Based Access Control, see [Sandhu et al. 1996]) proposes to structure the expression of authorization policy around the concept of role. A role is an organizational concept: roles are assigned to users according to the function that the user plays in the organization.
The basic principle of the RBAC model is to consider that permissions are directly associated with roles. In RBAC, roles receive authorizations to perform actions on objects. As IBAC, R-BAC model considers only positive authorizations (permissions) and therefore it is assumed that the policy is closed.
Furthermore, the RBAC model introduces the concept of session. To perform an action on an object, a user must first create a session and in this session, activates a role which has the permission to perform this action on this object. If such a role exists and the user was assigned to this role, then this user will have the permission to perform this action on this object once this role is activated.
When a new subject is created in the IS, it is sufficient to assign roles to this subject, thus this subject can access to the IS according to the permissions granted to this set of roles. Therefore, compared to the IBAC model, the management of authorization policy is simplified since there is no need to update this policy every time a new subject is created.
In general, any set of roles can be assigned to a user, and a user can activate in a session any subset of roles which was assigned to him/her. The RBAC model introduces the notion of constraint to specify authorization policies including more restrictive situations. Thus, a static separation constraint specifies that certain roles (e.g., nurse and physician) can not be assigned to a user simultaneously. A dynamic separation constraint specifies that, although some roles can be assigned to a user (for example liberal doctor and surgeon) these roles can not be active simultaneously in the same session.
In RBAC, it is also possible to organize roles hierarchically. Roles inherit permissions from other roles that are hierarchically below them. When a role r_1 is hierarchically superior to a role r_2, we say that r_1 is a senior role of r_2. Today the RBAC model is a standard [Ferraiolo et al. 2001]. Many IS implement this standard, for example Unix

Solaris from version 8 or the API Authorization Manager RBAC of Windows Server 2003.

2.2.3 Task based access control model

In IBAC and RBAC models, actions correspond generally to basic commands, such as reading the contents of an object or writing in an object. In recent applications, there is a need to control the execution of composite actions, called tasks or activities. The need to control composite activities is particularly present in Workflow applications [Atluri and Huang 1996].
The TBAC model (Task Based Access Control, see [Thomas and Sandhu 1998]) is the first model which introduced the concept of task. Then, other models were developed to monitor the implementation of activities in a workflow (see [Bertino et al. 1999, Atluri et al. 2000]). In particular, the user must obtain permission just in time when there is a need to continue the execution of the activity.

2.2.4 View based access control model

The RBAC and TBAC models introduced, respectively, the concepts of role and task in order to structure subjects and actions. To facilitate the expression and the management of authorization policy, there is also a need for a concept which enables to structure objects.
Among the access control models proposing a structuration of objects, we quote the security model proposed by SQL for relational databases. The expression of a security policy in SQL is based on the concept of view. We denote by VBAC (View Based Access Control) this kind of access control models.
Intuitively, in a relational database, a view is the result of an SQL query which was denoted by a given name. This concept of view is then used to structure the expression of authorization policy using GRANT statements (which allows granting a new permission to a user) and REVOKE (which allows deleting a permission that a user had). Thus, a view is an effective way to provide an access to all objects in the view. Note that these objects are sometimes virtual insofar as a SQL view is not materialized.
SQL/3 [Lentzner 2004], which is the latest evolution of the SQL standard, proposes to extend the VBAC model by combining the concepts of view and role in order to structure objects and subjects. Thus, the VR-BAC model is defined.
The concept of view is not limited to relational models. It could also be used in an oper-

ating system. Currently, most operating systems propose just the concept of directory in order to structure the expression of authorization policies.

2.2.5 Team-based access control model

In recent applications, it is often necessary to consider several organizations simultaneously. For example, in web services applications, a user of a certain organization may wish to access data belonging to another organization. An organization is a structured entity. For example, a hospital is an organization that is divided into several sub-organizations: the different departments of the hospital, the various services of these departments, etc. Each organization generally manages its own authorization policy. Some organizations may also be created dynamically depending on the activities that must support the hospital. For example, a health-care team can be created to support a particular patient. This organization could be deleted once the activities were made. Notice that the permissions of a subject not only depend on the role of the subject but also of the structure within which the subject performs its role. This problem was identified in the TMAC model. The TMAC (Team-Based Access Control, see [Thomas 1997]) model introduces the concept of team. In TMAC, permissions are associated with roles as well as teams. The subject's permissions result from the combination of permissions associated with the roles played by the subject and permissions associated with the team in which the subject is assigned. Various combinations (for example, the union of permissions) are considered. In fact, the TMAC model is incorrect because it introduces two binary relations: role-permission and team-permission. If the team concept is introduced, a ternary relation team-role authorization must be introduced. This is necessary to specify that authorizations do not depend only on role but also on the team in which this role is exercised. Using such a ternary relation, it is easy to specify the fact that the authorizations of the role doctor can change depending on the doctor is in the team of health-care or in the emergency team. This imperfection of the TMAC model has been corrected in the Or-BAC model, which we present latter in this section.

2.2.6 Dynamic and contextual authorization models

In practice, many permissions are not static but depend on contexts. When these contexts hold, the permissions are activated dynamically. This is called contextual authorizations.
Permissions may depend on temporal contexts, geographical contexts, or provisional

contexts (permission if other actions were previously performed as in the case of a workflow). Other types of contexts may be defined ([Cuppens and Miege 2003]).
To represent these contextual authorizations, several access control models based on rules have been proposed (model of type Rule-BAC, see [Jajodia et al. 2001, Bertino et al. 2001]). In these models, an authorization policy is considered as a set of rules of the form *condition* $\rightarrow$ *permission* which specify that a permission can be derived when a certain condition holds.
Models of type Rule-BAC are based on first-order logic which is, in general, undecidable. To deal with this problem, most of these models propose to use Datalog to get a decision procedure in a polynomial time. However rules must be conform to certain syntactic restrictions ([Ullman 1988]).
Compared to the models presented in the previous sections, Rule-BAC models have a greater expressiveness as it is possible to specify contextual authorizations. However, there is a lack in the structuration of the authorization policy which calls for the introduction of the concepts of role, activity, view and organization.

2.2.7 Organization based access control model

The OrBAC model (Organization Based Access Control, see [Kalam et al. 2003]) exploits the concepts of role, activity, view and organization introduced respectively in the RBAC models, TBAC, VBAC and TMAC. In OrBAC, an expression of authorization policy is focused on the concept of organization.
The concepts of roles, views and activities are organizational concepts. Each organization defines the roles, activities and views, in which it wants to regulate access by applying an authorization policy. Therefore, the OrBAC model introduces three relationships: relevant-role, relevant-activity and relevant-view, in order to specify respectively the roles, activities and views managed by the organization.
Thereafter each organization specifies assignments of subjects to roles using the ternary relation *empower*. We can notice that this model allows, for example, to consider that the same subject is assigned to different roles according to the concerned organization. Similarly, two other ternary relationship *consider* and *use* are used. The ternary relationship *consider* is used to specify, for each organization, the relationship between action and activity. The ternary relationship *use* is used to specify, for each organization, the relationship between object and view.
The OrBAC model also provides the ability to specify *role-definition*, *view-definition* and *activity-definition*. A role-definition is a logical condition that, if satisfied, leads to the conclusion that a subject is automatically assigned to the role corresponding to the role-definition. Similarly, a view-definition and an activity-definition correspond to

logical conditions which respectively manage views and activities.
Furthermore, the concept of context is explicitly introduced in the OrBAC model. The definition of context is a logical condition. When this condition is satisfied, it can conclude that the corresponding context holds. Note that each organization defines its own contexts. This is used to express a context which may vary from an organization to another.
Therefore, an authorization policy is expressed by specifying for each concerned organization, the activities that the roles have the permission to perform on the views and in which context. Thus, the authorization policy is expressed independently from the sets of the subjects, the actions and the objects managed by the IS. This is called organizational authorization policies.
The model proposes the rules of passage, to derive automatically, from the organizational authorization policy, the concrete permissions to apply to specific subjects, actions and objects. This rule specifies that in a given organization, if a subject is assigned to a certain role, an object is used in a certain view, an action implements some activity, and if the authorization policy of this organization specifies that the role is permitted to perform this activity in this view, then it can derive that the subject is permitted to perform this action on this object if the conditions corresponding to the context hold [Cuppens and Miege 2003].
The OrBAC model defines also role hierarchies (as in RBAC), in addition to the hierarchy of activities, views and organizations. Each of these hierarchies is associated with a permission inheritance mechanism [F. Cuppens and Miége 2004]. The definition of these hierarchies allows a more modular organizational expression of the authorization policy.
In conclusion, the Or-BAC model imposes the same syntactic constraints as imposed by Datalog. This allows to maintain a decision procedure computable in polynomial time even when considering contextual authorizations. Thus, the Or-BAC model provides an expressiveness comparable to the other models such as Rule-BAC models while providing a structured expression of the authorization policy.

2.3 Usage control models

Classical access control models specify whether a subject has the authorization to perform an action on an object of IS. Optionally, a contextual condition can be associated with the authorization. This condition must be met before the execution of the action. With the development of digital rights management applications (DRM), there is a need to specify conditions that must be met not only before but also during or after

an action was performed. For example, a server which enables listening music should be able to specify that the payment must be made before, during or after listening to a song. To express this kind of authorization policies, access control models are not sufficient. Therefore, usage control models start to be proposed. In this section, we present some of these frameworks which support obligations to meet usage control requirements.

2.3.1 Policy rule management including provisions and obligations

The model proposed in [Bettini et al. 2002b] extends rule-based access control models to support provisions and obligations. Provisions denote actions which a subject must perform before access, while obligations are actions which a subject will have to perform after access. This model makes a distinction between obligations and provisions by associating to each one of them a different set of predicates. The set of required provisions and obligations is calculated beforehand for each atom in the security rules. This set is denoted Global Provisions and Obligations Set (GPOS). Concerning the provision and obligation requirements corresponding to a given access request, they are calculated using the set GPOS, when the request is issued. If these requirements are satisfied, access is granted, otherwise, the user is informed about the provisions and obligations that he must meet to satisfy his/her request.
This framework has been extended later in [Bettini et al. 2002a, Bettini et al. 2003] to support obligations specification and enforcement. Obligation rules have been enriched by the specification of the set of actions to execute when the obligations are fulfilled. When an obligation is violated, this triggers another obligation and another set of actions should be taken.
Concerning the mechanism of obligation enforcement, the authors propose to set up triggers. When obligations are accepted by users, triggers are derived from the definition of obligations. These triggers will be used to check if the obligations are fulfilled. The derivation of the time when the obligations have to be verified is based on temporal reasoning techniques.

The specification of rules in this model is not uniform. In fact, each rule can specify any number of parameters. This complicates the interpretation of the policy because of the hierarchical definition of obligations.

2.3.2 Privacy policies in access control model

In [Ni et al. 2008], authors defined an obligation model to enable the specification of privacy requirements in access control models. This model introduced repeating and conditional obligations in addition to pre and post obligations.
The number of times that an obligation must be fulfilled is specified through variables in a temporal constraint. The temporal constraint specifies also the time interval in which the obligation must be fulfilled. The distinction between pre-obligation and post-obligation depends on the lower endpoint of this interval.

In this model, there is a hierarchical dependency between obligations and permissions. In fact, obligations are activated only if a request for access is issued. In addition, the semantic of conditional obligation is not completely formalized. Furthermore, the management and the enforcement of obligations in this model is complicated due to the lack of clarity in how the fulfillment and violation of obligations affects the state of policy.

2.3.3 The $UCON_{ABC}$ model

The $UCON_{ABC}$ model [Park and Sandhu 2004] is based on three components for usage control: Authorizations, oBligations and Conditions. The authorization component evaluates conditions related to the subject's and object's attributes. Obligations specify actions or operations that must be executed. The environmental conditions are specified through the condition component. The conditions that must be satisfied before access are verified by pre functional predicates. While conditions that must be true in current access are verified by ongoing predicates. Thus, if the pre-functional predicates are true, the access is allowed. When the ongoing functional predicates become false, the ongoing access is revoked. $UCON_{ABC}$ enables also the update of subject and object attributes as side effect of usage. This is called attribute mutability.
As pointed out in [Janicke et al. 2007], the lack of obligation monitoring and obligation fulfillment notions in $UCON_{ABC}$ explain why this model could fail to express properly a desired requirement. This is certainly the reason why this model does not support obligations which are not related to the usage session. Authors in [Janicke et al. 2007] discussed other limitations of $UCON_{ABC}$ as the fact that it does not support conflict detection and resolution techniques.

2.3.4 Obligation specification language

The Obligation Specification Language (OSL) [Hilty et al. 2007] enables the specification of restrictions on usages and mandatory actions. The language specification is based on typed set theory and First-Order Logic with equality. The semantics is defined over event traces with discrete time steps. The semantics formalize the fact that the trace of events (occurrence of actions) is consistent with the policy.
The OSL model does not formalize how states evolve when events occur. Thus, it is not useful for analysing usage control policies. Furthermore, OSL policies are difficult to specify and interpret.

2.3.5 Enforcement and management of obligation policies

In [Elrakaiby et al. 2012b], the authors present an obligation model based on the concepts of Event Condition Action. It supports pre and post-obligations as well as ongoing, and continuous obligations. The specification and the enforcement of sanctions for users that violate obligations, and re-compensation for users that fulfill their obligations are also supported in this model. To manage conflicts and lack of permissions, they propose to cancel obligations and the delay of obligations.
In contrast, the presented model does not manage accountability [Irwin et al. 2006, Irwin et al. 2008, Pontual et al. 2010a, Pontual et al. 2010b]. In this sense, the authors in [Pontual et al. 2011] give means for administrator to find the user responsible for violation. Furthermore, the model manages dependencies between obligations. This enables the administrator to find the obligations that can be affected by the obligations which have not been fulfilled.

2.4 Managing policy conflicts

Initially, access control models allowed only the expression of positive authorizations (permissions). Recently, models enable also to express negative authorizations (prohibitions). Combining positive and negative authorizations in access control policies is interesting for several reasons:

- Some authorization policies are easier to describe using prohibitions than permissions.
- When the authorization policy must be updated, sometimes it is easier to insert a prohibition than deleting an existing permission.

- The combination of permissions and prohibitions is a simple way to express rules with exceptions.

However, one problem arises when permissions and prohibitions are considered together in authorization policies [Lupu and Sloman 1999, Capitani et al. 2005, Cuppens et al. 2007]. Indeed, for a subject, an action and a given object, it is possible that the authorization policy allows deriving that the access is both permitted and forbidden. To solve these conflicting situations, several strategies have been proposed in the literature. In the following, we present some works from the literature wich deals with conflicts in access control policies. Furthermore, we discuss conflicts which arise when obligations are used in the policy language.

2.4.1 Access control policy analysis techniques

To solve conflicts between permissions and prohibitions, several approaches have been proposed in the literature. Some approaches (see for example [Jajodia et al. 2001]) consider only simple strategies, such as prohibitions always outweigh permissions, or permissions always outweigh prohibitions. These strategies are not suitable for addressing the problem of exceptions. Indeed, if a prohibition is an exception of a permission, then this prohibition shall prevail over this permission. Otherwise, if a permission is an exception of a prohibition, then the permission must prevail over the prohibition. To manage exceptions, several models (e.g., [Benferhat et al. 2003]) proposed to introduce priorities between the rules of the authorization policy. Most firewalls provide representative examples to implement such strategies. Indeed, the rules for filtering a firewall can be interpreted as a set of positive or negative authorizations. Generally, the priority of filter rules corresponds to the order in which the rules were written. Thus, when several rules are applied to a single packet, the final decision corresponds to the first applicable rule (first matching strategy). Therefore, the ordering of rules constitutes a simple, but effective, way to resolve conflicts in authorization policies which include permissions and prohibitions.
Concerning the detection of such conflicts, authors in [Becker and Nanz 2010] define a logic which extends datalog [Ullman 1988, Abiteboul et al. 1995] to specify state modifications after access requests. A proof system has been developed for this logic. This system allows determining the sequences of command to reach a particular system state.
However, the work in [Becker and Nanz 2010] considers only authorization policies. Therefore, it does not support obligations.

2.4.2 Obligation policies analysis techniques

Conflicts in obligation policies arise when a subject is imposed obligations, which cannot be fulfilled simultaneously. A conflict can also occur when a subject is obliged and prohibited to execute the same action. In this section, we first discuss how conflicts between system obligations can be resolved. Then, we present a framework which analysis policy conflicts when obligations are supported.

Conflicts between System-obligations

Authors in [Chomicki and Lobo 2001, Chomicki et al. 2003], present a framework based on the policy language PDL [Lobo et al. 1999] to resolve conflicts between system obligations.
An action constraint is used to specify actions and conditions under which these actions must never be executed together. There are two approaches to solve such kind of conflicts:

- Cancellation of some actions causing the conflict. The set of canceled actions must be minimal.
- The event cancellation strategy. This strategy consists of ignoring the events that caused the conflict.

In this work, a disjunctive logic program is used to formalize the semantics of conflict resolution. This program calculates the maximal sets of non-conflicting actions using the sets of events.
Sometimes it is enough to grant a delay on the execution of conflicting actions to resolve a conflict. This is the purpose of the work presented in [Chomicki and Lobo 2001]. This approach is based on the work previously presented in temporal action constraints [Chomicki 1995] where temporal action constraints were discussed. The specification of the actions previously executed is made using past temporal constraints.

Policy analysis in dynamic systems

The framework presented in [Craven et al. 2009] provide means to analyze authorization and obligation policies. In this work, authors distinguish the policy representation language from the domain description language. To describe dynamic domains, the Event Calculus [Kowalski and Sergot 1986, Miller and Shanahan 2002] is used. The policy and the domain description constitues what the autors called *domain-constrained*

policy P. The analysis of domain constrained policies is based on the abductive constraint logic programming proof procedure described in [Nuffelen 2004].

Given the domain-constrained policy P and a goal G, the analysis problem consists of finding the set of ground abducible predicates that if added as input to P, the goal G is achieved. The abductive reasoning enables to detect modality conflicts. Indeed, the goal to reach can be a situation where some subject is both permitted and prohibited to execute some action on some target, or situations where a subject has an obligation to execute some action which s/he is not permitted. This framework does not support the detection of conflict between obligations with deadlines.

In the following, we give an overview of situation calculus. The situation calculus is the formalism that we use to detect conflicts between obligations with deadlines and other conflicts that we present in the following chapters.

2.5 Situation calculus

The situation calculus [McCarthy 1983] is a second-order logic language specially designed to represent the change in dynamic worlds. The ontology and axiomatization of the sequential situation calculus was extended to include time [Reiter 1998], concurrency and natural actions [Reiter 1996], etc, but in all cases the basic elements of language are actions, situations and fluents. The situation calculus language used in this thesis is described below.

2.5.1 The language

The language consists of the following ontology:

- All changes in the world are the results of actions. They are designated by terms of first order logic. To represent the time in the situation calculus, we add a time argument in all instantaneous actions which is used to specify the exact time or time range in which the actions occur in world history. For example, $sign(Jean, dischargeNote(Mary), 100)$ is the instantaneous action of signing the discharge note of $Mary$ by $Jean$ at the moment 100. The actions are instantaneous, but we can express actions with duration. For example, consider the following two instantaneous actions, $startConsultation(d, p, t)$, meaning d starts consultation of p at time t, and $endConsultation(d, p, t')$, meaning d ends consultation of p at time t'. The fluent $inConsultation(p, s)$, expressing the patient

p is in consultation in the situation s, turns from false to true if there exists a time t and doctor d when the action $startConsultation(d, p, t)$ is performed, and turns to false if there exists a time t' when the action $endConsultation(d, p, t')$ is performed. Thus in situations where fluent $inConsultation(p, s)$ is true, we can describe the properties of the world, such as the heartbeat of p per unit of time, as a function of time that must be true during the progress of consultation.

- A possible history of the world, which is a sequence of actions is represented by the first order terms denoted *situations*. The constant S_0 is the initial situation.

- There is a binary function symbol do; $do(\alpha, s)$ denotes the situation resulting from the execution of the action α in the situation s. For example, $do(write(Jean, dischargeNote(Mary), 5)$, $do(write(Jean, consultationReport(Mary), 8)$, $do(write(Jean, admissionNote(Mary), 10), S_0)))$ is the situation indicating the history of the world which consists of the execution of the sequence of actions $[write(Jean, admissionNote(Mary), 10), write(Jean,$ $consultationReport(Mary), 8), write(Jean, dischargeNote(Mary), 5)]$.

- *Fluents* describing the facts of a state. There are two types of fluents; *relational fluents* and *functional fluents*. *Relational fluents* are symbols of predicates which take a term of type *situation* as the last argument, which their truth values may vary from one situation to another. For example, $inConsultation(Mary, s)$, means that $Mary$ is in consultation at situation s. *Functional fluents* are denoted by function symbols that take a situation as the last argument, which the truth of their function values change from one situation to another. For example, $heartbeat(Mary, s)$ denotes the number of heartbeats of $Mary$ in situation s.

- There are also symbols of predicates and functions (including constants) denoting relations and functions independent of situations.

- A particular binary predicate symbol $<$, defines a strict order relation on situations; $s < s'$ means that we can reach s' by a sequence of actions starting from s. For instance, $\text{do}(a_2\ , \text{do}(a_1\ , S_0\ )) < \text{do}(a_4\ , \text{do}(a_3\ , \text{do}(a_2\ , \text{do}(a_1\ , S_0\ ))))$.

- A second particular binary predicate symbol $Poss$, defines when an action is possible. $Poss(a, s)$ means that the action a can be executed in the situation s.

- A function symbol *time*: $time(a)$ denotes the time when the action a occurs.

- A function symbol *start*: $start(s)$ denotes the start time of the situation s.

2.5.2 fundamental axioms

The basic axioms for the situation calculus, as defined in [Lin and Reiter 1994] and [Reiter 1991] are as follows:

- The second order induction axiom:

$$(\forall P).[P(S_0) \wedge (\forall a, \sigma)(P(\sigma) \rightarrow P(do(a, \sigma)))] \rightarrow (\forall \sigma)P(\sigma)$$

  The induction axiom says that to prove that property P is true in all situations it is sufficient to prove that P is true in the initial situation S_0 (initialization step) and for all actions a and situations σ, if P is true in the situation σ, then P is still true in the situation $do(a, \sigma)$ (induction step). The axiom is necessary to prove properties true in all situations [Reiter 1993].

- The unique names axioms:

$$\begin{aligned} &S_0 \neq do(a, s), \\ &do(a, s) = do(a', s') \rightarrow a = a' \wedge s = s' \end{aligned} \tag{2.1}$$

- Axioms that define an order relation $<$ on situations:

$$\begin{aligned} &\neg s < S_0, \\ &s < do(a, s') \leftrightarrow (Poss(a, s') \wedge start(s') \leq time(a) \wedge s \leq s'). \end{aligned}$$

- The axiom: $start(do(a, s)) = time(a)$.

In addition to the axioms described above, we need to describe a class of axioms when we formalize an application domain:

- *Action precondition axioms*, one for each action:

$$Poss(A(\vec{x}, t), s) \leftrightarrow \phi(\vec{x}, t, s)$$

  where $\phi(\vec{x}, t, s)$ characterizes the preconditions of the action A, it is any first order formula with free variables among $\vec{x}$, t, and whose only term of sort of *situation* is s.
  For example, a patient can leave hospital if he is in the hospital.

$$Poss(leave(p, t), s) \leftrightarrow inpatient(p, s)$$

  Using predicate $poss(a)$, we can then recursively specify that a given situation s is executable.

$$\begin{aligned} &Executable(s) \leftrightarrow \\ &(\forall a, s').do(a, s') \leq s \rightarrow (Poss(a, s') \wedge start(s') \leq time(a)). \end{aligned}$$

- *Successor state axioms*, one for each fluent. These axioms characterize the effects of actions on fluents and they embody a solution to the frame problem[1] for deterministic actions [Reiter 1991].
  The syntactic form of successor state axiom for a relational fluent F is:

$$\begin{aligned} &Poss(a, s) \rightarrow \\ &[F(\vec{x}, do(a, s)) \leftrightarrow \gamma_F^+(\vec{x}, a, s) \vee \\ &(F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))] \end{aligned}$$

  where $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ indicate the conditions under which if the action a is executed in situation s, $F(\vec{x}, do(a, s))$ becomes true and false respectively.
  For example, the succession state axiom of fluent $assigned(p, d, s)$ meaning a patient p is assigned to a doctor d can be defined as follows:

$$\begin{aligned} &Poss(a, s) \rightarrow \\ &assigned(p, d, do(a, s)) \leftrightarrow [(\exists t)a = assign(p, d, t) \vee \\ &(assigned(p, d, s) \wedge \neg(\exists t)a = revokeAssignment(p, d, t) \wedge \\ &\neg(\exists t)a = leave(p, t))] \end{aligned}$$

  Here $\gamma_F^+(\vec{x}, a, s)$ corresponds to the formula: $(\exists t)a = assign(p, d, t)$ and $\gamma_F^-(\vec{x}, a, s)$ is the formula: $(\exists t)a = revokeAssignment(p, d, t) \vee (\exists t)a = leave(p, t)$. The action *assign* makes the fluent *assigned* true and the actions *revokeAssignment* and *leave* turn the fluent *assigned* to false.
  It is assumed that no action can turn F to be both true and false in a situation, i.e. $\neg\exists s\exists a\gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s)$.
  For a functional fluent, the syntactic form of successor state axiom is:

$$\begin{aligned} &Poss(a, s) \rightarrow \\ &[F(\vec{x}, do(a, s)) = y \leftrightarrow \gamma_F^+(\vec{x}, y, a, s) \vee \\ &(y = F(\vec{x}, s) \wedge \neg(\exists y')\gamma_F^-(\vec{x}, y', a, s))] \end{aligned}$$

  where, $\gamma_F(\vec{x}, y, a, s)$ is any first order formula with free variables among $\vec{x}$, y, a, t, and whose only term of sort of *situation* is s.

- Axioms describing the initial situation.

- In each application involving a particular action $A(\vec{x}, t)$, an axiom that gives the time of the action A: $time(A(\vec{x}, t)) = t$.

[1] The difficulty in logic of expressing the dynamics of a situation without explicitly specifying everything that is not affected by the actions.

In the rest of this report, we denote $Axioms = \Sigma \cup A_{ss} \cup A_{ap} \cup A_{S_0}$ where:

- Σ is the foundational axiomatic of the situation calculus.
- A_{ss} is a set of successor state axioms.
- A_{ap} is a set of action precondition axioms.
- A_{S_0} is a set of initial situation axioms. A_{S_0} is a set of sentences with the property that S_0 is the only term of sort situation mentioned by the fluents of a sentence of A_{S_0}. Thus, no fluent of a formula of A_{S_0} mentions a variable of sort situation or the function symbol *do*.

We denote $Axioms \vdash p$ the fact that the sentence p can be derived from the set of axioms $Axioms$. This kind of domain theories provides us with various reasoning capabilities, for instance planning [Green 1969b]. Given a domain theory $Axioms$ as above, and a goal formula $G(s)$ with a single free-variable s, the planing task is to find a sequence of actions $\vec{a}$ such that:

$$Axioms \vdash s_0 \leq do(\vec{a}, s_0) \wedge Executable(do(\vec{a}, s_0)) \wedge G(do(\vec{a}, s_0)),$$

where $do([a_1, ..., a_n], s)$ is an abbreviation for $do(a_n, do(a_{n-1}, ..., do(a_1, s)...))$

2.6 Conclusions

In this chapter, we give an overview of security policy specification and analysis. We then discuss the limitations of current access and usage control policy languages and models. This allows us to identify necessary points to support obligation and usage control policies. In particular, this study illustrates how describing change in states is necessary to clarify the semantics of obligations and usage controls. Policy analysis is finally examined and an overview of situation calculus is presented.

CHAPTER 3

Specification of Obligation with Deadline Policies

3.1 Introduction

A security policy is often defined as permission, prohibition, obligation and exemption rules. Permission and prohibition rules are used to specify access control policies. Obligation and exemption rules are useful to specify other security requirements corresponding to usage control policies [Cuppens et al. 2005][Elrakaiby et al. 2012a]. In the usage control literature, two different types of obligation are generally considered called system obligation and user obligation [Hilty et al. 2007]. When the security policy includes user obligation, these obligations should be associated with deadlines. When these obligations are activated, these deadlines provide the user with some time to enforce the obligation before violation occurs.

In this chapter, we use a language based on deontic logic to specify security policies that include obligations with deadline. The advantage of deontic logic is that it provides means to consistently reason about deontic concepts as obligation and permission. Then we suggest an approach based on the sequential temporal situation calculus [Reiter 1998] to give semantic to our language. The Situation Calculus allows us to analyze decidability and complexity of several useful problems as the temporal projection problem [Hanks and McDermott 1987]. This problem consists of asking whether a formula holds after a sequence of actions is performed in the initial situation. In this chapter, we will see how this is useful to decide which rule can be applied to a given situation and detect violation. Furthermore, the situation calculus provides a solution to the frame problem through the specification of succession state axioms [Reiter 1991]. The frame problem is the difficulty in logic of expressing the dynamics of a situation without explicitly specifying everything that is not affected by the actions.

This chapter is organized as follows. In section 3.2, we give a motivation example. Section 3.3 explains how to define security policies that include obligations with deadline. This model is based on deontic logic, and a security policy is viewed as a set of deontic norms. Section 3.4 extends situation calculus to formally derive which actual norms apply in a given situation. In this section, we also formally define when an obligation with deadline is violated. Finally, we present the conclusion and contribution.

3.2 Motivation example

In the medical community, the patient's record contains information about care provided to the patient during his stay in the hospital. The medical records are regulated by hospitals through legal texts. These laws specify, in particular, the time given to doctors to complete patient records assigned to them. In hospitals where medical records are digitally stored, these rules may be expressed as obligations with deadlines. These rules aim to ensure the availability of medical information in expected time. In this section, we describe the impact of availability of medical information in expected time on the quality of patient care and we give an example of obligations with deadline concerning completion of medical records.

