Enforcement of Privacy Preferences in Data Services: A SPARQL Query Rewriting Approach
Said Oulmakhzoune

To cite this version:

HAL Id: tel-00833895
https://tel.archives-ouvertes.fr/tel-00833895
Submitted on 13 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Enforcement of Privacy Policy Preferences in Data Services: A SPARQL Query Rewriting Approach

soutenue le 29 avril 2013

Composition du Jury :

Président : - M. Dominique MÉRY, Professeur, LORIA & Université de Lorraine

Rapporteurs : - M. Abdelmalek BENZEKRI, Professeur, IRIT Université Paul Sabatier
- M. Ernesto DAMIANI, Professeur, Universita’ degli Studi di Milano

Examinateurs : - M. Frédéric CUPPENS, Professeur, Telecom Bretagne
- Mme Nora CUPPENS, Chercheur associé, Telecom Bretagne
- M. Vincent FREY, Ingénieur de recherche - Chef de projet, Orange
- M. Marcel GOLDBERG, Professeur, INSERM
- M. Stéphane MORUCCI, Directeur Général, SWID
Abstract

With the constant proliferation of information systems around the globe, the need for decentralized and scalable data sharing mechanisms has become a major factor of integration in a wide range of applications. Literature on information integration across autonomous entities has tacitly assumed that the data of each party can be revealed and shared to other parties. A lot of research, concerning the management of heterogeneous sources and database integration, has been proposed, for example based on centralized or distributed mediators that control access to data managed by different parties.

On the other hand, real life data sharing scenarios in many application domains like healthcare, e-commerce market, e-government show that data integration and sharing are often hampered by legitimate and widespread data privacy and security concerns. Thus, protecting the individual data may be a prerequisite for organizations to share their data in open environments such as Internet.

Work undertaken in this thesis aims to ensure security and privacy requirements of software systems, which take the form of web services, using query rewriting principles. The user query (SPARQL query) is rewritten in such a way that only authorized data are returned with respect to some confidentiality and privacy preferences policy. Moreover, the rewriting algorithm is instrumented by an access control model (OrBAC) for confidentiality constraints and a privacy-aware model (PrivOrBAC) for privacy constraints.

A secure and privacy-preserving execution model for data services is then defined. Our model exploits the services’ semantics to allow service providers to enforce locally their privacy and security policies without changing the implementation of their data services i.e., data services are considered as black boxes. We integrate our model to the architecture of Axis 2.0 and evaluate its efficiency in the healthcare application domain.
Résumé

Avec la prolifération constante des systèmes d’information à travers le monde, la nécessité d’une décentralisation des mécanismes de partage de données est devenue un facteur important d’intégration dans une large gamme d’applications. La littérature sur l’intégration d’information entre les entités autonomes a tacitement admis que les données de chacune des parties peuvent être révélées et partagées avec d’autres parties. Plusieurs travaux de recherches, concernant la gestion des sources hétérogènes et l’intégration de base de données, ont été proposés, par exemple les systèmes à base de médiateurs centralisés ou distribués qui contrôlent l’accès aux données gérées par des différentes parties.

D’autre part, les scénarios réels de partage des données de nombreux domaines d’application tels que la santé, l’e-commerce, e-gouvernement montrent que l’intégration et le partage de données sont souvent entravés par la confidentialité des données privées et les problèmes de sécurité. Ainsi, la protection des données individuelles peut être une condition préalable aux organisations pour partager leurs données dans des environnements ouverts tels que l’Internet.

Les travaux entrepris dans cette thèse ont pour objectif d’assurer les exigences de sécurité et de confidentialité des systèmes informatiques, qui prennent la forme des services web, en utilisant le principe de réécriture de requêtes. La requête de l’utilisateur, exprimée en SPARQL, est réécrite de sorte que seules les données autorisées sont retournées conformément à la politique de confidentialité et aux préférences des possesseurs des données. En outre, l’algorithme de réécriture est instrumenté, dans le cas d’une politique de confidentialité, par un modèle de contrôle d’accès (OrBAC). Dans le cas d’une politique de préférences utilisateurs, il est instrumenté par un modèle de politique de privacy (PrivOrBAC).

Ensuite, nous avons défini un modèle d’exécution sécurisé et préservant la privacy pour les services de données. Notre modèle exploite la sémantique des services afin de permettre aux fournisseurs de services d’assurer localement leurs politiques de sécurité et de privacy sans changer l’implémentation de leurs services. C’est-à-dire que les services de données sont considérés comme des boîtes noires. Enfin, nous avons intégré notre modèle dans l’architecture Axis 2.0 et nous avons aussi évalué ses performances sur des données du domaine médical.
1 Introduction

1.1 Motivation and background ............................................ 1
1.2 Contributions ............................................................. 1
1.3 Outline of the dissertation ............................................ 2

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

2.1 Background ............................................................... 5
  2.1.1 RDF ................................................................. 5
  2.1.2 OWL ................................................................. 6
  2.1.3 SPARQL ............................................................ 6
  2.1.4 SPARQL/Update .................................................. 9
  2.2 Semantic Mediation .................................................... 9
  2.3 Security enforcement for mediators ................................ 10
  2.4 Security of query evaluation ....................................... 12
    2.4.1 View-based approach ......................................... 13
    2.4.2 Pre-processing approach ..................................... 14
    2.4.3 Post-processing approach .................................... 16

3 SPARQL Select Query Rewriting to Enforce Data Confidentiality 19

3.1 Introduction ............................................................. 19
3.2 Rewriting SPARQL Query: Basic Principles .......................... 20
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Notations, Definitions and Theorems</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Security policy</td>
<td>27</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Permission</td>
<td>27</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Prohibition</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
case  | Query: Our query rewriting model                                      | 29   |
| 3.5.1   | Case of simple condition $\varnothing$                                | 29   |
| 3.5.2   | Case of involved condition $\varnothing$                              | 32   |
| 3.5.3   | Case of complex condition $\varnothing$                               | 34   |
| 3.5.4   | Composition of simple and involved conditions                         | 37   |
| 3.6     | Conclusion and Contribution                                           | 39   |
| 4       | Rewriting of SPARQL/Update Queries for Securing Data access           | 41   |
| 4.1     | Introduction                                                          | 41   |
| 4.2     | Motivating example                                                    | 42   |
| 4.3     | Principle of our approach                                             | 44   |
| 4.3.1   | Update access control                                                 | 44   |
| 4.3.2   | Consistency between consultation and modification                     | 46   |
| 4.4     | Conclusion and Contribution                                           | 51   |
| 5       | SPARQL Query Rewriting Instrumented by an Access Control Model         | 53   |
| 5.1     | Introduction                                                          | 53   |
| 5.2     | The OrBAC model                                                       | 54   |
| 5.2.1   | Basic predicates                                                      | 55   |
| 5.2.2   | Role, activity and view definition                                    | 56   |
| 5.2.3   | Context definition                                                    | 57   |
| 5.2.4   | Hierarchy and inheritance                                             | 58   |
| 5.3     | Principle of the approach                                             | 59   |
| 5.4     | Modelling RDF Condition within OrBAC                                  | 61   |
5.4.1 RDF condition as context ........................................ 61
5.4.2 RDF condition as view ........................................... 63
5.4.3 RDF condition as view and context .......................... 66
5.5 Rewriting Query Instrumented by OrBAC rules .............. 67
5.6 Conclusion .......................................................... 68

6 Privacy policy preferences enforced by SPARQL Query Rewriting 69
6.1 Introduction .......................................................... 69
6.2 Approach principle .................................................. 71
6.3 Privacy-aware Ontology ............................................ 73
6.4 The correctness criteria ............................................ 77
6.5 Rewriting Algorithm principle .................................... 79
   6.5.1 Normalization of triple patterns ......................... 79
   6.5.2 Preferences acquisition ............................... 80
   6.5.3 Preferences enforcement ........................... 81
   6.5.4 SPARQL query without filter ....................... 83
   6.5.5 SPARQL query with filters ...................... 86
6.6 Conclusion .......................................................... 87

7 Privacy query rewriting algorithm instrumented by a privacy-aware access control model 89
7.1 Introduction .......................................................... 89
7.2 The privacy-aware OrBAC model (PrivOrBAC) ................ 90
   7.2.1 Consent ..................................................... 91
   7.2.2 Purpose .................................................... 92
   7.2.3 Accuracy ................................................... 94
7.3 Our approach: PrivOrBAC query rewriting algorithm ........ 95
   7.3.1 PrivOrBAC services ........................................ 95
   7.3.2 PrivOrBAC SPARQL Service ......................... 98
CONTENTS

7.4 Conclusion ............................................. 101

8 Secure and Privacy-preserving Execution Model for Data Services ........ 103

8.1 Introduction ........................................... 103

8.1.1 Motivating Scenario .................................. 104

8.1.2 Challenges ........................................ 105

8.1.3 Contributions ....................................... 105

8.2 A Secure and Privacy-Preserving Execution Model for Data Services ... 106

8.2.1 Model Overview ..................................... 107

8.2.2 Semantic models for data services and policies .................. 107

8.2.3 RDF views rewriting to integrate security and privacy constraints ... 109

8.2.4 Rewriting the extended view in terms of data services ............ 111

8.2.5 Enforcing security and privacy constraints ..................... 113

8.3 Conclusion and Perspectives .................................. 115

9 Architectures and Implementations .................................... 117

9.1 Implementation of fQuery .................................. 117

9.1.1 MoToR-BAC tool ...................................... 117

9.1.2 Implementation of fQuery-AC ............................. 118

9.1.3 Implementation of fQuery-Privacy .......................... 123

9.2 Performance of fQuery-Privacy instrumented by PrivOrBAC ......... 123

9.2.1 Architecture ........................................... 125

9.2.2 Use case ............................................... 126

9.2.3 Experimental results ................................... 129

9.3 Performance of Secure and Privacy-preserving Execution Model for Data Services ...................................................... 131

9.3.1 Implementation ........................................ 132

9.3.2 Evaluation ........................................... 133

9.4 Use cases .................................................. 135
CONTENTS

9.4.1 AGGREGO Server .................................................. 135
9.4.2 PAIRSE ................................................................. 138

10 Conclusion and perspectives ........................................ 141

A fQuery: réécriture de requêtes SPARQL .......................... 145
A.1 Introduction .............................................................. 145
A.2 Généralités ............................................................... 147
A.3 fQuery-AC: Principe de base ....................................... 148
A.4 fQuery-Privacy: Principe de base ................................. 149
A.5 Le cas des services de données .................................... 151
A.6 Implémentation .......................................................... 154

B Proof of theorem 3 chapter 3 ....................................... 155

C fQuery-AC Aspect ........................................................ 159

D fQuery-AC Visitor ......................................................... 163

List of Publications ....................................................... 167

Bibliography ...................................................................... 168

List of Figures .................................................................... 183
CHAPTER 1
Introduction

1.1 Motivation and background

The problem of data sharing has been investigated for several years. Nowadays, several companies as well as scientific communities and governments, feel a large need of sharing their data (data of different structures). Indeed, there are many and varied data sources (relational databases, object databases, files, etc.). This has led to a lot of research work on the management of heterogenous sources and database integration. For instance, TSIMMIS [1], Information Manifold [2], HERMES [3], DISCO [4], Garlic [5] and MMM [6]. The goal of such systems is to exploit of several independent data sources as if they were a single source, with a single global schema. They allow users to make complex queries over heterogeneous databases, as if they were a single one. Whenever a user expresses a query in terms of relations in the global schema, the system (mediator) translates the query into sub-queries using a query-reformulation procedure. These sub-queries can be executed in sources and the system can collect and combine returned results as the answer to the query.

However, Several proposed approaches, for data integration, do not take into account security issues and privacy requirements. Their main goal is the management of heterogenous sources and database integration. They help data sharing among different distributed sources but they increase the risk for data security, such as violating access control rule [7]. Moreover, they suppose that the security and privacy issues are only handled at the level of each data source.

1.2 Contributions

The integration of heterogeneous data sources must also consider security and privacy concerns. It is important to consider whether the client has the necessary credentials to access the data before the integration can occur.
We propose an approach that enforces the security and privacy requirements for a semantic mediator that uses SPARQL as query language. Our approach is based on a query rewriting principle. It is to rewrite the user query (SPARQL query) such that only authorized data are returned with respect to some confidentiality and privacy preferences policy.

In the case of confidentiality constraints, the rewriting algorithm is instrumented by an access control model like the OrBAC model [8]. For instance, if we suppose that a user $U$ issues a SPARQL query $Q_i$, the approach aims to get security constraints defined for the user $U$ and transform them into SPARQL filters and SPARQL triples that will be inserted into the initial query $Q_i$. The results returned by the rewritten query are compliant with the confidentiality policy.

In the case of update queries, we show how to rewrite SPARQL update queries without disclosing some other sensitive data whose access would be forbidden through select queries. We present an approach based on rewriting SPARQL/Update queries. It involves two steps. The first one satisfies the update constraints. The second one handles consistency between select and update operators. Query rewriting is done by adding positive and negative filters (corresponding respectively to permissions and prohibitions) to the initial query.

In the case of privacy constraints, the rewriting algorithm is instrumented by a privacy-aware model like the PrivOrBAC model [9]. Our approach aims to enforce the privacy policy preferences by query transformation. We take into account various dimensions of privacy preferences through the concepts of consent, accuracy, purpose and recipient.

Finally, we propose a secure, privacy-preserving execution model for data services based on our rewriting algorithms. This model allows service providers to enforce their privacy and security policies without changing the implementation of their data services i.e., data services are considered as black boxes.

### 1.3 Outline of the dissertation

In chapter 2 we present some key definitions and we start by introducing the concept of semantic mediation system. Then we discuss about some existing security enforcement for mediators and analyze other possible approaches for securing mediators, mainly the security of query evaluation.
1.3. OUTLINE OF THE DISSERTATION

In chapter 3, we define $f$Query, an approach used to protect SPARQL queries using query transformation. We present a generic approach to specify and apply an access control policy to protect resources viewed as RDF format. In this chapter, we consider the case of select queries.

In chapter 4, we propose an extension that considers how to transform update queries with respect to an access control policy. The access control policy is modelled as a set of filters.

In chapter 5, we define a user friendly specification language to express such an access control policy. We show how to derive the filter definition from the specification of an access control policy based on OrBAC [8].

In chapter 6, we present an extension of our approach $f$Query that aims to enforce the privacy requirements by query rewriting in the case of SPARQL queries.

In chapter 7, we show how to instrument our privacy rewriting algorithm using an existing privacy-aware model like the PrivOrBAC model.

In chapter 8, we present a use case of our rewriting approach. We propose a secure and privacy-preserving execution model for data services based on our $f$Query approach.

In chapter 9, we present concrete implementation of our approach $f$Query. Finally, chapter 10 makes an overview of the thesis, analyzes its strong and weak points and suggests future work directions.
In this chapter, we present some key definitions. Then we introduce the concept of semantic mediation system. After, we discuss about existing security enforcement for mediators. Finally we analyze other possible approaches for securing mediators, mainly security of query evaluation.

2.1 Background

2.1.1 RDF

RDF [10](Resource Definition Framework) is a graph data model. It is based upon the idea of making statements about resources (in particular Web resources) in the form of subject-predicate-object expressions. These expressions are known as triples in RDF terminology. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. For example, one way to represent the proposition "Bob’s salary is 60k" in RDF is as the triple: a subject denoting "Bob", a predicate denoting "has salary", and an object denoting "60k". A collection of RDF statements intrinsically represents a labeled, directed multi-graph. As such, an RDF-based data model is more naturally suited to certain kinds of knowledge representation than the relational model and other ontological models traditionally used in information systems today.

In practice, as more data is being stored in RDF format, a need has arisen for a simple way to locate specific information. SPARQL [11] is a powerful query language which fills that space, making it easy to find the data you need in the RDF graphs.
2.1.2 OWL

Ontology formally represents knowledge as a set of concepts within a domain, and the relationships between pairs of concepts. It can be used to model a domain and support reasoning about entities [12].

The Web Ontology Language (OWL) is a family of knowledge representation (KR) languages for authoring ontologies [12]. OWL is endorsed by the World Wide Web Consortium (W3C) and has attracted academic, medical and commercial interest.

There are three variants of OWL with different levels of expressiveness: (i) OWL Lite, (ii) OWL DL and (iii) OWL Full.

- OWL Lite: a sublanguage of OWL that was originally intended to support a classification hierarchy and simple constraints (e.g. a set is limited to 0 or 1 element).

- OWL DL (OWL Description Logics): a sublanguage of OWL that was designed to provide the maximum expressiveness possible while retaining computational completeness, decidability and the availability of practical reasoning algorithms.

- OWL Full: a sublanguage of OWL that was designed to preserve some compatibility with RDF Schema. It is based on a different semantics from OWL Lite or OWL DL. OWL Full is undecidable, so no reasoning software is able to perform complete reasoning for it.

2.1.3 SPARQL

SPARQL is the acronym of Simple Protocol And RDF Query Language. It is the query language of the Semantic Web for accessing RDF databases. It was standardized by the RDF Data Access Working Group of the World Wide Web Consortium, and is considered a key semantic web technology.

A SPARQL query consists of triple patterns, conjunctions, disjunctions, and optional patterns. SPARQL allows users to write globally unambiguous queries. For example, the following query returns the name of all patients and their drug name.
2.1. BACKGROUND

Variables are indicated by a "?" or "$" prefix. Bindings for ?name and ?drugName will be returned.

Basically, the SPARQL syntax resembles SQL, but the advantage of SPARQL is that it enables queries spanning multiple disparate (local or remote) data sources containing heterogeneous semi-structured data. In the rest of this section we present two main keywords that allow us to query local and remote data sources.

**SPARQL Named Graph**

SPARQL query is executed against RDF Dataset. An RDF Dataset is a collection of RDF graphs. It always contains one default graph which does not have a name. It contains also zero or more named graphs where each named graph is identified by an IRI.

The RDF Dataset is referenced in the SPARQL query using keywords FROM and FROM NAMED. A default graph consisting of the merge of the graphs referred to in the FROM clauses. Named graphs are presented as (IRI, graph) pairs, one from each FROM NAMED clause.