3.2.1 Impact of deadlines to complete medical records on the availability of information

Studies have shown that patient care can be improved by timely sending a complete and accurate information on patient hospitalization to the practitioner ([J. I. Balla 1994, Bolton 2001]). In contrast, a breakdown of communication, due to delays in the transfer of information or incomplete information can have serious consequences. For example, the physician who does not have access to the summary sheet of a patient hospitalization prepared by acute care services is in an uncomfortable situation when the patient's life is in danger.
Despite what has been raised by these studies on the importance of time when transferring patient information, it is observed a latency in transferring this information in practice. Studies have noted that discharge summaries for example were not fulfilled on time and in many cases, doctors who examined patients after their leaving from hospital, do not receive them at all [C. van Walraven 2002]. Other studies noticed a significant delay between the time when the patient receives his leave and when the generalist physician received the advice ([Mageean 1986], [A. N. Raval 2003]).

Therefore the hospitals are required to establish regulations so that the medical records are fulfilled timely to ensure continuity of patient care.

3.2.2 Rules regarding the completion of the patient's medical record

The law on public hospitals specifies that medical records must be fulfilled for any person registered or admitted to a health facility [R.R.O 1990]. Also it specifies the elements that a medical record must contain. The law may specify the deadline given to doctors so that each element is present in the patient's record, and the appropriate measures when these deadlines are not respected, see for example the Ontario regulations [R.R.O 1990]. Among the documents that must be found in the medical records: summary sheet, admission note, medical observation, operating protocol and discharge note. The time to make these documents present in the folder of the user differs from one document to the other. For example, writing the medical summary sheet of a patient must be completed before this latter leaves the hospital. An admission note must be completed within 48 hours following the admission of patient. See [CMQ 2005] for a complete description of the time accorded to complete the remaining documents. This example shows a real need to have obligations with deadline in security policy to ensure the availability of information in a predefined time. We were inspired by rules in document [CMQ 2005] for building the following example of policy rules.

Example 1

- **Rule1**: The doctor **must** complete the admission note of the patient assigned to him within 30 units of time following his admission to the hospital.
- **Rule2**: The doctor **must** complete the medical observation of the patient assigned to him within 40 units of time following his admission to the hospital.
- **Rule3**: End deadline for completing the admission note of a patient must occur after 30 units of time of his admission to the hospital.
- **Rule4**: End deadline for completing the medical observation of a patient must occur after 40 units of time of his admission to the hospital.
- **Rule5**: The doctor is permitted to write observation or admission note of an inpatient assigned to him.
- **Rule6**: The doctor is permitted to complete an observation or admission note, that are being written, of an inpatient assigned to him.

In this chapter, we first show how to express these rules in our language. Then we show how to derive active obligations, actual permissions and detect violations.

3.3 Security policy specification

In this section, we define our language, based on deontic logic of actions, to specify permissions and obligations in security policies. Then, we give the specification of rules specified in the example 1. This example will be subsequently enriched by other needs throughout this thesis.

3.3.1 Specification using the deontic logic of actions

We consider two modalities: Permissions and Obligations with deadline. They are called normative modalities in the following. Normative modalities are represented as dyadic conditional modalities. Permissions are specified using dyadic modality $P(\alpha|p)$ where α is an action of $\mathcal{A}$ and p is the condition of the permission. The condition is any formula built using fluents of $\mathcal{F}$. $P(\alpha|p)$ means that the action α is permitted when condition p holds. Obligations with deadline are specified using modality $O(\alpha < d|p)$ which intuitively means that when formula p starts to hold, there is an obligation to execute action α before the deadline condition d starts to hold. In the following, we assume that the deadline condition must be an atomic fluent predicate of $\mathcal{F}$. If the action α is executed before the deadline condition d starts to hold, then we shall say that the obligation is fulfilled. Else we shall consider that the obligation is violated. We call *norm* a formula corresponding to a conditional permission or obligation with deadline. A security policy, $\mathcal{P}$ is a finite set of norms.

3.3.2 Example of rule specification

We give, here, the specification of the policy rules described in 1. For that, we should determine the set of fluents $\mathcal{F}$ and the set of the actions $\mathcal{A}$ before given the specification.

- Set $\mathcal{F}$ of fluents:
  - $Assigned(p, d, \sigma)$. The patient p is assigned to a doctor d in situation σ.
  - $Inpatient(p, t, \sigma)$. The patient p is admitted to the hospital at time t in the situation σ.

  - $Deadline(type, p, t, \sigma)$. The deadline to write document of type *type* concerning patient p created at time t is elapsed in σ.
  - $WritingDoc(d, type, p, t, t', \sigma)$. Doctor d is writing the document of type *type* concerning patient p created at time t and began to be written at time t' in σ.
  - $Doctor(d)$. d is a doctor.

- Set $\mathcal{A}$ of actions:
  - $Assign(p, d, t)$. The action to assign at time t the patient p to the doctor d.
  - $RevokeAssignment(p, d, t)$. The action to revoke at time t assignment of the patient p to the doctor d.
  - $PatientAdmission(p, t)$. The action to admit at time t the patient p at the hospital.
  - $Leave(p, t)$. The patient p leaves the hospital at time t.
  - $EndDeadline(type, p, t, t')$. The action to warn at time t' that the accorded deadline for writing document of patient p expires.
  - $StartWrite(d, type, p, t)$, $EndWrite(d, type, p, t)$: d starts (resp. ends) to write document of type corresponding to patient p at time t; Type is one of the following elements: $Observation$ or $AdmissionNote$.

- Specification:
  - Obligations with deadline

$$\begin{aligned}
\mathbf{O1}: \quad & O(write(d, type, p, t, t') < Deadline(type, p, t, \sigma) | Doctor(d) \wedge \\
& Assigned(p, d, \sigma) \wedge Inpatient(p, t, \sigma) \wedge type = AdmissionNote) \\
\mathbf{O2}: \quad & O(write(d, type, p, t, t') < Deadline(type, p, t, \sigma) | Doctor(d) \wedge \\
& Assigned(p, d, \sigma) \wedge Inpatient(p, t, \sigma) \wedge type = Observation)
\end{aligned}$$

  - System obligations

$$\begin{aligned}
\mathbf{O3}: \quad & O(EndDeadline(admissionNote, p, t, t') | Inpatient(p, t, \sigma) \wedge t' = t + 30) \\
\mathbf{O4}: \quad & O(EndDeadline(observation, p, t, t') | Inpatient(p, t, \sigma) \wedge t' = t + 40)
\end{aligned}$$

  - Permissions

$$\begin{aligned}
\mathbf{P1}: \quad & P(StartWrite(d, type, p, t, t_{sw}) \mid Doctor(d) \wedge Assigned(p, d, \sigma) \wedge \\
& Inpatient(p, t, \sigma) \wedge type = (Observation \vee AdmissionNote)) \wedge t_{sw} > t \\
\mathbf{P2}: \quad & P(EndWrite(d, type, p, t, t_{ew}) \mid (\exists t') WritingDoc(d, type, p, t, t', \sigma) \wedge \\
& Assigned(p, d, \sigma) \wedge Inpatient(p, t, \sigma) \wedge \\
& type = (Observation \vee AdmissionNote) \wedge (t_{ew} > t) \wedge (t_{ew} > t'))
\end{aligned}$$

We shall now use the situation calculus to formally define the semantics of the different modalities which we defined in this section.

3.4 Actual norm derivation and violation detection

There are papers that only deal with system obligations [Lobo et al. 1999]. In that case, the objective is generally to check that obligations are immediately enforced and there is no room for violation. In this section, we consider user obligations with deadline and want to consider that these user obligations may be violated in some situations. Before showing this, let us point out how we can derive actual permissions.

3.4.1 The semantic of actual permission

We extend the situation calculus with fluent $Perm(\alpha, \sigma)$ meaning there is an actual permission to do α. Then the set of Axioms previously defined is extended with a permission definition axiom for every fluent predicate $Perm(\alpha)$, $\alpha \in \mathcal{A}$. For this purpose, let P_α be the set of conditional permissions having the form $P(\alpha|p)$. We denote $\psi_{P_\alpha} = p_1 \vee ... \vee p_n$ where each p_i for $i \in [1, ..., n]$ corresponds to the condition of a permission in P_α. If $P_\alpha = \emptyset$, then we assume that $\psi_{P_\alpha} = false$. Using ψ_{P_α}, the successor state axiom for $Perm(\alpha, \sigma)$ is defined as follows:

$$Poss(a, \sigma) \rightarrow$$
$$Perm(\alpha, do(a, \sigma)) \leftrightarrow [\gamma^{+}_{\psi_{P_\alpha}}(a, \sigma) \vee (Perm(\alpha, \sigma) \wedge \neg\gamma^{-}_{\psi_{P_\alpha}}(a, \sigma))]$$

This axiom specifies that the permission to do an action becomes effective after that the action that activates the context of the permission rule is executed. This permission remains effective until an action that turns the activation context to false is executed.

Example 2 Let us give when the permission rule P1 is active. According to its specification, we should first have the succession state axiom of fluents $Assigned(p, d, \sigma)$ and $Inpatient(p, t, \sigma)$. We assume that a patient p will be assigned to doctor d when the action of assignment is executed, since then p remains assigned to d unless there is a revocation of assignment or the patient leaves the hospital.

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &Assigned(p, d, do(a, \sigma)) \leftrightarrow [(\exists t)a = Assign(p, d, t) \vee \\ &(Assigned(p, d, \sigma) \wedge \neg(\exists t)a = RevokeAssignment(p, d, t) \wedge \\ &\neg(\exists t)a = Leave(p, t))] \end{aligned} \tag{3.1}$$

Patient p is hospitalized if he was admitted to the hospital and did not leave.

$$\begin{aligned}&Poss(a,\sigma) \rightarrow \\ &Inpatient(p,t,do(a,\sigma)) \leftrightarrow [a = PatientAdmission(p,t) \vee \\ &(Inpatient(p,t,\sigma) \wedge \neg(\exists t')a = Leave(p,t'))]\end{aligned} \tag{3.2}$$

Using the specification rule P1 and the succession state axioms above, we can calculate the conditions under which $StartWrite$ will be permitted:

$$\begin{aligned}&\gamma^{+}_{\psi_{P_{StartWrite(d,type,p,t,t_{sw})}}}(a,\sigma) \leftrightarrow type = (Observation \vee AdmissionNote) \wedge \\ &[(Assigned(p,d,\sigma) \wedge a = PatientAdmission(p,t)) \vee \\ &((\exists t').Inpatient(p,t',\sigma) \wedge a = Assign(d,p,t))]\end{aligned}$$

On the other side, the actions and the conditions under which $StartWrite$ will be no longer permitted are given by the following formula:

$$\begin{aligned}&\gamma^{-}_{\psi_{P_{StartWrite(d,type,p,t,t_{sw})}}}(a,\sigma) \leftrightarrow \\ &(\exists t')a = (RevokeAssignment(p,d,t') \vee Leave(p,t'))\end{aligned}$$

Then the active permission for $StartWrite$ is calculated using axiom (3.1).

$$\begin{aligned}&Poss(a,\sigma) \rightarrow \\ &Perm(StartWrite(d,type,p,t,t_{sw}),do(a,\sigma)) \leftrightarrow \\ &type = (Observation \vee AdmissionNote) \wedge && (3.3)\\ &[(Assigned(p,d,\sigma) \wedge a = PatientAdmission(p,t)) \vee && (3.4)\\ &((\exists t').Inpatient(p,t',\sigma) \wedge a = Assign(d,p,t))] \vee && (3.5)\\ &(Perm(StartWrite(d,type,p,t,t_{sw}),\sigma) \wedge && (3.6)\\ &\neg(\exists t')a = RevokeAssignment(p,d,t') \wedge \neg(\exists t')a = Leave(p,t') && (3.7)\end{aligned}$$

The lines 3.3, and 3.4 of the axiom above express the fact that a doctor is permitted to write the observation and the admission note of a patient assigned to him as soon as this patient is admitted in the hospital. The lines 3.3 and 3.5 express the fact that when a patient is hospitalized, a doctor d will be permitted to write its observation and admission note after that this patient is assigned to him. Finally the lines 3.6 and 3.7 express the fact that a current permission for a doctor to write the admission note and the observation of a patient is disabled, once the patient leaves the hospital or the assignment of this patient to the doctor is revoked.

The actual permission corresponding to the rule P2 is calculated in appendix A.

3.4.2 The semantic of active obligation

We extend the situation calculus with fluents $Ob(\alpha < d, \sigma)$ (the obligation to do α before deadline d starts to be effective) where α is an action of $\mathcal{A}$ and d is a fluent of $\mathcal{F}$. As permissions, we need the obligation definition axiom for every fluent predicate $Ob(\alpha < d)$, where $\alpha \in \mathcal{A}$ and $d \in \mathcal{F}$. Notice that since the sets $\mathcal{A}$ and $\mathcal{F}$ are finite, we have a finite set of successor state axioms to define for $Ob(\alpha < d)$. We define $O_{\alpha,d}$ to be the set of conditional obligations with deadline in P having the form $O(\alpha' < d'|p)$ such that $\alpha = \alpha'$ and d and d' are logically equivalent. We say that two fluent predicates d and d' are logically equivalent with respect to a set of *Axioms* if we can prove that $d \leftrightarrow d'$ is an integrity constraint of *Axioms*. We denote $\psi_{O_{\alpha,d}} = p_1 \vee ... \vee p_n$ where each p_i for $i \in [1, ..., n]$ corresponds to the condition of an obligation in $O_{\alpha,d}$. If $O_{\alpha,d} = \emptyset$, then we assume that $\psi_{O_{\alpha,d}} = false$. Using $\psi_{O_{\alpha,d}}$, the successor state axiom for $Ob(\alpha < d)$ is defined as follows :

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &Ob(\alpha < d, do(a, \sigma)) \leftrightarrow [\gamma^{+}_{\psi_{O_{\alpha,d}}}(a, \sigma) \vee \\ &(Ob(\alpha < d, \sigma) \wedge \neg(a = \alpha) \wedge \neg\gamma^{+}_{d}(a, \sigma) \wedge \neg\gamma^{-}_{\psi_{O_{\alpha,d}}}(a, \sigma))] \end{aligned} \tag{3.8}$$

This axiom says that the obligation to do α before deadline d is activated when $\psi_{O_{\alpha,d}}$ starts to be true. This obligation is deactivated when it is fulfilled (i.e. action α is done) or it is violated (i.e. deadline d starts to be true) or condition $\psi_{O_{\alpha,d}}$ ends to be true (i.e. it is no longer relevant to do α).

Example 3 In this example, we will see how to derive active obligations of rule O1 and O2.
In addition to the succession state axioms of fluents Assigned and Inpatient already specified in the example above, we need the succession state axiom of fluent Deadline.
Action *EndDeadline* is executed to denote that the delay granted to write documents is elapsed. When the deadline is considered expired, it remains expired forever.

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &Deadline(type, p, t, do(a, \sigma)) \leftrightarrow \\ &(\exists t')a = EndDeadline(type, p, t, t') \vee Deadline(type, p, t, \sigma) \end{aligned} \tag{3.9}$$

According to the specification of obligation rules, we can see that we have one set of conditional obligations with deadlines: $O_{write(d,type,p,t,t_w),Deadline(type,p,t,\sigma)}$.

The corresponding formula $\psi_{O_{write(d,type,p,t,t_w),Deadline(type,p,t,\sigma)}}$, after simplification, is as follows:

$$\psi_{O_{write(d,type,p,t,t_w),Deadline(type,p,t,\sigma)}} \leftrightarrow$$
$$Doctor(d) \wedge Assigned(p,d,\sigma) \wedge Inpatient(p,t,\sigma) \wedge$$
$$type = (Observation \vee AdmissionNote))$$

According to axiom (3.8), to derive concrete obligations we should calculate the following formulas:

$$\gamma^{+}_{\psi_{O_{write(d,type,p,t,t_w),Deadline(type,p,t,\sigma)}}}(a,\sigma) \leftrightarrow$$
$$[(Assigned(p,d,\sigma) \wedge a = PatientAdmission(p,t)) \vee$$
$$((\exists t')Inpatient(p,t',\sigma) \wedge a = Assign(p,t))]$$
$$\gamma^{-}_{\psi_{O_{write(d,type,p,t,t_w),Deadline(type,p,t,\sigma)}}}(a,\sigma) \leftrightarrow$$
$$(\exists t')a = (RevokeAssignment(p,d,t') \vee Leave(p,t'))$$

The above formulas are calculated using the succession state axiom of fluents Assigned (3.1) and Inpatient (3.2).
Finally the formula $\gamma^{+}_{Deadline(type,p,t)}(a,\sigma)$ is calculated using succession state axiom of fluent *Deadline* (3.9).

$$\gamma^{+}_{Deadline(type,p,t)}(a,\sigma) \leftrightarrow a = EndDeadline(type,p,t,t')$$

Thus the concrete obligations concerning rules $O1$ and $O2$ are calculated using axiom 3.8 as follows:

$$Poss(a,\sigma) \rightarrow$$
$$Ob(write(d,type,p,t,t_w) < Deadline(type,p,t), do(a,\sigma)) \leftrightarrow \quad (3.10)$$
$$(Assigned(p,d,\sigma) \wedge a = PatientAdmission(p,t)) \vee$$
$$((\exists t')Inpatient(p,t',\sigma) \wedge a = Assign(p,t)) \vee$$
$$[Ob(write(d,type,p,t),\sigma) \wedge \neg(\exists t')a = (EndDeadline(type,p,t,t') \vee$$
$$RevokeAssignment(p,d,t') \vee Leave(p,t'))]$$

The active obligations corresponding to the rules O3 and O4 are calculated in appendix A.

3.4.3 Fulfillment and violation detection

An obligation with deadline to do an action is considered satisfied, when the action is executed while the obligation is still active, and before the deadline of the obligation

becomes true. We characterize situations where the obligations are fulfilled by using the fluent Fulfil defined as follows:

$$\begin{aligned} &Poss(a,\sigma) \rightarrow \\ &Fulfil(\alpha < d, do(a,\sigma)) \leftrightarrow [(Ob(\alpha < d,\sigma) \wedge a = \alpha \wedge \neg\gamma_d^+(\alpha,\sigma)) \vee \\ &Fulfil(\alpha < d,\sigma)] \end{aligned} \tag{3.11}$$

Notice that, if in a given situation σ, it simultaneously happens that the obligatory action is executed and the associated deadline is activated, then the decision is to consider that the obligation is violated and not fulfilled. This is called obligation with strict deadline. We can also define $O(\alpha \leq d|p)$ so that, in the same situation, the obligation is fulfilled and not violated.
Finally we define the succession state axiom of the fluent $Violated_O(\alpha < d, \sigma)$, meaning the obligation to do the action α before the deadline d is violated in situation σ:

$$\begin{aligned} &Poss(a,\sigma) \rightarrow \\ &Violated_O(\alpha < d, do(a,\sigma)) \leftrightarrow [(Ob(\alpha < d,\sigma) \wedge \gamma_d^+(a,\sigma)) \vee \\ &Violated_O(\alpha < d,\sigma)] \end{aligned} \tag{3.12}$$

This axiom specifies that an obligation to do α is violated, when the associated deadline comes true when it was still active, and it was never executed. The axiom also specifies that in a given situation σ, if it simultaneously happens that the obligatory action is executed and the associated deadline is activated, then the decision is to consider that the obligation is violated.
Concerning system obligations we consider them as a special case of obligations with deadline, written as follows: $O(\alpha)$. As there is no deadline associated with these obligations, we assume that: $\gamma_d^+(a,\sigma) = \gamma_d^-(a,\sigma) = false$. Thus we can derive the succession state axiom characterizing the situations when system obligations are active using axiom 3.8.

$$Poss(a,\sigma) \rightarrow (Ob(\alpha, do(a,\sigma)) \leftrightarrow \gamma_{\psi_{O_\alpha}}^+(a,\sigma)) \tag{3.13}$$

This axiom says that the system obligation to do α is activated only in the situations when ψ_{O_α} starts to be true and they are deactivated immediately after. Thus a system obligation should be fulfilled immediately after its activation. This can be derived using the axiom 3.14 as follows:

$$\begin{aligned} &Poss(a,\sigma) \rightarrow \\ &Fulfil(\alpha, do(a,\sigma)) \leftrightarrow [(Ob(\alpha,\sigma) \wedge a = \alpha) \vee Fulfil(\alpha,\sigma)] \end{aligned}$$

When an obligation system is not executed immediately after its activation, a violation is detected using the following axiom:

$$Poss(a, \sigma) \rightarrow$$
$$Violated_O(\alpha, do(a, \sigma)) \leftrightarrow [(Ob(\alpha, \sigma) \wedge \neg(a = \alpha)) \vee Violated_O(\alpha, \sigma)]$$

Example 4 Situations where the rules O1 and O2 are violated are characterized by the following formula:

$$Poss(a, \sigma) \rightarrow$$
$$Violated_O(write(d, type, p, t, t_w) < Deadline(type, p, t), do(a, \sigma)) \leftrightarrow$$
$$[(Ob(write(d, type, p, t, t_w) < Deadline(type, p, t), \sigma) \wedge$$
$$a = EndDeadline(type, p, t, t_d)) \vee$$
$$Violated(write(d, type, p, t, t_w) < Deadline(type, p, t), \sigma)]$$

In the axiom above, action EndDeadline is the only action which turns the fluent Deadline to true as specified in the succession state axiom of the fluent Deadline 3.9. Concerning the fluent characterizing where these obligations are active, Ob(write(d, type, p, t, t_w) < Deadline(type, p, t), σ), is given by axiom 3.10.

3.4.4 How the situation calculus helps us by resolving the frame problem

Reiter [Reiter 1991] combined two different solutions to the frame problem in the situation calculus provided in [Pednault 1989] and [Schubert 1990]. In the case where the effects of all actions on all fluents are determined, the proposed solution by Reiter reduces the number of axioms necessary to describe a dynamic world. Indeed, it reduces the number of axioms to $F + A$, where F is the number of fluents and A is the number of actions, compared with the $2 \times A \times F$ explicit frame axioms that would otherwise be required.
Let us take an example. Consider the following sequence of actions:

$$\sigma = do([Assign(Alice, Jean, 4), PatientAdmission(Alice, 11),$$
$$StartWrite(Jean, Observation, Alice, 11), PatientAdmission(Bob, 12),$$
$$Assign(Bob, Jean, 13)], \sigma_0)$$

Where σ_0 is the initial situation containing the following effect: $Doctor(Jean)$. This sequence of actions is shown in the big gray block in figure 3.1. In this figure, we can

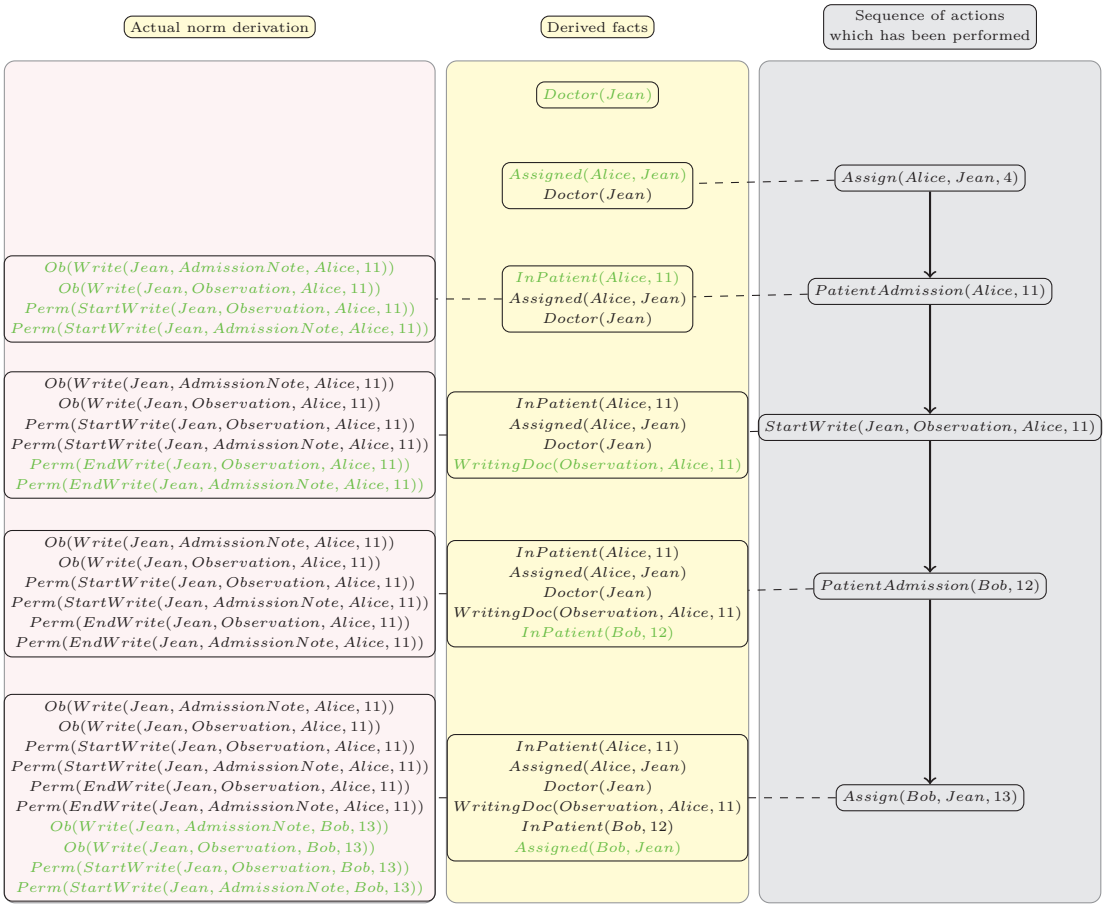

Figure 3.1: Example of actual norm derivation

see the resulting facts after the execution of each action. The facts in green are those that have just become true. They become black after the execution of actions which have no effect on them. The pink block represents what we derive using the succession state axioms of active obligations and concrete permissions which we have shown in the examples above.

Let see now, through this example, how the provided solution of the frame problem using the situation calculus helps us in deriving actual norms.

Let us consider the obligations of writing documents of Alice. According to the specification of policy rules, they are active in any situation where InPatient(Alice, t) and Assigned(Alice, Jean) are true. They start to be active from the moment when action PatientAdmission(Alice, 11) is executed (in the box where they are green). To derive that these obligations remain active in following situations (in all the boxes where they are black), normally, we should specify that each executed action from the execution of PatientAdmission(Alice, 11) until the execution of Assign(Bob, Jean, 13) has no impact on these obligations. Instead, it is simpler to specify, through succession state axioms, that action Leave(Alice) (which turns Inpatient(Alice) to false), and RevokeAssign-

ment (which turns Assigned(Alice, Jean)) to false) are the only ones that deactivate these obligations (in addition to the actions that activate the corresponding deadlines as we do not consider persistent obligations). Then because these actions were not executed, it can inferred that these obligations remain active.

3.5 Conclusion and Contribution

In this chapter, we proposed to use deontic modalities to specify security policies including obligations with deadline. Then we use the temporal sequential situations calculus to derive concrete permissions and obligations. Furthermore, we have seen how our model can detect violations. We have presented an example from an existing law regulating deadlines for completion of patient medical records in the hospital of Ontario [R.R.O 1990]. This example shows that there is a real need to include the rule of obligations with deadline in usage control policy to ensure the availability of information within an allotted time.
In addition to requiring a specific deadline for completing the document of patients, it also requires that they are with a better quality. For example, it is stated in [ANAES 1996] that the quality of documents is directly related to the time spent on their outfit. Thus, it appears to be natural to have constraints in the usage control policy specifying for example that the documents should be written sequentially by the same doctor and a doctor should spend at least 5 units of time writing each document. Thereby, we need to enrich our language so it can express and ensure the compliance of constraints policy. This is what we will see in the next chapter.

CHAPTER 4

Specifying and Enforcing Constraints Policy in a Dynamic World

4.1 Introduction

Constraints are undoubtedly an important aspect of access control models. They are usually used to prevent fraudulent situations. In the literature, several types of constraints have been identified. Most of them belong to one of few basic types and are used, in general, to implement the least privilege principle and separation of duty requirement [Clark and Wilson 1987], [Saltzer and Schroeder 1975]. Our goals in this chapter are, firstly, to develop a language in which these contraints can be expressed. And secondly, provide a means to prove that these constraints are not violated despite how the system evolves.

Furthermore, beyond the fact that there is a need to prove the satisfiability of these constraints, it is more suitable to organize the decisions taken and actions performed in the system to ensure it. In transactional databases for example, to ensure the enforcement of integrity constraints, the classical approach is to verify if after the transaction execution the database is consistent (i.e., does not violate the constraints). If an inconsistency is detected, all operations of the transaction are canceled (rollback). This approach cannot be applied in systems where a rollback is not possible (e.g., operating systems). In this kind of system, it should be ensured that the constraints are met before executing the actions. The approach we propose is to build the precondition axioms of actions so it is proved that the constraints remain ensured after any execution of actions verifying these axioms.