The GRAPH clause directs queries to particular named graphs. The figure 2.1 illustrates an example of use of GRAPH clauses. The default graph is the merge of the two graphs dt:drugs and dt:infos. The GRAPH keyword is used to match group pattern P against the named graph pg/preferences.

**SPARQL Service**

SPARQL is also used to express queries across diverse data sources that are viewed as RDF via middleware. The specification of SPARQL 1.1 defines the syntax and semantics for executing distributed queries. It is presented in [13] as “SPARQL 1.1 Federation Extensions”. It presents some features that allow us to merge data distributed across the web. In particular, a SERVICE feature enables expression of the merging queries.
It allows to direct a portion of a query to a particular SPARQL query service, like a `GRAPH` which directs queries to particular named graphs. It is often referred to by the informal term *SPARQL endpoint* (services that accept SPARQL queries and return results).

The mechanics of executing a query over a graph differ from those of querying a service. Typically, a `GRAPH` pattern is matched against an RDF graph which is in the querying system. However the `SERVICE` pattern is matched against a SPARQL endpoint which is not necessarily in the querying system.

The figure 2.2 illustrates an example of SPARQL query with preferences service `ps:preferences`.

---

**Figure 2.1: SPARQL Query with Graph clause**

```
PREFIX pg:<http://orbac.org/graph/>
PREFIX dt:<http://hospital.fr/patients/>
SELECT ...
FROM dt:drugs
FROM dt:infos
FROM NAMED pg:preferences
WHERE {
  A ...
  GRAPH pg:preferences {
    P ...
  }
}
```

**Figure 2.2: SPARQL Federated Query**

```
PREFIX ps:<http://orbac.org/endpoint/>
SELECT ...
WHERE {
  A ...
  SERVICE ps:preferences {
    P ...
  }
}
```
2.2. SEMANTIC MEDIATION

2.1.4 SPARQL/Update

There are also some recent proposals to extend SPARQL to specify queries for updating RDF documents. SPARUL, also called SPARQL/Update [14], is an extension to SPARQL. It provides the ability to insert, update, and delete RDF triples. The update principle presented by this extension is to delete concerned triples and then insert new ones. For example the following query illustrates an update of the graph ‘http://swid.fr/employees’ to rename all employees with the name ‘Safa’ to ‘Nora’.

```
PREFIX emp:<http://swid.fr/emp/0.1/>
WITH <http://swid.fr/employees>
DELETE { ?emp emp:name 'Safa' }
INSERT { ?emp emp:name 'Nora' }
WHERE {
?emp rdf:type emp:Employee.
?emp emp:name 'Safa'
}
```

The ‘WITH’ clause defines the graph that will be modified. ‘DELETE’ defines triples to be deleted. ‘INSERT’ defines triples to be inserted. Finally, ‘WHERE’ defines the quantification portion.

2.2 Semantic Mediation

A mediator has been introduced since 1992 by Wiederhold in [15]. Wiederhold defines a mediator as “a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications”.

A mediator refers the problem of combining data residing at autonomous and heterogeneous sources, and providing users with a unified global schema (master schema). It maintains a global schema and mappings between the global and source schemas.

The mediated query answering process can be divided into two phases [16]:

**Request phase**

- (a) A client $C$ sends a global query $Q$ to a mediator $M$.
- (b) The mediator $M$ decomposes the query $Q$ into a set of subqueries $Q_S$, where the subquery $Q_S$ is supposed to be appropriate for some source $S$.
- (c) The mediator sends the subquery $Q_S$ to the source $S$, for each relevant source $S$. 
CHAPTER 2. PRELIMINARIES AND STATE OF THE ART

Delivery phase

- (d) Each relevant source $S$ evaluates its subquery $Q_S$ and produces a sub-answer consisting of data $D_S$.

- (e) Each relevant source $S$ sends its sub-answer $D_S$ back to the mediator $M$.

- (f) The mediator $M$ integrates the received sub-answers $D_S$ into a global answer $D$.

- (g) The mediator sends back the global answer $D$ following the directions given by the return information.

Currently, there are two main basic approaches of mediators: Global-as-view (GAV) \[1, 17, 18, 19\] and Local-as-View (LAV) \[17, 19, 20, 21, 22\]. The two approaches differ in terms of how the mappings between the global schema and the schema of original sources are defined. In the GAV integration, the relations in the global schema are defined in terms of the relations in the source schemas. In the LAV approach, (the definitions go the other way) the relations in the source schema are defined in terms of the relations in the global schema. The LAV approach has all of the benefits of the GAV approach but makes things easier for database administrators \[23\].

The semantic mediation is the mediation that is capable of taking advantage of semantics \[24\]. It addresses not only the structure of the architecture of the data integration, but how to resolve semantic conflicts between heterogeneous data sources. For example, in the case of web services, identical services could use different vocabulary on their descriptions, and vice-versa, i.e. different web services can use same vocabulary for different meaning. The semantic mediation is used to resolve these heterogeneities of meaning between the vocabularies used for describing Web services.

Table 2.1 shows an example of mediators and their used query language.

2.3 Security enforcement for mediators

As the distribution and sharing of information over the World Wide Web becomes increasingly important, the needs for efficient yet secure access of data naturally arise. Unfortunately, several approaches of mediation that have been proposed do not take into account the security issues and privacy requirements. Moreover, they suppose that the security and privacy issues are only handled locally at each data source level.

Considerable research efforts have been made to handle the issues related to access control in a mediated environment \[16, 41, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55\].
Several proposed approaches aim to secure the mediation protocol mainly aspects of authentication and communication between clients and datasources [16, 52, 53, 54]. For instance [54] aims to preserve the anonymity of clients and confidentiality of data when transmitting data from datasources to clients via a mediator. Authors of [54] present three approaches that allow a mediator to compute a JOIN operation on encrypted relations, based on specific encryption schema (Database-As-Service, commutative encryption and homomorphic encryption/private matching).

However, there are few works that investigate the aspect related to the security query evaluation. TIHI [56] (Trusted Interoperation of Healthcare Information) establishes security by having a single security mediator that applies rules based pre-processing (processing of query) and post-processing (processing of result) data filtering to external client requests. The security mediator is responsible for the security in the mediation system. It insures that unauthorized data does not exit the mediator system. It filters incoming and out-going data from TIHI’s data sources. The security mediator is under the control of the security officer who may override the security me-

<table>
<thead>
<tr>
<th>System</th>
<th>Query Language</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humboldt Discoverer</td>
<td>SPARQL</td>
<td>[25]</td>
</tr>
<tr>
<td>Semantic Agreement</td>
<td>SPARQL</td>
<td>[26]</td>
</tr>
<tr>
<td>AGGREGO SERVER</td>
<td>SPARQL</td>
<td>[27]</td>
</tr>
<tr>
<td>TSIMMIS</td>
<td>LOREL</td>
<td>[28]</td>
</tr>
<tr>
<td>SQPeer</td>
<td>RQL</td>
<td>[29]</td>
</tr>
<tr>
<td>Edutella</td>
<td>RDF-QEL</td>
<td>[30]</td>
</tr>
<tr>
<td>Bibster</td>
<td>SeRQL</td>
<td>[31]</td>
</tr>
<tr>
<td>RDFPeers</td>
<td>RDQL</td>
<td>[32]</td>
</tr>
<tr>
<td>GridVine</td>
<td>RDQL</td>
<td>[33]</td>
</tr>
<tr>
<td>PEPSINT</td>
<td>c-XQuery/c-RDQL</td>
<td>[34]</td>
</tr>
<tr>
<td>Piazza</td>
<td>Subset of XQuery</td>
<td>[35]</td>
</tr>
<tr>
<td>XPeer</td>
<td>Subset of XQuery</td>
<td>[36]</td>
</tr>
<tr>
<td>APPA</td>
<td>XQuery</td>
<td>[37]</td>
</tr>
<tr>
<td>PeerDB</td>
<td>SQL</td>
<td>[38]</td>
</tr>
<tr>
<td>AmbientDB</td>
<td>SQL</td>
<td>[39]</td>
</tr>
<tr>
<td>QueryFlow</td>
<td>SQL</td>
<td>[40]</td>
</tr>
<tr>
<td>Hyperion</td>
<td>SQL</td>
<td>[41]</td>
</tr>
<tr>
<td>FREDdies</td>
<td>SQL</td>
<td>[42]</td>
</tr>
<tr>
<td>SwAP</td>
<td>SQL</td>
<td>[43]</td>
</tr>
</tbody>
</table>

Table 2.1: Example of mediators
CHAOS [51] (Configurable Heterogeneous Active Object System) tries to address the limitation of TIHI by integrating objects that are dynamically loaded into a mediator. CHAOS incorporates the security policies into the data objects as active nodes to form active objects. In this case, the active objects are represented by a special type of XML objects that encompass data elements as well as active elements. An active object contains one active node by mean of a Java class that must be interpreted by a runtime environment. When a query is passed to one of the active objects, its active node is dynamically loaded and executed by the security mediator. CHAOS does not need to rely on a set of primitive security rules. To establish security policies CHAOS provides an API to set the security policies with an active object. The CHAOS model moves the responsibility of security to the source data provider, rather than through a central authority.

Yang et al. [46, 47] proposed an approach based on RBAC [57] by utilizing the mediation system’s Access Control and View Expander components and the mediation spec database. Requests that are sent by the external clients are processed by the Access Control unit that utilizes RBAC authorization. The View Expander enforces proper client access right to insure that only proper database views are made available to the external client [58].

2.4 Security of query evaluation

In this section we made an attempt to identify necessary building blocks for securing mediators, mainly the aspect of security query evaluation. Then we present some existing approaches that are based on these blocks and that could be used and/or adapted for the security of mediation system.

Current mediators rely two main entities: (i) a user query and (ii) data sources. For securing mediators we need a new entity that represents the security policy.

Secure mediator consists of three building blocks:

- **Data** $D$ indicates data sources that contain the answers which users are looking for.

- **Query** $Q$ describes the information that users want. It is expressed in the query language of associated mediator.
2.4. SECURITY OF QUERY EVALUATION

- **Security policy** $P$ represents the access control rules describing the security policy associated to each user. It represents also the privacy policy preferences in case of private data sources.

Note that $D$, $Q$ and $P$ are independent components, and thus can be located independently and processed separately. Based on these elements we present different ways to enforce the security policy, namely, view-based approach, pre-processing, post-processing.

### 2.4.1 View-based approach

View based approach aims to process security policy $P$ and data $D$ first (see figure [2.3]). The idea of view-based enforcement is to create and maintain a *view* for each user who is authorized to access a specific portion/block of data. The view is generated by using the set of authorizations granted to the user to filter off the nodes that the user should not access. The view contains exactly the set of data that the user is authorized to access. During runtime, each user can simply run his queries against his view. Although views can be prepared offline. View-based enforcement has two serious limitations: (1) not scalable in managing and maintaining views when there are a large number of roles (or users), (2) high storage cost [59].
The examples of view-based approaches recently proposed for XML documents include [60, 61, 62, 63, 64, 65]. Depending on the details of the algorithms, the views can be maintained either physically or virtually.

In the case of XML, view-based approaches identify accessible XML nodes for each user (role) to create a view and evaluate user queries on the view. Such approaches provide fast access to the authorized data, especially when views are materialized, but need to deal with view maintenance issues. Most proposals are actually based on view materialization. In this case, for each user, the base of XML documents is transformed to extract the sub-part called the authorized view which is compliant with the access control policy. The query is then evaluated on the authorized view without modification. Unfortunately, it is generally considered that the view materialization process creates an intolerable overhead with respect to performance.

### 2.4.2 Pre-processing approach

This approach aims to handle a user query $Q$ and security policy $P$ prior to data $D$ as illustrated in figure 2.4. It consists in rewriting $Q$ based on the corresponding $P$ to construct a secure query $Q'$ which will be executed directly over $D$. The returned result of $Q'$ is conform to the security policy $P$.

Pre-processing approaches are also known in the literature as query rewriting, query modification and query transformation. An interesting approach for relational DBMS based on query transformation was suggested by Stonebraker [66]. In this case, the query transformation is specified by adding conditions to the WHERE clause of the original query. [66] assumes that a similar mechanism would apply to both select and update operators, which is not generally true (see chapter 4 for more detail). He does not handle the problem of consistency between data selection and update.

An interesting variant to transform SQL queries was suggested by Oracle with VPD [67] (Virtual Private Database) mechanism. In this case, the security policy is specified through the definition of predicates in PL-SQL that will apply as filters to transform the query. The approach suggested by Oracle requires to know PL-SQL in order to implement the access control policy. This may lead to security policies complex to define and maintain. VPD supports the fine-grained access control (FGAC) through policy functions. When a query is issued, it is dynamically modified by appending predicates, returned by the policy function, to the where clause of the query. Moreover, the policy functions are stored and managed locally.
A more recent approach was proposed by Wang et al. [68] where the objective is to securely maximize the answer provided to the user. [68] proposed a formal notion of correctness for fine-grained access control in relational databases. They presented three correctness criteria (sound, secure and maximum) that should be satisfied by any query processing algorithm in order to be “correct”.

**Soundness**: An algorithm is sound if and only if its rewritten query \( Q_{rw} \) returns only correct answers i.e. answers to the initial query.

**Maximality**: An algorithm is maximum if and only if \( Q_{rw} \) returns as much information as possible.

**Security**: An algorithm is secure if and only if the result of \( Q_{rw} \) respects the security and privacy policy of the queried system.

LeFevre et al. [69] presented an approach that enforces limited disclosure expressed by privacy policy in the case of Hippocratic databases. It is implemented by rewriting queries. When a query \( Q \) is issued, it is transformed to \( Q' \) so that the result of \( Q' \)
respects the cell-level disclosure policy \( P \). Their approach is based on replacing all the cells that are not allowed to be seen by \( P \) with NULL. After that, \( Q' \) is evaluated as a normal query with evaluation rules “NULL ≠ NULL” and “NULL ≠ cst” for any constant value cst. [69] does not take into account the privacy requirement accuracy. They only replace unauthorized data with “NULL”. Moreover, [69] does not satisfy the sound property and maximum property [68]. Another issue is that they only mask variables (fields) used in the header of the SQL query. They do not handle the qualification portion. For instance, for the following query \( \text{SELECT name FROM Patients WHERE age=25} \), they only mask unauthorized name. However, if a data-owner Alice, who is 25 years old, choses to disclose her name but not her age, anyone looking at the results concludes that Alice is 25 years old. One way to correct this problem is to normalize the SQL query by adding all variables used in the where clause to the corresponding header of the query, e.g. the query above is normalized as follows: \( \text{SELECT name, age FROM Patients WHERE age=25} \). Then we rewrite the normalized query and applies filter after transformation so that Alice will not appear in the result.

In the case of XML documents, more recent proposals suggest using query transformation, see for instance [70] that shows how to transform XPath queries and QFilter [71] which is based on query rewriting. [71] is based on non-deterministic finite automata (NFA) and rewrites user’s queries such that parts violating access control rules are pre-pruned. However, there is a main difference compared to RDF and SPARQL: XML documents correspond to oriented graphs. As noticed in [65], this may lead to complication to protect some relationships in an XML document. This issue has been addressed using two different approaches: In [70], protection of XML relationships is embedded in document transformation whereas [65] suggests specifying access control policies using the concept of blocks in order to break some relationships that must be protected.

### 2.4.3 Post-processing approach

Figures [2.5(a) and (c)] illustrate the post-processing scenario, where \( Q \) is applied to \( D \) first. No security enforcement is engaged. Then the security policy is examined second. \( Q \) is processed by DBMS to produce unsafe answers, which goes through post-filtering process to prune out the parts that violate security policy \( P \) and return only safe parts. Checking the security after the query evaluation may disclose some confidential information. For instance, a malicious user could issue a query that returns the name of patients who are 25 years old. The returned result corresponds exactly to patients
2.4. SECURITY OF QUERY EVALUATION

who are 25 years old even if the value of the age is hidden by the security policy. It is necessary to keep additional information related to the query, and take it into account at security filtering stage.

Post-processing approach is useful when security policy itself carries security conscious information and has to be stored securely. For instance, Figure 2.5(b) shows a use case of post-processing approach, where \( Q \) and \( D \) are stored together and security policy \( P \) is stored elsewhere. Note that in this case \( Q \) is not known from users. So the problem of disclosing confidential information mentioned before, does not occur in this case.

An interesting approach based on post-processing principle, to enforce access control rules in the case of XML document, is presented in [72]. Authors of [72] present an approach that extends regular query processing by going through a “post-filtering” stage, named as \( AFilter \), to filter out un-safe answers. [72] adopts \( YFilter \) [73], a query processor for streaming XML data, as an implementation of \( AFilter \).
3.1 Introduction

SPARQL has been defined to easily locate and extract data in an RDF graph. It is also used by several semantic mediators as a high level query language to express a user’s query. SPARQL queries must be filtered so that only authorized data are returned with respect to some confidentiality policy. In this chapter we propose an approach that enforces the data confidentiality (access control constraints) in the case of SPARQL queries. We model a confidentiality policy as a set of positive and negative filters (corresponding respectively to permissions and prohibitions) that apply to SPARQL queries. We then define rewriting algorithms that transform the queries so that the results returned by transformed queries are compliant with the confidentiality policy.

Basically, the SPARQL syntax is similar to SQL, but the advantage of SPARQL is that it enables queries spanning multiple disparate (local or remote) data sources containing heterogeneous semi-structured data. However, since a SPARQL query may access confidential data, it is necessary to design security mechanisms to control the evaluation of SPARQL queries and prevent these queries from illegally disclosing confidential data.

The approach is to rewrite the user SPARQL query by adding some SPARQL filters to that query. When, the user sends his or her SPARQL query to the server, our system will intercept this query and checks the security rules corresponding to that user (Figure 3.1). Then it rewrites the query by adding the corresponding SPARQL filters. The execution result of the rewritten query is returned to the user. The figure 3.1 illustrates our approach called fQuery [74].
In our approach, the answer to the rewritten query may differ from the user’s initial query. In that case and as suggested in [75], we can check the query validity of the rewritten query with respect to the initial query and notify the user when the query validity is not guaranteed.