In this chapter, we propose to extend our proposal language in chapter 3, based on deon-

tic logic of actions and the Situation Calculus [McCarthy 1968], to express and enforce policy constraints. We define two kinds of constraints which are referred throughout this chapter as non historical (called ahistorical in the following) and historical constraints. We show through examples how these two types of constraints can be used to express Statically Mutually Exclusive Roles (SMER), Dynamically Mutually Exclusive Roles (DMER)[Ferraiolo et al. 2001] and also periodic temporal constraints.
Our approach is based on the work of authors in [Lin and Reiter 1994]. In this work, the authors use the regression operator [Waldinger 1977, Pednault 1988, Reiter 1991] to build action precondition axioms taking into account constraints formulated as a simple state formula. These axioms are calculated so that, if the constraints are satisfied in the initial situation, they will be satisfied in all situations resulting from the execution of actions with respect to these axioms. In this chapter, we show how we extend their approach to take into account the historical constraints which cannot be expressed as a simple state formula.
This chapter is organized as follows. In section 4.2, we define our specification of the constraints. In section 4.3, we describe how regression is used to implement constraints written as simple formulas (ahistorical constraints), then we show how the historical constraints can be rewritten as simple formulas to generalize the concept of enforcement of specified constraint policies. In this section, we also define and characterize a secure system with respect to access control requirements and constraints policy. In section 4.4, we present related work. Section 4.5 concludes the chapter and summarizes contributions.

4.2 Specifying constraints

In our model, we have two kinds of constraints, ahistorical constraints and historical ones.

- An ahistorical constraint: It is a simple formula of the form:

$$(\forall\sigma)\neg(\exists\vec{x_1},..\vec{x_n})(F_1(\vec{x_1},\sigma)\wedge F_2(\vec{x_2},\sigma)\wedge ...\wedge F_n(\vec{x_n},\sigma))$$

  This constraint specifies that the fluents $F_1, ..., F_n$ must not all be true in the same situation. This also means that if, in some situation, some fluents were true and become false before the remaining fluents in the constraint become true, the constraint is not violated.

- A historical constraint is expressed using a formula of the form:

$$(\forall \sigma, \sigma'), \sigma \leq \sigma' \neg (\exists \vec{x_1}, ..\vec{x_n}) \\ (F_1(\vec{x_1}, \sigma) \wedge ... \wedge F_i(\vec{x_i}, \sigma) \wedge F_{i+1}(\vec{x_{i+1}}, \sigma') \wedge ...F_n(\vec{x_n}, \sigma'))$$

  Unlike ahistorical constraint, a historical constraint contains two parts. If it happened that the fluents specified in the first part were true in the same situation, the second part of the constraint should never be true in the future.

4.2.1 Example

In access control models based on roles (RBAC, see [Li et al. 2007]), two kinds of constraints have been defined: *static mutually exclusive roles* (SMER) and *dynamic mutually exclusive roles* (DMER) [Ferraiolo et al. 2001]. The SMER constraint specifies that some roles should not be assigned mutually to the same users, while the DMER constraint specifies that some roles should not be activated simultaneously in the same session. Managing such constraints becomes more complex when a hierarchy of roles is defined and administrative operations like deleting and adding roles are used. In this section, we show how to formalize the fragment of the RBAC model related to role assignment and role hierarchy using our language. We then give examples of SMER and DMER constraints.

Formalizing the fragment of the RBAC model related to role assignment and role hierarchy

We need the following fluents and actions to formalize the fragment of the RBAC model related to role assignment and role hierarchy.

- Fluents:
  - $Assign(s, r, \sigma)$: Subject s is assigned to role r in σ.
  - $Sub\_role(r_1, r_2, \sigma)$: r_1 is a sub-role of role r_2 in σ.
  - $Empower(s, r_2, ss, \sigma)$: Subject s is empowered to the role r in the session ss in σ.
  - $DMER(r_2, r_1, ss, \sigma)$: Role r_1 is separated from role r_2 in the session ss in σ.
  - $SMER(r_1, r_2, \sigma)$: Role r_1 is statically separated from role r_2 in σ.
- Actions:

- Create_hierarchy(s, r_1, r_2) (resp. Revoke_hierarchy): Subject s creates (resp. revokes) a hierarchy relation between role r_1 and role r_2.
- Create_assignment(s, s_1, r) (resp. Revoke_assignment): Subject s assigns (resp.revokes) subject s_1 to role r.
- Create_SMER(s, r_1, r_2) (resp. Revoke_SMER): Subject s creates (resp.revokes) a $SMER$ relation between role r_1 and role r_2.
- Create_DMER(s, r_1, r_2, ss), σ)(resp. Revoke_DMER): Subject s creates (resp. revokes) a $DMER$ relation between role r_1 and role r_2 in the session ss.

We give the semantic of the fluents specified below using succession state axioms.

- Succession state axiom of fluent SMER:

$$
\begin{aligned}
&SMER(r_1, r_2, do(a, \sigma)) \leftrightarrow \\
&(\exists s)[a = Create\_SMER(s, r_1, r_2) \vee a = Create\_SMER(s, r_2, r_1)] \vee \\
&(\exists r_3, s) Sub\_role(r_2, r_3, \sigma) \wedge a = (Create\_SMER(s, r_3, r_1) \vee \\
&Create\_SMER(s, r_1, r_3)) \vee \\
&Sub\_role(r_1, r_3, \sigma) \wedge (a = Create\_SMER(s, r_3, r_2) \vee \\
&Create\_SMER(s, r_2, r_3)) \vee \\
&SMER(r_1, r_3, \sigma) \wedge a = Create\_hierarchy(s, r_2, r_3) \vee \\
&SMER(r_2, r_3, \sigma) \wedge a = Create\_hierarchy(s, r_1, r_3) \vee \\
&SMER(r_1, r_2, \sigma) \wedge \neg(\exists s) a = (Revoke\_SMER(s, r_1, r_2) \vee \\
&Revoke\_SMER(s, r_2, r_1).
\end{aligned}
$$

  This axiom specifies that a static mutually exclusion roles between r_1 and r_2 can be created by applying the action $Create\_SMER(s, r_1, r_2)$. And for all situations σ, if SMER(r_2, r_1, σ) then SMER(r_1, r_2, σ). We admit that the action *Revoke_SMER* is the unique action that turns the fluent *SMER* from true to false. Furthermore in all situation σ, we assume if there exists a role r_3 such that $SMER(r_1, r_3, \sigma)$ and $Sub\_role(r_2, r_3, \sigma)$ then $SMER(r_1, r_2, \sigma)$. The succession state axiom of the fluent $DMER$ is the same like $SMER$.

- Succession state axiom of fluent Sub_role:

$$
\begin{aligned}
&Sub\_role(r_1, r_2, do(a, \sigma)) \leftrightarrow \\
&(\exists s)a = Create\_hierarchy(s, r_1, r_2) \vee \\
&(\exists r_3, s)[Sub\_role(r_1, r_3, \sigma) \wedge a = Create\_hierarchy(s, r_3, r_2) \vee \\
&Sub\_role(r_3, r_2, \sigma) \wedge a = Create\_hierarchy(s, r_1, r_3)] \vee \\
&Sub\_role(r_1, r_2, \sigma) \wedge \neg(\exists s)a = Revoke\_hierarchy(s, r_1, r_2).
\end{aligned}
$$

  When the action of creating hierarchy between two roles r_1 and r_2 is executed, the role r_1 is then a sub-role of role r_2. We assume for any situation σ, if there exists a role r_3 such that $Sub\_role(r_1, r_3, \sigma)$ and $Sub\_role(r_3, r_2, \sigma)$ then $Sub\_role(r_1, r_2, \sigma)$. We also assume that a hierarchy between roles r_1 and r_2 can only be revoked by applying the action of revoking hierarchy between r_1 and r_2.

- Succession state axiom of fluent Assign:

$$
\begin{aligned}
&Assign(s_1, r_1, do(a, \sigma)) \leftrightarrow \\
&(\exists s)a = Create\_assignment(s, s_1, r_1) \vee \\
&(\exists r_2, s)[Assign(s_1, r_2, \sigma) \wedge a = Create\_hierarchy(s, r_1, r_2) \vee \\
&Sub\_role(r_1, r_2, \sigma) \wedge a = Create\_assignment(s, s_1, r_2)] \vee \\
&Assign(s_1, r_1, \sigma) \wedge \neg(\exists s)a = Revoke\_assignment(s, s_1, r_1)
\end{aligned}
$$

  The role assignment (resp. revocation) can be done by the action Create_assignment (resp. Revoke_assignment). If a subject is assigned to some role, he is assigned to all its lower roles.

- Succession state axiom of fluent Empower:

$$
\begin{aligned}
&Empower(s_1, r, ss, do(a, \sigma)) \leftrightarrow \\
&Assign(s_1, r, \sigma) \wedge (\exists s)a = Activate\_role(s, s_1, r, ss) \vee \\
&(\exists r_1)Empower(s_1, r_1, ss, \sigma) \wedge (\exists s)a = Create\_hierarchy(s, r, r_1) \vee \\
&(\exists r_1)Sub\_role(r, r_1, \sigma) \wedge Assign(s_1, r_1, \sigma) \wedge (\exists s)a = Activate\_role(s, s_1, r_1, ss) \vee \\
&Empower(s_1, r, ss, \sigma) \wedge \neg(\exists s)a = Deactive\_role(s, s_1, r, ss)
\end{aligned}
$$

  This axiom specifies that when the action Activate_role(s, s_1, r, ss) is executed, the subject s_1 is empowered to the role r in the session ss if r is assigned to him. He is also empowered to all the roles that are lower than the role r. The unique action that turns the fluent Empower(s_1, r, ss, σ) from true to false is the action Deactive_role(s, s_1, r, ss).

Specifying SMER and DMER constraints

An SMER constraint specifies that two roles r_1 and r_2 cannot be simultaneously assigned to a user. This constraint can be expressed as an ahistorical constraint.

$$\neg(SMER(r_1, r_2, \sigma) \wedge Assign(s_1, r_1, \sigma) \wedge Assign(s_1, r_2, \sigma)) \tag{4.1}$$

Note that with the assignment revocation concept, if some subject s is assigned to the role r_1 and is revoked before that the role r_2 is assigned to him, the $SMER$ is logically not violated as the roles r_1 and r_2 are not assigned to the subject s at the same time. In addition to constraint 4.1, we consider the following ahistorical constraints:

$$\neg SMER(r, r, \sigma) \tag{4.2}$$

$$\neg Sub\_role(r, r, \sigma) \tag{4.3}$$

$$\neg DMER(r, r, ss, \sigma) \tag{4.4}$$

Concerning the DMER constraint, it can be expressed in the historical form as follows:

$$\neg(Empower(s, r_1, ss, \sigma) \wedge Empower(s, r_2, ss, \sigma') \wedge DMER(r_1, r_2, ss, \sigma')) \tag{4.5}$$

The constraint 4.5 specifies that roles r_1 and r_2 cannot be activated in the same *session*, although these roles may be assigned to the same user.
We show in the following section how these constraints can be enforced.

4.3 Enforcing constraints

The idea to implement constraint policies is, for each action, build the preconditions that must be satisfied so that the constraints are not violated when this action is executed. In [Lin and Reiter 1994], the authors use the regression concept to generate a set of action precondition axioms, one for each action prototype using state constraint formulas which are of the form: $(\forall\sigma)C(\sigma)$, where $C(\sigma)$ is a simple formula where the unique free variable of type situation is σ. The historical constraints in our model are in the form of state constraints. In this section, we show how this approach can be applied to enforce them. Then we extend it to implement historical constraints which are not simple formulas.

4.3.1 Enforcing ahistorical constraints

For each constraint C in A_{ahis}, there is an associated simple state formula Reg_C such that:

$$A_{una} \cup A_{ahis} \vdash (\forall x_1, ..., x_n, \sigma).R[C(do(A(x_1, ..., x_n), \sigma))] \leftrightarrow Reg_C$$

Then the following action precondition axiom is obtained by predicate completion ([Clark 1987]) of $Poss$:

$$Poss(a(x_1, ..., x_n), \sigma) \leftrightarrow \Pi_a \wedge \bigwedge Reg_C$$

where the big conjunction ranges over the ahistorical constraints in A_{ahis}.
Let A_{pre} the set of action precondition axioms thus obtained. For every simple constraint $C(\sigma)$ in A_{ahis}, it is proved in [Lin and Reiter 1994] that:

$$\Sigma \cup A_{una} \cup A_{ss} \cup A_{pre} \cup A_{cls} \vdash C(\sigma_0) \rightarrow (\forall \sigma).\sigma_0 \leq \sigma \rightarrow C(\sigma).$$

To generalize this concept to the historical constraints, the idea is to rewrite them in the simple form. This is what we present in the following.

4.3.2 Rewrite historical constraints in the form of simple state constraint

The formulas contained in A_{his} are in the following form:

$$(\forall \sigma, \sigma')\sigma \leq \sigma' \rightarrow \neg(C(\sigma) \wedge C'(\sigma'))$$

where $C(\sigma)$ (resp. $C(\sigma')$) is a simple formula with a unique free variable σ (resp. σ'). $C(\sigma)$ will be mentioned in the following as the first part of the historical constraint. With each fluent F contained in $C(\sigma)$, we associate the *backup* fluent F_{backup} meaning there exists a situation where the fluent F was true, the fluent F_{backup} becomes true when the associated fluent F becomes true and it will never be false, so we can build the succession state axiom for every fluent F_{backup} from the succession state axiom of F by eliminating the part that specifies when the fluent F becomes false. Thus the succession state axiom of the fluent F_{backup} is defined as follows:

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &[F_{backup}(\vec{x}, do(a, \sigma)) \leftrightarrow \gamma_F^+(\vec{x}, a, \sigma) \vee F_{backup}(\vec{x}, \sigma)] \end{aligned}$$

where $\gamma_F^+(\vec{x}, a, \sigma)$ indicates the conditions under which the fluent $F(\vec{x}, do(a, \sigma))$ becomes true if the action a is executed in situation σ. Let $C_{backup}(\sigma)$ be the formula obtained from $C(\sigma)$ by replacing each fluent in $C(\sigma)$ by its corresponding backup fluent. The constraint $\neg(C_{backup}(\sigma) \wedge C'(\sigma))$ is the simple form of the historical constraint $\neg(C(\sigma) \wedge C'(\sigma'))$.

4.3.3 Enforcing the historical constraints

Let A'_{ss} be the new set of succession state axioms obtained by adding the succession state axioms to A_{ss} for all backup fluents corresponding to fluents mentioned in every first part of constraints in A_{his}. We can then verify that:

$$A'_{ss} \vdash [(\forall \sigma, \sigma')\sigma \leq \sigma' \rightarrow \neg(C(\sigma) \wedge C'(\sigma'))] \leftrightarrow (\forall \sigma)\neg(C_{backup}(\sigma) \wedge C'(\sigma)) \tag{4.6}$$

as

$$(\exists \sigma, \sigma')\sigma \leq \sigma' \wedge (C(\sigma) \wedge C'(\sigma')) \leftrightarrow (\exists \sigma')C_{backup}(\sigma') \wedge C'(\sigma')$$

Let A'_{ahis} be the generated finite set of all rewritten constraints in A_{his}. Note that according to the unique name axioms, we can verify that for each historical constraint in A_{his}, there is a unique corresponding ahistorical one. As shown in the previous section, we have:

$$\begin{aligned} &\Sigma \cup A_{una} \cup A'_{ss} \cup A_{pre} \cup A_{cls} \vdash \\ &\neg(C_{backup}(\sigma_0) \wedge C'(\sigma_0)) \rightarrow (\forall \sigma).\sigma_0 \leq \sigma \rightarrow \neg(C_{backup}(\sigma) \wedge C'(\sigma)) \end{aligned} \tag{4.7}$$

where, A_{pre} is the set of precondition axioms generated as shown in the previous section by considering the set of simple constraints $A'_{ahis} \cup A_{ahis}$.

Theorem 1. *Let A'_{ss}, A_{his}, A_{pre}, A_{una} and A_{cls} be as given above. For every historical constraint $(\forall \sigma, \sigma')[(\sigma \leq \sigma') \wedge \neg(C(\sigma) \wedge C'(\sigma'))] \in A_{his}$,*

$$\begin{aligned} &\Sigma \cup A_{una} \cup A'_{ss} \cup A_{pre} \cup A_{cls} \vdash \\ &\quad \neg(C(\sigma_0) \wedge C'(\sigma_0)) \rightarrow [(\forall \sigma, \sigma').\sigma_0 \leq \sigma \leq \sigma' \rightarrow \neg(C(\sigma) \wedge C'(\sigma'))]. \end{aligned}$$

Proof. Using the formulas 4.6 and 4.7. □

4.3.4 Defining and characterizing a secure system

Before defining the property that a given situation is secure with respect to access control requirements and a constraint policy, we define legal situations.

Definition 1. *Legal situations*
A situation is legal if it is the result of execution of permitted actions. They are defined recursively as follows:

$$\begin{aligned} &Legal(\sigma_0) \wedge \\ &\quad \forall a \forall \sigma, Legal(do(a, \sigma)) \leftrightarrow Perm(a, \sigma) \wedge Legal(\sigma) \end{aligned}$$

A situation is secure if it is legal and it does not violate any constraint of constraints policy.

Definition 2. *Secure situations*
For each constraint $\neg C(\sigma)$ *in* A_{ahis} *and for each constraint* $\neg(C'(\sigma) \wedge C''(\sigma'))$ *in* A_{his}, *the situation* σ *is secure iff:*

$$Secure(\sigma) \leftrightarrow$$
$$Legal(\sigma) \wedge \neg C(\sigma) \wedge (\neg C''(\sigma) \vee \neg(\exists \sigma')\sigma' \leq \sigma \wedge C'(\sigma'))$$

Definition 3. *We say that the system evolves in a secure way with respect to the access control requirement and constraints policy iff all executable situations are secure.*

$$(\forall \sigma) Executable(\sigma) \rightarrow Secure(\sigma)$$

Theorem 2. *The system is secure when the action precondition axiom of every action* α *is strengthened with the guarded condition that this action must be permitted and the regression of all ahistorical and rewritten historical constraints.*

$$Secure(\sigma_0) \wedge (\forall \alpha, Poss(\alpha, \sigma) \leftrightarrow \Pi_\alpha(\sigma) \wedge \bigwedge Reg_C(\sigma) \wedge Perm(\alpha, \sigma))$$

where the big conjunction ranges over the ahistorical constraints in A_{ahis} *and the rewritten historical constraints in* A'_{ahis}.

Proof. The proof is by induction. The started situation σ_0 is secure by applying the hypothesis. Let σ be the situation such that $Secure(\sigma)$. We should demonstrate that $do(a, \sigma)$ is secure where a is any action in $\mathcal{A}$ such that $Poss(a, \sigma)$. The situation $do(a, \sigma)$ verifies $Legal(do(a, \sigma))$ as σ is a legal situation and $Perm(a, \sigma)$. Regarding constraints, for every constraint C, $Reg_C(\sigma)$ means $C(do(a, \sigma))$. □

4.3.5 Example

In this example, we show how we can calculate the precondition axiom of the action *Create_hierarchy* using regression.
The action of creating hierarchy between roles impacts all specified constraints in the example of section 4.2.
The regression of applying $Create\_hierarchy(s, r_1, r_2)$ to the constraint 4.1, is given by the following formulas:

$$R[C1(do(Create\_hierarchy(s, r_1, r_2), \sigma))] \leftrightarrow$$
$$\neg(\exists s_1)[SMER(r_1, r_2, \sigma) \wedge Assign(s_1, r_2, \sigma)] \wedge \quad (4.8)$$
$$\neg(\exists r_3, s_1)[SMER(r_3, r_2, \sigma) \wedge Assign(s_1, r_3, \sigma) \wedge Assign(s_1, r_1, \sigma)] \wedge \quad (4.9)$$
$$\neg(\exists r_3, s_1)[SMER(r_1, r_3, \sigma) \wedge Assign(s_1, r_2, \sigma) \wedge Assign(s_1, r_3, \sigma)] \quad (4.10)$$

Let $\sigma' = do(Create\_hierarchy(s, r_1, r_2), \sigma)$, then in the situation σ', we will have $sub\_role(r_1, r_2, \sigma')$. Suppose that 4.8 is false, according to the succession state axiom of the fluent $Assign$, we can derive also that $Assign(s_1, r_1, \sigma')$. The separation of duty between r_1 and r_2 will remain true in the situation σ', as the action $Create\_hierarchy(s, r_1, r_2)$ has no effect on the fluent $SMER$. The assignment of the role r_2 to subject s_1 will also remain true in the situation σ'. Then we will have $SMER(r_1, r_2, \sigma') \wedge Assign(s_1, r_1, \sigma') \wedge Assign(s_1, r_2, \sigma')$ which violates the constraint 4.1. Now suppose that 4.9 is false, according to the succession state axiom of the fluent $SMER$, we will also have static mutually exclusion roles between r_3 and r_1 in the situation σ'. As the action $Create\_hierarchy(s, r_1, r_2)$ has no effect on the fluent $Assign$, the roles r_1, and r_3 remain assigned to a subject s_1 in the situation σ', thus the constraint 4.1 will be violated in the situation σ'. If 4.10 is false, the role r_1 will be assigned to a subject s_1 in the situation σ' according to the succession state axiom of the fluent $Assign$ which is a violation of the constraint 4.1 as the static mutually exclusion roles between r_1 and r_3 remains true in the situation σ' and also $Assign(s_1, r_3, \sigma')$.
The regression of applying $Create\_hierarchy(s, r_1, r_2)$ to the constraint 4.2 is given by the following formulas:

$$R[C2(do(Create\_hierarchy(s, r_1, r_2), \sigma))] \leftrightarrow \neg SMER(r_1, r_2) \tag{4.11}$$

The regression of applying $Create\_hierarchy(s, r_1, r_2)$ to the constraint 4.3 is given by the following formulas:

$$\begin{aligned} &R[C3(do(Create\_hierarchy(s, r_1, r_2), \sigma))] \leftrightarrow \\ &\neg(r_1 = r_2) \wedge \neg Sub\_role(r_2, r_1) \end{aligned} \tag{4.12}$$

The regression of applying $Create\_hierarchy(s, r_1, r_2)$ to the constraint 4.4 is given by the following formulas:

$$R[C4(do(Create\_hierarchy(r_1, r_2)))] \leftrightarrow \neg(DMER(r_1, r_2, ss, \sigma)) \tag{4.13}$$

It is clear that if 4.11 (resp. 4.12, 4.13) are false, the constraints 4.2 (resp. 4.3, 4.4) will be violated. The regression of applying $Create\_hierarchy(s, r_1, r_2)$ to the constraint 4.5, after the rewriting process, is given by the following formulas :

$$R[C5(do(Create\_hierarchy(r_1, r_2)))] \leftrightarrow$$

$$\neg(\exists r_3)(DMER(r_3, r_2, ss, \sigma) \wedge$$

$$Empower_{backup}(s, r_3, ss, \sigma) \wedge Empower(s, r_1, ss, \sigma)) \wedge \tag{4.14}$$

$$\neg(DMER(r_1, r_2, ss, \sigma) \wedge Empower(s, r_2, ss, \sigma)) \wedge \tag{4.15}$$

$$\neg(DMER(r_3, r_2, ss, \sigma) \wedge$$

$$Empower_{backup}(s, r_1, ss, \sigma) \wedge Empower(s, r_3, ss, \sigma)) \tag{4.16}$$

Suppose that 4.14 is false in the situation σ. According to the succession state axiom of the fluent $DMER$, we will also have in session ss in the situation σ', the dynamic mutually exclusion roles between r_3 and r_1. As the action $Create\_hierarchy(r_1, r_2)$ has no effect on the fluents $Empower_{backup}(s, r_3, ss, \sigma)$ and $Empower(s, r_1, ss, \sigma)$, they reminded true in the situation σ' and then the constraint 4.5 will be violated. If 4.15 is false in the situation σ, from the succession state axiom of the fluent $Empower$, we can derive that the subject s will empower the role r_1 in the situation σ'. As the the fluents $DMER(r_1, r_2, ss, \sigma)$ and $Empower(s, r_2, ss, \sigma)$ will remain true in the situation σ', the constraint 4.5 will be violated in this situation. For 4.16, we can follow the same reasoning as in 4.15.
Then the action precondition axiom of the action $Create\_hierarchy(s, r_1, r_2)$ is as follows :

$$
\begin{aligned}
&Poss(do(Create\_hierarchy(s, r_1, r_2), \sigma)) \leftrightarrow \\
&\neg(r_1 = r_2) \wedge \\
&\neg Sub\_role(r_2, r_1, \sigma) \wedge \\
&\neg SMER(r_1, r_2, \sigma) \wedge \neg DMER(r_1, r_2, ss, \sigma) \wedge \\
&\neg(\exists s_1)[SMER(r_1, r_2, \sigma) \wedge Assign(s_1, r_2, \sigma)] \wedge \\
&\neg(\exists r_3, s_1)[SMER(r_3, r_2, \sigma) \wedge Assign(s_1, r_3, \sigma) \wedge Assign(s_1, r_1, \sigma)] \wedge \\
&\neg(\exists r_3, s_1)[SMER(r_1, r_3, \sigma) \wedge Assign(s_1, r_2, \sigma) \wedge Assign(s_1, r_3, \sigma)] \wedge \\
&\neg(DMER(r_3, r_2, ss, \sigma) \wedge Empower_{backup}(s, r_3, ss, \sigma) \wedge Empower(s, r_1, ss, \sigma)) \wedge \\
&\neg(DMER(r_1, r_2, ss, \sigma) \wedge Empower(s, r_2, ss, \sigma)) \wedge \\
&\neg(DMER(r_3, r_2, ss, \sigma) \wedge Empower_{backup}(s, r_1, ss, \sigma) \wedge Empower(s, r_3, ss, \sigma))
\end{aligned}
$$

Up here, we have shown how historical and ahistorical constraints are implemented in our model in order to ensure the secure system evolution with respect to constraint policy. In the following, we show how we are able to express interesting requirements using these two types of constraints.

4.3.6 Expressive power

In our model, we can express constraints on obligations and permissions using fluents Ob and $Perm$. These constraints can be implemented using our approach because we have the succession state axioms corresponding to these fluents that we presented in the previous chapter. Furthermore, we can express temporal constraints by using fluents taking time as parameters. They are enforced in the same way as non temporal ones. In this category of constraints, we are able, in particular, to express periodic

constraints. For illustration, let us take the following example concerning a payment incident which occurs when the bank rejects a check payment order.

Example 5 The regulation like for example in France Monetary and Financial Code[1] specifies that the bank can apply tax to payment incident but after sending to the bank issuer an injunction mail informing him that he will be taxed for the bad check. A rejected check can be returned to the beneficiary, then to solve the payment incident, the check issuer must replenish his account with the appropriate amount and ask the beneficiary to make a new presentation of the check. The regulation specifies that the bank cannot apply a new tax on payment incident that may happen a second time within 30 days after the first rejection. We assume that mails are received after being sent and there is no delete action applied to the received mails. So we can suppose we have the following succession state axiom for the fluent $ackInjMailRCV$:

$$ackInjMailRCV(mail, t, do(a, \sigma)) \leftrightarrow$$
$$a = sendAckInjMail(mail, t) \vee ackInjMailRCV(mail, t, \sigma)$$

The fluent $payIncTaxed(check, t)$ means that the *check* is taxed at the time t. We assume there is no revocation on the tax operation. Then the succession state axiom of the fluent $payIncTaxed$ is as follows:

$$payIncTaxed(check, t, do(a, \sigma)) \leftrightarrow$$
$$a = applyTax(check, t) \vee payIncTaxed(check, t, \sigma)$$

We suppose that a taxing operation can be always possible:

$$Poss(do(applyTax(check, t), \sigma)) \rightarrow True$$

And consider the two constraints:

$$\neg(\exists mail)[check = subjOfInj(mail, \sigma) \wedge \\ payIncTaxed(check, t, \sigma) \wedge payIncTaxed(check, t', \sigma) \wedge \neg(t' = t)) \quad (4.17)$$

$$\neg(\exists mail, mail')[check = subjOfInj(mail, \sigma) \wedge \\ check = subjOfInj(mail', \sigma) \wedge payIncTaxed(check, t, \sigma) \wedge \\ payIncTaxed(check, t', \sigma) \wedge (t' - t) \leq 30)] \quad (4.18)$$

The constraint 4.17 specifies that a tax should not be applied to check more than once when the injunction mail is received. The constraint 4.18 specifies that a check should never be taxed more than once within 30 units of

[1] Article L131-73 and Article D131-25

time even if it is rejected again. These constraints are impacted just by the action $applyTax$, so in this example for simplicity, we omit to speak about the precondition axiom of $sendAckInjMail$ action and the context when it is permitted to apply it. To enforce these constraints, we can use the regression concept to calculate precondition axiom of the action $applyTax$. Another approach consists in introducing the concept of temporary activation of roles related to objects. This kind of constraints is known in the literature under the name of *object-based separation of duty* [Jaeger and Tidswell 2001]. Thereby we introduce the following:

- The fluent $Empower_{tmp}(s, r, o, t, ss, \sigma)$, namely the subject s plays the role r to act on object o at time t in the session ss.
- The action $Activate\_role_{tmp}(s, r, o, ss, t)$ activates the role r for the subject s to act on the object o at the time t.

The succession state axiom for $Empower_{tmp}$ is defined as:

$$\begin{aligned}
&Empower_{tmp}(s, r, o, t, ss, do(a, \sigma)) \leftrightarrow \\
&Assign(s, r, \sigma) \wedge a = Activate_role_{tmp}(s, r, o, ss, t) \vee \\
&(\exists r_1)[Sub_role(r, r_1, \sigma) \wedge Assign(s, r_1, \sigma) \wedge \\
&a = Activate_role_{tmp}(s, r_1, o, ss, t)] \vee False
\end{aligned}$$

This above axiom specifies that a temporary activated role is true just in the situation resulting from the execution of the action $Activate\_role_{tmp}(s, r, o, ss, t)$. Any other action is performed after restoring the fluent at a false value. Suppose now that the authorization to apply tax to a check which is subject to an injunction mail is associated with a temporary activation of role *Clerk*. Then a subject that temporary empowers the role $Clerk$ is permitted to apply a tax to the bad check but when he does, he will not yet empower the role $Clark$ to act again on the check, but there is nothing that prevents him to activate the *Clerk* role again. For this purpose, we should add the following constraint:

$$\begin{aligned}
&\neg\exists(mail, t, t')[check = subjOfInj(mail, \sigma) \wedge \\
&Empower_{tmp}(s, Clerk, check, t, ss, \sigma) \wedge \\
&check = subjOfInj(mail, \sigma') \wedge Empower_{tmp}(s, Clerk, check, t', ss, \sigma')
\end{aligned} \tag{4.19}$$

The check that is rejected one more time activates two different injunction mails. Normally it is permitted to apply tax to the check that is rejected

one more time but not during the 30 days after the last rejection. This is expressed by the following historical constraint:

$$\neg(\exists mail, mail')[check = subjOfInj(mail, \sigma) \wedge \quad (4.20)$$
$$Empower_{tmp}(s, Clerk, check, t, ss, \sigma) \wedge check = subjOfInj(mail', \sigma') \wedge$$
$$Empower_{tmp}(s, Clerk, check, t', ss, \sigma') \wedge t' - t \leq 30$$

4.4 Related work

Ahn and Sandhu [Ahn and Sandhu 1999, Ahn and Sandhu 2000, Bertino et al. 1999] propose a Role-based constraints specification language in which they identify useful role-based authorization constraints such as prohibition and obligation constraints. However, the proposed language is complex to use and it is difficult for an administrator to check if the specified constraints actually reflect the needed safety requirements. In our model there are just two kinds of constraints easy to grasp. Note that this does not affect the expressiveness of our model that allows us to express several constraints known in the literature. In fact, we can express obligation constraints using fluent *Ob* defined in chapter 3. We have seen how to derive the succession state axiom of this fluent. Thus we can calculate the regression of any formula involving fluent *Ob* and then enforce obligation constraints.

Tidswell and Jaeger propose a graphical access control model implementing administrative controls [Tidswell and Jaeger 2000b, Tidswell and Jaeger 2000a, Jaeger and Tidswell 2001]. In this model, the nodes represent sets of subjects, objects, etc. And the edges represent binary relationships on those sets. The constraints are expressed using a set of operators on graph nodes. The verification of various safety properties is done at run time involving a comparator function. With the construction of precondition axioms of actions using regression, our approach ensures in advance enforcement of constraints. The constraints will always be satisfied for all possible evolutions of the system from an initial situation that respects the constraints to situations resulting from execution of actions in accordance with precondition axioms (executable situations). At runtime execution of the system, ensuring that the precondition axiom of actions is satisfied amounts to solve the temporal projection problem which is decidable in polynomial time. In addition our approach allows us to express and enforce periodic temporary constraints which are not handled in other related works. Simpler specification scheme for separation of duty constraints for RBAC model is defined by Crampton in [Crampton 2003], but with this scheme, it is not possible to express for example that a subject is restricted from executing an operation on a particular object twice. This is what we express in the example related to taxing bad check presented

in the previous section. And more interesting, in this example we also show how we can specify the restrictions of acting on objects during a time interval. Similarly, we can express a periodic temporary constraints concerning obligations.
It is clear that in our model there is a step that requires more simplification, this consists in the definition of succession state axioms for fluents which belong to domain application. Our proposal in this case is to specify axioms of positive (resp.negative) effects on fluents and derivation rules and then build automatically succession state axioms.

4.5 Conclusions and contribution

We proposed in this chapter a formal language to express constraints policy. In our language, there is two kinds of constraints which we call historical and ahistorical constraints. We show how this language is adequate to express well-known constraints in the literature. To enforce constraints, first we propose to rewrite the historical constraints into simple formulas. We then give a procedure based on the regression concept to enforce these constraints. Furthermore, we formally specify the condition to prove that the system specification is secure with respect to the access control requirements and constraints policy.
Admittedly, constraints in access control models are essential to avoid fraudulent situations and ensure some desired properties which must be verified throughout the system execution. However their use leads to restrict the conditions under which actions can be performed. This may produce conflicting situations in obligation with deadline policies. For example, consider a situation where it is obligatory to do an action but it is impossible to execute it. The goal of the next chapter is to identify this kind of conflict and many others.

CHAPTER 5

Conflict Detection in Obligation with Deadline Policies

5.1 Introduction

The use of obligation rules in a security policies may cause conflicting situations. Preliminary work on the classification of conflicts are reported in [Moffett and Sloman 1993], where several types of conflicts have been defined (see also [Bertino et al. 1996, Dinolt et al. 1994]). [Benferhat et al. 2003] presents an approach based on possibilistic logic to deal with conflicts in prioritized security policies.
In this chapter, we deal with new types of conflict which we managed in [Essaouini et al. 2014a]. The first conflict arises between obligations with deadlines, namely conflict in feasibility of obligations. This kind of conflict could happen in the case of overlapping deadlines. For example: (i) The doctor is obliged to fill in the summary sheet within 1 hour after the patient leaves. (ii) The surgeon must be vigilant in the operating room. If the doctor is a surgeon and he is in the operating room during a patient's leaving, and if the duration of the surgery ends 2 hours after the patient's leaving, the surgeon cannot fill in the summary sheet of the patient because the surgery will end after the deadline associated with filling the summary sheet. Thus, there may be situations where it is impossible to meet certain obligation requirements of the security policy before their deadlines. We show in this chapter how we identify this conflict using executable plan [Green 1969a]. Given a goal formula, executable plan consists in finding a possible sequence of actions so that the goal is satisfied after executing this sequence of actions. The second conflict is between permissions and obligations with deadline. This conflict occurs when it is impossible to find a sequence of permitted actions which leads to a situation where obligations are fulfilled within their deadlines. We formally define the situations which correspond to such conflicts

by introducing the concept of a legal plan. A legal plan is the sequence of permitted actions.
This chapter is organized as follows. In section 5.2, we enrich the example of completion of medical records described in chapter 3 by adding some constraints and show how this causes some conflicts. In section 5.3, we define formally situations which present a conflict in the feasibility of obligations and a conflict between obligations and permissions. In this section, we also give the algorithm allowing the detection of these conflicts. In section 5.4, we implement our model using the programming language GOLOG [Levesque et al. 1997]. In this section, we make assessment on different situations that we build to simulate our model on the motivation example and discuss some performance evaluation. The related work is presented in section 5.5. Finally, we present the conclusion and contributions.

5.2 Motivation example

Let us consider the rules of example 1 specified in chapter 3. We now add the following constraints:

- C1: A doctor should spend at least 5 units of time writing each document.
- C2: Only one document should be written by the same doctor at the same time.

For expressing these constraints, we need to add the fluent WrittenDoc(d, type, p, t, t_e, σ), meaning the document of type *type* concerning patient p and created at time t has been written at time t_e in σ by d. The corresponding succession state axiom is defined as follows:

$$\begin{aligned}&Poss(a,\sigma) \rightarrow \\ &WrittenDoc(d,type,p,t,t_e,do(a,\sigma)) \leftrightarrow \\ &[a = EndWrite(d,type,p,t,t_e) \vee WrittenDoc(d,type,p,t,t_e,\sigma)]\end{aligned}$$

This axiom specifies that a document is considered written if the writing process is completed.
Thus the constraint $C1$ can be expressed as follows:

$$(\forall\sigma,\sigma').WritingDoc(d,type,p,t,t_s,\sigma) \wedge WrittenDoc(d,type,p,t,t_e,\sigma') \wedge (t_e - t_s) \geq 5$$

And the constraint $C2$:

$$\begin{aligned}(\forall\sigma).\neg(&WritingDoc(d,type,p,t,t_s,\sigma) \wedge WritingDoc(d,type',p',t',t'_s,\sigma) \wedge \\ &\neg(type = type') \wedge \neg(p = p') \wedge \neg(t = t'))\end{aligned}$$

Recall that the fluent $WritingDoc(d, type, p, t, t_s, \sigma)$, means that the document of type $type$ concerning patient p and created at time t is in writing process by d, and d began to write it at time t_s. The corresponding succession state axiom A.2 is described in the appendix A.
Using the regression principle which we discussed in chapter 4, we can build the precondition axiom of action EndWrite as follows:

$$Poss(EndWrite(d, type, p, t, t_e), \sigma) \leftrightarrow WritingDoc(d, type, p, t, t_s, \sigma) \wedge t_e \geq t_s + 5$$

In this axiom, we admit that the writing end is applied to an ongoing writing document. However the second constraint restricts the possibility of starting to write documents. Thus, the precondition axiom of StartWrite is given by the following formula:

$$Poss(StartWrite(d, type, p, t, t_s), \sigma) \rightarrow$$
$$\neg(\exists type', p', t', t'_s).WritingDoc(d, type', p', t', t'_s, \sigma)$$

Now let us reconsider the same sequence of actions described in figure 3.1, while considering the precondition axioms of actions that we specified above. The new figure 5.1 shows the truth value of predicate $Poss$ concerning the actions of writing documents. In this figure, we can see that there are situations where there are obligations to do actions but it is impossible to execute them. If these obligations are system obligations, we can conclude immediately that there is a conflict in the policy. However the purpose of obligations with deadline is to fulfill them before their deadlines expire. Thus, it is possible that there are situations in the future where these actions become possible to execute before the deadlines expire. In which case these situations will not be considered conflictual. The purpose of the following sections is to show how we can prove the existence or not of these situations in the future using a planning process. We make the same reasoning about situations where there are obligations with deadline to do actions which are not permitted.

5.3 Conflict detection

In this section, we distinguish two kinds of conflict:

- A conflict in feasibility of obligations, which occurs when it is impossible to fulfill obligations without violating some of them. This conflict is detected through the definition of executable plan. In this kind of conflict, we distinguish two types:
  - Conflict in the feasibility of an obligation with deadline.

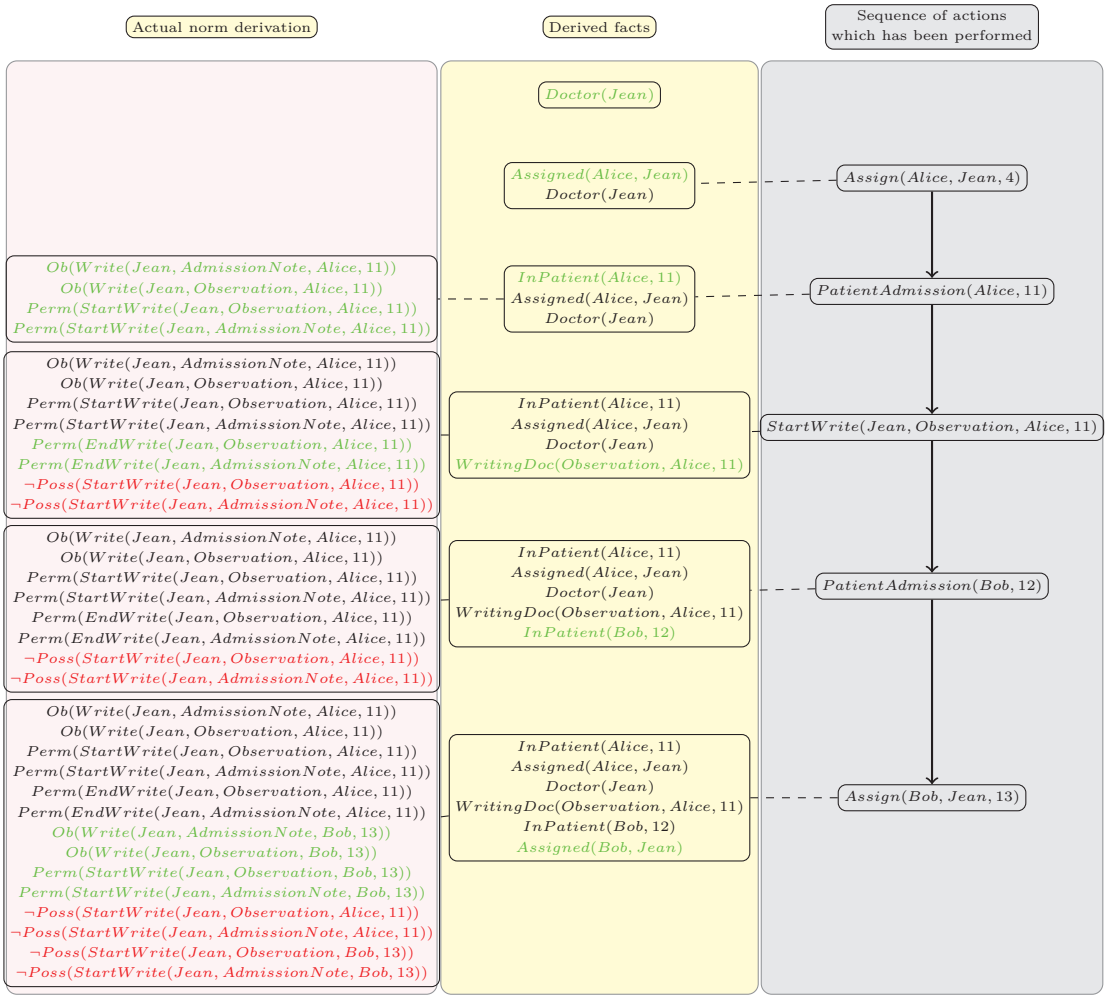

Figure 5.1: Motivation example

  - Global conflict in the feasibility between obligations with deadline.

- A conflict between obligations and permissions, which occurs when obligations can not be fulfilled in their deadlines by executing permitted actions. This conflict is detected through the definition of legal plan. In this kind of conflict, we distinguish two types:

  - Conflict between an obligation with deadline and permissions.
  - Global conflict between obligations with deadline and permissions.

5.3.1 Conflict in feasibility of obligations

An obligation is feasible in a situation σ if it is possible to execute it within its deadline by following an executable plan. To formalize this, we define fluent $Feasible(\alpha < d, \sigma)$.

$$Feasible(\alpha < d, \sigma) \leftrightarrow [Ob(\alpha < d, \sigma) \wedge$$
$$(\exists \sigma', \sigma < \sigma')(Fulfil(\alpha < d, \sigma') \wedge Executable(\sigma'))]$$

In the above formula, the obligation to do the action α is active in the situation σ and fulfilled in the situation σ' which it is an executable situation. When an obligation is not feasible in a situation σ, we say that there is a conflict in the feasibility of this obligation in the policy for this situation.
It may happen that in one situation, every active obligation is feasible, but it is still not possible to do all of them without violating the associated deadlines. If all the active obligations in a situation σ can be executed without violating at least one of them, we say that this situation is globally feasible. To characterize this, we introduce the formula $G\text{-}Feasible(\sigma)$.

$$G\text{-}Feasible(\sigma) \leftrightarrow (\exists \sigma')\sigma' > \sigma \wedge$$
$$(\forall \alpha, d)(Ob(\alpha < d, \sigma) \rightarrow (Fulfil(\alpha < d, \sigma') \wedge Executable(\sigma')))$$

Proving that a given situation σ is globally feasible, amounts to proving the existence of an executable situation where all the active obligations in σ are fulfilled. If the set of actions and the set of fluents are finite, we can prove that the existence of such a situation is decidable and can be solved in NEXPTIME complexity. This complexity of planning in the situation calculus is high but is similar to other planners, like Strips for example [Fikes and Nilsson 1971].
If a given situation is not globally feasible, then we shall say that there is a global conflict in the feasibility of obligations in the policy in this situation. There are several ways to solve this conflict:

- Changing the deadline of some obligations
- Creating an exemption which cancels some obligations
- Offering the possibility to delegate some obligations

Whereas obligations can be globally feasible, it is not possible to fulfill them by executing permitted actions. The purpose of the next section is to detect such situations.

5.3.2 Conflict between permission and obligation rules

There is a conflict between an obligation and a permission if the obligation cannot be done within its deadline by following an execution of a permitted action. In general, we call a sequence of permitted actions that achieves a given goal, a *legal plan*. Formally, a legal plan to achieve a goal G, is the task of establishing that:

$$Axiom \vDash (\exists\sigma)G(\sigma) \wedge Legal(\sigma)$$

If the set of actions and the set of fluents are finite, similarly to finding an executable plan, finding a legal plan is a decidable problem.
Through a legal situation, we define the fluent predicate $L\text{-}Enforceable(\alpha < d, \sigma)$, meaning the active obligation $Ob(\alpha < d, \sigma)$ in σ is enforceable in a legal situation σ'.

$$\begin{aligned} &L\text{-}Enforceable(\alpha < d, \sigma) \leftrightarrow \\ &(\exists\sigma').\sigma' > \sigma \wedge Fulfil(\alpha < d, \sigma') \wedge Legal(\sigma') \end{aligned}$$

In the formula above, finding the legal situation σ', consists in finding a legal plan leading to fulfill the active obligation α in σ.
Similarly to the global conflict between obligations, we define the global conflict between obligations and permissions using a legal plan. There is a global conflict between obligation and permissions in a policy in a given situation σ, if this situation is not globally enforceable by following a legal plan. A globally enforceable situation by following a legal plan is defined as follows:

$$\begin{aligned} &LG\text{-}Enforceable(\sigma) \leftrightarrow \exists\sigma', \sigma' > \sigma \wedge (\forall\alpha, d) \\ &Ob(\alpha < d, \sigma) \rightarrow Fulfil(\alpha < d, \sigma') \wedge Legal(\sigma') \end{aligned}$$

In the above formula, all the active obligations in σ are fulfilled in the legal situation σ'. In the next section, we give an algorithm to detect the conflicts we formalized in this section.

5.3.3 Conflict detection algorithm

In what follows, we assume the existence of a temporal reasoning component that allows us to infer, for example, that $T_1 = T_2$ when $T_1 \leq T_1$ and $T_2 \leq T_2$, and we are able to solve linear equations and inequalities over the reals using the Simplex algorithm [Thom Fruhwirth 1992].
The Algorithm 1 detects the different types of conflict we have defined using recursive search as defined in Algorithm 2. In this algorithm, we allow the execution of parallel

actions otherwise we can use constraints to specify the actions which cannot be done in parallel. Note that we suppose that if the situation we check is globally feasible (resp. legally globally enforceable), then this situation must be executable (resp. legal). Proving that a situation is executable (resp. legal) can be done using regression [Reiter 1991], where testing is reduced to proving first order theorems in the initial situation.

Furthermore, if the set of actions and the set of values are finite, we can estimate the

Algorithm 1 ConflictDetection(σ, N, conflictType)

Require: σ: the situation to check
N: the maximal depth
conflictType: the type of the searched conflict, "FC"for feasibility conflict, "LC"for legally conflict and "SC"for strong conflict

Ensure: No: if there is no conflict of type conflictType in the policy at situation s otherwise Yes.

$\mathcal{O} = \{\alpha \in \mathcal{A}$ **such that** $Ob(\alpha < d, \sigma)\}$ {set of active obligations in s}
$s' \leftarrow$ recursiveSearch(s, N, $\mathcal{O}$, conflictType)
if $\neg(s' = NULL)$ **then**
  return No {there is no conflict of type conflictType in the policy at σ and σ' is the plan which leads to fulfill all the active obligations in σ}
else
  return Yes {there is a conflict of type conflictType in the policy in situation σ}
end if

maximum length of the plan, N, allowing to achieve the goal.
In our algorithm, we explore the tree of all possible worlds that can be very large. Indeed, if we suppose that on average, there are k actions which are possible to execute from a given situation, then the number of worlds to explore is the order of k^N.
In the following section, we show how to optimize the search tree for finding a situation where obligations are fulfilled if it exists.

5.4 Implementation

We implement our model using the logic programming language Golog ([Levesque et al. 1997, Reiter 1998]), based on the situation calculus. Regarding our need to solve linear equations and inequalities, we use the Common Logic Programming System ECLIPSE 3.5.2, which provides a built-in Simplex algorithm for

Algorithm 2 recursiveSearch(σ, N, $\mathcal{O}$, conflictType)

Require: s: the current situation
N: the current depth (initially the given maximum depth)
$\mathcal{O}$: set of active obligations in σ
conflictType: the type of the searched conflict, "FC"for feasibility conflict, "LC"for legally conflict and "SC"for strong conflict

Ensure: Null: if the depth of the current path exceeds the given maximum depth or, situation when all obligations in $\mathcal{O}$ are fulfilled if it exists otherwise, the next situation to give to the next call for recursion

switch (conflictType)
case "FC"**:**
$\mathcal{E} \leftarrow \{a \in \mathcal{A}, Poss(a, \sigma) \wedge Start(\sigma) \leq Time(a)\}$ {the set of actions that can lead from σ to an eventual executable situation}
case "LC"**:**
$\mathcal{E} \leftarrow \{a \in \mathcal{A}, Perm(a, \sigma)\}$ {the set of actions that can lead from σ to an eventual legal situation}
case "SC"**:**
$\mathcal{E} \leftarrow \{a \in \mathcal{A}, Poss(a, \sigma) \wedge Start(\sigma) \leq Time(a) \wedge Perm(a, \sigma)\}$ {the set of actions that can lead from σ to an eventual legal and executable situation}
end switch
while true **do**
if $N < 0$ **then**
return NULL
end if
for all $a \in \mathcal{E}$ **do**
$\sigma' \leftarrow do(a, \sigma)$
$N \leftarrow N - 1$
if $(\forall \alpha, d \in \mathcal{O}) Fulfil(\alpha < d, \sigma')$ **then**
return σ'
end if
$\sigma'' \leftarrow recursiveSearch(\sigma', N, \mathcal{O}, conflictType)$
if $\neg(\sigma'' = NULL) \wedge (\forall \alpha, d \in \mathcal{O}) Fulfil(\alpha < d, \sigma'')$ **then**
return σ''
end if
$N \leftarrow N + 1$
end for
return NULL
end while

solving linear equations and inequalities over the reals.

The point of departure for the implementation is to get the list of all active obligations in a given situation S. This is given using the predicate *activeObligations(ActiveObligationsList,S)*:

```
activeObligations(ActiveObligationsList,S):- findall(Rule,ob(Rule,S),
                                              ActiveObligationsList).
```

Given the list of active obligations, seeking the situation where all these obligations are fulfilled is made using the procedure *plan*.

```
proc(plan(N,L),
     ?(all(r,member(r,L) => fulfil(r))):?(reportStats)#
     ?(N > 0):
     pi(a,?(primitiveAction(a)):a):
     ?(-badSituation):
     pi(n,?(n is N-1):plan(n,L))).
```

Here, *primitiveAction* is a predicate characterizing all the actions of the domain. If $N = 0$ the execution of the procedure ends. If $N > 0$, a primitive action a is selected. The Golog interpreter checks if the selected action a is possible and verifies that $start(s) \leq time(a)$, where s is the current situation. If so, a is executed and $do(a, s)$ becomes the new current executable situation.

we can change the following instruction of Golog:

```
do(E,S,do(E, S)) :- primitiveAction(E), poss(E,S),
                    start(S,T1),time(E, T2),T1 <= T2.
```

and replace it with the following statement for searching a legal plan:

```
do(E,S,do(E, S)) :- primitiveAction(E),perm(E, S),
                    start(S,T1),time(E,T2),T1 <= T2.
```

In our implementation we make no change in the Golog interpreter but every precondition axiom of an action includes the fact that this action is permitted using the fluent *perm*. Thus we test whether a situation is strongly enforceable or not using the predicate *sEnforceable(N,S1)*.

```
sEnforceable(N,S1):- initializeCPU,
                     activeObligations(ActiveObligationsList,S1),
                     do(plan(N, ActiveObligationsList),S1,S),
                     prettyPrintSituation(S).
```

Let us start by showing how we can write axioms of our example using Golog. The complete description of axioms is described in the file *conflictualSituation.pl*.

Examples of succession state axioms:

```
ob(write(D,Type,P,T),do(A,S)):- (assigned(P,D,S),
                                 A=patientAdmission(P,T));
```

```
                                    (ob(write(D,observation,P,T),S),
                                    not A=endWrite(D,observation,P,T,T1),
                                    not A=endDeadline(observation,P,T,T2),
                                    not A=leave(P,T3),
                                    not A=revokeAssignation(P,D,T4)).

fulfil(write(D,Type,P,T),do(A,S)):- (ob(write(D,Type,P,T),S),
                                    (A=endWrite(D,Type,P,T,T2));
                                    fulfil(write(D,Type,P,T),S)).
```

Examples of action precondition axioms:

```
poss(startWrite(D,observation,P,T,T1),S):- inpatient(P,T,S),
                                           assigned(P,D,S),
                                           not writingDoc(D,Type1,P1,T3,T4,S),
                                           not writtenDoc(D,observation,P,T,S).

poss(endWrite(D,Type,P,T,T1),S):- writingDoc(D,Type,P,T,T2,S),T1 $>= T2+5.
```

In addition to the succession state axioms and precondition axioms of actions, we suppose having the following axioms in the initial situation s_0.

```
start(s0, 0).
doctor(jean).
```

Description of bad situations:

The badSituation test is used to remove partial plans which are known in advance to be unsuccessful. For example, a branch resulting from the execution of an action that disables an active obligation can be eliminated. A branch resulting from the execution of an action that activates the deadline corresponding to an active obligation may also be removed. We will see in the following how this can be done in the implementation of our example.

If a violation of an active obligation occurs after the execution of an action, it is no longer necessary to continue searching a solution from the resulting situation.

```
badSituation(do(A,S)):- A=endDeadline(Type,P,T,T1),
                        not fulfil(write(D,Type,P,T),S),!.
badSituation(do(A,S)):- A=endDeadline(Type,P,T,T1),
                        poss(endDeadline(Type1,P1,T2,T3),S),T3 $< T1,!.
badSituation(S):- ob(endDeadline(Type,P,T,T1),S),start(S,T2),
                  not (T1$>=T2),!.
```

If an active obligation is deactivated after the execution of an action, it is no longer necessary to continue searching a solution from the resulting situation.

```
badSituation(do(A,S)):- A=leave(P,T),!.
badSituation(do(A,S)):- A=revokeAssignation(P,D,T),!.
```

The construction of these bad situations can be done using the succession state axioms of fluents *Ob* and *Fulfil*. Thus these optimizations can be generalized to any policy

without losing the completeness of the planning search. In our example, the execution of actions *patientAdmission* and *assign* activates other obligations and has no impact on the fulfillment of obligations which are already active. Therefore the path resulting from their execution can be eliminated in the solution search.

```
badSituation(do(A,S)):- A=assign(P,D,T),!.
badSituation(do(A,S)):- A=patientAdmission(P,T),!.
```

This optimization is closely related to our example because there is nothing that prevents to have actions in the policy that are necessary to fulfill obligations but their execution leads to activate other obligations simultaneously.
We have performed tests that check the strongly enforceability of situations constructed as follows: the first situation checked by the first test, denoted $s1$ is the result of the assignment of a single patient $p1$ to the doctor jean at time 4 followed by his admission in the hospital at time 5.

```
test1:- sEnforceable(6,do(patientAdmission(p1,5),do(assign(p1,jean,4),s0))).
```

The next situation $s2$ is the assignment of another patient $p2$ to jean at time 6 from the situation $s1$ followed by the admission of $p2$ in the hospital at time 7.

```
test2:- sEnforceable(12,do(patientAdmission(p2,7),do(assign(p2,jean,6),
                        do(patientAdmission(p1,5),do(assign(p1,jean,4),s0))))).
```

We build 20 tests. Their complete description is in the file *test.pl*. The planning depth research is calculated as follows. In our application domain, there are seven actions, four of them are removed from the planning through the specification of *badSituation*. The remainder actions are: *startWrite*, *endWrite* and *endDeadline*. In the database, there is one doctor Jean and two types of documents, so for each patient, these actions are possible twice, one for each type of document. When one of these actions is executed, it is not possible to execute it again according to their precondition axioms. Thus when all these actions are performed one after the other, it is no longer possible to perform other actions except those which are discarded from the planning search. Thereby whenever a patient is added, the minimum depth ensuring the decidability of solution research is increased by six.
We conducted two series of tests depending on the deadlines associated with the obligations to write documents. The experiment was run on a machine equipped with an Intel 32bit 2.60GHzx4 processor and 3,8GB RAM, running ECLIPSE 3.5.2 on ubuntu Linux(v.13.04).
The first series of tests: the deadline for writing admission note is 30 units of time and the observation is 40 units of time. In this series of tests, the maximum number of patients, who can be admitted in the hospital and assigned to *jean*, without causing

conflict between the obligations is 4. Indeed, the policy is not conflictual in the first four situations. As example, the following legal plan generates a situation when all the active obligations in the situation $s1$ are fulfilled which means that $s1$ is strongly globally enforceable.

```
[eclipse 2]: test1.

  CPU time (sec): 0.00

[assign(p1,jean,4),patientAdmission(p1,5),startWrite(jean,
admissionNote,p1,5,_374),endWrite(jean,admissionNote,p1,5,_1095),
startWrite(jean,observation,p1,5,_2264),endWrite(jean,observation,p1,5,_3112),
endDeadline(admissionNote,p1,5,35),endDeadline(observation,p1,5,45)]
more? n.

Linear Store:

_3112 $>= 15+1*_2321+1*_1187+1*_431
_2264 $>= 10+1*_1187+1*_431
_1095 $>= 10+1*_431
_374 $>= 5
```

The above plan contains uninstantiated temporal variables. The value of these variables is just constrained by the inequalities in ECLIPSE's linear constraint store. Although, there may be cases when plans are fully specified like the following third test which proves that the policy remains consistent after the admission of three patients.

```
[eclipse 4]: test3.

 CPU time (sec): 0.03

[assign(p1,jean,4),patientAdmission(p1,5),assign(p2,jean,6),
patientAdmission(p2,7),assign(p3,jean,8),patientAdmission(p3,9),
startWrite(jean,admissionNote,p3,9,9),endWrite(jean,
admissionNote,p3,9,14),startWrite(jean,admissionNote,p2,7,14),
endWrite(jean,admissionNote,p2,7,19),startWrite(jean,
admissionNote,p1,5,19),endWrite(jean,admissionNote,p1,5,24),
startWrite(jean,observation,p3,9,24),endWrite(jean,
observation,p3,9,29),startWrite(jean,observation,p2,7,29),
endWrite(jean,observation,p2,7,34),startWrite(jean,
observation,p1,5,34),endDeadline(admissionNote,p1,5,35),
endDeadline(admissionNote,p2,7,37),endWrite(jean,observation,p1,5,39),
endDeadline(admissionNote,p3,9,39),endDeadline(observation,p1,5,45),
endDeadline(observation,p2,7,47),endDeadline(observation,p3,9,49)]
more? n.
```

Finally the fifth test shows how the admission of a fifth patient produces a conflict.

```
[eclipse 6]: test5.

No (1460.33s cpu)
```

The second series of tests: the deadline for writing admission note is 1000 units of time and the observation is 1100 units of time.