This chapter is organized as follows. Section 3.2 presents the basic principles of rewriting SPARQL query and illustrate these principles through some examples. Section 3.3 presents some definitions and theorems that are used in other sections. Section 3.4 defines the security policy model for SPARQL and some of its properties. In section 3.5 we specify the rewriting query algorithm and finally section 3.6 concludes this chapter.

3.2 Rewriting SPARQL Query: Basic Principles

Let us take an example of query transformation. We assume that the user Bob tries to select the name and the salary of each employee. We assume also that Bob is not permitted to see salaries of employees who earn more than 60K. The table 3.1 shows Bob’s SPARQL query before and after transformation. The presence of the OPTIONAL construct in the transformed query makes it a non-conjunctive (disjunctive) one. It means
3.3. NOTATIONS, DEFINITIONS AND THEOREMS

Before transformation

```triples
SELECT ?name ?salary
WHERE
{
  ?employee rdf:type emp:Employee.
  ?employee foaf:name ?name.
}
```

After transformation

```triples
SELECT ?name ?salary
WHERE
{
  ?employee rdf:type emp:Employee.
  ?employee foaf:name ?name.
  Optional {
    Filter (?salary <60000)
  }
}
```

Table 3.1: Example of query transformation

that: if the condition inside the OPTIONAL clause is False then the value of the salary variable is assigned to Null.

The access control policy is based on filter definitions. To each user or group of users, we assign a set of filters. Depending on the policy type, we consider two different types of filter: (1) Positive filters corresponding to permissions and (2) Negative filters corresponding to prohibitions.

These filters may be associated with a simple condition or an involved condition. Filters associated with involved condition provides means to protect relationships. In our approach we assume that when a user asks a query, we can get additional information like the user identity. This additional information may be used in the filter definition (see the example [10] page 34).

Filters actually provide a generic approach to represent an access control policy for RDF documents which does not rely on a specific language. However, it would be also interesting to define a user friendly specification language to express an access control policy for RDF documents (see chapter 5).

3.3 Notations, Definitions and Theorems

As mentioned in the introduction, an RDF database is represented by a set of triples. So, we denote $E$ the set of all RDF triples of our database. We denote $E_{subject}$ (respectively $E_{predicate}, E_{object}$) the projection of $E$ on subject (resp. predicate and object). $E_{subject}$ represents (resp. $E_{predicate}, E_{object}$) the set of all subjects (resp. predicates, objects) of the RDF triples of $E$. 
Definition 1: RDF condition
We define a “condition of RDF triples” as the application \( \omega : E \rightarrow \text{Boolean} \) which assigns each RDF triple \( x = (s,p,o) \) of \( E \) to an element of the set \( \text{Boolean} = \{\text{True}, \text{False}\} \).

\[
\omega : E \rightarrow \text{Boolean} \\
x \rightarrow \omega(x)
\]

\( \omega(x) \) is expressed in terms of \( s, p \) and \( o \) where \( x = (s,p,o) \). We define also the negation of \( \omega \) denoted \( \bar{\omega} \) as follows:

\[
\bar{\omega} : E \rightarrow \text{Boolean} \\
x \rightarrow \bar{\omega}(x)
\]
such that \( (\forall x \in E)\bar{\omega}(x) = \overline{\omega(x)} = \neg(\omega(x)) \)

For each element \( x \) of \( E \), we say that \( \omega(x) \) is satisfied if \( \omega(x) = \text{True} \). Otherwise we say that \( \omega(x) \) is not satisfied.

Definition 2: simple condition
We define the “simple condition of RDF triples” as the condition of RDF triples that uses the same operators as the SPARQL filter (regex, bound, =, <, > ...) and constants (see [11] for a complete list of possible operators).

Example 1 \((\forall x = (s,p,o) \in E)\)
\[\omega(x) = (s \neq \text{emp:Alice}) \lor ((p = \text{foaf:name}) \land (o \neq 'Alice'))\]

Definition 3: involved condition
Let \( n \in \mathbb{N}^* \), \( \{p_i\}_{1 \leq i \leq n} \) be a set of predicates of \( E_{\text{predicat}} \) and \( \{\omega_i\}_{1 \leq i \leq n} \) be a set of simple conditions. Let \( x \) be an element of \( E \) where \( x = (s,p,o) \). The condition expressing that \( s \) (subject of \( x \)) must have the properties \( \{p_i\}_{1 \leq i \leq n} \) such that the value of each property \( p_i \) satisfies the condition \( \omega_i \), is called an involved condition associated with \( \{(p_i, \omega_i)\}_{1 \leq i \leq n} \). This condition, denoted \( \omega \), could be expressed as follows:

\[
(\forall x = (s,p,o) \in E) \\
\omega(x) = \begin{cases} 
\text{True} & \text{if } (\exists(x_1, ..., x_n) \in E^n)/(\forall 1 \leq i \leq n)x_i = (s, p_i, o_i) \\
\text{where } o_i \in E_{\text{object}} \text{ and } \omega_i(o_i) = \text{True} \\
\text{False} & \text{Otherwise}
\end{cases}
\]

\( \omega(x) \) does not depend only on the RDF triple \( x \) but it also depends on other RDF triples \( (x_1, ..., x_n) \) that share the same subject of \( x \) and satisfy respectively the simple
conditions \((\omega_1, \ldots, \omega_n)\). The involved condition for an element \(x\) of \(E\) is the existence of other properties \(\{p_i\}_{1 \leq i \leq n}\) (predicates) of the subject of \(x\) and the value of each property \(p_i\) satisfies the simple condition \(\omega_i\).

Let \(x = (s, p, o)\) be an element of \(E\). In the case of a simple condition \(\omega_{\text{simple}}\), we only need the \(s\), \(p\) and \(o\) to evaluate \(\omega_{\text{simple}}(x)\). But in the case of an involved condition \(\omega_{\text{involved}}\) associated with \(\{(p_i, \omega_i)\}_{1 \leq i \leq n}\), the value of \(x\) is not sufficient to evaluate \(\omega_{\text{involved}}(x)\). It requires knowledge about other elements of \(E\).

**Example 2**  
Bob is permitted to select the information of the network department employees. The condition \(\omega\) associated with this rule could be expressed as follows:

\[
(\forall x = (s, p, o) \in E) \omega(x) = \begin{cases} 
\text{True} & \text{if } (\exists y \in E) | y = (s, \text{emp:dept}, '\text{Network}') \\
\text{False} & \text{Otherwise}
\end{cases}
\]

It means that Bob can select only the RDF triples where the subject has also the predicate \text{emp:dept} with the value ‘Network’. \(\omega(x)\) does not depend only on the RDF triple \(x\) but it also depends on another RDF triple \(y\) that shares the same subject of \(x\) and its predicate is \text{emp:dept} with the value ‘Network’.

An involved condition can be expressed as a SPARQL ‘Basic Graph Pattern’ (BGP) \([11]\) (a set of triple patterns) where all its triple patterns have the same subject \(s\).

\[
\text{BGP} = \{ \text{\L}_i=1^n (tp_i, \text{Filter}(\omega_i)) \} \text{ such that } \forall i \in [0, n], \text{tp}_i = (s, p_i, o_i).
\]

**Example 3**  
The following condition means that it is not allowed to see the data of employees of network and security department who are more than 30 years old.

```sparql
\text{NOT EXISTS} \{ 
\text{\?s rdf:type O:Employee;}
\text{O:dept ?dep;}
\text{O:age ?age.}
\text{FILTER(?age >30)}
\text{FILTER(?dep IN ('Network', 'Security'))}
\}
```

**Definition 4: complex condition**

Let \(n\) be an integer, \(\omega\) an RDF condition, \(\{p_i\}_{1 \leq i \leq n}\) be a set of predicates and \(\{\omega_i\}_{1 \leq i \leq n}\) be a set of RDF conditions. The condition \(\omega\) is a complex condition associated with \(\{(p_i, \omega_i)\}_{1 \leq i \leq n}\) with level \(L \geq 1\) if and only if \(\{\omega_i\}_{1 \leq i \leq n}\) are complex conditions such that \(\text{max}(\text{level}(\omega_i)) = L - 1\).
By definition a simple condition is a complex condition with level \( L = 0 \). So an involved condition is a complex condition with level \( L = 1 \).

A complex condition could be also expressed as a SPARQL Group pattern (GP) \[11\].

**Example 4** Doctor is allowed to see only the information of its own patients.

```
GP={?s rdf:type O:Patient;
    O:mydoc ?doc.
?doc rdf:type O:Doctor;
    O:hasId ?id.
FILTER(?id=doctor_id)
}
```

**Definition 5: projection**

Let \( tp \) be a triple pattern of the where clause of a SPARQL query and \( \omega \) be a condition on RDF triples. We define the projection of \( \omega \) relative to \( tp \) as the condition \( \omega(tp) \) expressed in terms of the \( tp \) SPARQL variables. We denote that projection as \( \pi_{\omega/tp}, \pi_{\omega/tp} = \omega(tp) \).

**Example 5** Let \( x = (s, p, o) \in E \) such that

\[
\omega(x) = (s \neq \text{emp:Alice}) \land (p = \text{foaf:name}) \land (s \neq o)
\]

and \( tp = (\text{emp:Charlie}, ?m, ?n) \)

\[
\pi_{\omega/tp}(?) = (?m = \text{foaf:name}) \land (?n \neq \text{emp:Charlie})
\]

We denote constants of conditions of RDF triple \( \Omega_{True} \) and \( \Omega_{False} \) applications defined as follows:

\[
\Omega_{True} : E \rightarrow Boolean \quad \Omega_{False} : E \rightarrow Boolean \\
x \rightarrow True \quad x \rightarrow False
\]

**Definition 6: conjunction and disjunction**

Let \( \omega_1 \) and \( \omega_2 \) be two conditions on RDF triples. We define the conditions \( \omega_1 \land \omega_2 \) and \( \omega_1 \lor \omega_2 \) as follows:

\[
\omega_1 \land \omega_2 : E \rightarrow Boolean \quad \omega_1 \lor \omega_2 : E \rightarrow Boolean \\
x \rightarrow \omega_1(x) \land \omega_2(x) \quad x \rightarrow \omega_1(x) \lor \omega_2(x)
\]
Definition 7: set $I(\omega)$
Let $\omega$ be a condition on RDF triples. We define the subset of $E$ that satisfies the condition $\omega$, denoted $I(\omega)$, as follows:

$$I(\omega) = \{ x \in E | \omega(x) = True \}$$

We define the complement of the set $I(\omega)$ in $E$, denoted $\overline{I(\omega)}$, as follows:

$$\overline{I(\omega)} = \{ x \in E | x \notin I(\omega) \} = E \setminus I(\omega)$$

Theorem 1: Let $\omega$ be a condition on RDF triples, $I(\overline{\omega}) = I(\omega) = E \setminus I(\omega)$

Proof of theorem 1:

$$x \in I(\overline{\omega}) \iff \{ x \in E | \overline{\omega}(x) = True \}$$

$$\iff \{ x \in E | \overline{\omega}(x) = True \}$$

$$\iff \{ x \in E | \omega(x) = False \}$$

$$\iff \{ x \in E | x \notin I(\omega) \}$$

$$\iff x \in I(\overline{\omega}).$$

Theorem 2: Let $\omega_1$ and $\omega_2$ be two conditions on RDF triples. We have the following properties: $I(\omega_1 \land \omega_2) = I(\omega_1) \cap I(\omega_2)$ and $I(\omega_1 \lor \omega_2) = I(\omega_1) \cup I(\omega_2)$

Proof of theorem 2:

$$x \in I(\omega_1 \land \omega_2) \iff \omega_1(x) \land \omega_2(x) = True$$

$$\iff \omega_1(x) = True \text{ and } \omega_2(x) = True$$

$$\iff x \in I(\omega_1) \text{ and } x \in I(\omega_2)$$

$$\iff x \in I(\omega_1) \cap I(\omega_2).$$

Then $I(\omega_1 \land \omega_2) = I(\omega_1) \cap I(\omega_2)$

With the same reasoning we can prove that $I(\omega_1 \lor \omega_2) = I(\omega_1) \cup I(\omega_2)$.

By induction (recurrence) we can prove the properties bellow.

Generalization of theorem 2: Let $n \in \mathbb{N}^*$ and $\{\omega_i\}_{0 \leq i \leq n}$ be a set of conditions on RDF triples.

$$I(\land_{i=0}^{n} \omega_i) = \bigcap_{i=0}^{n} I(\omega_i)$$

$$I(\lor_{i=0}^{n} \omega_i) = \bigcup_{i=0}^{n} I(\omega_i)$$

Definition 8: Abstract property
Let $G$ be an RDF graph. Let $A$ and $B$ be two elements of $G$ such that there exist an
indirect relation $R$ (a path of length greater than or equal to 2) between $A$ and $B$. We define the “abstract property” $P_{A,B}^R$ as a predicate which is not part of predicates of $G$ and which represents the relation $R$.

The relation $R$ is called the mapping associated with the abstract property $P_{A,B}^R$.

**Example 6** Let us take an example. We use the prefix ‘o’ to nominate a concrete element and ‘ap’ to nominate abstract properties. Let us take the following group of patterns:

$$\begin{align*}
\{ & \text{?s rdf:type o:Patient.} \\
& \text{?s o:physician ?doc.} \\
& \text{?doc o:hasName ?docName.} \\
\}
\end{align*}$$

This group of patterns could be represented by the following group of patterns:

$$\begin{align*}
\{ & \text{?s rdf:type o:Patient.} \\
& \text{?s ap:myDocName ?docName.} \\
\}
\end{align*}$$

such that $ap:myDocName$ is the abstract property represented by the following mapping between $\text{?s}$ and $\text{?docName}$:

$$\begin{align*}
\{ & \text{?s ap:myDocName ?docName} \} \iff \{ & \text{?s o:physician ?doc.} \\
& \text{?doc o:hasName ?docName.} \\
\}
\end{align*}$$

Let us take another example. We consider the following group of patterns:

$$\begin{align*}
\{ & \text{?s rdf:type o:Patient.} \\
& \text{?s o:name ?name.} \\
& \text{?pref rdf:type o:Preference.} \\
& \text{?pref o:dataOwner ?do.} \\
& \text{?do o:hasName ?name.} \\
\}
\end{align*}$$

This group of patterns could be represented by the following group of patterns:

$$\begin{align*}
\{ & \text{?s rdf:type o:Patient.} \\
& \text{?s ap:hasPreference ?pref.} \\
\}
\end{align*}$$

such that $ap:hasPreference$ is the abstract property represented by the following mapping between $\text{?s}$ and $\text{?pref}$:
3.4 SECURITY POLICY

| {             | }                                                   |
| { s ap:hasPreference ?pref } ⇐⇒                   |
| s o:name ?name.                                  |
| pref rdf:type o:Preference.                      |
| pref o:dataOwner ?do.                           |
| do o:hasName ?name.                              |

Theorem 3:
A complex condition $\omega$ with level $l \geq 2$ could be represented by an involved condition using abstract properties.

Proof of theorem 3: see appendix B.

3.4 Security policy

In this section we show how to factorize a set of permission rules (resp. prohibition rules), assigned to a given user, as a single permission rule (resp. prohibition rule). In our proposal we define the security policy as a set of permissions or a set of prohibitions. We also assume that the policy is closed.

3.4.1 Permission

A security policy rule is defined as the permission for a user to select a set of RDF triples of $E$ that satisfies a condition on RDF triples denoted $\omega$. It means that the user is permitted to select only the RDF triples of the subset $I(\omega)$. We denote this permission as $\text{Permission}(\omega)$.

Example 7  Bob is permitted to see the name and email of all employees stored in the RDF database. This rule can be expressed as the permission to select a set of RDF triples of $E$ that satisfies the condition $\omega$ defined as follows:

$$(\forall x = (s, p, o) \in E) \quad \omega(x) = \begin{cases} 
\text{True} & \text{if } p \in P \\
\text{False} & \text{if } p \notin P 
\end{cases}$$

Such that $P = \{\text{foaf:name, foaf:mbox}\}$ is a set of predicates associated with the information name and email.

Let $\{\text{Rule}_i\}_{1 \leq i \leq n}$ be a set of security rules (permission rules) associated with a user and $\{\omega_i\}_{1 \leq i \leq n}$ be a set of conditions on RDF triples such that $n \in \mathbb{N}^*$ and
Rule\(_i = Permission(\omega_i)\). So the user could select the RDF triples of each set \(I(\omega_i)\).

It means that the user could select the RDF triples of the set \(\bigcup_{i=1}^{n} I(\omega_i)\). According to the result of the theorem 2, the user is permitted to select the RDF triples of \(I(\bigvee_{i=1}^{n} \omega_i)\).

So the user is permitted to select RDF triples that satisfies the condition \(\omega = \bigvee_{i=1}^{n} \omega_i\).

We deduce that:

\[
\bigcup_{i=1}^{n} Permission(\omega_i) = Permission(\bigvee_{i=1}^{n} \omega_i)
\]

It means that a set of permission rules \(\{Permission(\omega_i)\}_{1 \leq i \leq n}\) could be expressed as one permission rule defined as the permission to select RDF triples that satisfies the condition \(\omega = \bigvee_{i=1}^{n} \omega_i\).

### 3.4.2 Prohibition

In the case of prohibition we define the security policy rule as the prohibition for a user to select a set of RDF triples of \(E\) that satisfies a condition on RDF triples denoted \(\omega\).

It means that the user is prohibited to select any RDF triples of the subset \(I(\omega)\). We denote this prohibition as \(Prohibition(\omega)\).