In this series of tests we check 20 situations. The table 5.1 summarizes the results obtained and the execution time for each tested situation. This table describes the most important parameters influencing the time of executions: the number of active obligations and the depth of the solution search.
On average, there are $2 \times nb$ actions, which can be executed from a given situation, where nb is a number of admitted patients. Moreover, the plan to achieve the desired goal is made up of $nb \times 6$ actions. Then, the number of situations to explore is in the order of $(2 \times nb)^{nb \times 6}$. This explains the increment in the time duration, each time a patient is admitted.

Table 5.1: The summary of second series of tests

Tests	Patient number	Active obligation number	The minimum research depth	CPU time (sec)	Strongly globally enforceable ?
$s1$	1	4	6	0	yes
$s2$	2	8	12	0.01	yes
$s3$	3	12	18	0.03	yes
$s4$	4	16	24	0.06	yes
$s5$	5	20	30	0.1	yes
$s6$	6	24	36	0.2	yes
$s7$	7	28	42	0.36	yes
$s8$	8	32	48	0.66	yes
$s9$	9	36	54	1.1	yes
$s10$	10	40	60	1.78	yes
$s11$	11	44	66	2.78	yes
$s12$	12	48	72	4.16	yes
$s13$	13	52	78	6.03	yes
$s14$	14	56	84	6.04	yes
$s15$	15	60	90	8.60	yes
$s16$	16	64	96	11.95	yes
$s17$	17	68	102	16.43	yes
$s18$	18	72	108	21.75	yes
$s19$	19	76	114	28.89	yes
$s20$	20	80	120	37.97	yes

5.5 Related work

Most traditional security models are static and respond to access requests just by yes (accept) or no (deny). Recently, there are more and more works on security models that handlle obligations [Bettini et al. 2002a, Damianou et al. 2001, Ni et al. 2008, Hilty et al. 2007, Craven et al. 2009]. Formalization of obligations differs from one model to another. In XACML [XACML], obligations are all operations that must be met in conjunction with the application of the authorization decision. In [Bettini et al. 2002a, Hilty et al. 2005, Hilty et al. 2007], distinction is made between provisions and obligations. Provisions are actions or conditions that must be met before authorizing access. Obligations are actions that must be executed by users or system after the access is given. The ABC model (Authorization, Obligation and

Condition) [Park and Sandhu 2004] was specifically designed to express security policies including usage control constraints. The expression of a constraint to be satisfied before the use of an object can be expressed as contextual authorization. However, constraints to meet during or after the use of an object relate to obligations that the user must follow. The NOMAD model [Cuppens et al. 2005] is based on a formalization in temporal and deontic logic to express contextual obligations which should be met before, during or after the execution of an action. It is also possible to specify a deadline after which some obligation will be considered violated if the action was not performed. Authors in [Sans et al. 2007], define a core language to specify the access and usage control requirements and then give a formalism based on the logic of Temporary Actions (TLA) [Lamport 1994] to specify the behavior of the policy controller in charge of evaluating such policy. In this approach, a permission is associated with two conditions, the first must be true at the time of query evaluation, and the second must always be true as long as access is in progress. The authors also introduce a concept to reset a current access. Regarding obligations, they are associated with two conditions. Once the first condition is satisfied, the obligation is triggered then the controller sends a notification to the user to perform the appropriate obligation, the second condition determines when the obligation should be considered violated. If the user does not satisfy the obligation before the second condition becomes true, a penalty is applied to him. The authors in [Goedertier and Vanthienen 2006] talk about what they called deontic conflicts. The types of conflict that the authors have classified in this category are those that occur between permission and prohibition and those which occurs between obligation and obligation waiver. As in the used formalism the authors do not use prohibition and obligation waiver modalities, they do not deal with these conflicts in their work. But in this category, there is another kind of conflict which is the conflict between the obligations with deadlines and permissions. In our work, this conflict is detected when there is no plan consisting of permitted actions that lead to fulfilling an obligation requirement within its deadline. In other words it is possible that in a given situation, a mandatory action is permitted and it can be fulfilled within its deadline but it is not possible to execute because it is necessary to first execute other actions which are not permitted. Certainly, the authors define another type of conflict called temporal conflicts which occur when two deontic assignments at the same time initiate and terminate obligation. This is a particular case of detection that we called the global conflict between the obligations with deadlines. Indeed, in a given situation, it may be possible to fulfill an active obligation within its deadline but given that there are other active obligations at the same time it is not possible to fulfill them together without violating one of them. The conflict in the temporal constraints is actually a special case of a “logical”conflict that we detect with the concept of executable plan.

5.6 Conclusion and Contribution

In this chapter, we first begin by formally defining the conflicting situations using the situation calculus. Afterwards, we provide an algorithm for searching a plan of actions, when it exists, which fulfills all the active obligations in a given situation within their deadlines with respect to the permission rules. The length of the plan is set in advance and can be calculated in the case where the sets of actions and fluents are finite to ensure the decidability of the solution search. Furthermore, in the plan search, the choice of the execution time of the elected actions obeys to equations and inequalities which need to be solved. For this purpose, we use a component allowing these equations and inequalities resolution. To illustrate our approach, we take the example described in chapter 3 inspired from existing laws in hospitals regulating deadlines for completing patient medical records. The example is implemented in ECRC Common Logic Programming System ECLIPSE 3.5.2, which is equipped with the Simplex algorithm for solving linear equations and inequalities over the reals. In the implementation, we show how the plan search can be optimized through the use of some heuristics and make some evaluation tests.
However, the existence of conflicting situations, does not necessarily imply that there is a conflict in the policy. Sometimes a conflict occurs because there is an accumulation of old obligations that have not been fulfilled and remained active. We will argue about the nature of the activities that led to the accumulation of such obligations for determining if the conflict comes from the policy or it is the responsibility of agents. For that we need to enrich our language with another modality which it is the modality of right. Indeed we need to distinguish an activity resulting from executing a permitted action from an activity resulting from executing an action when there is a right to perform it. This is the purpose of the following chapter.

CHAPTER 6

Enriching Obligation with Deadline Policies with Rights

6.1 Introduction

In this chapter, our main concern is to show how enriching security policies with right rules allows us to improve the expressiveness of policy languages. A right to do an action is associated with a context which corresponds to the set of conditions that must hold in order to make the right active. Whenever a right is active, the execution of the corresponding action should be possible, making the right ensured, otherwise there is a violation of right. However making an action possible does not mean that the action must be executed, while when an action must be executed (i.e., obligation), it must necessarily be possible. Otherwise, there is a conflict in the feasibility of the obligation rule. Thus, unlike obligations, it remains to the user to choose to exercise its right or not.

However, we consider that obligations and rights have one thing in common insofar as both can lead to situations of violation and sanction. But the semantic we give to the violation of a right is different from that of obligation with deadline. The violation of right is not associated with the fact that the user does not perform the action before a deadline, but the fact that there are circumstances in a system or some users' behaviors which made the action impossible to be executed. These situations should be detected even if the users concerned by the right did not try to execute the action. For example, consider a user who has paid to access an online document. If the online document is deleted or the corresponding host is stopped for some reasons, the access is no longer possible (ex. HTTP Error 503-Service unavailable) but not forbidden (ex. HTTP 403-Forbidden). Thus, this can not be considered as a conflict between permissions and interdictions but more specifically an access permission impossible to meet. In

addition, assume that the failure to access to this document induces the refund of the costs that the user paid. Such an approach is a penalty resulting from a situation where an access permission has not been satisfied. In these circumstances, the access to this document should be expressed in the policy as a right. Situations where the server is down or the document is deleted correspond to situations where the right has been violated. The violation here has no connection with the fact that the user has not fulfill an obligation to access to this document. But would protect the users's right to access to a document which they paid for, while keeping their own choice to execute this access or not. In general, rights are often associated with penalties in case of violations. For example, CNIL (Commission National de l'informatique et des libertés) [1] gives a list of rights and made provisions of steps to follow if these rights are violated[2].

In this chapter, we first extend our language by considering a right modality. Then, we show how to capture every situation where something happens which prevents right to be executed. Furthermore, using right rules in security policies leads to a new type of conflict which is, to the best of our knowledge, has received scarce attention in the literature on the topic. It is a conflict that arises when the execution of a right prevents an obligation from to be fulfilled in its deadline or conversely; fulfilling an obligation in its deadline necessarily leads to the violation of a right. In this chapter, we introduce the concept of *preserved plan* to formaly define a conflict between obligation with deadline and right rules. Indeed, a preserved plan consists of a sequence of actions that causes no violation of any current right. When there is no preserved plan that leads to fulfill active obligations in their deadlines, we say that there is a conflict between obligations and rights.

This chapter is organized as follows. Section 6.2 presents a motivation example. In section 6.3, we extend our language with right modality to enable the specification of right rules. Section 6.4 extends situation calculus to formally derive where a right is effective. In this section, we also formally define right violation. Section 6.5 shows how to detect the presence of conflicting norms in the policy. Our right modeling is then compared to some of existing work modeling rights in section 6.6. Finally section 6.7 concludes this chapter.

[1] An independent French administrative authority. It is responsible for ensuring that information technology is at the service of citizens and it does not affect human identity, nor the rights or privacy, or individual and public freedoms. It carries out its tasks in accordance with Law No. 78-17 of 6 January 1978 amended August 6, 2004

[2] http://www.cnil.fr/vos-droits/plainte-en-ligne/

6.2 Motivation example

In a code of medical ethics [3], it is stated that physicians are encouraged to attend training and this is expressed as a right by this law. Thus, let us enrich the motivation example described in 5.2 by adding the following rules:

- **Right rule**:
  - r: The doctor has **right** to join training.
- **Permission rules**:
  - p_1: It is permitted to start training at unit time 30.
  - p_2: It is permitted to end training at unit time 90.
  - p_3: The doctor has a permission to join training.
  - p_4: The doctor has a permission to leave training.
  - p_5: It is permitted to start the coffee break at unit time 10.
  - p_6: It is permitted to end the coffee break at unit time 30.
  - p_7: The doctor has a permission to start to take a coffee break at the coffee break moment
  - p_8: The doctor has a permission to end to take a coffee break when he is in coffee break.

  Furthermore, we saw in the previous chapters that there may be requirements on the circumstances under which the medical records must be written to ensure better quality. We saw that it is stated in [ANAES 1996] that the quality of documents is directly related to the time spent on their outfit. This prompted us to add constraints specifying the minimum time needed for the completion of documents. We think that be vigilant can be also an important requirements to ensure the better quality of documents. Thus, it appears to be natural to add the following constraint to our example.
- **Constraint**:
  - C3: No patient document is written by a doctor while he is in training or in coffee break.

[3] CODE DE DEONTOLOGIE MEDICALE: http://www.comores-droit.com/code/deontologiemedicale

Let us consider the sequence of actions described in figure 6.1. Jean has a possibility between starting writing the admission note of Alice, or her observation note, or joining a training. He can not do any of these actions in parallel otherwise he will not satisfy the policy constraints. However, if Jean exercises his right to attend the training until the end, i.e., 90 units of time, the obligations concerning writing the documents of Alice will be violated. Similarly, requiring from Jean to fulfill these obligations deprives him from his right to attend training which lead also to a violation of right. The violation in this case is not due to the fact that Jean will not attend a training (attending a training is not an obligation for Jean) but the fact that the succession of events prevents him from having the choice to do it or not. In that case, the action is no longer possible to be executed if he starts fulfilling his obligations. Keeping for the user the choice to exercise his right or not at any moment leads us to consider that the defined policy is conflictual in this kind of situation.
In the remainder of this chapter, we show how we detect this kind of conflict.

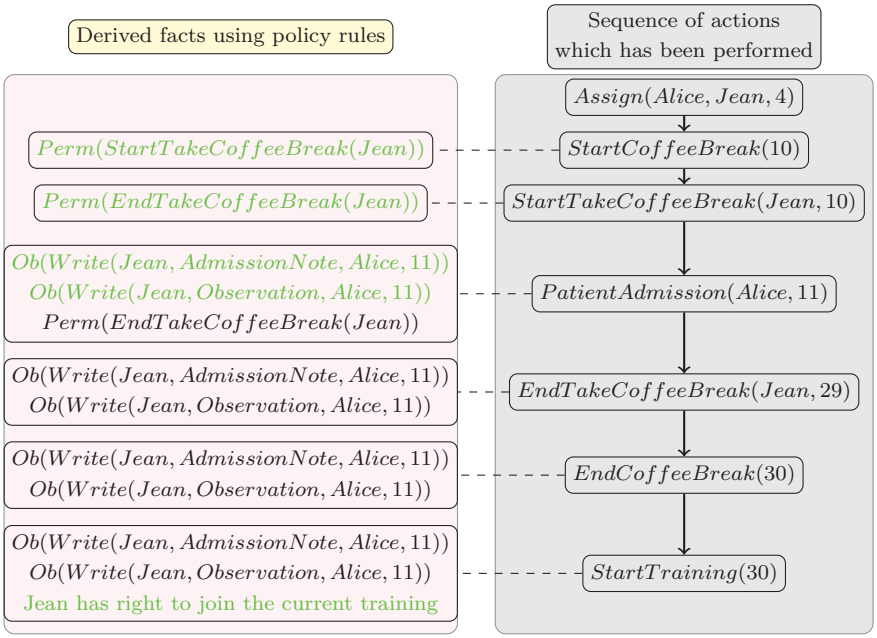

Figure 6.1: Right example

6.3 Specification of right rules

We extend our language by adding the right modality. Modality $R(\alpha|p)$ means there is a right to do α when condition p holds. Thus, we extend a security policy $\mathcal{P}$ with a *right norm* corresponding to a conditional right.

Example 6 In this example, we show how to specify the rule r given in section 6.2 using our language. For this purpose, we extend the set of fluents $\mathcal{F}$ and the set of actions $\mathcal{A}$.
The set $\mathcal{F}$ is extended by the following fluent:

- $Training(t, \sigma)$: There is a training in σ started at time t.

The set $\mathcal{A}$ is extended by the following actions:

- $StartTraining(t)$, $EndTraining(t)$: The action to start (resp. end) a training.
- $JoinTraining(s, t)$, $LeaveTraining(s, t)$: s joins (resp.leave) training.

Specification of the right rule:

$$(r): \quad R(JoinTraining(d, t) \mid Doctor(d) \wedge (\exists t')Training(t', \sigma) \wedge t > t')$$

We shall now use the situation calculus to formally define the semantic of right.

6.4 Derivation of effective rights and detection of rights violation

In this section, we show how to derive effective right, characterizing situations where right is ensured and detecting the violation of rights.

6.4.1 Derivation of effective rights

The situation calculus is extended with fluent $Right(\alpha, \sigma)$, meaning there is an actual right to do α in a situation σ. Let R_α be the set of conditional rights of $\mathcal{P}$ having the form $R(\alpha|p)$. We denote $\psi_{R_\alpha} = p_1 \vee ... \vee p_n$ where each p_i for $i \in [1, ..., n]$ corresponds to the condition of a right in R_α. If $R_\alpha = \emptyset$, then we assume that $\psi_{R_\alpha} = false$.
The succession state axiom of fluent $Right(\alpha, \sigma)$ is defined as follows:

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &\quad Right(\alpha, do(a, \sigma)) \leftrightarrow \gamma^{+}_{\psi_{R_\alpha}}(a, \sigma) \vee (Right(\alpha, \sigma) \wedge \neg\gamma^{-}_{\psi_{R_\alpha}}(a, \sigma)) \end{aligned} \tag{6.1}$$

This axiom specifies that a right becomes effective immediately after the execution of action that activates the condition associated with it. Then this right will stay effective in all following situations unless an action which disables the condition associated with it is executed.

Example 7 Let give where the right of a doctor to join a training is effective.
We suppose that a training is underway in a given situation if action StartTraining was executed and EndTraining did not.

$$\begin{aligned} &Poss(a,\sigma) \rightarrow \\ &\quad Training(t, do(a,\sigma)) \leftrightarrow a = StartTraining(t) \vee \\ &\quad Training(t,\sigma) \wedge \neg(\exists t')a = EndTraining(t') \end{aligned}$$

From the specification of the rule r and the succession state axiom above, we can derive that the right of a doctor to join a training becomes effective upon the execution of action StartTraining.

$$\gamma^{+}_{\psi_{R_{JoinTraining(d,t)}}}(a,\sigma) \leftrightarrow Doctor(d) \wedge (\exists t')a = StartTraining(t')$$

This right is disabled once the training is completed (i.e., action EndTraining is executed).

$$\gamma^{-}_{\psi_{R_{JoinTraining(d,t)}}}(a,\sigma) \leftrightarrow (\exists t')a = EndTraining(t')$$

Thus using axiom (6.1), situations where there is a right to join a training are characterized by the following axiom:

$$\begin{aligned} &Poss(a,s) \rightarrow \\ &Right(JoinTraining(d,t), do(a,\sigma)) \leftrightarrow Doctor(d) \wedge \\ &(\exists t')a = StartTraining(t') \vee \\ &Right(JoinTraining(d,t),\sigma) \wedge \neg(\exists t')a = EndTraining(t') \end{aligned} \tag{6.2}$$

Note that so far, there is really no distinction between permissions and rights. The distinction will be highlighted in the next section concerning the detection of violations.

6.4.2 Violation detection

In our formalism, no action can prevent the enforcement of a granted right. Otherwise there is a violation of right. The language is then extended by fluent $Ensured(\alpha,\sigma)$ meaning the right to do α is assured in the situation σ. In other words, in all situations if the right to do an action is active the action must be possible.

$$Ensured(\alpha,\sigma) \overset{def}{\leftrightarrow} Right(\alpha,\sigma) \wedge Poss(\alpha,\sigma)$$

If the precondition axiom of α is written as: $Poss(\alpha, s) \leftrightarrow \phi_\alpha(\sigma)$, then the succession state axiom of $Ensured(\alpha, \sigma)$ can be derived as follows:

$$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &Ensured(\alpha, do(a, \sigma)) \leftrightarrow (Right(\alpha, \sigma) \wedge \gamma^+_{\phi_\alpha}(a, \sigma) \wedge \neg\gamma^-_{\psi_{R_\alpha}}(a, \sigma)) \vee \\ &(\phi(\sigma) \wedge \gamma^+_{\psi_{R_\alpha}}(a, \sigma) \wedge \neg\gamma^-_{\phi_\alpha}(a, \sigma)) \vee \\ &(Ensured(\alpha, \sigma) \wedge \neg\gamma^-_{\psi_{R_\alpha}}(a, \sigma) \wedge \neg\gamma^-_{\phi_\alpha}(a, \sigma)) \end{aligned} \tag{6.3}$$

Example 8 Let us consider the right of doctor to join a training. To characterize the situations where this right is ensured, we must first give the precondition axiom of action JoinTraining.
A doctor d can join a training if he is not writing a document of a patient assigned to him, this is to not violate the constraint $C3$. Thus we admit that the precondition axiom of action JoinTraining is as follows:

$$\begin{aligned} &Poss(JoinTraining(d, t), \sigma) \leftrightarrow \\ &\neg(\exists type, p, t', t'')WritingDoc(d, type, p, t', t'', \sigma) \end{aligned} \tag{6.4}$$

Here, $\phi_\alpha(\sigma)$ corresponds to formula:

$$\neg(\exists type, p, t', t'')WritingDoc(d, type, p, t', t'', \sigma)$$

Thus, to build the succession state axiom of Ensured, according to axiom 6.3, we use the succession state axiom of WritingDoc A.2. Thereby,

$$\gamma^+_{\phi_\alpha}(a, \sigma) \leftrightarrow (\exists t_e)a = EndWrite(d, type, p, t', t_e)$$

and,

$$\gamma^-_{\phi_\alpha}(a, \sigma) \leftrightarrow a = StartWrite(d, type, p, t', t'')$$

Finally, using the axiom 6.3, we can derive situations where right to join training is ensured.

$$\begin{aligned} &Poss(a, s) \rightarrow \\ &Ensured(JoinTraining(d, t), do(a, \sigma)) \leftrightarrow \\ &[Right(JoinTraining(d, t), \sigma) \wedge \\ &(\exists type, p, t', t'')a = EndWrite(d, type, p, t', t'')] \vee \\ &[\neg(\exists type, p, t')WritingDoc(d, type, p, t', \sigma) \wedge (\exists t'')a = StartTraining(t'')] \vee \\ &[Ensured(StartAttendTraining(d, t), \sigma) \wedge \\ &\neg(\exists type', p', t', t'')a = StartWrite(d, type, p, t', t'')] \end{aligned}$$

The violation of a right occurs in situations where the right is not ensured. The violation of right is captured by the fluent $Violated_R(\alpha, \sigma)$.

$$Violated_R(\alpha, \sigma) \overset{def}{\leftrightarrow} Right(\alpha, \sigma) \wedge \neg Ensured(\alpha, \sigma)$$

Thus we can detect the situation where there is violation of right. However, we do not distinguish who is responsible of this violation. For example, if a doctor starts writing a document at the same time as the starting of the training, his right to join the training will not be ensured for some moments. In this case, it is a doctor who choice to waive his own right.

Furthermore, it might be useful to get information about exercised rights. A right to do an action is exercised when the action is done in situations where the right is active. This is characterized by the following axiom:

$$Poss(a, \sigma) \rightarrow$$
$$Exercised(\alpha, do(a, \sigma)) \leftrightarrow (Right(\alpha, \sigma) \wedge a = \alpha) \vee Exercised(\alpha, \sigma)$$

Exercising a right does not automatically deactivate it. Like permission, as long as a user has a right to do an action, he can perform it if the related conditions are satisfied.

6.5 Conflicting situations

In this section, we investigate three types of conflict:

- A conflict in an exemplary feasibility of obligations. This conflict occurs when there is no executable plan to fulfill obligations within their deadlines without violating new active obligations.
- A conflict in the feasibility of rights. This conflict occurs when rights are not possible to be executed.
- A conflict between obligations and rights. This conflict occurs when obligations can not be fulfilled in their deadlines without violating rights.

6.5.1 Conflict in exemplarity of feasibility of obligations

An obligation is exemplary feasible in a situation σ if it is possible to execute it within its deadline without causing any violation. To formalize this, we define what we call *exemplary situations*. A situation is exemplary if it does not contain violation of any

obligation. Formally the exemplary situations can be constructed by recursion as follows:

$$Exemplary(\sigma_0) \wedge$$
$$Exemplary(do(a,\sigma)) \leftrightarrow Exemplary(\sigma) \wedge \neg(\exists\alpha, d)[Ob(\alpha < d, \sigma) \wedge \gamma_d^+(a,\sigma)]$$

In the above formula, if σ is an exemplary situation, $do(a,\sigma)$ will be also an exemplary situation if the action a does not activate the deadline of current active obligation ($\gamma_d^+(a,\sigma)$). By construction, in situation $do(a,\sigma)$, there is no violation of obligations because it is the result of an action that does not generate a new violations and there is no old ones because σ is itself an exemplary situation. We can now give the formal definition of feasible obligation by introducing fluent $E\text{-}Feasible(\alpha < d, \sigma)$:

$$E\text{-}Feasible(\alpha < d,\sigma) \leftrightarrow (\exists\sigma').\sigma' > \sigma \wedge Executable(\sigma') \wedge$$
$$Exemplary(\sigma') \wedge Fulfil(\alpha < d, \sigma')$$

The active obligation in σ is fulfilled in σ' which it is an exemplary situation. The research of σ' can be blocked due to the existence of old violations. Thus, we can define *Pseudo-Exemplary* situations as situations resulting from the execution of action which not causes a new violation of obligations. These situations may contain old violation of obligations.

$$Pseudo - Exemplary(do(a,\sigma)) \leftrightarrow \neg(\exists\alpha, d)[Ob(\alpha < d, \sigma) \wedge \gamma_d^+(a,\sigma)]$$

Using Pseudo-Exemplary situations, we can define an obligation which is feasible without causing new violations, $PE\text{-}Feasible(\alpha < d, \sigma)$.

$$PE\text{-}Feasible(\alpha < d,\sigma) \leftrightarrow (\exists\sigma').\sigma' > \sigma \wedge Executable(\sigma') \wedge$$
$$(\forall\sigma'', \sigma < \sigma'' \leq \sigma')Pseudo\text{-}Exemplary(\sigma'') \wedge Fulfil(\alpha < d, \sigma')$$

In the formula above, between σ and σ', there is no violation of new obligations. If an obligation is not feasible exemplarily (resp.pseudo exemplarily) in a situation, we shall consider that there is a conflict in the exemplarity (resp. pseudo exemplarity) of feasibility of an obligation in the policy in this situation.
It may happen that every active obligation is exemplarily feasible in a situation σ, but it is still not possible to do all of them without violating the associated deadlines. For this purpose, we define the fluent predicate *GE-Feasible*(σ), meaning a situation σ is globally exemplary feasible, as follows:

$$GE\text{-}Feasible(\sigma) \leftrightarrow \exists\sigma', \sigma' > \sigma \wedge Executable(\sigma') \wedge Exemplary(\sigma') \wedge$$
$$(\forall\alpha, d).Ob(\alpha < d, \sigma) \rightarrow Fulfil(\alpha < d, \sigma')$$

In the situation σ', all the obligations that were active in the situation σ are fulfilled. And there are no violations of any obligation. If it requires just to not produce new violations between situation σ and σ', we can use Pseudo-Exemplary situations like what we do in the definition of $PE\text{-}Feasible$.
If a given situation is not globally exemplary feasible, then we shall say that there is a global conflict in the exemplarity of feasibility of obligations in the policy in this situation. There are several ways to solve this conflict:

- Changing the deadline of some obligations
- Creating an exemption which cancels some obligations
- Offering the possibility to delegate some obligations

6.5.2 A conflict in feasibility of rights

Let us consider a conditional right rule $R(\alpha|p)$ and the precondition axiom of action α: $Poss(\alpha) \leftrightarrow \phi_\alpha$.
A conflict in feasibility of right exists if the following condition holds:

$$Axioms \vdash (\exists\sigma).p(\sigma) \wedge \neg\phi_\alpha(\sigma)$$

For the resolution of this conflict, it is not always wise to rewrite the condition that activates the right (i.e., $R(\alpha|p \wedge \neg\phi_\alpha)$. Let us take the example of right to join a training (rule r).
It does not make sense that a subject joins a training which he is in process to follow it. Thus, we can have $\neg AttendingTraining(d, \sigma)$ as precondition of action JoinTraining. Furthermore, it is necessary to have a training for to join it (precondition: $Training(t, \sigma)$). We assume at this stage that these conditions are sufficient, thus the precondition axiom of action JoinTraining can be specified as below:

$$Poss(JoinTraining(d, t), \sigma) \leftrightarrow (\exists t')Training(t', \sigma) \wedge \neg AttendingTraining(d, \sigma)$$

In this case, we can rewrite the rule r to join a training to avoid a conflict like follows:

$$(r') : R(JoinTraining(d, t) \mid Doctor(d) \wedge (\exists t')Training(t', \sigma) \wedge \neg AttendingTraining(d, \sigma)).$$

Hence, the axiom 6.2 is rewritten as follows:

$$\begin{aligned}
&Poss(a, s) \rightarrow \\
&Right(JoinTraining(d, t), do(a, \sigma)) \leftrightarrow Doctor(d) \wedge \\
&(\exists t')a = StartTraining(t') \vee \\
&(\exists t')Training(t', \sigma) \wedge (\exists t'')a = LeaveTraining(d, t'') \vee \\
&Right(JoinTraining(d, t), \sigma) \wedge \\
&\neg(\exists t')a = (EndTraining(t') \vee JoinTraining(d, t'))
\end{aligned} \tag{6.5}$$

Furthermore, in the policy of our example, we specified a constraint that restricts the circumstances under which it is possible to write documents for patients. Thus the constraint $C3$ prevents a doctor from writing a document of a patient who is assigned to him when he is following a training. To meet this requirement, the precondition axiom of action JoinTraining should be as follows:

$$\begin{aligned}
&Poss(JoinTraining(d, t), \sigma) \leftrightarrow Training(\sigma) \wedge \\
&\neg AttendingTraining(d, \sigma) \wedge \neg(\exists type, p, t, t')WritingDoc(d, type, p, t, t', \sigma)
\end{aligned} \tag{6.6}$$

This axiom specifies that if a doctor is writing a document, he can not execute the action JoinTraining to satisfy the constraint $C3$. In this case, rewriting the rule r' as follows:

$$\begin{aligned}
&R(JoinTraining(d, t) \mid Doctor(d) \wedge (\exists t')Training(t', \sigma) \wedge \\
&\neg AttendingTraining(d, \sigma) \wedge \neg(\exists type, p, t, t')WritingDoc(d, type, p, t, t', \sigma)).
\end{aligned}$$

changes completely the rule. This is not always a good solution. The rules of rights can be established by another organization different from the one which established a policy. And perhaps there is another organization which monitors compliance with its rights and applies sanctions if the right is not ensured (For example, CNIL[4]). In this case, it is not wise to change the context of activation of right. As in the case of voting, the right of an employee to vote can not be restricted by the company where he works with a condition that the company will applied to avoid a conflict in its policy.
Thus, among the preconditions of actions, we must distinguish those which must be verified otherwise the action cannot, logically, be executed. These conditions must be added to the conditions of activation of the corresponding rights. Concerning the other

[4]An independent French administrative authority. It is responsible for ensuring that information technology is at the service of citizens and it does not affect human identity, nor the rights or privacy, or individual and public freedoms. It carries out its tasks in accordance with Law No. 78-17 of 6 January 1978 amended August 6, 2004

conditions, to resolve the conflict, we can assign a prioritization between constraints and rights.

Note that in this work, we are defining a persistent right, which must be ensured at every moment (ex: read document). It is easy to extend this work for expressing right with deadlines like what we do with obligations with deadlines. We mean by a right with deadline, a right to do an action before some condition holds. Unlike persistent right, it is not necessary to ensure this right every time. It is sufficient to ensure it before that the condition holds so the right can be exercised. For example, it is sufficient to ensure the right of voting before the closing time. In this case, the detection of conflict is done using a planning task like detection of conflict concerning obligations with deadlines.

Furthermore, there may be situations where there are several active rights. Each one can be ensured separately. But it is not possible to ensure them simultaneously. We call this conflict a global conflict in the feasibility of rights. It is detected when the following condition occurs:

$$(\exists \alpha, \alpha').Right(\alpha, \sigma) \wedge Right(\alpha', \sigma) \wedge \neg Poss(\alpha', do(\alpha, \sigma))$$

Let us consider the same example like before, but suppose that there is possibility of having several types of trainings ($Training(Title, \sigma)$: meaning there is a training titled Title). Logically, it is not possible for a subject to be simultaneously physically in two different training if they are given in two different spaces. In this case, we can have the following precondition axiom for action joining a training:

$$Poss(JoinTraining(d, t), \sigma) \leftrightarrow (\exists Title).Training(Title, \sigma) \wedge$$
$$\neg(\exists Title')AttendingTraining(d, Title', \sigma)$$

There is a global conflict in the feasibility of rights in a situation σ if the following condition holds:

$$Training(Title, \sigma) \wedge Training(Title', \sigma) \wedge \neg(Title = Title')$$

Thus to solve a conflict, the rule r should be rewritten as follows:

$$(r) : R(JoinTraining(d, t) \mid Doctor(d) \wedge (\exists Title)Training(Title, \sigma) \wedge$$
$$\neg(\exists Title')AttendingTraining(d, Title', \sigma)).$$

It is clear that rewriting rule of right r as below restricts the basic rule. A doctor in the new rule has a right to attend just one training in the same time. But it is not possible to do otherwise. Because it is a logical conflict.

Note here another difference between permissions and rights. In fact, we can do the same thing for permissions as we did for rights, i.e., ensure that every time an action is permitted, it is possible to execute it to ensure for example the property of availability.

$$Availableness(\sigma) \leftrightarrow [(\forall\alpha)Perm(\alpha,\sigma) \rightarrow Poss(\alpha,\sigma)]$$

However in presence of a conflict, there are more flexibility to rewrite the rule of permissions than rewriting the rule of rights because as we mentioned before, rights are often associated with penalties.

6.5.3 A conflict between obligations and rights

A conflict between obligation and right occurs when it is not possible to do an obligation within its deadline without violating any right. To characterize this, we first introduce the notion of *preserved situation*. A preserved situation is the result of execution of an action that not causes any violation of rights. The preserved situations are constructed recursively like follows:

$$Preserved(\sigma_0) \wedge$$
$$Preserved(do(a,\sigma)) \leftrightarrow Preserved(\sigma) \wedge \neg(\exists\alpha)[Right(\alpha,\sigma) \wedge \gamma^{-}_{\phi_\alpha}(a,\sigma)]$$

The formula above specifies that the action a does not deactivate the preconditions of any right to do an action α.
We can define a pseudo preserved situations as situations which are not the result of action that causes a new violation of rights. This means that it is possible to have violation of right in these situations which were inherited from precedent situations.

$$Pseudo\text{-}Preserved(do(a,\sigma)) \leftrightarrow \neg(\exists\alpha)[Right(\alpha,\sigma) \wedge \gamma^{-}_{\phi_\alpha}(a,\sigma)]$$

Thus, we define the fluent $P\text{-}Enforceable(\alpha < d,\sigma)$ meaning that the obligation to do α is enforceable before that the deadline d holds while preserving rights.

$$P\text{-}Enforceable(\alpha < d,\sigma) \leftrightarrow (\exists\sigma').\sigma' > \sigma \wedge Preserved(\sigma') \wedge$$
$$Fulfil(\alpha < d,\sigma')$$

Between σ, where the obligation is active, and σ', where the obligation is fulfilled, there is no violation of right. This is done through the recursive construction of preserved situations. When an obligation is not enforceable while ensuring rights in a given situation, we say that there is a conflict between obligations and rights in the policy.

It is possible that each active obligation in a given situation is enforceable with preserving rights. However fulfilling all these obligations together necessarily leads to

a violation of a right. To characterize this, we define the fluent $GP\text{-}Enforceable(\sigma)$, meaning a situation σ is globally enforceable while preserving rights.

$$GP\text{-}Enforceable(\sigma) \leftrightarrow \exists \sigma', \sigma' > \sigma \wedge (\forall \alpha, d)$$
$$Ob(\alpha < d, \sigma) \rightarrow Fulfil(\alpha < d, \sigma') \wedge Preserved(\sigma')$$

In the formula above, all active obligations in σ are fulfilled in σ'. The fact that σ' is a preserved situation ensures that there is no violation of right between σ and σ'. If a situation is not globally enforceable while preserving rights, we shall say that there is a global conflict in the policy between obligations and rights in this situation. To resolve this conflict, we can adopt the same strategies that we propose to resolve a conflict between obligations. However, in addition, we can negotiate a waiver of some rights against compensation. This is a common solution in real life. For example, in a company, employees have a right to get holidays. The employer may negotiate with the employee by asking her to renounce to this right to meet the delivery deadline of a given project in exchange of a monetary compensation.

Example 9 The sequence of actions represented in figure 6.1 is a conflicting situation. More accurately, there is a global conflict between right and obligations in this situation. In fact, the only way to fulfill the active obligations in σ_c before their deadlines, is to fulfill them in a moment when the training is given. However writing documents by doctors when they are attending a training is not possible. Thus it is necessary that Jean does not join a training in some moments, which violates his right to join a current training whenever he wants.

In the figure 6.2, we can see an example of a legal and not preserved plan leading to fulfill all obligation in σ_c. In this plan, the right of Jean to join training is disabled when Jean joins the training (succession state axiom 6.5). However to fulfill the obligations before their deadlines, it is necessarily to execute the action LeaveTraining before starting writing documents (action precondition axiom 6.6), but the execution of a leaving action activates another time the right of Jean to join the training (succession state axiom 6.5). Thus starting writing document after the execution of LeaveTraining violates the right of Jean to join the training.

6.6 Related work and discussion

Normally, when someone refers to an action as a permitted one, this means implicitly that there is a choice between doing the action or not [Sartor 2005]. Indeed, when

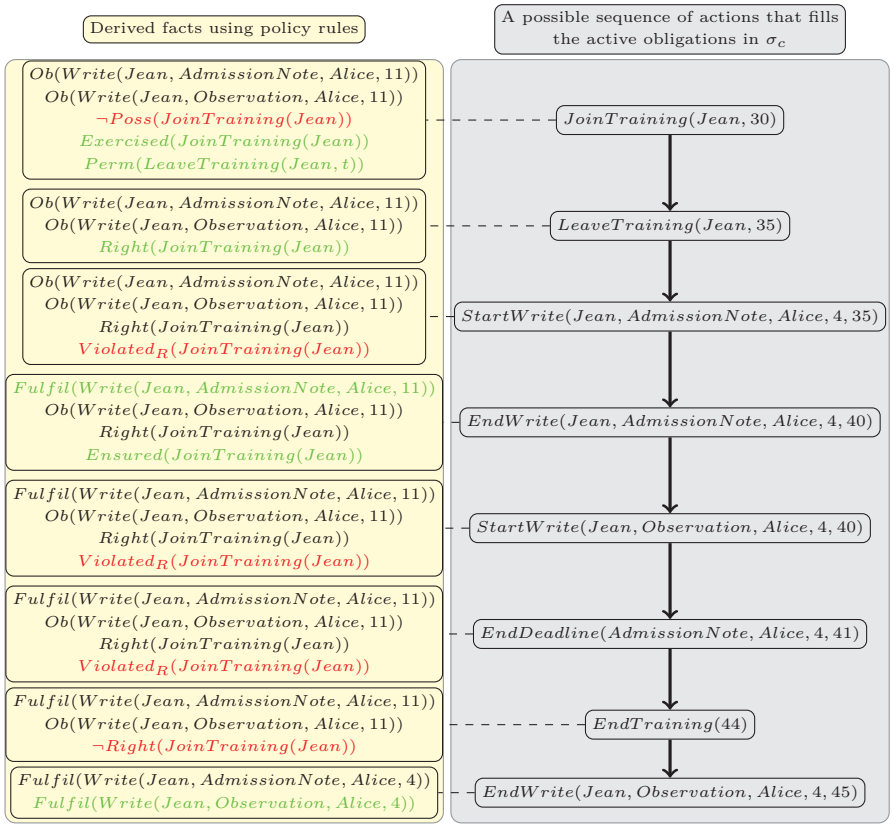

Figure 6.2: Example of a legal and not preserved plan: violation of right to join training in many situations

an action is permitted, it is sufficient to provide some opportunities that allow to execute this action in the presence of general prohibitions. For instance, a user who has permission to access the on line document may be prevented from accessing it because for example it is also permitted in the policy to remove this document for some reasons or stop the host server for maintenance. This leads to the concept of preserving permissions [Sartor 2005] using *directed obligations* which means actions that individuals must perform to ensure an interest of someone else. Then from the directed obligations, Sartor in [Sartor 2005] defines obligation right. When a person J has an obligation toward a person K to ensure an interest of K, then it said that K has an obligation right toward J. Our conception of right is different from Sartor. To show this, consider an example concerning right of data rectification [5]: *A user has right to send request to correct informations concerning him.* We admit that a necessary condition to make a request to correct information is to have an available email address of the manager holding the information. Thus, the right to send request is ensured in the situations when the web-master has created the email address of the

[5]http://www.cnil.fr/vos-droits/vos-droits/le-droit-de-rectification/

manager holding the information and there is no deleting action that has been applied on this email. We can then consider that the obligation of the web-master to create an email address of the manager of data modification ensured an interest of user to make a request for data rectification. Then, this corresponds to the obligation right of the user toward the web-master to create an email address of data manager. With this approach, it is not possible to explicitly express the right of the user to make a request to change her data. This is because it is her right to perform the action by herself and not someone else. On the other side, if there is an obligation which requires for the manager to respond to a request for modification of data transmitted by a user, in this case it is an obligation right of the user toward manager to have an answer which is certainly different from the right to issue the query.
To preserve rights, it is possible to derive prohibitions and obligations which are not directly specified in the policy. Let's see the example concerning the right of data rectification. And consider the following precondition axiom of action for sending modification request:

$$Poss(SendRectificationRequest(EmailAddress, req), \sigma) \leftrightarrow$$
$$ContactAddresses(DataManager, EmailAddress, \sigma)$$

However the information concerning the email address of the manager exists on website if if it was created at a given time and has not been removed.

$$Poss(a, s) \rightarrow$$
$$ContactAddress(DataManager, EmailAddress, do(a, \sigma)) \leftrightarrow$$
$$a = CreateContactAddress(DataManager, EmailAddress) \vee$$
$$ContactAddresses(DataManager, EmailAdress, \sigma) \wedge$$
$$\neg a = DelateContactAddresse(DataManager, EmailAddress)$$

From the succession state axiom of fluent ContactAddress(DataManager, emailAddress), we can see that if we add an obligation rule to create an email of the manager of data rectification and a rule to prohibit its deletion, then situations where the right rule is preserved are situations where the obligation to create an address email is fulfilled and the deletion prohibition rule is not violated.

6.7 Conclusion and contributions

In this chapter, we extend our formal language based on deontic logic of actions to specify rights rules. Throughout the chapter we give examples to approve the usefulness of

the distinction between rights and permissions. Then, we use situation calculus to give a semantics to our specification which enables this distinction. The use of right rules leads to a new conflict which it is the conflict between the rights and obligations with deadlines that occurs when it is impossible to fulfill the obligations in their deadlines without violating an active right. We show how using the planning task, we can prove the existence of such conflict.
The introduction of right rules in security policy could be used for any purpose other than the one we have proposed in this chapter. In fact they can be used to model the property of availability as illustrated in the example concerning on line documents which we mentioned previously. In the next chapter, we will talk about another interest to use the rule of right in a security policy.

CHAPTER 7

Conflict resolution based on delegation and renunciation of right

7.1 Introduction

In the previous chapters, we saw that the use of right and obligation rules in security policies may cause a conflictual situations. In this chapter, we propose a possible solution to resolve a conflict between obligations and permissions based on delegation. And offer the possibility to waive right in order to resolve a conflict between obligations and rights.

However, a conflict does not come necessarily from security policies. Sometimes a conflict could be avoided if users had adopted a different behavior within an interval of time before that conflict occurs. For example, suppose that there are two users Bob and Alice, each of them has an active obligation with deadline to write different information in the same document. Suppose also that the write access on this document can not be done at the same time by two different users. An example of conflict can hold in the following situation: Alice starts to write information to fulfill her own obligation in the document; Bob can not fulfill his obligation, he must first wait for Alice to finish her access to the document. However after that Alice fulfills her obligation, it is too late for Bob to fulfill his obligation within its deadline. The problem is that it is possible that the obligation concerning Bob was active before this conflict occurs. And maybe it was possible that if Bob had adopted another trace of execution of actions between the activation of his own obligation and the activation of Alice's one, the conflict could have been avoided. But what are the bases on which we can challenge the actions executed by Bob in this time interval ? For example, can we have the same judgment if Bob was attending an online training in this time of interval and if Bob was taking a coffee break. It certainly depends on what is specified in the policy. If both actions

are permitted, and assuming that obligations have higher priority than permissions, we can say that Bob should have begun to fulfill his obligation instead of taking a coffee break or attending a training. The purpose in this circumstances, is to consider that the conflict is the responsibility of Bob. Contrariwise, assume that in a policy, attending training is considered as a right. This implicitly leads to a commitment by the policy of preserving the execution of this action (the principle of violation of right). In this case, Bob could not be held responsible for the conflict. This is where modeling of right in security policies becomes more challenging and interesting.

Indeed, we believe that the actions that were executed because there is a right to do them should not be revoked. It is a way to be fair with a user to not revoke a right which he exercised. In other words, the rights and obligations are considered at the same level. This is justifiable because a right, which is not preserved, as an obligation, which is not fulfilled in its deadline, raises violations and can lead to penalties. Of course, it is interesting to consider prioritization between obligations and rights. However, rights as obligations have always priority over permissions.

The characterization of conflicts responsibilities is a complex task. Indeed, responsibility may be shared by multiple stakeholders. This may happen if the conflict can be avoided if each stakeholder would have adopted another behavior in a given interval of time. Responsibility can also be multiple. This occurs when the change in the behavior of multiple users independently of each other would solve the conflict. For example, if obligations concerning Bob and Alice are activated before the conflict happened and both of them were in coffee break. Being fair with Bob implies that Alice should also be considered responsible for the current conflict.

The remainder of this chapter is organized as follows. In section 7.2, we formalize a conflict which does not come from the security policy and determine responsibilities. In section 7.3, we propose to redistribute obligations using delegation to resolve a conflict between obligations and permissions. In this section, we also introduce the principle of right renunciation in order to resolve a conflict between rights and obligations. Finally section 7.4 concludes this chapter.

7.2 Detected conflict and responsibility

Let us assume that a conflict between obligations with deadlines is detected in a given situation σ. It is possible that some of them have been active in situations prior to σ and remain active because they have not been executed. Thus, it is interesting to

analyze the previously executed actions in order to show if the conflict could have been avoided or not and determine responsibilities before trying to resolve a conflict.

7.2.1 Situations where the conflict could have been avoided

Our approach consists firstly to identify the previous situation where the first active obligation in σ was activated. This situation is called the *earliest situation* in the rest of this chapter and denoted σ_e. Then from σ_e, we search a plan to fulfill active obligations that cause the conflict by reproducing what has been done with these obligations in σ while preserving rights which are triggered. If the plan exists, then there is no conflict in the policy as the conflict could have been avoided.

Let us start by giving the definition of the earliest situation.

Definition 4. *Earliest situation.*
The earliest situation denoted σ_e related to a conflictual situation σ_c is a prior situation to σ_c where the older obligation among all active obligations in σ_c was activated.

$$
\begin{aligned}
&Earliest(\sigma_e, \sigma_c) \leftrightarrow \sigma_e \leq \sigma_c \wedge \\
&(\exists \alpha_e, d_e)[Ob(\alpha_e < d_e, \sigma_c) \wedge Ob(\alpha_e < d_e, \sigma_e) \wedge (\forall \sigma, \sigma_e \leq \sigma \leq \sigma_c) Ob(\alpha_e < d_e, \sigma)] \wedge \\
&\forall (\alpha', d', \sigma') Ob(\alpha' < d', \sigma') \wedge Ob(\alpha' < d', \sigma_c) \rightarrow \sigma_e \leq \sigma'
\end{aligned}
$$

Note that by definition, the active obligation which characterizes the earliest situation should necessarily be active throughout the path between σ_e and the target situation σ_c.

Proposition 1. *The earliest situation σ_e corresponding to a situation σ_c is unique.*

Proof. If there are two earliest situations σ_e and σ'_e corresponding to the same situation σ_c, then $\sigma_e \leq \sigma'_e$ and $\sigma'_e \leq \sigma_e$ which leads to $\sigma_e = \sigma'_e$ according to unique name axiom 2.1 □

Between a conflictual situation σ_c and its corresponding earliest situation σ_e, we need to reproduce some of the facts when they actually happened. For this, we introduce the following definition:

Definition 5. *Earliest situation vs fact.*
The earliest situation related to fact F is the earliest situation when the fluent F becomes true.

$$
\begin{aligned}
&Earliest_F(\sigma) \leftrightarrow \\
&(\exists \sigma', a) \sigma = do(a, \sigma') \wedge \gamma^+_F(a, \sigma') \wedge \neg F(\sigma')
\end{aligned}
$$

We can now characterize the set of agents whose behavior might have prevented that a conflict occurs.

Definition 6. *The set of agents which could avoid conflict.*
Let σ_c be a situation which is not strongly enforceable and σ_e its corresponding earliest situation. The set of agents which could avoid the conflict in σ_c consists of agents which while we rearrange their activities between σ_e and σ_c, we can reach a legal and preserved situation where the active obligations in σ_c are fulfilled. More formally:

$$AvoidingConflict(r, \sigma_c) \overset{def}{\leftrightarrow} \neg SG\text{-}Enforceable(\sigma_c) \wedge$$
$$(\exists \sigma_g, \sigma_e) Earliest(\sigma_e, \sigma_c) \wedge \sigma_g > \sigma_e \wedge Legal(\sigma_g) \wedge Preserved(\sigma_g) \wedge$$
$$(\forall \sigma, \alpha, s').\sigma_e \leq \sigma \leq \sigma_c \wedge Right(\alpha(s'), \sigma) \rightarrow \tag{7.1}$$
$$\quad (\exists \sigma').\sigma_e \leq \sigma' \leq \sigma_g \wedge Right(\alpha(s'), \sigma') \wedge start(\sigma) = start(\sigma')$$
$$(\forall \sigma, \alpha, d, s').(\sigma_e \leq \sigma \leq \sigma_c) \wedge Earliest_{Ob(\alpha(s')<d)}(\sigma) \rightarrow \tag{7.2}$$
$$\quad (\exists \sigma').\sigma_e \leq \sigma' \leq \sigma_g \wedge Ob(\alpha(s') < d, \sigma') \wedge start(\sigma) = start(\sigma') \wedge$$
$$(\forall \sigma, \alpha, d, s').s' \notin r \wedge \sigma_e \leq \sigma \leq \sigma_c \wedge Earliest_{Fulfil(\alpha(s')<d)}(\sigma) \rightarrow \tag{7.3}$$
$$\quad (\exists \sigma')\sigma_e \leq \sigma' \leq \sigma_g \wedge Fulfil(\alpha(s') < d, \sigma') \wedge start(\sigma) = start(\sigma') \wedge$$
$$(\forall \alpha, d, s).s \in r \wedge Fulfil(\alpha(s) < d, \sigma_c) \rightarrow Fulfil(\alpha(s) < d, \sigma_g) \wedge \tag{7.4}$$
$$(\forall \alpha, d, s').Ob(\alpha(s') < d, \sigma_c) \rightarrow Fulfil(\alpha(s') < d, \sigma_g) \tag{7.5}$$

In the formula above, σ_g is the searched solution. The plan allowing to reach this situation verifies the following:

- *All the rights which were active between σ_e and σ_c are activated at the same time by this plan (7.1)*
- *All the obligations which were activated before σ_c are active at the same time by this plan (7.2)*
- *The previous fulfilled obligations between σ_e and σ_c by agents not belonging to the set r are fulfilled at the same time by this plan (7.3)*
- *The previous fulfilled obligations between σ_e and σ_c by agents in the set r are fulfilled by this plan (7.4)*
- *The active obligations that cause the conflict are fulfilled in σ_g (7.5)*

Example 10 Consider again the situation σ_c represented in figure 6.1. The conflict in σ_c could have been avoided. As shown in figure 7.1, there is a legal, executed and preserved plan that fulfills all the active obligations in σ_c in their

deadlines starting from the earliest situation σ_e corresponding to σ_c, where σ_e is given as follows:

$$\sigma_e = do([Assign(Alice, Jean, 4), StartCoffeeBreak(10), StartTakeCoffeeBreak(Jean, 10), PatientAdmission(Alice, 11)], \sigma_0)$$

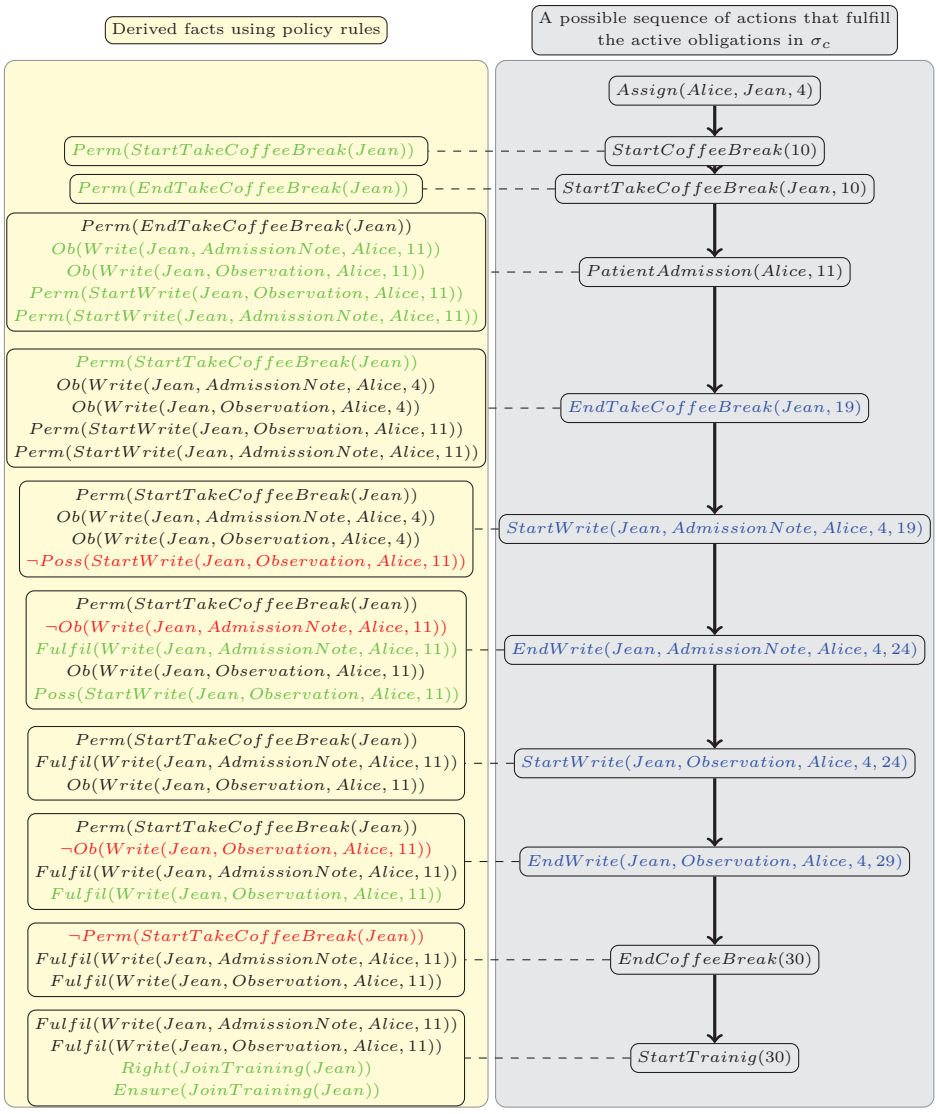

Figure 7.1: Example of legal, executed and preserved plans which prove that the conflict in σ_c could have been avoided

The members in the set r verifying AvoidingConflict(r, σ_c), are not necessarily responsible for the conflict in σ_c. In the next section, we distinguish among them who is actually responsible. We then prove that our definition of responsibility ensures the fairness property.

7.2.2 Shared and multiple responsibilities

In this section, we define the group of agents who are considered responsible for a current conflict. Then, we show that this definition enables the distinction between the notion of multiple responsibility and shared responsibility. Furthermore, we prove that our semantic of responsibility ensures the fairness property.

Definition 7. *Responsibility*
The agents responsible of a conflict in a given situation σ_c are those that constitute the smallest set which satisfies AvoidingConflict in σ_c.

$$Responsible(r, \sigma_c) \leftrightarrow AvoidingConflict(r, \sigma_c) \wedge$$
$$(\forall r')[(AvoidingConflict(r', \sigma_c) \wedge r' \subseteq r) \rightarrow r = r']$$

We assume that $AvoidingConflict(\emptyset, \sigma_c) = False$. In this case we say that in σ_c there is an *inevitable conflict* which means that the conflict comes from the security policy. When r is different from singleton, the formula above expresses a *shared responsibility* between the elements of r. Intuitively, we mean by *shared responsibility* that there is a nested composition of actions that the elements of r should have executed to avoid the conflict. Note that there may be different sets verifying the responsibility formula in which case we say that there is *multiple responsibilty*. Let us take an example.

> **Example 11** Suppose that in a situation σ_c, there are active obligations concerning agents s_1, s_2, s_3 and s_4 which cause conflict in σ_c and from the earliest situation σ_e corresponding to σ_c there are three different sets of agents which lead to avoid conflict, $r_1 = \{s_1, s_2\}$, $r_2 = \{s_1\}$ and $r_4 = \{s_3, s_4\}$. Here there is no shared responsibility between s_1 and s_2 as it exists a plan which allows fulfilling the obligations of all users when the executed actions of s_2 are kept unchanged and the activities of s_1 are rearranged (i.e., $r_1 \subset r_2$). But there is a shared responsibility between s_3 and s_4 as if we keep the executed actions by s_1 unchanged and rearrange the executed actions by s_3 and s_4 together, the conflict could have been avoided. Then to be fair back to s_1, the agents s_3 and s_4 are also responsible for this conflict.

In the following we assess formally the fairness of our model of responsibility.

Proposition 2. *Fairness*
Our overview of fairness is based on the following principle: for any happened conflict,

if it is possible for other agents to change their behavior to avoid the conflict, then they should be also considered responsible.

$$(\exists s).s \in r \wedge Responsible(r, \sigma_c) \wedge (\exists r').s \notin r' \wedge AvoidingConflict(r', \sigma_c) \rightarrow$$
$$Responsible(r', \sigma_c) \vee (\exists r'').r'' \subset r' \wedge Responsible(r'', \sigma_c)$$

Furthermore, the formalization of responsibility ensures that there is no agent that can share the responsibility of a conflict with other agents when it is proved that this conflict could be avoided if just the other agents had changed their behavior. In the example 7.1, s_2 does not share responsibility with s_1 because there is a plan which proves that only the change in the behavior of s_1 can lead to avoid the conflict.

$$(\forall s).(s \in r \wedge Responsible(r, \sigma_c)) \rightarrow \neg(\exists r')(r' = r \backslash \{s\} \wedge AvoidingConflict(r', \sigma_c))$$

Where $r\backslash\{s\}$ denotes the set consisting of all elements of r except the element s.

In this section, we distinguished between the conflictual situations, a conflict in a policy and a conflict caused by users. This distinction allows the determination of responsibilities before trying to resolve the conflict by redistributing obligations. In the following, we show how the reallocation of obligations can be done using delegation.

7.3 Possible conflict resolution

When a conflict occurs, one possible solution is to change the deadlines associated with obligations. However sometimes these deadlines can not be modified. For example the deadlines concerning the completion of medical records are calculated based on studies that have been made to estimate the most suitable deadlines to give the most effective service to patients. Thus, the delegation may be in some cases the only way to avoid that the violations occurs. For this purpose, we extend our model to support the delegation of obligations.

7.3.1 Obligation delegation

We extend our model by the following actions and fluents.

- $Propound(s, s', \alpha < d)$, meaning a subject s suggests the obligation to do α before deadline d to a subject s'.
- $Decline(s, s', \alpha < d)$, meaning a subject s rejects the obligation to do α before deadline d which was suggested to him by s'.

- $Accept(s, s', \alpha < d)$, meaning a subject s accepts the obligation to do α before deadline d which was suggested to him by s'.