**Example 8** Bob is not permitted to select the salary and the birth day of all employees stored in a RDF database. This rule can be expressed as \(Prohibition(\omega)\) such that \(\omega\) is defined as follows:

\[
(\forall x = (s, p, o) \in E) \quad \omega(x) = \begin{cases} 
    True & \text{if } p \in P \\
    False & \text{if } p \notin P
  \end{cases}
\]

Such that \(P = \{\text{emp:salary, foaf:birthday}\}\) is a set of the predicates associated with the information salary and birth day. So \(\omega\) could be written as:

\[
(\forall x = (s, p, o) \in E) \quad \omega(x) = (p = \text{emp:salary}) \lor (p = \text{foaf:birthday})
\]

With the same reasoning as in the previous section \[3.4.1\] we deduce that:

\[
\bigcup_{i=1}^{n} Prohibition(\omega_i) = Prohibition(\bigvee_{i=1}^{n} \omega_i)
\]

Assuming that \(\{Prohibition(\omega_i)\}_{1 \leq i \leq n}\) are all security rules associated with a user, we can prove the following result:

\[
\bigcup_{i=1}^{n} Prohibition(\omega_i) = Permission(\bigwedge_{i=1}^{n} \omega_i)
\]
3.5 \( f \text{Query}: \) Our query rewriting model

We rewrite the user query by adding filters and/or removing triples of pattern from the where clause following the associated security policy (see section 3.5.1). Sometimes it is also necessary to add triples of pattern to the query in order to handle involved condition (see section 3.5.2).

Our query rewriting algorithm treats each BGP [11] (Basic Graph Pattern) of a SPARQL query. Each BGP is handled separately from the others.

### 3.5.1 Case of simple condition \( \omega \)

Let \( Bgp \) be a basic graph pattern of the where clause of a SPARQL query. We check the security rule associated with the condition \( \omega \) (\( \text{Permission}(\omega) \) or \( \text{Prohibition}(\omega) \)) for each triple pattern \( tp = (s, p, o) \) of \( Bgp \) by calculating the projection \( \pi_{\omega/tp} \). There are three cases depending on the \( \pi_{\omega/tp} \) value.

**Permission case**

- \( \pi_{\omega/tp} = \Omega_{True} \)
  
  It means that \( \pi_{\omega/tp} \) is always true for each SPARQL variable of the triple pattern \( tp \). In this case the triple pattern \( tp \) matches the security policy. So there is no action to do for \( tp \). We check the security condition \( \omega \) for the next triple pattern.

- \( \pi_{\omega/tp} = \Omega_{False} \)
  
  It means that \( \pi_{\omega/tp} \) is always false for each SPARQL variable of the triple pattern \( tp \). In this case the triple pattern \( tp \) does not match the security policy. So we delete this triple pattern \( tp \) from \( Bgp \). Then we check the security condition \( \omega \) for the next triple pattern.

- Otherwise \( \pi_{\omega/tp} \) is expressed in terms of \( tp \) variables. In this case, we put \( tp \) in an OPTIONAL construct and we add the positive filter \( \varphi \) to it. Then we add this optional construct to \( Bgp \). The positive filter \( \varphi \) is defined as follows:

\[
\varphi(tp) = \text{FILTER}(\pi_{\omega/tp}) = \text{FILTER}(\omega(tp))
\]

This filter filters the RDF triples that satisfy the condition \( \omega \). The presence of the OPTIONAL construct in the transformed query makes it a non-conjunctive one.
CHAPTER 3. SPARQL SELECT QUERY REWRITING TO ENFORCE ...

Prohibition case

- \( \pi_{\omega/tp} = \Omega_{\text{True}} \)
  
  It means that \( \pi_{\omega/tp} \) is always \text{true} for each SPARQL variable of the triple pattern \( tp \). In this case, RDF triples that match with the triple pattern \( tp \) are prohibited. So we delete this triple pattern \( tp \) from the basic graph pattern \( Bgp \). Then we check the security condition \( \omega \) for the next triple pattern.

- \( \pi_{\omega/tp} = \Omega_{\text{False}} \)
  
  It means that \( \pi_{\omega/tp} \) is always \text{false} for each SPARQL variable of the triple pattern \( tp \). In this case RDF triples that match with the triple pattern \( tp \) are allowed to be selected. So there is no action to do for \( tp \). We check the security condition \( \omega \) for the next triple pattern.

- Otherwise \( \pi_{\omega/tp} \) is expressed in terms of \( tp \) variables. In this case, we put \( tp \) in an \text{OPTIONAL} construct and we add the filter \( \varphi \) to it. Then we add this optional construct to \( Bgp \). The filter \( \varphi \) is defined as follows:

  \[
  \varphi(tp) = \text{FILTER}(\pi_{\omega/tp}) = \text{FILTER}(\omega(tp))
  \]

  This filter filters the RDF triples that do not satisfy the condition \( \omega \).

We define Algorithm \[ \] the query rewriting algorithm for a simple condition.

Example 9  Bob is not permitted to see salaries of employees who earn more than 50K and their premiums if it is greater than 9K. This prohibition could be expressed as \( \text{Prohibition}(\omega) \) where \( \omega \) is defined as: \( (\forall x = (s,p,o) \in E) \)

\[
\omega(x) = ((p = \text{emp:salary} \land (o \geq 50000)) \lor ((p = \text{emp:premium}) \land (o \geq 9000))
\]

Bob tries to select the name, the salary of each employee and their premium if it is greater than 10K. He executes the following query:

```
SELECT ?name ?salary ?premium
WHERE {
  ?s1 foaf:name ?name,
  emp:salary ?salary.
  Optional{
    ?s1 emp:premium ?premium. Filter(?premium > 10000)
  }
}
```

Let \( tp_1 = (?s1,\text{foaf:name},?\text{name}) \) and \( tp_2 = (?s1,\text{emp:salary},?\text{salary}) \) and \( tp_3 = (?s1,\text{emp:premium},10000) \) be triples of pattern of the where clause of
Algorithm 1 Algo1 (Query, ω, ruleType). Query rewriting Algorithm for a simple condition

**Require:** ω is a simple condition

for each basic graph pattern Bgp of Query do
    for each triple pattern tp of Bgp do
        if \( \pi_{\omega/tp} = \Omega_{True} \) then
            if ruleType = PROHIBITION then
                delete tp from Bgp
            end if
        else if \( \pi_{\omega/tp} = \Omega_{False} \) then
            if ruleType = PERMISSION then
                delete tp from Bgp
            end if
        else
            create new optional element opEl
            move tp to opEl
            if ruleType = PERMISSION then
                add the filter \( \text{FILTER}(\pi_{\omega/tp}) \) to opEl
            else if ruleType = PROHIBITION then
                add the filter \( \text{FILTER}(\pi_{\omega/tp}) \) to opEl
            end if
        end if
    end for
end for

Bob's query. The query has two basic graph patterns \( Bgp_1 = \{tp_1, tp_2\} \) and \( Bgp_2 = \{tp_3\} \).

We have \( \pi_{\omega/tp_1} = \omega(tp_1) = False = \Omega_{False} \)
\( \pi_{\omega/tp_2} = \omega(tp_2) = (?, salary \geq 50000) \)
\( \pi_{\omega/tp_3} = \omega(tp_3) = (?, premium \geq 9000) \)

\( \pi_{\omega/tp_1} = \Omega_{False} \) so there is nothing to do with the triple pattern \( tp_1 \).
\( \pi_{\omega/tp_2} = (?, salary \geq 50000) \) so we add the filter \( \text{FILTER}(?, salary < 50000) = \text{FILTER}(\pi_{\omega/tp_2}) \) to the \( Bgp_1 \).
\( \pi_{\omega/tp_3} = (?, premium \geq 9000) \) so we add the filter \( \text{FILTER}(?, premium < 9000) \) to \( Bgp_2 \). The rewritten query will be as follows:
3.5.2 Case of involved condition $\omega$

In this section, we define Algorithm 2 the query rewriting algorithm for an involved condition.

Let $Bgp$ be a basic graph pattern, $\{s_i\}_{1 \leq i \leq m}$ the set of subjects of the $Bgp$ triple patterns and $\{Gp_i\}_{1 \leq i \leq m}$ a set of group patterns where $Gp_i$ is a set of triple patterns of $Bgp$ which has the same subject $s_i$. There are two cases to consider: Permission($\omega$) and Prohibition($\omega$).

Permission case: We handle each $Gp_i$ separately from the others. For each $1 \leq j \leq n$, the subject $s_i$ should have the property $p_j$ such that its value should satisfy the simple condition $\omega_j$. We verify if there exists a triple pattern $tp = (s, p, o)$ of $Gp_i$ which has the property $p_j$ ($p = p_j$). If this triple exists, then it should satisfy the simple condition $\omega_j$. For this purpose, we add a new SPARQL filter with the condition $\omega_j(tp)$. If there is no triple pattern with the property $p_j$ on $Gp_i$ then we create a new one $tp_{ij} = (s_i, p_j, ?\alpha_j)$ and we add it to $Gp_i$ (where $?\alpha_j$ is a SPARQL variable). $tp_{ij}$ should then satisfy the simple condition $\omega_j$. So we add a new SPARQL filter with the condition $\omega_j(tp_{ij})$.

Prohibition case: In this case we verify for each $1 \leq j \leq n$ if there exists a triple pattern $tp = (s, p, o)$ of $Gp_i$ with the property $p_j$ ($p = p_j$). If this pattern exists, then there are two cases. If its value 'o' is a SPARQL variable then it should not satisfy the condition $\omega_j$ or it should be unbound (i.e. $s_i$ does not have the property $p_j$). In this case we add a new SPARQL filter with the condition $(\omega_j(tp) \lor !\text{bound}(o))$. Otherwise, the value 'o' could not be unbound, then the triple pattern $tp$ should not satisfy $\omega_j$. In this case we add a new filter with the condition $\omega_j(tp)$.
Algorithm 2 (Query, \( \omega \), ruleType). Query rewriting Algorithm for an involved condition

**Require:** \( \omega \) is an involved condition

**for** each basic graph pattern \( Bgp \) of Query **do**

Let \( \{ s_i \}_{1 \leq i \leq m} \) be a set of the subjects of the \( Bgp \) triples pattern

**for** each subject \( s_i \) **do**

Let \( Gp_i \) be a set of triple pattern of \( Bgp \) with the same subject \( s_i \)

\( \{ \omega_j \}_{1 \leq i \leq n} \) a set of simple condition associated with \( \omega \)

\( \{ p_j \}_{1 \leq i \leq n} \) a set of predicates associated with \( \omega \)

**for** \( j = 1 \) to \( n \) **do**

if \( \exists tp = (s, p, o) \in Gp_i \mid p = p_j \) then

**for** each \( tp = (s, p, o) \in Gp_i \mid p = p_j \) **do**

if \( \text{ruleType} = \text{PERMISSION} \) then

add \( \text{FILTER}(\pi_{\omega_j/tp}) \) to \( Bgp \)

else if \( \text{ruleType} = \text{PROHIBITION} \) then

if \( o \) is a SPARQL variable then

add \( \text{FILTER}(\pi_{\omega_j/tp} \lor \text{bound}(o)) \) to \( Bgp \)

else

add \( \text{FILTER}(\pi_{\omega_j/tp}) \) to \( Bgp \)

end if

end if

end for

else

let \( tp_j = (s, p_j, ?\alpha_j) \) be a triple of pattern

add \( tp_j \) to \( Gp_i \)

if \( \text{ruleType} = \text{PERMISSION} \) then

add \( \text{FILTER}(\pi_{\omega_j/tp_j}) \) to \( Bgp \)

else if \( \text{ruleType} = \text{PROHIBITION} \) then

add \( \text{FILTER}(\pi_{\omega_j/tp_j} \lor \text{bound}(?\alpha_j)) \) to \( Bgp \)

end if

end if

end for

end for

end for

Now if there is no triple pattern with the property \( p_j \) on \( Gp_i \), then we create a new one \( tp_{ij} = (s_i, p_j, ?\alpha_j) \) and we add it to \( Gp_i \). So \( tp_{ij} \) should not satisfy the condition \( \omega_j \) or it should be unbound. In this case we add a new SPARQL filter with
CHAPTER 3. SPARQL SELECT QUERY REWRITING TO ENFORCE ...

Before transformation

```sparql
SELECT ?name ?location
WHERE
{
}
```

After transformation

```sparql
SELECT ?name ?location
WHERE
{
  FILTER (?doct=Bob_id)
}
```

Table 3.2: Bob’s query transformation

the condition \( \omega_j(t_{p,j}) \lor \neg\text{bound}(\alpha_j) \). The expression \( \text{bound}(\text{variable}) \) returns true if ‘variable’ is bound to a value. It returns false otherwise [11].

**Example 10**  Bob is a doctor and he can see only the information of his patients. The involved condition assigned (in the case of permission) to doctor role could be expressed as:

\[
(\forall x = (s,p,o) \in E) \omega(x) = \begin{cases} 
  \text{True} & \text{if } (\exists y \in E) | y = (s,\text{pat:doctor},$Bob_id) \\
  \text{False} & \text{Otherwise}
\end{cases}
\]

where $Bob_id$ is the identifier of Bob. Bob tries to select names and locations of all patients. The table 3.2 shows Bob’s query before and after transformation.

### 3.5.3 Case of complex condition \( \omega \)

The query transformation could be done by using the SPARQL filter **EXISTS** and **NOT EXISTS** or by using SPARQL **optional** portion. **EXISTS** and **NOT EXISTS** are not supported by the standard SPARQL 1.0. They are defined in SPARQL1.1 [14]. However **OPTIONAL** operator is supported by the standard SPARQL 1.0.

In this section we presents the rewriting approach for both operators.

Let us take an example to illustrate the approach.

**Example 11**  Doctor Alice is allowed to see the data of her own patients. The complex condition associated with this permission is as follows:
### 3.5. *FQUERY: OUR QUERY REWRITING MODEL*

<table>
<thead>
<tr>
<th>EXISTS operator</th>
<th>OPTIONAL operator</th>
</tr>
</thead>
</table>
| **SELECT** ?name  
**WHERE** {  
?p rdf:type O:Patient;  
O:name ?name;  
O:disease ?d.  
?d O:name 'X'.  
**FILTER EXISTS** {  
?md rdf:type O:Doctor;  
O:hasId ?docId.  
**FILTER** (?docId = Alice_id)  
}  
} | **SELECT** ?name  
**WHERE** {  
?p rdf:type O:Patient;  
O:name ?n;  
O:disease ?d.  
?d O:name 'X'.  
**OPTIONAL** {  
?md rdf:type O:Doctor;  
O:hasId ?docId.  
**FILTER** (?docId = Alice_id)  
}  
**FILTER** (bound(?p) and bound(?md) and bound(?docId))  
} |

Table 3.3: Complex condition : FILTER EXISTS and OPTIONAL operator

```
{  
?s rdf:type O:Patient;  
O:myDoctor ?md.  
?md rdf:type O:Doctor;  
O:hasId ?docId.  
**FILTER** (?docId = Alice_id)  
}
```

We assume that Doctor Alice is trying to select names of patients who have disease ‘X’. Alice’s query before transformation is as follows:

```
**SELECT** ?name  
**WHERE** {  
?p rdf:type O:Patient;  
O:name ?name;  
O:disease ?d.  
?d O:name 'X'.  
}
```

The Table 3.3 shows the query transformation in both cases.

Let $Q$ be the initial query, $W_{Q_{query}}$ the where clause of $Q$ and $\{?h_i\}_{1 \leq i \leq n}$ the variables of $W_{Q_{query}}$ used on the query head of $Q$. 
EXISTS and NOT EXISTS filters

The use of EXISTS filter consists in selecting explicitly the data that satisfies the condition expressed in the block of EXISTS FILTER. That block corresponds to $FILTER(\omega)$. The transformed query $Q'$ is defined as follows:

$$Q' = \text{SELECT } \{ ?h_i \}_{1 \leq i \leq n} \text{WHERE} \{ W_{\text{Query}}. \text{FILTER EXISTS}(GP(\omega)) \}$$

Such that $GP(\omega)$ represents the group pattern associated with the complex condition $\omega$. In the case of prohibition, we use $\text{NOT EXISTS}$ instead of the EXISTS operator.

OPTIONAL Operator

The second way to do the transformation is to use the optional portion $OP$. It is used to check the satisfaction of the condition $\omega$. It contains the condition $\omega$ and filters over the variables used on the group pattern of $\omega$.

$$OP = \text{OPTIONAL}\{ \text{GP}(\omega). \text{FILTER}(\land_{i=1}^{m}\text{bound}(?k_i)) \}$$

Such that $\{ ?k_i \}_{1 \leq i \leq m}$ are all variables used in $GP(\omega)$. If $\omega$ is satisfied then the variable $?ok$ is bound to value ‘true’ and vice-versa. At the end of the where clause we check if the variable $?ok$ is bound to a value then the condition is satisfied. Since we are in the permission case, the user $u$ is allowed to see the selected data.

$$Q' = \text{SELECT } \{ ?h_i \}_{1 \leq i \leq n} \text{WHERE} \{ \begin{array}{c}
W_{\text{Query}}.
\text{OPTIONAL}\{ \text{GP}(\omega) \}
\text{FILTER}(\land_{i=1}^{m}\text{bound}(?k_i))
\end{array} \}$$

In the prohibition case we used the filter $FILTER(\lnot \land_{i=1}^{m}\text{bound}(?k_i))$. 
3.5.4 Composition of simple and involved conditions

In this section we study the rewriting algorithm in case of composition of simple and involved conditions. The study is valid also for the composition of simple and complex conditions.

Let Algo be a rewriting query algorithm which takes a query \( Q \), a condition \( \omega \) and the type of security rule (permission, prohibition) as inputs and returns a new query \( Q' \). In the case of a permission rule, the execution result of the query \( Q' \), denoted \( RQ' \), is composed of elements of \( I(\omega) \), i.e. the execution result of \( Q' \) satisfies the condition \( \omega \). If we suppose that \( RQ \) is the execution result of \( Q \), then \( RQ' = RQ \cap I(\omega) \) (Figure 3.2-A). In the case of a prohibition rule, the execution result of the query \( Q' \) is composed of elements of \( \overline{I(\omega)} = E \setminus I(\omega) \), i.e. \( RQ' = RQ \cap \overline{I(\omega)} = RQ \setminus I(\omega) \) (Figure 3.2-B).

![Figure 3.2: (A) Permission case. (B) Prohibition case](image-url)

**Composition in the case of permission**

Let \( \omega_1 \) be a simple condition, \( \omega_2 \) be an involved condition, \( \omega \) the condition \( \omega_1 \land \omega_2 \) and \( \omega' \) the condition \( \omega_1 \lor \omega_2 \). Let \( Algo_1 \) (section 3.5.1) and \( Algo_2 \) (section 3.5.2) be respectively the rewriting query algorithms of the simple conditions and involved conditions. Let \( Q, Q_1 \) and \( Q_2 \) be SPARQL queries, \( RQ, RQ_1 \) and \( RQ_2 \) be respectively the execution result of \( Q, Q_1 \) and \( Q_2 \) such that \( Q_1 = Algo_1(Q, \omega_1, \text{permission}) \) and \( Q_2 = Algo_2(Q_1, \omega_2, \text{permission}) \).

**Logical AND:** \( \omega = \omega_1 \land \omega_2 \) (Figure 3.3-A)

We have \( RQ_2 = RQ_1 \cap I(\omega_2) \) and \( RQ_1 = RQ \cap I(\omega_1) \) then \( RQ_2 = RQ \cap I(\omega_1) \cap I(\omega_2) \). According to the result of theorem 2 we deduce that \( RQ_2 = RQ \cap I(\omega_1 \land \omega_2) = RQ \cap I(\omega) \).
Thus we can use $\text{Algo}_1$ and $\text{Algo}_2$ to rewrite the query $Q$ in order to satisfy the security rule $\text{Permission}(\omega) = \text{Permission}(\omega_1 \lor \omega_2)$. The rewriting query algorithm corresponding to this case is defined as follows:

$$\text{Algo}(Q, \omega_1 \land \omega_2, \text{permission}) = \text{Algo}_2(\text{Algo}_1(Q, \omega_1, \text{permission}), \omega_2, \text{permission})$$

**Example 12**  
Bob is permitted to select salaries of the network department employees. This rule could be expressed as $\text{Permission}(\omega) = \text{Permission}(\omega_1 \lor \omega_2)$ where: $\forall x = (s, p, o) \in E \omega_1(x) = (p = \text{emp:salary})$

$$\omega_2(x) = \begin{cases} True & \text{if } (\exists y \in E) | y = (s, \text{emp:dept}, \text{'Network'}) \\ False & \text{Otherwise} \end{cases}$$

**Logical OR:** $\omega = \omega_1 \lor \omega_2$ (Figure 3.3-B)

Let $Q'_1$ and $Q'_2$ be SPARQL queries, $RQ'_1$ and $RQ'_2$ be respectively the execution result of $Q'_1$ and $Q'_2$ such that $Q'_1 = \text{Algo}_1(Q, \omega_1, \text{permission})$ and $Q'_2 = \text{Algo}_2(Q, \omega_2, \text{permission})$.

We have $RQ'_2 = RQ \cap I(\omega_2)$ and $RQ'_1 = RQ \cap I(\omega_1)$ then $RQ'_1 \cup RQ'_2 = (RQ \cap I(\omega_1)) \cup (RQ \cap I(\omega_2)) = RQ \cap (RQ'_1 \cup RQ'_2)$.

So $RQ'_1 \cup RQ'_2 = RQ \cap I(\omega_1 \lor \omega_2)$.

We deduce that the rewriting query $Q_{\text{final}}$ corresponding to $\text{Permission}(\omega') = \text{Permission}(\omega_1 \lor \omega_2)$ is the union of the queries $Q'_1$ and $Q'_2$. So we can write $Q_{\text{final}} = Q'_1 \cup Q'_2$ as well as

$$\text{Algo}(Q, \omega_1 \lor \omega_2, \text{permission}) = \text{Algo}_1(Q, \omega_1, \text{permission}) \cup \text{Algo}_2(Q, \omega_2, \text{permission})$$

**Example 13**  
Bob is permitted to select the employees’ salaries. He is also permitted to select all the information of the network department employees. This rule could be expressed as $\text{Permission}(\omega) = \text{Permission}(\omega_1 \lor \omega_2)$ where: $\forall x = (s, p, o) \in E, \omega_1(x) = (p = \text{emp:salary})$

$$\omega_2(x) = \begin{cases} True & \text{if } (\exists y \in E) | y = (s, \text{emp:dept}, \text{'Network'}) \\ False & \text{Otherwise} \end{cases}$$

**Composition in the case of prohibition**

Let $\omega_1$ be a simple condition, $\omega_2$ be an involved condition. Let $\text{Algo}_1$ and $\text{Algo}_2$ be respectively the rewriting query algorithms of the simple condition and the involved
condition. We use the same reasoning as in the previous section and by applying De Morgan’s laws for sets, we obtain the following results:

**Logical AND:** \( \omega = \omega_1 \land \omega_2 \) (Figure 3.3-C)

\[
\text{Algo}(Q, \omega_1 \land \omega_2, \text{prohibition}) = \text{Algo}_1(Q, \omega_1, \text{prohibition}) \cup \text{Algo}_2(Q, \omega_2, \text{prohibition})
\]

**Logical OR:** \( \omega = \omega_1 \lor \omega_2 \) (Figure 3.3-D)

\[
\text{Algo}(Q, \omega_1 \lor \omega_2, \text{prohibition}) = \text{Algo}_2(\text{Algo}_1(Q, \omega_1, \text{prohibition}), \omega_2, \text{prohibition})
\]

### 3.6 Conclusion and Contribution

In this chapter, we have defined an approach to protect SPARQL select queries using query transformation. It is a generic approach to specify and apply an access control policy to protect RDF documents and/or resources viewed as RDF. An access control policy is modelled as a set of filters. A filter may be associated with a simple condition or an involved condition. Involved conditions provide means to protect relationships. We consider two different types of filters: Positive filters corresponding to permission and negative filters corresponding to prohibition.
In this chapter, we only consider the case of select queries. There are some recent proposals to extend SPARQL to specify queries for updating RDF documents. The next chapter (ch. 4) presents an extension that considers how to transform update queries with respect to an access control policy.

Second, the access control policy is specified through a set of filters. This provides a generic approach to represent an access control policy for RDF documents which does not rely on a specific language. However and as suggested in section 3.2, a possible extension would be to define a user friendly specification language to express such an access control policy. For this purpose, the chapter 5 shows how to derive the filter definition from the specification of an access control policy based on an existing access control model such as the OrBAC model [8].
4.1 Introduction

Several access control models for database management systems (DBMS) only consider how to manage select queries and then assume that similar mechanism would apply to update queries. However they do not take into account that updating data may possibly disclose some other sensitive data whose access would be forbidden through select queries. This is typically the case of current relational DBMS managed through SQL which are wrongly specified and lead to inconsistency between select and update queries. In this chapter, we show how to solve this problem in the case of SPARQL queries. We present an approach based on rewriting SPARQL/Update queries. It involves two steps. The first one satisfies the update constraints. The second one handles consistency between select and update operators. Query rewriting is done by adding positive and negative filters (corresponding respectively to permissions and prohibitions) to the initial query.

In the literature, several access control models for database management systems have been defined to implement a security mechanism that controls access to confidential data. For instance, the view-based access control model for relational database, Stonebraker’s model [66] for Ingres database, query transformation model, etc. These proposals assume that similar mechanism would apply to both select and update operators, which is not generally true. They do not enforce consistency between consultation and modification of data.
Table 4.1: result of select query on Employee table

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Said</td>
<td>Rennes</td>
<td>45 000</td>
</tr>
<tr>
<td>Toutou</td>
<td>Madrid</td>
<td>60 000</td>
</tr>
<tr>
<td>Aymane</td>
<td>London</td>
<td>55 000</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
<td>90 000</td>
</tr>
<tr>
<td>Safa</td>
<td>Paris</td>
<td>45 000</td>
</tr>
</tbody>
</table>

For example in the case of a SPARQL query, let us assume that for two given predicates $p$ and $q$, we are allowed to select and modify the value of $p$, but we are not allowed to select the value of $q$. If we update the value of $p$ using a condition on the predicate $q$, we can deduce the value of $q$ (see the examples 2 and 3 in section 4.3.2). Here we use permission to update in order to disclose confidential information which we are not allowed to see. Unfortunately, this problem is not taken into account by several access control models and still exists in many implementations including current relational DBMS compliant with SQL.

Our approach is to rewrite the user SPARQL update query by adding some filters to that query. It involves two steps: (1) Satisfy the security constraints associated with ‘update’ and (2) handle the consistency between select and update operators.

This chapter is organized as follows. Section 4.2 presents our motivating example. Section 4.3 formally defines our approach to manage SPARQL update queries. Finally, section 4.4 concludes this chapter.

### 4.2 Motivating example

The model of view is an interesting access control model for relational databases. It works pretty well for select operator, but when we move to update, there may be some illegal disclosure of confidential data. Let us take an example to illustrate this problem.

We suppose that we have an ‘Employee’ table with fields ‘name’, ‘city’ and ‘salary’ and with the following data (see Table 4.1). We create a user named Bob. We suppose then that Bob is not allowed to see the salary of employees. According to the view model, we create a view with fields ‘name’ and ‘city’. Then, we give to Bob the permission to ‘select’ on this view. Let Employee_view be that view which is defined as follows:

Query: CREATE VIEW ‘Employee_view’ AS (select name, city from Employee)
4.2. MOTIVATING EXAMPLE

We suppose now that Bob is allowed to select fields of ‘Employee_view’. So, he can execute the following query:

**Query:** SELECT * FROM 'Employee_view'

The result of that query is presented on the table 4.2. We note that Bob cannot see the employees’ salary. Now let us assume that he is allowed to update data of the ‘Employee’ table. He updates, for example, the city of employees who earn 45 000 using the following SQL query:

**Query:** UPDATE Employee SET city="Brest" WHERE salary=45 000

Now, he takes a look at Employee_view in order to see if its content has been changed or not. So, he executes the following query:

**Query:** SELECT * FROM 'Employee_view'

As we can see if we compare with the content of the initial Employee_view (Table 4.3), the city of employees Said and Safa is changed to “Brest”. So Bob deduces that their salary is 45000, which he is not allowed to. Although Bob is not permitted to see the salary of the employee table, he is able to learn, through an update command, that there are two employees, Said and Safa, with a salary equal to 45000.

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Said</td>
<td>Rennes</td>
</tr>
<tr>
<td>Toutou</td>
<td>Madrid</td>
</tr>
<tr>
<td>Aymane</td>
<td>London</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
</tr>
<tr>
<td>Safa</td>
<td>Paris</td>
</tr>
</tbody>
</table>

Table 4.2: result of select query on Employee_view

<table>
<thead>
<tr>
<th>name</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Said</td>
<td>Brest</td>
</tr>
<tr>
<td>Toutou</td>
<td>Madrid</td>
</tr>
<tr>
<td>Aymane</td>
<td>London</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
</tr>
<tr>
<td>Safa</td>
<td>Brest</td>
</tr>
</tbody>
</table>

Table 4.3: result of select query on Employee_view
This kind of problem exists in all relational databases such as Oracle. It lies in the SQL specification. It does not come from the security policy (Bob could have permission to update the salaries without necessarily being allowed to consult them). It comes from inadequate control on the update query. Let us show how to handle this in the case of SPARQL queries.

### 4.3 Principle of our approach

Let $\omega$ be a condition of RDF triples and $\{p_i\}_{1 \leq i \leq n}$ a set of predicates of $E_{predicate}$. We define our security rules as the permission or prohibition to select (or update) the value of predicates $\{p_i\}_{1 \leq i \leq n}$ if the condition $\omega$ is satisfied.

Our approach involves two steps:

- (1) Satisfy the security constraints associated with ‘update’.
- (2) Handle the inconsistency between ‘select’ and ‘update’.

#### 4.3.1 Update access control

In this case we similarly treat the ‘DELETE’ and ‘INSERT’ clause of the update query.

Let $D_{\text{Query}}$, $I_{\text{Query}}$ and $W_{\text{Query}}$ be respectively the DELETE, INSERT and WHERE clause of a user’s update query. There are two cases, the case of prohibition and the case of permission.

**Prohibition case:**

We assume that a user $u$ is not allowed to update the value of predicates $\{p_i\}_{1 \leq i \leq n}$ if the condition $\omega$ is satisfied. Since the security policy is closed then this user is allowed to update the value of predicates that are not in $\{p_i\}_{1 \leq i \leq n}$. Now, if this user tries to update at least one predicate $p$ of $\{p_i\}_{1 \leq i \leq n}$, then we must check if the condition $\omega$ is not satisfied (prohibition case). Which means adding the negative filter of $\omega$ to $W_{\text{Query}}$.

We deduce then the following expression in the case of prohibition:

$$\left[ \exists p \in D_{\text{Query}} \cup I_{\text{Query}} \right] \left[ p \in \{p_i\}_{1 \leq i \leq n} \right] \rightarrow \left[ \text{Filter}(W_{\text{Query}}, \omega) \right] \quad (4.1)$$
4.3. PRINCIPLE OF OUR APPROACH

Where $\text{Filter}(GP,C)$ means adding the SPARQL filter of the RDF condition $C$ to the group of patterns $GP$.

Let $I^u_{\text{Integ}}$ be the set of elements that the user $u$ is permitted to update. For a given predicate $q$ we denote $S_q$ a set of triple of $E$ that has the predicate $q$. Let $S^u_{\text{pred}} = \bigcup_{i=1}^n S_{p_i}$.

The user $u$ is not allowed to update the value of predicates $\{p_i\}_{1 \leq i \leq n}$ if the condition $\omega$ is satisfied. This is equivalent to: $I^u_{\text{Integ}} = S^u_{\text{pred}} \cap I(\omega)$. According to the result of Theorem 1 (section 3.3), we deduce that:

$$I^u_{\text{Integ}} = S^u_{\text{pred}} \cup I(\overline{\omega})$$

Proof of integrity:

Let $x$ be an element of $E$ that has been updated by the user’s transformed query. There are two cases: (i) $x \notin S^u_{\text{pred}}$ (ii) $x \in S^u_{\text{pred}}$. In the case (i) we have $x \in S^u_{\text{pred}} \subseteq I^u_{\text{Integ}}$. So, $x \in I^u_{\text{Integ}}$. In the case (ii), we have $x \in S^u_{\text{pred}}$. According to the expression 4.1 above, $x$ satisfies the negation of the condition $\omega$, which means that $x \in I(\overline{\omega}) \subseteq I^u_{\text{Integ}}$. So, $x \in I^u_{\text{Integ}}$. In both cases, $x \in I^u_{\text{Integ}}$, which proves that the expression 4.1 above preserves the integrity.

Permission case:

We assume that a user $u$ is allowed to update the value of predicates $\{p_i\}_{1 \leq i \leq n}$ if the condition $\omega$ is satisfied. So:

- (i) The user $u$ is not allowed to update the value of predicates $\{p_i\}_{1 \leq i \leq n}$ if the condition $\overline{\omega}$ is satisfied.
- (ii) He is not allowed also to update the value of predicates that are not in $\{p_i\}_{1 \leq i \leq n}$. Which means that he is not allowed to update the value of predicates $E_{\text{predicate}} \setminus \{p_i\}_{1 \leq i \leq n}$ if the condition $\Omega_{\text{True}}$ is satisfied.

According to the expression (4.1), the prohibition (i) is equivalent to:

$$[(\exists p \in D_{Query} \cup I_{Query})|p \in \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{[Filter}(W_{Query},\overline{\omega})]$$

We have $\overline{\omega} = \omega$. So,

$$[(\exists p \in D_{Query} \cup I_{Query})|p \in \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{[Filter}(W_{Query},\omega)]$$  \hspace{1cm} (4.2)

The prohibition (ii) is equivalent to:

$$[(\exists p \in D_{Query} \cup I_{Query})|p \in E_{\text{predicate}} \setminus \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{[Filter}(W_{Query},\overline{\Omega_{\text{True}}})]$$
CHAPTER 4. REWRITING OF SPARQL/UPDATE QUERIES FOR...

We have $\Omega_{\text{True}} = False$, so (ii) is equivalent to:

$$[(\exists p \in D_{\text{Query}} \cup I_{\text{Query}}) | p \notin \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{Filter}(W_{\text{Query}}, False)$$

Adding false filter to $W_{\text{Query}}$ means ignoring the execution of the query. So, we treat the case (ii) first. If we need to add a false filter then we do not have to treat the case (i), since the query is ignored. Otherwise, we treat the case (i).

4.3.2 Consistency between consultation and modification

This section treats the second step of our approach. It handles the consistency between the ‘select’ and ‘update’ operators. This treatment is done only by analysing the quantification portion of the update query.

Let $u$ be a user and $I^u_{\text{Conf}}$ (resp. $I^u_{\text{Integ}}$) be the set of elements that the user $u$ is permitted to select (resp. to update).

In general, in the case of the select operator, the result of the query must be a subset of $I^u_{\text{Conf}}$ (Figure 4.1 (A)) in order to preserve confidentiality. Similarly for the update operator, the modified data must be a subset of $I^u_{\text{Integ}}$ in order to preserve the integrity (Figure 4.1 (B)). As shown in the motivating example, these two rules are not sufficient to preserve confidentiality when the user has both authorizations to select
4.3. PRINCIPLE OF OUR APPROACH

and authorizations to update. In other words, the user can update an element $x$ in $I^u_{Conf} \cap I^u_{Integ}$ using reference to an element $y$ of $\overline{I^u_{Conf}} = E \setminus I^u_{Conf}$ in order to deduce the value of $y$ associated with $x$, which he is not allowed to see. There are two cases:

1. $I^u_{Conf} \cap I^u_{Integ} = \emptyset$,
2. $I^u_{Conf} \cap I^u_{Integ} \neq \emptyset$.

It is obvious that in the first case (1), there is no such problem. However, in the current SQL implementation, we still have another kind of problem of confidentiality. For example, in our motivating example, if we suppose that the user Alice is allowed only to select names and cities of employees and she is allowed to update only their salaries. This corresponds to the case (1). Alice updates the salary of employees who earn exactly 45000 using the following SQL query:

```sql
SQL> UPDATE Employee SET salary=salary+100 WHERE salary=45 000;
2 rows updated
```

Although, Alice is not permitted to see the salary of employees, she has been able to learn, through this update command, that there are two employees with a salary 45000. This would correspond to a write down prohibited by the Bell-LaPadula model [76].