The delegation of obligations requires two steps. The first step consists to suggest the obligation to a suitable subject. If the suggested obligation is accepted, then this obligation is considered delegated.

A suggested obligation for delegation is denoted $Proposal(\alpha < d, s, s', \sigma)$; meaning the obligation to do the action α before the deadline d is suggested by s to s' in the situation σ.

$$
\begin{aligned}
&Poss(a, \sigma) \rightarrow \\
&Proposal(\alpha < d, s, s', do(a, \sigma)) \leftrightarrow \\
&a = Propound(s', s, \alpha < d) \vee \\
&[Proposal(\alpha < d, s, s', \sigma) \wedge \neg(\exists s'')(a = Accept(s'', s, \alpha < d) \wedge \\
&\neg(a = Decline(s', s, \alpha < d)) \wedge \neg\gamma_d^+(a, \sigma) \wedge \neg\gamma^-_{\psi_{O_{\alpha,d}}}(a, \sigma) \wedge \neg(a = \alpha)]
\end{aligned}
\tag{7.6}
$$

This axiom specifies that an obligation is no longer proposed for delegation when it is accepted, declined or deactivated. Furthermore, an obligation can be proposed for delegation to multi users. Although, in this axiom it is sufficient that one of them accepts the proposed obligation, so it becomes no longer suggested for delegation to the others. However, it may be useful to delegate an obligation to multiple users for managing group obligations [Rakaiby et al. 2009] and shared obligations [Cole et al. 2001, Ben Ghorbel-Talbi et al. 2011]. In this case, the line 7.6 can be modified by replacing s'' by s'.

Certainly, a subject can propound an obligation for delegation if it is permitted by the policy. And it is only an active obligation which can be propound for delegation.

$$
\begin{aligned}
&Poss(Propound(s, s', \alpha < d), \sigma) \leftrightarrow Ob(\alpha(s) < d, \sigma) \wedge \\
&Perm(Propound(s, s', \alpha < d), \sigma) \wedge Perm(\alpha(s'), \sigma)
\end{aligned}
$$

An obligation is delegated when it was suggested and accepted. A delegated obligation is defined using the fluent $Delegated(\alpha < d, s, s', \sigma)$, meaning the obligation to do the action α before the deadline d is delegated to s' by s.

$$
\begin{aligned}
&Poss(a, \sigma) \rightarrow \\
&Delegated(\alpha < d, s, s', do(a, \sigma)) \leftrightarrow \\
&Proposal(\alpha < d, s, s', \sigma) \wedge a = Accept(s', \alpha < d) \vee \\
&Delegated(\alpha < d, s, s', \sigma) \wedge \neg\gamma_d^+(a, \sigma) \wedge \neg\gamma^-_{\psi_{O_{\alpha,d}}}(a, \sigma) \wedge \neg(a = \alpha)
\end{aligned}
\tag{7.7}
$$

The line 7.7 specifies that the delegated obligation to s' to do the action α before the deadline d (i.e., Delegated($\alpha <$ d, s, s', σ)) turns to false when the obligation is deactivated, when the corresponding deadline holds, or when it is fulfilled.

When a subject s delegates its own obligation to a subject s', this obligation becomes effective for s'. Thus we extend the axiom concerning active obligations 3.8 to support the delegation of obligations as follows:

$$
\begin{aligned}
&Poss(a,\sigma) \rightarrow \\
&Ob(\alpha(s) < d, do(a,\sigma)) \leftrightarrow [\gamma^{+}_{\psi_{O_{\alpha,d}}}(a,\sigma) \vee \\
&(\exists s')Proposal(\alpha < d, s', s, \sigma) \wedge a = Accept(s, s', \alpha < d) \vee \\
&[Ob(\alpha(s) < d, \sigma) \wedge \neg(a = \alpha) \wedge \neg\gamma^{+}_{d}(a,\sigma) \wedge \neg\gamma^{-}_{\psi_{O_{\alpha,d}}}(a,\sigma)]
\end{aligned}
$$

Whenever an obligation is delegated to a subject s, it is active for him. A delegated obligation could be delegated again.

Our model enables the propagation of obligation delegation [Ben Ghorbel-Talbi et al. 2011]. This happens when a delegated obligation is propounded for delegation another time and accepted. Nevertheless, when an obligation is disabled, the entire chain of delegation is disabled as well.

7.3.2 Renunciation of its own rights

In this section, we propose to give the possibility to waive rights in order to solve conflicting situations. Therefore, we extend our model as follows:

- *Renunciate*(s, α): a subject s renounces to its right to do α.
- *Abandoned*(α, σ): The right to do α is abandoned in a situation σ.

Once a subject renounces to his right, this latter is deactivated. Therefore, the succession state axiom of active right 6.1, is redefined as follows:

$$
\begin{aligned}
&Poss(a,\sigma) \rightarrow \\
&\quad Right(\alpha(s), do(a,\sigma)) \leftrightarrow \gamma^{+}_{\psi_{R_\alpha}}(a,\sigma) \vee \\
&\quad (Right(\alpha,\sigma) \wedge \neg\gamma^{-}_{\psi_{R_\alpha}}(a,\sigma) \wedge \neg(a = Renunciate(s,\alpha)))
\end{aligned}
$$

When a subject renounces to his right, it is called an abandoned right, namely $Abandoned(\alpha, \sigma)$. When a right is disabled, it is no longer an abandoned right.

$$Poss(a, \sigma) \rightarrow$$
$$Abandoned(\alpha(s), do(a, \sigma)) \leftrightarrow Right(\alpha(s), \sigma) \wedge a = Renunciate(s, \alpha) \vee$$
$$Abandoned(\alpha, \sigma) \wedge \neg\gamma^{-}_{\psi_{R_\alpha}}(a, \sigma)$$

Certainly, a subject can renounce to his right if it is permitted by the policy. And it is only an active right which can be waived.

$$Poss(Renunciate(s, \alpha), \sigma) \leftrightarrow Right(\alpha(s), \sigma) \wedge$$
$$Perm(Renunciate(s, \alpha), \sigma)$$

7.3.3 Solvable conflict

A conflict between obligations in a situation σ_c could be solved legally if using delegation, it is possible to find a legal situation σ where the active obligations in σ_c are fulfilled.

$$L - Solvable(\sigma_c) \leftrightarrow \exists\sigma, \sigma > \sigma_c \wedge Legal(\sigma) \wedge (\forall\alpha, d, s)$$
$$Ob(\alpha(s) < d, \sigma_c) \rightarrow Fulfil(\alpha(s) < d, \sigma) \vee$$
$$(\exists s')Fulfil(\alpha(s') < d, \sigma) \wedge (\exists s'', \sigma')(\sigma_c < \sigma' < \sigma) \wedge Delegated(\alpha < d, s'', s', \sigma')$$

In the above formula, the active obligation in the situation σ_c concerning the subject s, is fulfilled in the situation σ. This obligation is fulfilled by s, or by another subject s' which was delegated to him by a subject s''.
Furthermore, the searched of a legal situation σ, ensures that all subjects are permitted to do the actions that lead to this situation.

Similarly, the conflict between the obligations and rights could be solved, if using delegation and the renunciation of right, it is possible to find a preserved situation σ where the active obligations in σ_c are fulfilled.

$$P - Solvable(\sigma_c) \leftrightarrow \exists\sigma, \sigma > \sigma_c \wedge Preserved(\sigma) \wedge (\forall\alpha, d, s)$$
$$Ob(\alpha(s) < d, \sigma_c) \rightarrow Fulfil(\alpha(s) < d, \sigma) \vee$$
$$(\exists s')Fulfil(\alpha(s') < d, \sigma) \wedge (\exists s'', \sigma')(\sigma_c < \sigma' < \sigma) \wedge Delegated(\alpha < d, s'', s', \sigma')$$

In the above formula, searching for a preserved situation σ, ensures that all rights are ensured or some of them are ensured and others are waived.
Conflict due to delegation

Conflict could be caused by delegation. Indeed, one user delegates his obligation to the other one, and when this latter accepts the delegation, this causes a conflict in the policy.

$$(\exists s, \sigma) GL\text{-}Enforceable(\sigma) \wedge \neg(GL\text{-}Enforceable(do(Accept(s, \alpha < d), \sigma))$$

Note that our mechanism of responsibility detection considers the user which accepts the obligation as responsible of the occurrence of this conflict. Indeed, if we remove the action accept (i.e., Accept(s, α < d)), the resulting situation is not conflictual (GL-Enforceable(σ)), this means that s could have avoided the conflict. Furthermore, as the action Accept is necessarily preceded by the action Propound, we can also derive that the user s and the user who executed the action Propound could avoid the conflict. However, in our definition of responsibility we conclude that the user s who accepts the obligation is the only one responsible of this conflict. In the case of delegation propagation, it is the last user who accepts the delegation before the occurrence of the conflict which is considered responsible for this conflict.

7.4 Conclusion and contributions

In this chapter, we show how we can formally distinguish a conflict that arises from the policy and the one that is the responsibility of one or multiple users. We show that our judgment on the responsibility of a conflict is *fair* whenever it was possible that a change in the behavior of a user would have led to avoid a conflict. In this case, this user is considered responsible for this conflict. We also show that no user is sharing a responsibility for a conflict with a user group if there exists an arrangement of actions where it was sufficient for the other members of the group to change their behavior to avoid the conflict.

In addition, we proposed a possible solution based on the delegation of obligation and the renunciation of right to resolve a conflict that comes from a policy. If there is no solution based on delegation, the only way would be to deactivate certain obligations.

CHAPTER 8

Conclusion and perspectives

In this thesis, we use deontic modalities to specify security policies including obligations with deadline. Then we use the temporal sequential situations calculus to derive concrete permissions and obligations. Furthermore, we show how the situation calculus allows us to detect if there is a policy conflict in a given situation using the planning task. Moreover, we have illustrated our approach by using a case study from the health care community. Specifically, we are interested in obligations with deadlines concerning completion of the patients' medical records. We show how we can use our language to express the obligations of this example. In addition, we present the implementation that we did, using the logic programming language based on Golog, in order to prove that a given situation is globally enforceable or not.

Furthermore, we proposed a formal language to express constraints in access control policy. In our language, there are two kinds of constraints which we call historical and ahistorical constraints. We show how this language is adequate to express well-known constraints in the literature. To enforce constraints, first we propose to rewrite the historical constraints into simple formulas. We then give a procedure based on the regression concept to enforce these constraints.

In this thesis, we also propose to express the rules of right, in addition to permissions and obligations rules. Throughout the thesis, we give examples to approve the usefulness of the distinction between rights and permissions. Then, we use the situation calculus to give semantics to our specification which enables this distinction. The use of rules of right leads to a new conflict that has not been addressed in the literature, it is the conflict between the rights and obligations with deadlines. As a first step, we use the schedule to prove the existence of the conflict. Then the detection mechanism is refined to distinguish conflict from security policies and conflict due to the nature of the activities carried out before the conflict occurs. This refinement is due to the distinction that we make between permissions and rights to the extent that mandatory

activities have priority over those which are permitted and at the same levels as the rights. This is justified by the fact that rights are also associated with violations as we have shown in the examples from CNIL. In addition, we can determine whether the user is responsible for the ongoing conflict or not. Then, we give a property that characterizes the fairness of the evaluation mechanism that allows us to judge this responsibility. Furthermore, we propose a possible solution to solve conflicts based on delegation of obligations and renunciation of rights.

8.1 Perspectives

It is clear that in our model there is a step that requires more simplification, this consists in the definition of succession state axioms for fluents which belong to domain application. Our proposal in this case is to specify axioms of positive (resp.negative) effects on fluents and derivation rules and then build automatically succession state axioms.

Concerning the constraint model, we intend to use the planning task to distribute user roles in order to accomplish the complete workflow task in compliance with access control policies. Planning will also allow us to detect if there is a conflict between specified constraints. We will, in the future work, give a formal specification of constraint consistency and provide means to prove it.

Our future work also includes the implementation of the detection mechanism of responsibilities. In this regard, we should improve our algorithm of planning to allow the search of several goals.

We will implement the solution which we propose to resolve conflicts using delegation and renunciation of right. We should provide a plan which gives a solution with a minimal renunciation of rights. On the other hand, the redistribution of obligations could be guided by certain specific parameters. In the following, we propose how to manage delegation based on the reputation of users and their workload.
The calculation of users reputation is based on the occurrence of violations. However, the workload of users corresponds to the total amount of obligations assigned to them. We define for each fluent F, a fluent $Counter(F, n, \sigma)$ meaning the occurrence number of fluent F in the situation σ is n. Similarly, $Counter_{\vec{X}}(F, n, \sigma)$ denotes the number of parameters terrifying F in the situation σ where $\vec{X} = (X_0, ..., X_p)$ is fixed among the parameters of F. For example, $Counter_{Jean}$(Violated(Write(Jean, type, p, t) $<$ Deadline(type, p, t)), is the number of violation that Jean did concerning the obligation of writing patient documents. However, Counter(Violated(Write(d, type, p, t)

< Deadline(type, p, t)) is the total number of violations concerning writing patient documents. Using the Counter fluent, we can consider several perspectives to solve conflictual situations based on delegation.

8.1.1 Delegation based on workload assigned to users

We propose to calculate the workload assigned to a user s using the number of active obligations: $Counter_s(Ob, n, \sigma)$, the number of violations: $Counter_s(Violated, n, \sigma)$ and the number of fulfilled obligations: $Counter_s(Fulfil, n, \sigma)$. A workload assigned to a user between two situations is the total number of obligations that he must fulfill between these situations.

$$
\begin{aligned}
&Workload_{\sigma,\sigma'}(s, n) \leftrightarrow \sigma < \sigma' \wedge \\
&Counter_s(Violated, v, \sigma) \wedge Counter_s(Violated, v', \sigma') \wedge \\
&Counter_s(Ob, a, \sigma') \wedge \\
&Counter_s(Fulfil, f, \sigma) \wedge Counter_s(Fulfil, f', \sigma') \wedge \\
&n = [(v' - v) + (f' - f) + a]
\end{aligned}
$$

This formula could be improved if we give for each obligation a weight expressing the degrees of complexity to fulfill it. Assuming that all obligations have the same degree of complexity, we can evaluate the equity between users concerning the task assigned to them.

$$(\forall s, s')Workload_{\sigma,\sigma'}(s, n) \wedge Workload_{\sigma,\sigma'}(s, m) \rightarrow n = m$$

The above formula may be restricted to users having the same role in the model of access control based on roles.

Thus, we can search to resolve a conflict in a situation σ_c, while trying to balance obligations between users.

$$
\begin{aligned}
&Solvable_w(\sigma_c) \leftrightarrow \exists\sigma, \sigma > \sigma_c \wedge Legal(\sigma) \wedge (\forall\alpha, d, s) \\
&Ob(\alpha(s) < d, \sigma_c) \rightarrow Fulfil(\alpha(s) < d, \sigma) \vee \\
&(\exists s', s'')Fulfil(\alpha(s') < d, \sigma) \wedge Delegated(\alpha < d, s'', s', \sigma) \wedge \\
&(\exists m)(\forall s''')Workload_{\sigma_c,\sigma}(s''', m)
\end{aligned}
$$

The formula above specified that all users between the conflictual situation σ_c and the situation σ where all active obligations in σ_c are fulfilled, have the same workload.

8.1.2 Delegation based on user reputation

We can use the number of violations to calculate the reputation levels of users. We consider two levels of reputations, *Bad* and *Efficient*.

Definition 8. *A user s is considered having the Bad reputation of n % in a situation σ if in σ the following formula is verified:*

$$Reputation(s, Bad, n, \sigma) \overset{def}{\leftrightarrow}$$
$$(\exists v) Counter_s(Violated, v, \sigma) \wedge (\exists f) Counter_s(Fulfil, f, \sigma) \wedge$$
$$n = v \div (f + v)$$

A user s is considered having the *Efficient* reputation of n % in a situation σ if in σ he has the bad reputation of $1-n$ %. Using the reputation of users in a given situation σ, we can calculate a reputation linked to an evaluation interval and a reputation linked to a specific obligation.

Reputation linked to an evaluation interval

The reputation linked to an interval is calculated by reference to what happened in this interval regardless of what happened before or after.

$$Reputation_{\sigma,\sigma'}(s, Bad, n) \leftrightarrow \sigma < \sigma' \wedge$$
$$Counter_s(Violated, v, \sigma) \wedge Counter_s(Violated, v', \sigma') \wedge$$
$$Counter_s(Fulfil, va, \sigma) \wedge Counter_s(Fulfil, va', \sigma') \wedge$$
$$n = (v' - v) \div [(va' - va) + (v' - v)]$$

Reputation linked to a specific obligation

A reputation in a situation σ, linked to a specific obligation $O(\alpha < d)$ and a user s, denoted $Reputation_{\alpha,d}(s, r, \sigma)$, can be calculated using $Counter_{O(\alpha(s)<d)}(Violated, n, \sigma)$, $Counter_{O(\alpha(s)<d)}(Fulfil, n, \sigma)$.

Consider a given threshold of reputation; $Threshold(m)$. We could delegate an obligation to a user deemed to be effective to do this obligation with percentage m.

$$Solvable_r(\sigma_c) \wedge Threshold(m) \leftrightarrow \exists \sigma, \sigma > \sigma_c \wedge Legal(\sigma) \wedge (\forall \alpha, d, s)$$
$$Ob(\alpha(s) < d, \sigma_c) \rightarrow Fulfil(\alpha(s) < d, \sigma) \vee$$
$$(\exists s', s'') Fulfil(\alpha(s') < d, \sigma) \wedge Delegated(\alpha < d, s'', s', \sigma) \wedge$$
$$Reputation_{\alpha<d}(s', Efficient, n) \wedge n \geq m)]$$

In the above formula, the obligation to do α is fulfilled by a user s'. This obligation was delegated to s' by s'' and s' has a reputation to be efficient in doing this obligation with percentage greater or equal to m.

Finally, it is important to implement a mechanism that allows to warn users of a possible conflict and suggests the most appropriate plan of actions to follow.

APPENDIX A

Actual norm derivation of the case study described in chapter 3

In this appendix, we calculate where the remainder of rules, specified in the case study described in chapter 3, are effective. For this, let us first recall the specification of these rules and the succession state axiom of fluents used on.

- The specification of rules:

  O3 : $O(EndDeadline(AdmissionNote, p, t, t') | Inpatient(p, t) \wedge t' = t + 30)$
  O4 : $O(EndDeadline(Observation, p, t, t') | Inpatient(p, t) \wedge t' = t + 40)$
  P2 : $P(EndtWrite(d, type, p, t, t_{ew}) \mid WritingDoc(d, type, p, t, t') \wedge t_{ew} \geq t' + 5)$

- Succession state axioms:

  – Succession state axiom of fluent Inpatient:
  Patient p is hospitalized if he was admitted to the hospital and did not leave.

  $$\begin{aligned} &Poss(a, \sigma) \rightarrow \\ &Inpatient(p, t, do(a, \sigma)) \leftrightarrow [a = PatientAdmission(p, t) \vee \\ &(Inpatient(p, t, \sigma) \wedge \neg(\exists t') a = Leave(p, t'))] \end{aligned} \tag{A.1}$$

  – Succession state axiom of fluent WritingDoc:
  A document is in writing process if the write began before and has not been

completed.

$$
\begin{aligned}
&Poss(a, \sigma) \rightarrow \\
&WritingDoc(d, type, p, t, t_s, do(a, \sigma)) \leftrightarrow \\
&a = StartWrite(d, type, p, t, t_s) \vee \\
&WritingDoc(d, type, p, t, t_s, \sigma) \wedge \neg(\exists t_e) a = EndWrite(d, type, p, t, t_e)
\end{aligned} \tag{A.2}
$$

The formula characterizing where obligation rules O3 and O4 are effective is calculating using the succession state axiom of fluent WritingDoc.

$$
\begin{aligned}
&Poss(a, \sigma) \rightarrow \\
&Perm(EndWrite(d, type, p, t, t_e), do(a, \sigma)) \leftrightarrow \\
&a = StartWrite(d, type, p, t, t_s) \wedge t_e \geq t_s + 5 \vee \\
&Perm(endWrite(d, type, p, t, t_e), \sigma) \wedge \neg a = EndWrite(d, type, p, t, t'_e)
\end{aligned}
$$

The situations where the system obligations O3 and O4 are active are calculated using the succession state axiom of fluent Inpatient.

$$
\begin{aligned}
&Poss(a, \sigma) \rightarrow \\
&Ob(EndDeadline(type, p, t, t', do(a, \sigma)) \leftrightarrow \\
&[(a = PatientAdmission(p, t) \wedge \\
&((type = AdmissionNote \wedge t' = t + 30) \vee type = Observation \wedge t' = t + 40)) \vee \\
&(Ob(EndDeadline(type, p, t, t'), \sigma) \wedge \\
&\neg(\exists t') a = EndDeadline(type, p, t, t') \wedge \neg(\exists t') a = Leave(p, t'))]
\end{aligned}
$$

APPENDIX B

Prolog file: ConflictDetection.pl

```
1
:- include(gologinterpreter).
3 :- discontiguous(time/2).

5 % GOLOG Procedures

7 proc(plan(N, L),
       ?(all(r, member(r, L)=> fulfil(r))): ?(reportStats)#
9      ?(N > 0):
       pi(a, ?(primitiveAction(a)): a):
11     ?(-badSituation):
       pi(n, ?(n is N-1): plan(n, L))).
13
% Preconditions for Primitive Actions.
15
poss(assign(P, D, T), S):- doctor(D), not assigned(P, D, S).
17
poss(patientAdmission(P, T), S):-  assigned(P, D, S), not inpatient(P, T1, S).
19
poss(revokeAssignation(P, D, T), S):- assigned(P, D, S).
21
poss(leave(P, T), S):-   inpatient(P, T1, S).
23
poss(startWrite(D, admissionNote, P, T, T1), S):- perm(startWrite(D, admissionNote,
25                                                 P, T, T1), S), not writingDoc(D,
                                                   admissionNote, P, T, T2, S),
27                                                 not writtenDoc(D, admissionNote,
                                                   P, T, S).
29
poss(startWrite(D, observation, P, T, T1), S):- perm(startWrite(D, observation,
31                                               P, T, T1), S), not writingDoc(D,
                                                 observation, P, T, T2, S),
33                                               not writtenDoc(D, observation,
                                                 P, T, S).
35
poss(endWrite(D, Type, P, T, T1), S):- perm(endWrite(D, Type, P, T, T1), S).
37
poss(endDeadline(admissionNote, P, T, T1), S):- inpatient(P, T, S),
39                                              not deadline(admissionNote, P, T , S),
                                                adDeadline(Deadline),
41                                              T1 $= T + Deadline.
```

```
43 poss(endDeadline(observation, P, T, T1), S):- inpatient(P, T, S),
                                                  not deadline(observation, P, T , S),
45                                                obDeadline(Deadline),
                                                  T1 $= T + Deadline.
47
   % Successor State Axioms
49
   assigned(P, D, do(A, S)):- A = assign(P, D, T);
51                             (assigned(P, D, S),
                               not A = revokeAssignation(P, D, T1),
53                             not A = leave(P, T2)).

55 deadline(Type, P, T, do(A, S)):- A = endDeadline(Type, P, T, T1);
                                    deadline(Type, P, T, S).
57
   inpatient(P, T, do(A, S)):- A = patientAdmission(P, T);
59                             (inpatient(P, T, S),
                               not A = leave(P, T1)).
61
   writingDoc(D, Type, P, T, T1, do(A, S)):- A = startWrite(D, Type, P, T, T1);
63                                           (writingDoc(D, Type, P, T, T1, S),
                                             not A = endWrite(D, Type, P, T, T2)).
65
   writtenDoc(D, Type, P, T, do(A, S)):- A = endWrite(D, Type, P, T, T1);
67                                       writtenDoc(D, Type, P, T, S).

69 % Derived active permissions

71 perm(startWrite(D, observation, P, T, T1), do(A, S)):- assigned(P, D, S),
                                                         not writingDoc(D, Type1, P1,
73                                                       T2, T3, S),
                                                         A = patientAdmission(P, T).
75
   perm(startWrite(D, observation, P, T, T1), do(A, S)):- assigned(P, D, S),
77                                                        inpatient(P, T, S),
                                                          A = endWrite(D, Type1, P1,
79                                                        T2, T3).

81 perm(startWrite(D, observation, P, T, T1), do(A, S)):- perm(startWrite(D, observation,
                                                         P, T, T1), S),
83                                                       not A = revokeAssignment(P,
                                                         D, T2), not A = leave(P, T2),
85                                                       not A = startWrite(D, Type1,
                                                         P1, T3, T4).
87
   perm(startWrite(D, admissionNote, P, T, T1), do(A, S)):- assigned(P, D, S),
89                                                          not writingDoc(D, Type1, P1,
                                                            T2, T3, S),
91                                                          A = patientAdmission(P, T).

93 perm(startWrite(D, admissionNote, P, T, T1), do(A, S)):- assigned(P, D, S),
                                                           inpatient(P, T, S),
95                                                         A = endWrite(D, Type1, P1,
                                                           T2, T3).
97
   perm(startWrite(D, admissionNote, P, T, T1), do(A, S)):- perm(startWrite(D,
99                                                           admissionNote, P, T, T1), S),
```

```
                                                    not A = revokeAssignment(P,
101                                                 D, T2), not A = leave(P, T2),
                                                    not A = startWrite(D, Type1,
103                                                 P1, T3, T4).

105

perm(endWrite(D, Type, P, T, T1), do(A, S)):- A = startWrite(D, Type, P, T, T2),
107                                            T1 $>= T2 + 5.

109 perm(endWrite(D, Type, P, T, T1), do(A, S)):- perm(endWrite(D, Type, P, T, T1), S),
                                                  not A = endWrite(D, Type, P, T, T2).
111

113 % Derived active obligations

115 ob(write(D, admissionNote, P, T), do(A, S)):- (assigned(P, D, S),
                                                  A = patientAdmission(P, T));
117                                               (ob(write(D, admissionNote, P, T), S),
                                                  not A = endWrite(D, admissionNote, P,
119                                               T, T1),
                                                  not A = endDeadline(admissionNote, P,
121                                               T, T2), not A = leave(P, T3),
                                                  not A = revokeAssignation(P, D, T4)).
123

ob(write(D, observation, P, T), do(A, S)):- (assigned(P, D, S),
125                                         A = patientAdmission(P, T));
                                            (ob(write(D, observation, P, T), S),
127                                         not A = endWrite(D, observation,
                                            P, T, T1),
129                                         not A = endDeadline(observation,
                                            P, T, T2),
131                                         not A = leave(P, T3),
                                            not A = revokeAssignation(P, D, T4)).
133

ob(endDeadline(admissionNote, P, T, T1), do(A, S)):- (A = patientAdmission(P, T),
135                                                   adDeadline(Deadline),
                                                      T1 $= T + Deadline);
137                                                   (ob(endDeadline(admissionNote,
                                                      P, T, T1), S),
139                                                   not A = endDeadline(admissionNote,
                                                      P, T, T1), not A = leave(P, T2)).
141

ob(endDeadline(observation, P, T, T1), do(A, S)):- (A = patientAdmission(P, T),
143                                                 obDeadline(Deadline),
                                                    T1 $= T + Deadline);
145                                                 (ob(endDeadline(observation,
                                                    P, T, T1), S),
147                                                 not A = endDeadline(observation,
                                                    P, T, T1), not A = leave(P, T2)).
149

% Derived the fulfilled obligations.
151

fulfil(endDeadline(Type, P, T, T1), do(A, S)):- (ob(endDeadline(Type, P, T, T1), S),
153                                              A = endDeadline(Type, P, T, T1)) ;
                                                 fulfil(endDeadline(Type, P, T, T1), S).
155

fulfil(write(D, Type, P, T), do(A, S)):- (ob(write(D, Type, P, T), S),
157                                       (A = endWrite(D, Type, P, T, T2));
```

```
                                        fulfil(write(D, Type, P, T), S)).
159
badSituation(do(A,S)):- A = endDeadline(Type, P, T, T1),
161                      not fulfil(write(D, Type, P, T), S),!.
badSituation(do(A,S)):- A = endDeadline(Type, P, T, T1),
163                      poss(endDeadline(Type1, P1, T2, T3), S), T3 $< T1,!.
badSituation(S):- ob(endDeadline(Type, P, T, T1), S), start(S, T2), not (T1 $>= T2) ,!.
165 badSituation(do(A,S)):- A = leave(P, T),!.
badSituation(do(A,S)):- A = revokeAssignation(P, D, T),!.
167 badSituation(do(A,S)):- A = assign(P, D, T),!.
badSituation(do(A,S)):- A = patientAdmission(P, T),!.
169
/* Initial Situation.*/
171 start(s0,0).
doctor(jean).
173
% The time of an action occurrence is its last argument.
175
time(assign(P, D, T), T).
177 time(revokeAssignation(P, D, T), T).
time(patientAdmission(P, T), T).
179 time(leave(P, T), T).
time(endDeadline(Type, P, T, T1), T1).
181 time(startWrite(D, Type, P, T, T1), T1).
time(endWrite(D, Type, P, T, T1), T1).
183
% Restore situation arguments to fluents.
185
restoreSitArg(fulfil(Rule), S, fulfil(Rule, S)).
187 restoreSitArg(ob(Rule), S, ob(Rule, S)).
restoreSitArg(badSituation, S, badSituation(S)).
189
% Primitive Action Declarations.
191
primitiveAction(startWrite(D, Type, P, T, T1)).
193 primitiveAction(endWrite(D, Type, P, T, T1)).
primitiveAction(endDeadline(Type, P, T, T1)).
195 primitiveAction(revokeAssignation(P, D, T)).
primitiveAction(assign(P, D, T)).
197 primitiveAction(patientAdmission(P, T)).
primitiveAction(leave(P, T)).
199

201 %  Utilities.

203 makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1), append(L1, [A], L).
205 prettyPrintSituation(S) :- makeActionList(S,Alist), nl, write(Alist),nl.

207 activeObligations(ActiveObligationsList, S):- findall(Rule, ob(Rule, S),
                                                  ActiveObligationsList).
209


211 reportStats:- nl, cputime(T), write(' CPU time (sec): '),
                 getval(cpu, T1), T2 is T - T1, write(T2), nl.
213
initializeCPU:- cputime(T), setval(cpu, T).
215
```

```
sEnforceable(N, S1):- initializeCPU, activeObligations(ActiveObligationsList, S1), nl,
217                   do(plan(N, ActiveObligationsList), S1, S),
                      prettyPrintSituation(S).
```

APPENDIX C

Gestion Des Conflits dans les Politiques de Contrôle d' Usage

Dans une politique de sécurité, les règles de permissions et d'interdictions sont généralement utilisées pour spécifier les politiques de contrôle d'accès. Les obligations sont utiles pour exprimer des politiques de contrôle d'usage. Ces règles appliquées à un même objet conduisent assez souvent à des situations conflictuelles. Dans la littérature, on distingue plusieurs types de conflits. Un type de conflit qui n' a pas encore été géré dans la littérature est le conflit entre les obligations avec délais qui survient lorsque ces délais se chevauchent. Les conflits entre les obligations avec délais sont difficiles à détecter et à gérer. Nous avons besoin d'un modèle qui gére comment le système d'information évolue dans le temps. Le but de la thèse est de développer des mécanismes de contrôle d'accès et de contrôle d'usage qui permettent d'implémenter les règles d'obligations avec délais, de détecter et de gérer les conflits.
Dans cette thèse, nous proposons un langage reposant sur les modalités déontiques pour spécifier des politiques d,Äôobligations avec délais. Ce modèle est intégré dans le langage du calcul des situations séquentiel temporel. Le calcul des situations est un langage qui représente le changement des mondes dynamiques comme un ensemble de formules de la logique du premier ordre. Ses éléments de base sont les actions qui peuvent être effectuées dans le monde, les fluents qui décrivent l'état du monde et les situations qui représentent une histoire d'occurrences d'actions. Le calcul des situations permet d'analyser la décidabilité et la complexité de plusieurs problèmes utiles comme le problème de projection temporelle. Ce problème consiste à vérifier si une formule est vrai après qu'une séquence d'actions a été effectuée à partir de la situation initiale. Ainsi notre modèle permet de décider quelle règle peut être appliquée à une situation donnée et de détecter les violations d'obligations dans un temps polynomial.