To solve it, we have just to delete the information message (number of rows that has been updated).

Now let us assume that $I^u_{Conf} \cap I^u_{Integ} \neq \emptyset$. Let $x \in I^u_{Conf} \cap I^u_{Integ}$, we denote $I_{Ref}(x)$ a subset of elements of $\overline{I^u_{Conf}}$ that are in relation with $x$, i.e $I_{Ref}(x) = \{ y \in \overline{I^u_{Conf}} \mid \exists R, a\ binary\ relation\ over\ E\ such\ that\ xRy \}$. If $I_{Ref}(x)$ is not empty, then we can deduce the value of each element of $I_{Ref}(x)$. We simply update the value of $x$ using elements of $I_{Ref}(x)$ in the where clause of our query. So, our problem is equivalent to the following proposition:

$$[(\exists x \in I^u_{Conf} \cap I^u_{Integ}) | I_{Ref}(x) \neq \emptyset] \rightarrow [\text{Interference(select, update)}]$$

(4.4)

$\text{Interference(select, update)}$ means that there exists an interference between select and update operators. To solve this problem, we must control $W_{Query}$ (the where clause of the query) to avoid referencing a value that is not in $I^u_{Conf}$.

There are two cases:

**Prohibition case**

Let $\omega$ be a condition of RDF triples. We assume that a user $u$ is not allowed to see (select) values of predicates $\{p_i\}_{1 \leq i \leq n}$ where $n \in N^*$, if the condition $\omega$ is satisfied.

Let $S^u_{pred}$ be a set of all possible triples of $E$ where predicates are elements of $\{p_i\}_{1 \leq i \leq n}$. Since we are in the case of prohibition, so, $\overline{I^u_{Conf}} = S^u_{pred} \cap I(\omega)$. If the $W_{Query}$ of the update query uses at least one predicate of the set $\{p_i\}_{1 \leq i \leq n}$, then the
condition \(\omega\) should not be satisfied in order to enforce the security policy. In other words we have to introduce the negative filter of \(\omega\) in \(W_{\text{Query}}\). This is equivalent to:

\[
(\exists p \in W_{\text{Query}}) [p \in \{p_i\}_{1 \leq i \leq m} \rightarrow \text{Filter}(W_{\text{Query}}, \overline{\omega(x_p)})]
\]

(4.5)

where \(x_p\) is the triple pattern of \(W_{\text{Query}}\) using the predicate \(p\).

**Proof of Confidentiality:** We assume that the user \(u\) tries to update an element \(x\) of \(I^{u}_{\text{Conf}} \cap I^{u}_{\text{Integ}}\) by referencing an element \(y \in E\). It is obvious that if \(y \in I^{u}_{\text{Conf}}\) there is no such problem. We assume that \(y \notin I^{u}_{\text{Conf}}\) i.e \(y \in \overline{I^{u}_{\text{Conf}}} = S^{u}_{\text{pred}} \cap I(\omega)\), so, \(y \in I(\omega)\) and \(y \in S^{u}_{\text{pred}}\).

Let us proceed by contradiction to proof that the update of \(x\) has not occurred. We have \(y \in S^{u}_{\text{pred}}\), so according to the expression (4.5) above, the negative filter of \(\omega\) has been added to \(W_{\text{Query}}\). If we assume that the user update takes place, then \(y\) satisfies the negation of \(\omega\) i.e \(y \in I(\overline{\omega}) = \overline{I(\omega)}\). This means that \(y \notin I(\omega)\), so this is a Contradiction. So, the update of \(x\) has not occurred.

**Involved condition:** Let \(P\) be the graph pattern of \(W_{\text{Query}}\). Let \(\omega\) be an involved condition associated with \(\{(q_i, \omega_i)\}_{1 \leq i \leq m}\) where \(\{q_i\}_{1 \leq i \leq m}\) is a set of predicates and \(\{\omega_i\}_{1 \leq i \leq m}\) is a set of simple conditions. In the case of an involved condition, negative filter is equivalent to: (i) at least one simple condition of \(\{\omega_i\}_{1 \leq i \leq m}\) is not satisfied or (ii) at least one predicate of \(\{q_i\}_{1 \leq i \leq m}\) does not appear. That is \(W_{\text{Query}}\) corresponds to the following transformed query:

\[
W_{\text{Query}} = P_1 \text{ UNION } P_2
\]

where \(P_1\) and \(P_2\) are the following graph patterns:

\[
P_1 = (P \text{ AND } (\text{UNION}_{i=1}^{m}(tp_i \text{ FILTER } \overline{\omega_i(tp_i)})))
\]

\[
P_2 = (P \text{ OPT}_{i=1}^{m}(tp_i) \text{ FILTER } (\text{BOUND}_{i=0}^{m}(?\text{obj}_i)))
\]

such that \(\{tp_i = (?\text{emp}, q_i, ?\text{obj}_i)\}_{1 \leq i \leq m}\).

\(\text{AND}, \text{UNION}, \text{OPT}\) and \(\text{FILTER}\) are respectively the SPARQL binary operators (.), \(\text{UNION}\), \(\text{OPTIONAL}\) and \(\text{FILTER}\). \(P_1\) guarantees that at least one simple condition of \(\{\omega_i\}_{1 \leq i \leq m}\) is not satisfied. \(P_2\) represents the set of elements that does not have at least one predicate of \(\{q_i\}_{1 \leq i \leq m}\).

In SPARQL 1.1 version [14], \(\text{NOT EXIST}\) and \(\text{EXISTS}\) are respectively two filters using graph pattern in order to test for the absence or presence of a pattern. In the case of prohibition for an involved condition, it comes to test the non existence of triples pattern \(\{tp_i \text{ FILTER } \omega_i(tp_i)\}_{1 \leq i \leq m}\). In this case \(W_{\text{Query}}\) corresponds, for example, to the following transformed query: \(W_{\text{Query}} = P \text{ FILTER } \text{NOT EXIST } (\text{AND}_{i=1}^{m}(tp_i \text{ FILTER } \omega_i(tp_i)))\)
Permission case

Let $\omega$ be a condition of RDF triples. We assume that a user is allowed to see (select) the value of predicates $\{p_i\}_{1 \leq i \leq n}$ where $n \in \mathbb{N}^*$, if the condition $\omega$ is satisfied. Since our security policy is closed, this permission could be expressed as the following prohibitions:

- (a) The user is not allowed to see the value of predicates $E_{\text{Predicate}} \{p_i\}_{1 \leq i \leq n}$
- (b) The user is not allowed to see the value of predicates $\{p_i\}_{1 \leq i \leq n}$, if the condition $\omega$ is satisfied.

Let us apply the expression 4.5 to the case (a) and then to the case (b). The case (a) could be expressed as prohibition to see the value of predicates $E_{\text{Predicate}} \{p_i\}_{1 \leq i \leq n}$ if the condition $\Omega_{\text{True}}$ is satisfied. $\Omega_{\text{True}}$ is a simple condition, so according to the prohibition algorithm, if $W_{\text{Query}}$ uses at least one predicate of the set $E_{\text{Predicate}} \{p_i\}_{1 \leq i \leq n}$ then we add the corresponding negative filter of $\Omega_{\text{True}}$ to $W_{\text{Query}}$, i.e.:

$$[(\exists p \in W_{\text{Query}}) \mid p \in E_{\text{Predicate}} \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{Filter}(W_{\text{Query}}, \Omega_{\text{True}}(x_p))$$

where $x_p$ is the triple pattern of $W_{\text{Query}}$ using the predicate $p$. This is equivalent to:

$$[(\exists p \in W_{\text{Query}}) \mid p \notin \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{Filter}(W_{\text{Query}}, \text{False})$$

(4.6)

Adding the False filter to $W_{\text{Query}}$, means ignoring the execution of the query. So, we do not have to treat the case (b) since the query is ignored.

Now if all predicates of $W_{\text{Query}}$ belong to $\{p_i\}_{1 \leq i \leq n}$. We treat the case (b). According to the prohibition algorithm, (b) is equivalent to:

$$[(\forall p \in W_{\text{Query}}) \mid p \in \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{Filter}(W_{\text{Query}}, \omega(x_p))$$

The negative filter of $\omega$ is the positive filter of $\omega$. So, the case (b) is equivalent to the following proposition:

$$[(\forall p \in W_{\text{Query}}) \mid p \in \{p_i\}_{1 \leq i \leq n}] \rightarrow \text{Filter}(W_{\text{Query}}, \omega(x_p))$$

(4.7)

Example 14 (case of involved condition)

We assume that Bob is not allowed to see the name, age and salary of network department employees having an age greater than 30. This prohibition could be expressed as “Bob is not allowed to see values of predicates $\{\text{emp:name}, \text{emp:age}, \text{emp:salary}\}$ if the involved condition $\omega$, defined below, is satisfied”.

$$[(\exists p \in W_{\text{Query}}) \mid p \in E_{\text{Predicate}} \{p_i\}_{1 \leq i \leq n}]$$
\((\forall x = (s, p, o) \in E)\)
\[
\omega(x) = \begin{cases} 
\text{True} & \text{if } (\exists (value_1, value_2) \in E^2_{\text{Object}})|\omega_1(x_1) = \text{True and } \omega_1(x_2) = \text{True} \\
\text{False} & \text{Otherwise}
\end{cases}
\]

such that \((\forall y = (s', p', o') \in E)\)
\[
\omega_1(y) = ((p' = \text{emp:dept}) \land (o' = "\text{Network}")) \lor ((p' = \text{emp:age}) \land (o' \geq 30))
\]

We assume also that Bob is allowed to update the city of all employees. Bob tries to update the city of employees whose name is "Alice". So the corresponding Bob’s update query will be as follows:

```sparql
WITH <http://swid.fr/employees>
DELETE { ?emp emp:city ?city } 
INSERT { ?emp emp:city 'Rennes' } 
WHERE { 
  ?emp rdf:type emp:Employee; 
  emp:name 'Alice'.
}
```

We note that Bob uses the predicate name on \(W_\text{Query}\). We know also that this predicate is prohibited to be selected under the involved condition \(\omega\). We assume that the employee Alice works in the network department and she is 34 years old. So the execution of this query allows Bob to deduce that there is an employee named "Alice" on the network department and her age is greater than 30.

According to the result above, the transformed query will be as follows:

```sparql
WITH <http://swid.fr/employees>
DELETE { ?emp emp:city ?city } 
INSERT { ?emp emp:city 'Rennes' } 
WHERE { 
  ?emp rdf:type emp:Employee; 
  emp:name 'Alice'.
  { 
    ?emp emp:dept ?dept. FILTER(?dept != 'Network'))
    UNION { ?emp emp:age ?age. FILTER(?age<30) }
  }
}
UNION { 
  ?emp rdf:type emp:Employee; 
  emp:name 'Alice'.
  OPTIONAL{?emp emp:dept ?dept}
  OPTIONAL{?emp emp:age ?age}
  FILTER(!bound(?dept) || !bound(?age))
}
```

In the case of SPARQL 1.1, the transformed query will be as follows:
4.4 Conclusion and Contribution

In this chapter, we have defined an approach to protect SPARQL/Update queries using query transformation. It involves two steps. The first one is to satisfy the update constraints. The second one is to handle consistency between ‘select’ and ‘update’ operators by analyzing the WHERE clause of the update query. Query rewriting is done by adding filters to the initial query: Positive filters corresponding to permission and negative filters corresponding to prohibition. We also presented several proofs to show the correctness of the rewriting approach for different cases.

i.e. the update will be done only on employees who are not in the network department (or do not have the department property) or are less than 30 years old (or do not have the age property).
5.1 Introduction

A possible way to enforce access control requirements when evaluating SPARQL queries is based on rewriting the queries such that their evaluation fulfills the access control policy, as illustrated in chapters 3 and 4. One may then define a new dedicated language to express access control policies for SPARQL.

In this chapter, we suggest a different approach based on a generic access control policy model in order to express the access control requirements. We present how to instrument the rewriting algorithm using a set of security policy rules derived from the OrBAC model [8].

In chapter 3 the policy constraints are presented as conditions of RDF triples. An RDF condition is defined as an application $\omega : E \rightarrow \text{Boolean}$ where $E$ is the set of all RDF triples of our RDF database. The condition $\omega$ associates each RDF triple $x = (s, p, o)$ of $E$ with an element of the set $\text{Boolean} = \{\text{True, False}\}$.

Let us take an example:

**Example 15** We assume that a user with the role *manager* is allowed to read the information of its own employees. The RDF condition $\omega$ corresponding to that constraint could be expressed using SPARQL syntax as follows (question mark denotes variables):
One way to express this security rule is as follows:

\[ \text{Rule}_{\text{sparql}} = \text{Permission(} \text{manager, select, Employee, } \omega \text{)} \]

which means that the role manager is allowed to select data of the concept Employee if the RDF condition \( \omega \) is satisfied.

Every data model, like the relational model, has generally its own access control model for expressing the security policy. For RDF views or databases, we can also define a new model to express security policy as illustrated in the example above. However, in the case of heterogeneous databases, it becomes difficult to manage these different databases if they use different access control languages.

In this chapter we show how to define a SPARQL security policy in a generic high level access control model (OrBAC [8]). Then, from this policy definition, we extract the corresponding security constraints as an RDF condition \( \omega \) by a simple translation. Finally, the condition \( \omega \) is used to rewrite the initial query using the fQuery approach [74] presented in chapter 3.

This chapter is organized as follows. Section 5.2 presents the OrBAC model. Section 5.3 presents the principle of our approach. Section 5.4 shows how we model RDF condition in the OrBAC model. Then, section 5.5 presents how a rewriting query algorithm is instrumented by OrBAC rules. Finally, section 5.6 concludes this chapter.

5.2 The OrBAC model

The Organization-Based Access Control model (OrBAC) [8] is a generic and expressive access control model. It provides interesting concepts to express the security policy and enables making distinction between an abstract policy specifying organizational requirements and its implementation in a given information system. OrBAC is a model built on top of Rule based access control (Rule-BAC) termed models. In this kind of model, access control is defined as a set of rules \( \text{Condition} \rightarrow \text{Authorization} \) where \( \text{Condition} \) is a set of constraints over the subjects, actions and objects. The central entity in OrBAC is the entity Organization. Intuitively, an organization can be seen as any entity that is responsible for managing a given access control policy (e.g. hospitals
5.2. THE ORBAC MODEL

or companies are organizations). Therefore, instead of defining security rules that
directly apply to subject, action and object, the access control policy is defined at the
“organizational” level. For this purpose, subject, action and object are respectively
abstracted into role, activity and view. A view corresponds to a set of objects to
which the same security rules apply. An activity is similarly defined but for regrouping
actions. Finally, permissions and prohibitions only apply in specific contexts.

The OrBAC formalism is compatible with a stratified Datalog with negation pro-
gram [77]. Stratifying a Datalog program consists in ordering rules so that if a rule
contains a negative literal then the rule that defines this literal is computed first. A
stratified Datalog program is computable in polynomial time. To express rules and
facts, we shall actually use a prolog-like notation. Terms starting with a capital letter,
such as Subject, correspond to variables and terms starting with a lower case let-
ter, such as peter, correspond to constants. A fact, such as: \( \text{Parent} (\text{peter}, \text{john}) \). says that peter is a parent of john, and a rule such as: \( \text{Grand_parent}(X,Z) :- \text{Parent}(X,Y), \text{Parent}(Y,Z) \).
says that \( X \) is a grand-parent of \( Z \) if there is a subject \( Y \) such that \( X \) is a parent of \( Y \) and \( Y \) is a parent of \( Z \).

5.2.1 Basic predicates

There are eight basic sets of entities: \( \text{Org} \) (a set of organizations), \( S \) (a set of subjects),
\( A \) (a set of actions), \( O \) (a set of objects), \( R \) (a set of roles), \( A \) (a set of activities), \( V \) (a
set of views) and \( C \) (a set of contexts). In the following we present the basic built-in
predicates:

- \textbf{Empower} is a predicate over domains \( \text{Org} \times S \times R \). If \( \text{org} \) is an organization, 
  \( s \) is a subject and \( r \) is a role, then \( \text{Empower}(\text{org},s,r) \) means that \( \text{org} \) empowers
  subject \( s \) in role \( r \).

- \textbf{Use} is a predicate over domains \( \text{Org} \times O \times V \). If \( \text{org} \) is an organization, \( o \) is an
  object and \( v \) is a view, then \( \text{Use}(\text{org},o,v) \) means that \( \text{org} \) uses \( o \) in view \( v \).

- \textbf{Consider} is a predicate over domains \( \text{Org} \times A \times A \). If \( \text{org} \) is an organization, \( \alpha \)
  is an action and \( a \) is an activity, then \( \text{Consider}(\text{org},\alpha,a) \) means that \( \text{org} \) considers
  that action \( \alpha \) implements the activity \( a \).

- \textbf{Hold} is a predicate over domains \( \text{Org} \times S \times A \times O \times C \). If \( \text{org} \) is an organization, \( s \)
  is a subject, \( \alpha \) is an action, \( o \) is an object and \( c \) is a context, then \( \text{Hold}(\text{org},s,\alpha,o,c) \)
  means that within organization \( \text{org} \), context \( c \) holds between subject \( s \), action \( \alpha \)
  and object \( o \).
- **Permission** and **Prohibition** are predicates over domains \( \text{Org} \times \text{R} \times \text{A} \times \text{V} \times \text{C} \), where \( \text{R} = R \cup S \), \( \text{A} = A \cup A \) and \( \text{V} = V \cup O \). If \( \text{org} \) is an organization, \( g \) is a role or a subject, \( t \) is a view or an object, \( p \) is an activity or an action and \( c \) is a context, then \( \text{Permission}(\text{org}, g, p, t, c) \) (resp. \( \text{Prohibition}(\text{org}, g, p, t, c) \)) means that in organization \( \text{org} \), grantee \( g \) is granted permission (resp. prohibition) to perform privilege \( p \) on target \( t \) in context \( c \). These predicates enable a given organization to specify permissions and prohibitions between roles, activities and views in a given context.

- **Is_permitted** and **Is_prohibited** are predicates over domains \( \text{S} \times \text{A} \times \text{O} \). These predicates enable to specify permissions and prohibitions at the concrete level, which are based on subjects, actions and objects. A concrete permission (prohibition) is derived from an abstract permission (prohibition) when the associated context holds as illustrated in the following rule \( \text{Rule}_C \).

\[
\text{Rule}_C \\
\text{Is_permitted}(\text{Sub}, \text{Act}, \text{Obj}) \leftarrow \\
\text{Empower}(\text{Org}, \text{Sub}, \text{R}) \\
\land \text{Consider}(\text{Org}, \text{Act}, \text{A}) \\
\land \text{Use}(\text{Org}, \text{Obj}, \text{V}) \\
\land \text{Permission}(\text{Org}, \text{R}, \text{A}, \text{V}, \text{C}) \\
\land \text{Hold}(\text{Org}, \text{Sub}, \text{Act}, \text{Obj}, \text{C})
\]

### 5.2.2 Role, activity and view definition

Instead of enumerating facts corresponding to instances of predicates \( \text{Empower} \), \( \text{Consider} \) and \( \text{Use} \), it is also possible to specify role, activity and view definitions which correspond to logical conditions that, when satisfied, are used to automatically manage assignment of subject to role, action to activity and object to view, respectively.

For instance, a role definition corresponds to a logical rule that has the \( \text{Empower} \) predicate in the conclusion and respects the Datalog restrictions as follows:

\[
\text{Empower}(\text{BS}, s, \text{Gold_customer}) \leftarrow \\
\text{Empower}(\text{BS}, s, \text{Customer}) \\
\land \text{Membership}(\text{BS}, s, d) \\
\land d >= 10.
\]

where \( \text{Membership}(\text{BS}, s, d) \) is an application dependent predicate meaning that subject \( s \) has been a customer of bookshop \( \text{BS} \) for \( d \) years.
This rule means that in a bookshop BS a subject $s$ is empowered in role Gold_customer, if this subject is empowered in role Customer and if he or she has been a customer for more than 10 years.

Activity and view definitions are similarly used to automatically manage assignment of action to activity and object to view. We assume that activity and view definitions also respect the Datalog restrictions.

### 5.2.3 Context definition

Contexts are used to specify conditions, for example working hours, during vacation or urgency. Conditions that must be satisfied to derive that a context is active are modelled by a logical rule called context definition. More details and discussion about contexts are given in [78]. Contexts are very useful to specify fine grained access control requirements. For instance, let us consider the following context that simply says that a subject executes an action on an object in context during vacation if this subject is in vacation:

$$Hold(org, \text{Subj}, Act, Obj, \text{during\_vacation}) \leftarrow \text{In\_vacation}(\text{Subj}).$$

Five kinds of contexts have been defined [78]:

- the Temporal context that depends on the time at which the subject is requesting for an access to the system,

- the Spatial context that depends on the subject location,

- the User-declared context that depends on the subject objective (or purpose),

- the Prerequisite context that depends on characteristics that join the subject, the action and the object.

- the Provisional context that depends on previous actions the subject has performed in the system.

We can also combine these elementary contexts to define new composed contexts by using conjunction, disjunction and negation operators: $\&$, $\oplus$ and $\neg$. This means that if $c_1$ and $c_2$ are two contexts, then $c_1 \& c_2$ is a conjunctive context, $c_1 \oplus c_2$ is a disjunctive context and $\neg c$ is a negative context. These composed contexts are defined by the following rules:
\[
Hold(\text{org}, s, \alpha, o, c_1 \& c_2) \leftarrow Hold(\text{org}, s, \alpha, o, c_1) \land Hold(\text{org}, s, \alpha, o, c_2).
\]
\[
Hold(\text{org}, s, \alpha, o, c_1 \oplus c_2) \leftarrow Hold(\text{org}, s, \alpha, o, c_1) \lor Hold(\text{org}, s, \alpha, o, c_2).
\]
\[
Hold(\text{org}, s, \alpha, o, \bar{c}) \leftarrow \neg Hold(\text{org}, s, \alpha, o, c).
\]

There is also a context called \textit{nominal} that is always active for any subject, action and object.

Contexts are very useful and provide high flexibility to the OrBAC model. Indeed, it is possible to specify complex conditions, to manage the security policy, that are not supported by models based on RBAC. For instance, it is possible to specify the notion of emergency which is very important in the medical environments. More details and discussion about contexts are given in [78].

### 5.2.4 Hierarchy and inheritance

In the OrBAC model it is suggested to define hierarchies over roles as suggested in the RBAC model [57] but also activities and views, and to associate permission inheritance with these different hierarchies. This is modelled as follows:

- \textit{Sub\_role}, is a partial order relation over domains \textit{OrgRxR}. If \textit{Org} is an organization, \textit{R}_1 and \textit{R}_2 are roles, then \textit{Sub\_role}(\textit{Org}, \textit{R}_1, \textit{R}_2) means that, in organization \textit{Org}, role \textit{R}_1 is a sub-role (also called senior role) of role \textit{R}_2. Permissions and prohibitions are inherited through the role hierarchy. For instance, inheritance of permissions is modelled by the following rule:

\[
\text{Rule}_{SubR} \quad 
\begin{align*}
\text{Permission}(\text{Org}, \textit{R}_1, \textit{A}, \textit{V}, \textit{C}) & \leftarrow \text{Permission}(\text{Org}, \textit{R}_2, \textit{A}, \textit{V}, \textit{C}) \\
& \land \text{Sub\_role}(\text{Org}, \textit{R}_1, \textit{R}_2).
\end{align*}
\]

Similarly, there is a rule that derives prohibitions through the role hierarchy.

- Similar predicates \textit{Sub\_view}(\textit{Org}, \textit{V}_1, \textit{V}_2) and \textit{Sub\_activity}(\textit{Org}, \textit{A}_1, \textit{A}_2) are introduced to respectively specify hierarchies over views and activities. Permissions and prohibitions are also inherited through these hierarchies. For instance, permissions are inherited through the view and the activity hierarchy using \textit{Rule}_{Subv} and \textit{Rule}_{Suba}, that are similar to \textit{Rule}_{SubR} where predicate \textit{Sub\_role} is replaced by \textit{Sub\_view} and \textit{Sub\_activity}, respectively (for further details see [79]).

- \textit{Sub\_context}, is a relation over domains \textit{OrgxCxC}. If \textit{Org} is an organization, \textit{C}_1 and \textit{C}_2 are contexts, then \textit{Sub\_context}(\textit{Org}, \textit{C}_1, \textit{C}_2) means that in organization
5.3 Principle of the approach

One way to express the security rule corresponding to the example 15 in the OrBAC model is as follows:

\[ Rule_{Abst}^{SPARQL} \]
\[
\text{Permission}(org, Manager, Read, EmployeeView, Ctx_\omega) \]

Such that \( org \) is an organization, \( Manager \) is a role, the activity \( Read \) is mapped to the SPARQL action \( Select \), the view \( EmployeeView \) is mapped to the RDF concept \( Employee \). The RDF condition \( \omega \) is expressed in the form of a context definition in the context \( Ctx_\omega \) as follows:

\[
\forall s, \forall \alpha, \forall o \ Hold(\text{org}, s, \alpha, o, Ctx_\omega) \leftarrow Employee(o, e) \land IsManagerOf(s, e)
\]

where \( Employee(o, e) \) and \( IsManagerOf(s, e) \) are application dependent predicates. \( Employee(o, e) \) means that \( o \) is an information of the employee \( e \). \( IsManagerOf(s, e) \) means that \( s \) is a manager of the employee \( e \).

The principle of our approach is to generate RDF conditions from OrBAC rules. Then they will be used by the \( f \)Query algorithm \[74\] to rewrite the user query. Let us take an example to illustrate our approach.

\textbf{Example 16} Let us consider the OrBAC rule \( Rule_{Abst}^{SPARQL} \). We assume that, in the organization \( org \), \( Bob \) has the role \( Manager \) i.e \( Empower(\text{org}, Bob, Manager) \). The action \( Select \) implements the activity
Read i.e. Consider\((\text{org, Select, Read})\). The concept Employee corresponds to the OrBAC view EmployeeView.

Let us now assume that Bob is trying to select the name and salary of employees. He issues the following SPARQL query:

```
SELECT ?name ?salary
WHERE{
  ?e a Employee.
  ?e name ?name.
  ?e salary ?salary.
}
```

According to the query below, the user Bob is trying to select some information (name and salary) of the concept Employee. So, we get context definitions of all OrBAC contexts that are associated with permissions defined for the user Bob and the action Select on the view EmployeeView i.e. context definition of a context \( C \) that satisfies the following condition:

\[
\text{Permission}(\text{org, Role, Activity, EmployeeView, } C)
\wedge
\text{Empower}(\text{org, Bob, Role})
\wedge
\text{Consider}(\text{org, Select, Activity})
\]

According to the OrBAC rule Rule\(_{\text{SPARQL}}^{\text{Abst}}\) we deduce that \( C = \text{Ctx}_\omega \), Role = Manager and Activity = Read. We get the context definition of \( \text{Ctx}_\omega \). Then we translate it to an RDF condition as expressed in the example \[15\]. So we obtain the following SPARQL security rule:

\[
\text{Rule}_{\text{SPARQL}}^{C} = \text{Permission}(\text{Bob, Select, Employee, } \omega)
\]

Finally we apply the \( f \)Query algorithm using the condition \( \omega \) of Rule\(_{\text{SPARQL}}^{C}\). The rewritten query is as follows:

```
SELECT ?name ?salary
WHERE{
  ?e a Employee.
  ?e name ?name.
  ?e salary ?salary.
  FILTER Exists{
    ?e hasManager ?m.
    ?m hasId bob_id.
  }
}
```

Then, as explained in chapter \[3\] the evaluation of this rewritten query will only provide authorized answers with respect to the access control policy (rule Rule\(_{\text{SPARQL}}^{\text{Abst}}\)). This approach can be adapted to handle update queries expressed in SPARQL/update as explained in chapter \[4\].
5.4 Modelling RDF Condition within OrBAC

In this section we present how to model RDF condition (condition of RDF triples) using the OrBAC model. RDF conditions could be presented as context, as view or as both context and view.

5.4.1 RDF condition as context

Contexts are used to specify conditions, for example working hours, urgency... Conditions that must be satisfied to derive that context is active, are modelled by a logical rule called context definition. Contexts are very useful to deal with our requirements, since they allow to specify RDF conditions without adding new components.

Simple condition

A simple condition $\omega_{\text{simple}}$ is expressed over term of an RDF triple $tp = (x, y, z)$ where $x$ is a subject, $y$ is a predicate and $z$ is an object. Let us take an example to illustrate the approach.

**Example 17** We assume that a secretary is allowed to see salary of employees if its value is less than 45K, and is allowed to see name of all employees. This condition could be expressed as follows:

$$\forall tp = (x, y, z) \quad \omega_{\text{simple}}(tp) = (y = \text{name}) \lor ((y = \text{salary}) \land (z < 45000))$$

The OrBAC rule corresponding to this permission is expressed as follows:

$$\text{Permission}(org, Secretary, Read, ViewEmployee, C_{\omega_{\text{simple}}})$$

such that:

$$\text{Hold}(org, s, \alpha, o, C_{\omega_{\text{simple}}}) \leftarrow \text{Is\_name}(o, N) \lor (\text{Is\_salary}(o, Z) \land Z < 45000).$$

where $\text{Is\_salary}$ (resp. $\text{Is\_name}$) is an application dependent predicate meaning that object $o$ is a salary (resp. a name) with the value $Z$ (resp. $N$).
Complex condition

A complex condition could be also represented as a logical rule corresponding to a context definition.

**Example 18** We assume that a secretary is allowed to see the information of employees who are in the network department and are greater than 25 years old. The complex condition corresponding to that permission is as follow:

\[
\omega_{\text{complex}}(tp) = \begin{cases} 
\text{True} & \text{if } (\exists (a, b) \in E^2)((a = (x, \text{dept}, D)) \land (D = \text{Network}')) \\
& \land (b = (x, \text{age}, A)) \land (A \geq 25) \\
\text{False} & \text{Otherwise}
\end{cases}
\]

The OrBAC rule corresponding to this permission is expressed as follows:

\[
\text{Permission}(\text{org}, \text{Secretary}, \text{Read}, \text{ViewEmployee}, C_{\omega_{\text{complex}}})
\]
such that:

\[
\text{Hold}(\text{org}, s, \alpha, o, C_{\omega_{\text{complex}}}) \leftarrow \text{Employee}_\text{infos}(o, E) \\
& \land \text{Has}_\text{dept}(E, D) \land (D = \text{Network}') \\
& \land \text{Has}_\text{age}(E, A) \land (A \geq 25).
\]

where \(\text{Employee}_\text{infos}\), \(\text{Has}_\text{dept}\) and \(\text{Has}_\text{age}\) are application dependent predicates. \(\text{Employee}_\text{infos}(o, E)\) means that \(o\) is an information of the employee \(E\). \(\text{Has}_\text{dept}(E, D)\) means that the employee \(E\) is in the department \(D\). \(\text{Has}_\text{age}(E, A)\) means that the employee \(E\) has the age \(A\).

Composition of RDF conditions

Let \(\omega_1\) and \(\omega_2\) be two RDF conditions, \(C_{\omega_1}\) and \(C_{\omega_2}\) their corresponding representation as context respectively. The composition of the two RDF conditions \(\omega_1\) and \(\omega_2\) is expressed as a composition of their corresponding context \(C_{\omega_1}\) and \(C_{\omega_2}\). i.e.

\[
C_{\omega_1 \land \omega_2} = C_{\omega_1} \& C_{\omega_2}
\]

\[
C_{\omega_1 \lor \omega_2} = C_{\omega_1} \oplus C_{\omega_2}
\]

**Example 19** Let us take the conjonction of the two conditions \(\omega_{\text{simple}}\) of the example \[17\] and \(\omega_{\text{complex}}\) of the example \[18\]. We assume that a secretary is allowed to see the salary if it is less than 45K and the name, of employees
who are in the network department and are greater than 25 years old. The RDF condition \( \omega \) corresponding to that permission is as follow:

\[
\omega = \omega_{\text{simple}} \land \omega_{\text{complex}}
\]

\( \omega \) is then expressed by the context \( C_\omega = C_{\omega_{\text{simple}}} \land C_{\omega_{\text{complex}}} \). The context definition of \( C_\omega \) is defined as follow:

\[
\text{Hold}(org, s, \alpha, o, C_\omega) \leftarrow \ [Is\_name(o, N) \lor (Is\_salary(o, Z) \land Z < 45000)] \\
\land \ Employee\_infos(o, E) \\
\land \ Has\_dept(E, D) \land (D = 'Network') \\
\land \ Has\_age(E, A) \land (A \geq 25).
\]

The OrBAC rule corresponding to this permission is expressed as follows:

\[
\text{Permission}(org, Secretary, Read, ViewEmployee, C_\omega)
\]

### 5.4.2 RDF condition as view

In this section we present how to model an RDF condition as an OrBAC view.

The OrBAC model allows us to define constraints that manage the assignment of an object to a view. These constraints are expressed as logical conditions using a view definition.

#### Simple condition

Simple RDF conditions depends only on variables of an RDF triple pattern.

Let \( O \) be an ontology. We associate each concept \( C \) of the ontology \( O \) with an OrBAC view \( V_C \). Each property \( P \) of the concept \( C \) is also associated with an OrBAC view \( V_P \) such that \( V_P \) is a sub view of \( V_C \) (\( \text{sub\_view}(V_P, V_C) \)). For instance, figure 5.1 presents an example of a concept and its properties extracted from ontology of employees, and figure 5.2 shows their corresponding OrBAC views.

Let \( \omega \) be a simple condition. We decompose \( \omega \) to a disjunction of simple conditions \( \{\omega_i\}_{1 \leq i \leq n} \) such that each \( \omega_i \) handles only one property \( p_i \) of the ontology \( O \) and

\[
\forall i, j \in [1, n] \quad i \neq j \Rightarrow p_i \neq p_j.
\]

The sentence \( \omega_i \) handles only one property \( p_i \) of the ontology \( O \) means that :

\[
\forall j \in [1, n], \forall tp_j = (x, p_j, z) \in E \quad i \neq j \Rightarrow \omega_i(tp_j) = False.
\]
For each condition $\omega_i$ we create its corresponding OrBAC view $V_{\omega_i}$ as follow:

$$V_{\omega_i} : \begin{cases} V_{\omega_i} = V_{p_i} & \text{if } \forall tp = (x,y,z) \omega_i(tp) = (y = p_i) \\ \text{Sub\_view} (org, V_{\omega_i}, V_{p_i}) & \text{Otherwise} \end{cases}$$

Such that the view definition of $V_{\omega_i}$ is a logical rule that corresponds to $\omega_i$.

We define a new predicate denoted $\text{Sub\_or\_same\_view}$ defined as follow:

$$\text{Sub\_or\_same\_view} (org, V_1, V_2) \leftarrow (V_1 = V_2) \lor \text{Sub\_view} (org, V_1, V_2)$$

So, $\forall i \in [1, n]$ we have $\text{Sub\_or\_same\_view} (org, V_{\omega_i}, V_{p_i})$.

Let us take an example.