D'autre part, les contraintes constituent un aspect important des modèles de contrôle d'accès. Elles sont généralement utilisées pour éviter des situations frauduleuses. Ainsi notre modèle a été étendu pour exprimer une politique de contraintes.

Gestion des politiques de contraintes

Dans notre langage, il existe deux sortes de contraintes que nous appelons contraintes *historiques* et *non-historiques*. Les contraintes historiques servent à spécifier des exigences qui ne peuvent être vérifiées qu'en possédant des informations sur l'historique, alors que les contraintes non-historiques servent à exprimer des exigences liées á l'état courant. Nous montrons comment notre langage est suffisant pour exprimer des contraintes bien connues dans la littérature. Pour appliquer les contraintes, nous proposons d'abord de réécrire les contraintes historiques dans des formules simples (ne contenant qu'une seule variable de type situation). Nous proposons ensuite une procédure fondée sur le concept de régression pour faire respecter ces contraintes. En outre, nous spécifions formellement la condition dont la satisfaction permet de prouver que la spécification du système est sécurisée par rapport aux exigences de contrôle d'accès et de la politique des contraintes. Cependant l'utilisation des contraintes peut conduire à des situations conflictuelles.

Gestion des conflits dans les politiques d'obligations avec délais

L'utilisation des contraintes peut conduire à restreindre les situations dans lesquelles les actions peuvent être exécutées et par suite générer des situations où il est obligatoire de faire une action, mais il est impossible de l'exécuter. Dans ce sens, nous avons d'abord défini formellement les situations qui présentent ce genre de conflit que nous appelons un conflit de *faisabilité*. Ce conflit est détecté en utilisant les situations *exécutables*. Une situation exécutable est le résultat de l'exécution d'une action possible (les pré-conditions de l'action exécutée sont vérifiées) à partir d'une situation qui est elle même exécutable. Ainsi, sous l'hypothèse que la situation initiale est exécutable, ces situations sont construites par récursion. Nous dirons qu'il existe un conflit de faisabilité globale dans une politique de contrôle d'usage dans une situation donnée s si à partir de cette situation, il n'existe aucune situation exécutable s' où les obligations actives dans s peuvent être remplies.
De plus, notre modèle permet de détecter les conflits entre les permissions et les obligations. La détection de ce genre de conflit se fait en utilisant les situations *légales*. Comme les situations exécutables, ces situations sont construites récursivement. Une

situation légale est le résultat de l'exécution d'une action permise à partir d'une situation qui est elle même légale.
Nous avons proposé un algorithme de planification pour prouver si une politique de contrôle d'usage est globalement cohérente dans une situation donnée ou non (ne présentant pas de conflit de faisabilité ou un conflit entre les obligations et les permissions). La planification consiste à trouver une séquence d'actions qui permettent de satisfaire un but donné. Nous avons illustré notre approche en utilisant un exemple du milieu de la santé où les règles d'obligations avec délais sont utilisées pour assurer la disponibilité de l'information concernant les patients. Nous avons présenté la spécification formelle de ces règles en utilisant notre modèle. Puis nous avons réalisé l'implémentation en utilisant le langage de programmation d'actions Golog qui se fonde sur le calcul des situations. Dans notre implémentation, nous avons fait appel au Système de Programmation logique commune ECLIPSE 3.5.2 qui intègre l'algorithme du simplex pour résoudre les équations et les inégalités linéaires.
Il est possible qu'un conflit se produit parce qu'il existe une suite d'anciennes obligations qui n' ont pas été remplies et qui sont restées actives.
Déterminer si les utilisateurs sont responsables d'un conflit dépend de la nature des activités qui ont conduit à l'accumulation de ces obligations. En effet, nous devons distinguer parmi les actions qui ont été exécutées avant que le conflit se produise, les actions accompagnées d'une permission de les exécuter et les actions accompagnées d'un droit de les exécuter. Nous considérons que ce n'est pas équitable de révoquer le droit d'un utilisateur. Pour cela, nous avons enrichi notre langage avec une autre modalité qui est la modalité de droit.

Enrichir les politiques de sécurité par des règles de droits

Dans le modèle que nous proposons, un utilisateur ne peut être privé d'excercer son droit, sinon il y a une violation de droit. En général, les droits sont souvent associés à des sanctions en cas de violations. Par exemple, la CNIL (Commission nationale de l'informatique et des libertés) donne une liste de droits et prévoit des étapes à suivre si ces droits sont violés. Ainsi dans notre modèle, une action dont on détient le droit de l'exécuter doit toujours être possible tant que le droit est actif. En d'autre termes, les pré-conditions d'une action doivent être vérifiées dans toutes les situations où le droit de l'exécuter est actif. Dès lors qu'une des pré-conditions qui lui sont associées est désactivée, une violation de ce droit est détectée.
Cependant, l'introduction des règles de droit dans une politique de sécurité peut générer d'autre types de conflits comme le conflit entre les droits et les obligations

avec des délais. En effet, il peut y avoir des situations où il n' est pas possible de remplir les obligations dans leurs délais sans que cela entraîne une violation d'un droit. Pour détecter ce genre de conflit, nous avons défini les situations *préservant* les droits. Ces situations sont, comme les situations légales, construites par récursion et sont le résultat de l'exécution des actions qui ne désactivent aucune des pré-conditions des actions qui font l'objet d'un droit effectif. Ainsi, on dit qu'une politique présente un conflit entre les obligations et les droits dans une situation donnée s si il n'existe pas une situation préservant les droits, s', où les obligations actives dans s sont remplies. Pour prouver l'existence ou non de la situation s', on fait appel à la planification. L'approche est similaire à celle utilisée pour détecter les conflits entre les permissions et les obligations.
Notre modèle permet aussi de détecter les conflits entre les droits. Ce conflit survient dans les situations où l'exercice d'un droit empêche d'exercer un autre droit qu'on détient. Ce genre de conflit est détecté grâce aux situations exécutables.
Après avoir donné une sémantique au droit qui le distingue de la permission, nous avons formalisé les responsabilités sur les conflits.

Gestion des responsabilités

Le comportement des utilisateurs qui les rend responsables d'un conflit peut être un interval de temps mal exploité, comme il peut aussi être le résultat de l'exécution de certaines actions qui ont empêché de remplir certaines obligations actives. Notre approche comporte deux étapes :

- Déterminer l'interval de temps dans lequel on examine le comportement des utilisateurs. Cet interval de temps se situe entre la situation où la plus ancienne des obligations actives dans la situation conflictuelle s'est activée, et la situation conflictuelle. Cette situation est appelée *earlier* situation.
- A partir de la situation *earlier*, on cherche une situation exécutable, légale, qui préserve les droits et où les obligations actives dans la situation conflictuelle sont remplies. Dans cette recherche, on s'assure de reproduire les droits et les obligations qui ont été actives entre la situation *earlier* et la situation conflictuelle. Le temps d'activation est le même. On s'assure aussi de reproduire tout ce qui a été exercé comme droit et les actions qui ont été exécutées pour satisfaire des obligations.

Ainsi dans la recherche, toutes les combinaisons seront testées pour trouver une meilleur organisation des actions qui ont été effectuées (celle qui conduit à remplir les

obligations qui ont causé le conflit). Implicitement, cette recherche pourrait révoquer l'exécution de certaines actions qui n'ont servis ni à remplir une obligation ni à assurer un droit (même si elles sont permises). En outre, notre modèle assure une propriété de l'équité dans l'affectation des responsabilités. Dans la mesure où chaque fois qu'un utilisateur pourrait changer son comportement pour éviter les conflits, il est considéré comme responsable. Notre modèle permet également d'exprimer formellement une responsabilité partagée.
Finalement, nous proposons la délégation des obligations et la renonciation aux droits comme une éventuelle approche qui permettrait de résoudre les différents conflits que nous avons recensé dans les politiques de contrôle d'usages.

Résolution des conflits

Un conflit est dit *solvable* s'il est possible de trouver une situation dans le futur où les obligations causant le conflit seront remplies soit par le même utilisateur qui est concerné par l'obligation, soit par un utilisateur dont l'obligation lui a été déléguée. On propose que la redistribution des obligations causant le conflit obéisse à certains paramètres, comme par exemple choisir de déléguer une obligation à un utilisateur réputé d'être efficace dans l'accomplissement de cette obligation.
D'autre part, nous proposons une approche qui se fonde sur la renonciation aux droits pour résoudre les conflits entre les obligations et les droits. La renonciation à un droit est une action que l'utilisateur peut exécuter pour désactiver son propre droit. La renonciation au droit devrait être réglementée dans la politique par des règles de permissions contextuelles qui servent à cadrer le contexte où il peut y avoir un abandon de droit. Ces derniers points concernant la résolution des conflits ont été définis formellement et seront développés plus en détail dans des travaux futurs.

List of Publications

International Conferences

- Nada Essaouini, Frédéric Cuppens, Nora Cuppens-Boulahia, and Anas Abou El Kalam. Conflict management in obligation with deadline policies. In Proceedings of the 2013 International Conference on Availability, Reliability and Security, ARES '13, pages 52-61, Washington, DC, USA, 2013. IEEE Computer Society.
- Nada Essaouini, Frédéric Cuppens, Nora Cuppens-Boulahia, and Anas Abou El Kalam. Specifying and enforcing constraints in dynamic access control policies. In 2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014, pages 290-297. IEEE, 2014.

International Journals

- Nada Essaouini, Frédéric Cuppens, Nora Cuppens-Boulahia, and Anas El Kalam. Conflict detection in obligation with deadline policies. EURASIP Journal on Information Security, 2014(1):13, 2014.
- Frédéric Cuppens, Nora Cuppens-Boulahia, Meriam Ben-Ghorbel-Talbi, Stephane Morucci, and Nada Essaouini. Smatch: Formal dynamic session management model for RBAC. J. Inf. Sec. Appl., 18(1):30-44, 2013.

National Conferences

Nada Essaouini, Frédéric Cuppens, Nora Cuppens-Boulahia and Anas Abou El Kalam. Détection de conflit dans une politique de sécurité d'obligations avec délais. SARSSI 2013, 8ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes dÍnformation Mont de Marsan-Landes, France, 16-18 septembre 2013.

Bibliography

[A. N. Raval 2003] J. M. A. ,G. E. MarchioriA. N. Raval. Improving the continuity of care following discharge of patients hospitalized with heart failure: is the discharge summary adequate? *The Canadian Journal of Cardiology*, 19(4):365–370, 2003.

[Abiteboul et al. 1995] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Ahn and Sandhu 1999] G.-J. Ahn and R. Sandhu. The rsl99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM workshop on Role-based access control*, RBAC '99, pages 43–54, New York, NY, USA, 1999. ACM.

[Ahn and Sandhu 2000] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, November 2000.

[ANAES 1996] ANAES. Agence nationale d'accréditation et d'évalutaion en santé, la tenue du dossier médical en médecine générale:état des lieux et recommandations. SEPTEMBRE 1996.

[Atluri and Huang 1996] V. Atluri and W.-k. Huang. An authorization model for workflows. In *Proceedings of the 4th European Symposium on Research in Computer Security: Computer Security*, ESORICS '96, pages 44–64, London, UK, UK, 1996. Springer-Verlag.

[Atluri et al. 2000] V. Atluri, W.-K. Huang, and E. Bertino. A semantic-based execution model for multilevel secure workflows. *J. Comput. Secur.*, 8(1):3–41, January 2000.

[Becker and Nanz 2010] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3):20:1–20:28, July 2010.

[Ben Ghorbel-Talbi et al. 2011] M. Ben Ghorbel-Talbi, F. Cuppens, N. Cuppens-Boulahia, D. Le Métayer, and G. Piolle. Delegation of obligations and responsibility. In J. Camenisch, S. Fischer-Hubner, Y. Murayama, A. Portmann, and C. Rieder, editors, *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 197–209. Springer Berlin Heidelberg, 2011.

[Benferhat et al. 2003] S. Benferhat, R. E. Baida, and F. Cuppens. A stratification-based approach for handling conflicts in access control. In *SACMAT 2003, 8th ACM Symposium on Access Control Models and Technologies, June 2-3, 2003, Villa Gallia, Como, Italy, Proceedings*, pages 189–195, 2003. ACM.

[Bertino et al. 2001] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, SACMAT '01, pages 41–52, New York, NY, USA, 2001. ACM.

[Bertino et al. 1999] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, February 1999.

[Bertino et al. 1996] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 94–, Washington, DC, USA, 1996. IEEE Computer Society.

[Bettini et al. 2002a] C. Bettini, S. Jajodia, X. Wang, and D. Wijesekera. Obligation monitoring in policy management. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, POLICY '02, pages 2–, Washington, DC, USA, 2002. IEEE Computer Society.

[Bettini et al. 2002b] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 502–513, 2002. VLDB Endowment.

[Bettini et al. 2003] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. Provisions and obligations in policy rule management. *J. Netw. Syst. Manage.*, 11(3):351–372, September 2003.

[Bolton 2001] P. Bolton. A quality assurance activity to improve discharge communication with general practice. *Journal of Quality in Clinical Practice*, 21:69–70, 2001.

[C. van Walraven 2002] C. van A. L. ,R. SethWalraven. Dissemination of discharge summaries. not reaching follow-up physicians. *Canadian Family Physician*, 48:737–742, 2002.

[Capitani et al. 2005] S. D. Capitani, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. 2005.

[Chomicki 1995] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, June 1995.

[Chomicki and Lobo 2001] J. Chomicki and J. Lobo. Monitors for history-based policies. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 57–72, London, UK, UK, 2001. Springer-Verlag.

[Chomicki et al. 2003] J. Chomicki, J. Lobo, and S. Naqvi. Conflict resolution using logic programming. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):244–249, January 2003.

[Clark and Wilson 1987] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195, 1987. IEEE Computer Society.

[Clark 1987] K. L. Clark. Readings in nonmonotonic reasoning. chapter Negation as failure, pages 311–325. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[CMQ 2005] CMQ. Collège des médecins du québec: La tenue des dossiers par le médecin en centre hospitalier de soins généraux et spécialisés. Décembre 2005.

[Cole et al. 2001] J. Cole, J. Derrick, Z. Milosevic, and K. Raymond. Author obliged to submit paper before 4 july: Policies in an enterprise specification. In M. Sloman, E. Lupu, and J. Lobo, editors, *Policies for Distributed Systems and Networks*, volume 1995 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2001.

[Crampton 2003] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 43–50, New York, NY, USA, 2003. ACM.

[Craven et al. 2009] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, and A. Bandara. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 239–250, New York, NY, USA, 2009. ACM.

[Cuppens et al. 2007] F. Cuppens, N. Cuppens-Boulahia, and M. B. Ghorbel. High level conflict management strategies in advanced access control models. *Electr. Notes Theor. Comput. Sci.*, 186:3–26, 2007.

[Cuppens et al. 2005] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: a security model with non atomic actions and deadlines. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 186–196, June 2005.

[Cuppens and Miege 2003] F. Cuppens and A. Miege. Modelling contexts in the or-bac model. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 416–425, Dec 2003.

[Damianou et al. 2001] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, UK, 2001. Springer-Verlag.

[Dinolt et al. 1994] G. Dinolt, L. A. Benzinger, and M. G. Yatabe. Combining components and policies. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 22–33, Jun 1994.

[Elrakaiby et al. 2012a] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formal enforcement and management of obligation policies. *Data Knowl. Eng.*, 71(1):127–147, 2012.

[Elrakaiby et al. 2012b] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formal enforcement and management of obligation policies. *Data Knowl. Eng.*, 71(1):127–147, January 2012.

[Essaouini et al. 2014a] N. Essaouini, F. Cuppens, N. Cuppens-Boulahia, and A. El Kalam. Conflict detection in obligation with deadline policies. *EURASIP Journal on Information Security*, 2014(1):13, 2014.

[Essaouini et al. 2013] N. Essaouini, F. Cuppens, N. Cuppens-Boulahia, and A. A. El Kalam. Conflict management in obligation with deadline policies. In *Proceedings of the 2013 International Conference on Availability, Reliability and*

Security, ARES '13, pages 52–61, Washington, DC, USA, 2013. IEEE Computer Society.

[Essaouini et al. 2014b] N. Essaouini, F. Cuppens, N. Cuppens-Boulahia, and A. A. E. Kalam. Specifying and enforcing constraints in dynamic access control policies. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, pages 290–297, 2014. IEEE.

[F. Cuppens and Miége 2004] N. C.-B. F. Cuppens and A. Miége. Inheritance hierarchies in the or-bac model and application in a network environment. 2004.

[Ferraiolo et al. 2001] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, August 2001.

[Fikes and Nilsson 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[Goedertier and Vanthienen 2006] S. Goedertier and J. Vanthienen. Designing compliant business processes with obligations and permissions. In *Proceedings of the 2006 International Conference on Business Process Management Workshops*, BPM'06, pages 5–14, Berlin, Heidelberg, 2006. Springer-Verlag.

[Green 1969a] C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[Green 1969b] C. Green. Theorem-proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, chapter 11, pages 183–205. Edinburgh University Press, 1969.

[Hanks and McDermott 1987] S. Hanks and D. McDermott. Readings in nonmonotonic reasoning. chapter Default Reasoning, Nonmonotonic Logics, and the Frame Problem, pages 390–395. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[Harrison et al. 1976] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[Hilty et al. 2005] M. Hilty, D. Basin, and A. Pretschner. On obligations. In *Proceedings of the 10th European Conference on Research in Computer Security*, ESORICS'05, pages 98–117, Berlin, Heidelberg, 2005. Springer-Verlag.

[Hilty et al. 2007] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Conference on Research in Computer Security*, ESORICS'07, pages 531–546, Berlin, Heidelberg, 2007. Springer-Verlag.

[Irwin et al. 2008] K. Irwin, T. Yu, and W. Winsborough. Assigning responsibility for failed obligations. In Y. Karabulut, J. Mitchell, P. Herrmann, and C. Jensen, editors, *Trust Management II*, volume 263 of *IFIP ,Äì The International Federation for Information Processing*, pages 327–342. Springer US, 2008.

[Irwin et al. 2006] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 134–143, New York, NY, USA, 2006. ACM.

[J. I. Balla 1994] W. E. J. J. I. Balla. Improving the continuity of care between general practitioners and public hospitals. *Medical Journal of Australia*, 161(11-12):656–659, 1994.

[Jaeger and Tidswell 2001] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, 4(2):158–190, May 2001.

[Jajodia et al. 2001] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, June 2001.

[Janicke et al. 2007] H. Janicke, A. Cau, and H. Zedan. A note on the formalisation of ucon. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 163–168, New York, NY, USA, 2007. ACM.

[Kalam et al. 2003] A. Kalam, R. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization based access control. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131, June 2003.

[Kowalski and Sergot 1986] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, January 1986.

[Lamport 1994] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[Lampson 1971] B. Lampson. Protection. *5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, 1971.

[Lentzner 2004] R. Lentzner. Sql 3:initiation et programmation. 2004.

[Levesque et al. 1997] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[Li et al. 2007] N. Li, M. V. Tripunitara, and Z. Bizri. On mutually exclusive roles and separation-of-duty. *ACM Trans. Inf. Syst. Secur.*, 10(2), May 2007.

[Lin and Reiter 1994] F. Lin and R. Reiter. State constraints revisited. *J. Log. Comput.*, 4(5):655–678, 1994.

[Lobo et al. 1999] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[Lupu and Sloman 1999] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25:852–869, 1999.

[Mageean 1986] R. J. Mageean. Study of discharge communications from hospital. *British Medical Journal*, 293(6557):1283–1284, 1986.

[McCarthy 1968] J. McCarthy. Programs with common sense. In *Semantic Information Processing*, pages 403–418, 1968. MIT Press.

[McCarthy 1983] Stanford Artificial Intelligence Project. Situations, actions, and causal laws. Technical Report Memo 2, Stanford University, 1983.

[Miller and Shanahan 2002] R. Miller and M. Shanahan. Some alternative formulations of the event calculus. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 452–490, London, UK, UK, 2002. Springer-Verlag.

[Moffett and Sloman 1993] J. D. Moffett and M. S. Sloman. Policy conflict analysis in distributed system management. 1993.

[Ni et al. 2008] Q. Ni, E. Bertino, and J. Lobo. An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 133–142, New York, NY, USA, 2008. ACM.

[Nuffelen 2004] V. Nuffelen. Abductive constraint logic programming: Implementation and applications. 2004.

[Park and Sandhu 2004] J. Park and R. Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[Pednault 1988] E. P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4:356–372, 1988.

[Pednault 1989] E. P. D. Pednault. Adl: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Pontual et al. 2010a] M. Pontual, O. Chowdhury, W. H. Winsborough, T. Yu, and K. Irwin. Toward practical authorization-dependent user obligation systems. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 180–191, New York, NY, USA, 2010. ACM.

[Pontual et al. 2011] M. Pontual, O. Chowdhury, W. H. Winsborough, T. Yu, and K. Irwin. On the management of user obligations. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 175–184, New York, NY, USA, 2011. ACM.

[Pontual et al. 2010b] M. Pontual, K. Irwin, O. Chowdhury, W. Winsborough, and T. Yu. Failure feedback for user obligation systems. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*, pages 713–720, Aug 2010.

[Rakaiby et al. 2009] Y. E. Rakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formalization and management of group obligations. In *Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, POLICY '09, pages 158–165, Washington, DC, USA, 2009. IEEE Computer Society.

[Reiter 1991] R. Reiter. Artificial intelligence and mathematical theory of computation. chapter The frame problem in situation the calculus: a simple solution

(sometimes) and a completeness result for goal regression, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[Reiter 1993] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[Reiter 1996] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996.*, pages 2–13, 1996. Morgan Kaufmann.

[Reiter 1998] R. Reiter. Sequential, temporal GOLOG. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998.*, pages 547–556, 1998. Morgan Kaufmann.

[R.R.O 1990] R.R.O. Réglement.965: Gestion hospitalière: Dossiers de renseigments personnels sur la santé. 1990.

[Saltzer and Schroeder 1975] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[Sandhu et al. 1996] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[Sans et al. 2007] T. Sans, F. Cuppens, and N. Cuppens-Boulahia. A framework to enforce access control, usage control and obligations. *Annales des Télécommunications*, 62(11-12):1329–1352, 2007.

[Sartor 2005] G. Sartor. Legal reasoning: A cognitive approach to the law. *Springer*, 2005.

[Schubert 1990] L. K. Schubert. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In *Knowledge Representation and Defeasible Reasoning*, pages 23–67, 1990. Kluwer Academic Press.

[Thom Fruhwirth 1992] V. K. T. L. P. P. L. E. M. M. W. ,Alexander HeroldThom Fruhwirth. Constraint logic programming. logic programming in action. *Lecture Notes in Computer Science*, 636:3–35, 1992.

[Thomas 1997] R. K. Thomas. Team-based access control (tmac): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, RBAC '97, pages 13–19, New York, NY, USA, 1997. ACM.

[Thomas and Sandhu 1998] R. K. Thomas and R. S. Sandhu. Task-based authorization controls (tbac): A family of models for active and enterprise-oriented autorization management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects*, pages 166–181, London, UK, UK, 1998. Chapman & Hall, Ltd.

[Tidswell and Jaeger 2000a] J. E. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 154–163, New York, NY, USA, 2000. ACM.

[Tidswell and Jaeger 2000b] J. E. Tidswell and T. Jaeger. Integrated constraints and inheritance in dtac. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 93–102, New York, NY, USA, 2000. ACM.

[Ullman 1988] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[Waldinger 1977] R. Waldinger. Achieving several goals simultaneously. In E. W. Elcock and D. Michie, editors, *Machine Intelligence*, volume 8, pages 94–136. Wiley, 1977.

[XACML] XACML. Oasis, extensible access control markup language tc v2.0, normative xacml 2.0. http://www.oasis-open.org/specs/index.php.

List of Figures

3.1 Example of actual norm derivation 34

5.1 Motivation example 56

6.1 Right example 72

6.2 Example of a legal and not preserved plan: violation of right to join training in many situations 83

7.1 Example of legal, executed and preserved plans which prove that the conflict in σ_c could have been avoided 91

Résumé

Les politiques de sécurité s'expriment en général par des règles de permissions et d'interdictions. Plus récemment, les spécifications et mises en oeuvre des règles d'obligation commencent à voir le jour, notamment pour exprimer des politiques de contrôle d'usage. Dans cette thèse, nous proposons un langage reposant sur les modalités déontiques pour spécifier des politiques d'obligations avec délais. Ce modèle est intégré dans le langage du calcul des situations séquentiel temporel. Le modèle permet de prouver si un ensemble d'obligations actives dans une situation donnée est globalement satisfaisable ou non. La démarche repose sur une recherche de planification des obligations. Le modèle permet aussi d'exprimer les permissions et analyser un autre type de conflit lorsqu'il est impossible de trouver un plan d'actions permises qui permet de remplir les obligations avec deadline. Le modèle permet aussi de spécifier un ensemble de contraintes associées à la politique de sécurité. La démarche permet de prouver que les contraintes seront toujours satisfaites. Finalement, nous avons étendu notre modèle pour définir une politique incluant des règles de droit. La sémantique proposée permet de formaliser la différence entre permission et droit. Cette distinction permet de prouver si un conflit dans une situation donnée provient d'une anomalie dans la politique ou si elle relève de la responsabilité d'un utilisateur. De plus, le modèle formalise une propriété d'équité dans le jugement des responsabilités. Lorsqu'un utilisateur a la possibilité de changer son comportement pour éviter un conflit, il est considéré responsable. Le modèle permet également de formaliser les situations de responsabilité partagée.

Mots-clés : Sécurité informatique, Politique de sécurité, Contrôle d'accès, Contrôle d'usage, Obligations avec délais, Calcul des situations, Gestion des conflits, Planification

Abstract

The security policies are commonly specified through permissions, prohibitions and obligations. Permissions and prohibitions are generally used to specify access control policies, while obligations are useful to express usage control policies. Two different types of obligations are generally considered, namely system obligations and user obligations. User obligations are associated with deadlines. When these obligations are activated, these deadlines provide the user with some time to enforce the obligation before violation occurs. Using obligations with deadlines in security policies may cause a new type of conflict. This kind of conflict could happen in presence of overlapping deadlines. In this thesis, we propose a language based on a deontic logic of actions to express permissions and obligations. The semantic of the proposed language is defined using the situation calculus formalism. This allows us to analyze decidability and complexity of several problems such as planning tasks. We then use the planning task to prove the existence of conflictual situations.

Once the conflicts are detected, we use delegation to redistribute obligations in order to resolve these conflicts and thus avoid possible violations. Furthermore, we show that obligations and permissions are not sufficient to preserve interests of the system's users. Indeed, for fairness reasons, the possibility of executing actions achieving their interests should be always preserved. Otherwise, violations are triggered. However, these actions are not obligations since users have the choice to execute them or not. Thus, the violation is the consequence of system's failure in preserving the choice of users. Consequently, we enriched our model with right rules to enable this feature. Finally, we show how the use of rights allows the refinement of responsibility when conflicts occur.

Keywords : Computer security, Security policy, Access and usage control, Obligation with deadline, Situation calculus, Conflict management, Planning

TELECOM Bretagne

n° d'ordre : 2015telb0340

Télécom Bretagne

Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3

Tél : + 33(0) 29 00 11 11 - Fax : + 33(0) 29 00 10 00