**Example 20** The RDF condition $\omega_{\text{simple}}$ presented in the example could be expressed as a disjunction of two RDF conditions $\omega_{\text{simple}} = \omega_{\text{name}} \lor \omega_{\text{salary}}$ such that:

$$\forall tp = (x,y,z) \omega_{\text{name}} (tp) = (y = \text{name})$$

$$\forall tp = (x,y,z) \omega_{\text{salary}} (tp) = (y = \text{salary}) \land (z < 45000)$$
Let $V_{\omega_{\text{name}}}$ and $V_{\omega_{\text{salary}}}$ be respectively OrBAC views corresponding to RDF conditions $\omega_{\text{name}}$ and $\omega_{\text{salary}}$. View definitions of $V_{\omega_{\text{name}}}$ and $V_{\omega_{\text{salary}}}$ are defined as follows:

- $\text{Use}(\text{org}, o, V_{\omega_{\text{name}}}) \leftarrow \text{Is\_name}(o, N)$
- $\text{Use}(\text{org}, o, V_{\omega_{\text{salary}}}) \leftarrow \text{Is\_salary}(o, Z) \land (Z < 45000)$

According to the view definitions above, it is obvious that $V_{\omega_{\text{name}}}$ corresponds to ViewName (i.e. $\text{ViewName} = V_{\omega_{\text{name}}}$) and $V_{\omega_{\text{salary}}}$ is a sub view of ViewSalary. Figure 5.3 shows the relation between $V_{\omega_{\text{name}}}$, $V_{\omega_{\text{salary}}}$ and generated views.

![Figure 5.3: Relation between $V_{\omega_{\text{name}}}$, $V_{\omega_{\text{salary}}}$ and generated views](image)

Thus, if we decide to specify the OrBAC rules using view definition, then we shall obtain the two following rules:

- Rule_{name} : $\text{Permission(}\text{org, Secretary, Read, ViewName, Nominal)}$
- Rule_{salary} : $\text{Permission(}\text{org, Secretary, Read, V_{\omega_{\text{salary}}}, Nominal)}$

**Complex condition**

In the case of complex conditions $\omega$, we create a new OrBAC view where the view definition is the logical rule corresponding to $\omega$. Let us take an example to illustrate this case.

**Example 21** We consider the complex condition $\omega_{\text{complex}}$ presented in the example 18. Let $V_{\omega_{\text{complex}}}$ be its corresponding OrBAC view. View definition of $V_{\omega_{\text{complex}}}$ is defined as follow:
Use\( (org, o, V_{\omega_{\text{complex}}}) \leftarrow \text{Employee\_infos}(o, E) \)
\( \land \text{Has\_dept}(E, D) \land (D = 'Network') \)
\( \land \text{Has\_age}(E, A) \land (A \geq 25). \)

Figure 5.4 shows the relation between \( V_{\omega_{\text{complex}}} \) and generated views. The OrBAC rule corresponding to \( \omega_{\text{complex}} \) is as follows:

\( \text{Rule}_{\omega_{\text{complex}}} : \text{Permission}(org, Secretary, Read, V_{\omega_{\text{complex}}}, Nominal) \)

such that \( Nominal \) is the context that is always true.

![ViewEmployee](ViewEmployee.png)

Figure 5.4: Relation between \( V_{\omega_{\text{complex}}} \) and generated views

Complex condition as view has some limitation. We can only represent a complex condition that depends only on the object \( o \). For instance, the condition represented by the context \( Ctx_{\omega} \) on section 5.3 could not be expressed as a view because it depends on the subject \( s \) which is not part of the view definition.

### 5.4.3 RDF condition as view and context

In section 5.4.1 we present how to model RDF conditions as context. In section 5.4.2 we present how to model simple condition as view. We explain also that some complex conditions could not be represented as views.

In the rest of this chapter we assume that a complex condition is only expressed as context. However, simple condition could be expressed as context and/or as view. For instance, if we consider the RDF condition \( \omega = \omega_{\text{simple}} \land \omega_{\text{complex}} \) presented in the example 19. \( \omega \) could be expressed with the two following OrBAC rules:

\( \text{Rule}_1 : \text{Permission}(org, Secretary, Read, ViewName, C_{\omega_{\text{complex}}}) \)

\( \text{Rule}_2 : \text{Permission}(org, Secretary, Read, V_{\omega_{\text{salary}}}, C_{\omega_{\text{complex}}}) \)
5.5 Rewriting Query Instrumented by OrBAC rules

In this section we present how to instrument a rewriting algorithm \( f_{\text{Query}} \) using security rules based on the OrBAC model \cite{OrBAC}. In our case we only have two actions ‘read’ and ‘write’. The read action corresponds to SELECT query and the write action corresponds to UPDATE query.

For a given SPARQL query, the subject \( s \) is the requestor. The action \( \alpha \) corresponds to the query type. For instance, action ‘read’ for SELECT query and action ‘write’ for UPDATE query. Views can be retrieved by analyzing the clause WHERE of the query. It corresponds to the concepts and/or properties of a given concept that are queried. Each property \( p_i \) of the where clause, corresponds to the view \( V_{\omega_{p_i}} \).

We define a new predicate \( \text{When\_permitted} \) which allows us to get view definitions and context definitions corresponding to a given subject \( s \), action \( \alpha \) and view \( V \).

\( \text{When\_permitted} \) is a predicate over domains \( \text{Org}x\text{SxAxVxDxDefxDef} \) defined as follows:

\[
\text{When\_permitted}(\text{org}, s, \alpha, V, DCtx, DView) \leftarrow \text{Empower}(\text{org}, s, R) \\
\land \quad \text{Consider}(\text{org}, \alpha, A) \\
\land \quad \text{Sub\_or\_same\_view}(\text{org}, V_1, V) \\
\land \quad \text{Permission}(\text{org}, R, A, V_1, C) \\
\land \quad DCtx \\
\land \quad DView
\]

such that \( DCtx \) is the context definition of a given context \( C \). \( DView \) is the view definition of the given view \( V_1 \). The RDF condition will be the conjunction of context definition of \( C \) and view definition of \( V_1 \) i.e. \( \omega = DCtx \land DView \).

For a given view \( V \), \( \text{When\_permitted} \) allows us to get context definitions and view definitions of corresponding permission defined on the view \( V \) or sub view of \( V \). For instance, if we consider OrBAC rules \( \text{Rule}_1 \) and \( \text{Rule}_2 \) presented in section \ref{sec:OrBAC-rules}, we deduce that:
• for the property name of the employee ontology i.e. \( V = V_{name} = ViewName \): 
  
  \( \text{When\_permitted}(org, s, \alpha, ViewName, DCtx, DView) \) unifies \( DCtx \) to context definition of \( C_{\omega_{\text{complex}}} \) and \( DView \) to view definition of \( ViewName \).

• for the property salary of the employee ontology i.e. \( V = V_{salary} = ViewSalary \): 
  
  \( \text{When\_permitted}(org, s, \alpha, ViewSalary, DCtx, DView) \) unifies \( DCtx \) to context definition of \( C_{\omega_{\text{complex}}} \) and \( DView \) to view definition of \( V_{\omega_{\text{salary}}} \).

Let \( C \) be a concept of the ontology \( \mathcal{O} \) and \( \{ p_{i} \}_{1 \leq i \leq n} \) be a set of properties of \( C \). We have

\[
\forall i \in [1, n] \quad \text{Sub\_or\_same\_view}(org, V_{p_{i}}, V_{C})
\]

So, we can get constraints defined for all properties \( \{ p_{i} \}_{1 \leq i \leq n} \) of a given concept \( C \) by using \( \text{When\_permitted} \) with the view \( V = V_{C} \) corresponding to \( C \).

For instance, for the \( \text{Employee} \) concept and based on the OrBAC rules Rule1 and Rule2, \( \text{When\_permitted}(org, s, \alpha, ViewEmployee, DCtx, DView) \) unifies \( DCtx \) to context definition of \( C_{\omega_{\text{complex}}} \) and \( DView \) to view definitions \( \{ ViewName, V_{\omega_{\text{salary}}} \} \).

Afterwards, we parse the final logical rules in order to generate the corresponding RDF condition. Then that condition and the initial query are used as inputs of the query rewriting algorithm \( fQuery \).

### 5.6 Conclusion

In this chapter, we present an approach that instruments a rewriting query algorithm using the access control model OrBAC. The constraints used by the algorithm of query modification are expressed as OrBAC contexts using logical rules. Simple condition could be also represented as view. This approach remains valid for all rewriting query algorithm which are based on constraints that could be modeled in the form of logical rules. An advantage of our approach is that the policy is stored independently from a specific data model unlike \([66, 69]\). For example, in the case of an heterogeneous database we only have to express our security policy in a single security model like OrBAC or RBAC \([57]\), which facilitates security policy management. Another advantage is that the access control policy is not expressed only for rewriting query algorithm. It is syntactically independent from the algorithm. The process of rewriting query is responsible for translating policy constraints to the necessary format (e.g. to SPARQL filters). Which means that the same access control policy could be used by other services.
CHAPTER 6

Privacy policy preferences enforced by SPARQL Query Rewriting

6.1 Introduction

Privacy is the right of individuals to determine for themselves when, how and to what extent information about them is communicated to others\footnote{Alan Westin, Professor Emeritus of Public Law and Government, Columbia University}. The data subject (\textit{data owner}) is the one who specifies the privacy preferences, i.e. it decides how its data should be used (\textit{purposes}), to whom it may be disclosed (\textit{recipients}) and under which accuracy (anonymization, obfuscation, etc.).

Privacy principles express the conditions under which a requestor can access private information. We explore existing privacy directives and laws \cite{82, 83, 84, 85, 86, 87} to derive privacy principles, such as consent. \cite{9} specifies the most relevant privacy requirements which are compliant with the current legislations in the field of privacy protection \cite{82, 83, 84, 85, 86, 87}. These requirements are the consent, accuracy, provisional obligations and purposes. The \textit{consent} is the user agreement for accessing and/or processing his/her data. It is the requirement before delivering the personal data to third parties. The \textit{accuracy} is the level of anonymity and/or the level of the accuracy of the data. The \textit{provisional obligations}, refer to the actions to be taken by the requestors after the access (usage control). The \textit{purpose} is the goal that motivates the access request.

In this chapter we present an approach that enforces the privacy policy preferences by query transformation. We then present how to instrument this rewriting query
algorithm using a privacy-aware model like PrivOrBAC (see chapter 7 for more details). We take into account various dimensions of privacy preferences through the concepts of consent, accuracy, purpose and recipient.

The goal of our approach is to enforce the privacy requirements by SPARQL query rewriting. That rewriting algorithm is instrumented by a privacy-aware model. Without any loss of generality, we propose to use the privacy-aware OrBAC model PrivOrBAC [9] to express privacy policies.

In the literature there are two categories of approaches that handle the privacy dimension. The first one aims to define a model or language to express privacy requirements. For instance, models such as P-RBAC [88], Purpose-BAC [89] and PR-RBAC [90] define new languages to express access contexts, and they focus on purpose entity and on other privacy requirements [83]. PrivOrBAC [9] extends the OrBAC model [8]. It reuses most of existing mechanisms implemented in OrBAC to express privacy requirements.

The second category aims to apply security requirements on data and/or using their own new security model for a specific kind of database, for instance, LeFevre et al. [69] and Huey [67]. [69] presents an approach that enforces limited disclosure expressed by privacy policy in the case of Hippocratic databases. They store privacy requirements in relational tables in the same database where the data to be protected is stored. Then they enforce those requirements using a query rewriting approach. [67] is based on the Oracle Virtual Private Database (VPD) and supports fine-grained access control (FGAC) through policy functions. A function is associated with the table (or view) that needs protection. When it is invoked, it returns various pieces of SQL, called predicates, depending on the system context (e.g. current time, current user, etc.) to enforce FGAC. Policy functions are expressed in PL/SQL. Their approach is based on query modification. When a query is issued, it is dynamically modified by appending predicates, returned by the policy function, to the where clause of the query.

Regarding the enforcement of privacy requirements, our approach belongs to the second category. But, in our case, requirements specification is based on an existing privacy-aware model independently from the data. Thus, we avoid empiricity, manage conflicts and can move to another privacy model when needed. We choose Priv-OrBAC as it takes into account most of the privacy recommendations specified by well known standards [83, 84, 85, 86].

Our contribution aims to enforce privacy policy preferences with the following features:
6.2. APPROACH PRINCIPLE

- Enforcing privacy policy does not require any modification to existing databases viewed as RDF graphs.

- Use of an existing privacy-aware model to store and manage privacy policy preferences.

- The following privacy requirements are taken into account: consent, purpose, recipient and accuracy.

- In the case of distributed systems, we only have to manage our privacy policy preferences in a given system as SPARQL service (endpoint).

In our approach, the answer to the rewritten query may differ from the result of the user’s initial query. In that case and as suggested in [75], we can check the query validity of the rewritten query with respect to the initial query and notify the user when the answer to the query is not complete.

The rest of this chapter is organized as follows. Section 6.2 presents an overview of our approach. Section 6.3 presents an example of privacy preferences ontology used to illustrate our approach. Section 6.4 introduces some criteria that should be satisfied by our rewriting algorithm. Section 6.5 presents the principle of our rewriting algorithm. Finally, section 6.6 concludes this chapter.

6.2 Approach principle

A SPARQL query consists of triple patterns, conjunctions, disjunctions, and optional patterns. SPARQL allows users to write globally unambiguous queries. For example, the following query returns the name of all patients and their drug name.

```
PREFIX dt:<http://hospital.fr/patients/>
SELECT ?name ?drugName
WHERE {
}
```

Basically, the SPARQL syntax looks like SQL, but the advantage of SPARQL is that it enables queries spanning multiple disparate (local or remote) data sources containing heterogeneous semi-structured data.

The principle of our approach is to rewrite the initial SPARQL query in order to protect personal data from unauthorized access. The privacy policy preferences is
expressed and stored using the PrivOrBAC model. We take into account the concepts of consent, accuracy, purpose and recipient.

Figure 6.1 shows an overview of the approach. The SPARQL Rewriting Engine is the component that implements the rewriting algorithm. It takes as input (i) the initial SPARQL query, (ii) the preferences ontology OWLprivacy and (iii) a table mapping $M$. (ii) and (iii) are explained in detail in section 6.3. SPARQL Rewriting Engine adds security constraints (SPARQL conditions, filters and services) to the initial query such that the returned result is compliant with the privacy policy defined by each data-owner. The output is the rewritten SPARQL query denoted as $Q_{rw}$.

Let us first show a privacy unaware execution of a SPARQL query. We assume that doctor Alice tries to get the name of all patients, and patient Charlie does not want to disclose her information, including her name, to doctor Alice. Alice issues the following query $Q_i$:

```sparql
PREFIX dt:<http://hospital.fr/patients/>
SELECT ?name
WHERE {
}
```

The initial query $Q_i$ is executed by the SPARQL engine which evaluates it (with no modifications and no filters) and gets the corresponding result from the queried RDF data sources. Of course the returned result contains Charlie’s name.
PrivOrBAC proposes a set of web services used to access the privacy policy preferences. Using these web services we build our *PrivOrBAC SPARQL service* by implementing the approach proposed in [91]. In [91] the received SPARQL query is decomposed into a set of web services that covers the initial query requirements. *PrivOrBAC SPARQL service* is a server that receives a SPARQL query expressed in OWL$_ {privacy}$. It decomposes this SPARQL query into a set of PrivOrBAC web services that will be invoked later. The collected result is correctly merged, filtered and formatted, then transferred to the requestor (see chapter 7). In the rest of this chapter we assume that the privacy policy is accessed through SPARQL service based on preferences ontology OWL$_ {privacy}$ presented in section 6.3. That is, the privacy policy is viewed as RDF data.

In the case of privacy-aware model that does not provide web services, there exist other approaches that allow building SPARQL services. For instance [92] and [93] could be used when policies are stored in XML format.

Figure 6.2 illustrates the execution of the rewritten query $Q_{rw}$. It can be summarized as follows:

- SPARQL rewriting engine rewrites the initial query by inserting (i) a call to the service of policy preferences and (ii) inserting corresponding security constraints.
- SPARQL engine executes the rewritten query $Q_{rw}$. So, that execution will get the data from 'RDF databases' and the corresponding policy preferences from the service of preferences that is injected by the rewriting engine. Then it applies corresponding security filters and conditions to the query.

### 6.3 Privacy-aware Ontology

In this section we define an ontology of privacy preferences. This ontology allows us to query the privacy policy preferences of PrivOrBAC via SPARQL as explained in section 6.2. This ontology is independent from the data structure. It is denoted OWL$_ {private}$. Obligations [9] are not taken into account by this ontology. We handle the concepts of consent, accuracy, purpose and recipient as suggested in [9]. In the rest of the chapter, we use the prefix $P$ (resp. $dt$) for preference ontology (resp. data ontology). Figure 6.3 presents this ontology. It is composed of three classes:

- **P:Dataowner**: represents the data-owner, he/she is identified by the property $P:hasId$. Each data-owner has a set of preferences $P:Preference$ via the object...
property $P:\text{hasPreference}$. We can express this in our ontology, using the N3 syntax\(^2\) as:

\[
\begin{align*}
P:DataOwner & \text{rdf:type owl:Class}; \\
P:hasId & \text{rdf:type owl:DatatypeProperty,} \\
& \quad \text{owl:FunctionalProperty}; \\
& \quad \text{rdfs:domain P:DataOwner}; \\
& \quad \text{rdfs:range xsd:string}. \\
P:hasPreference & \text{rdf:type owl:ObjectProperty}; \\
& \quad \text{rdfs:domain P:DataOwner}; \\
& \quad \text{rdfs:range P:Preference}. \\
\end{align*}
\]

\(^{2}\text{http://www.w3.org/TeamSubmission/2008/SUBM-n3-20080114/}\)
• **P:Preference**: represents preferences associated with triples (data-owner, recipient, purpose). Each preference has a set of targets **P:Target** via the object property **P:hasTarget**. **P:hasPurpose** corresponds to the purpose associated with this preference. **P:hasRecipient** represents the *Requestor* targeted by this preference.

\[
P:\text{Preference} \text{ rdf:type owl:Class};
P:\text{hasTarget} \text{ rdf:type owl:ObjectProperty};
\text{rdfs:domain P:Preference;}
\text{rdfs:range P:Target}.
P:\text{hasPurpose} \text{ rdf:type owl:DatatypeProperty,}
\text{owl:FunctionalProperty;}
\text{rdfs:domain P:Preference;}
\text{rdfs:range xsd:string}.
P:\text{hasRecipient} \text{ rdf:type owl:DatatypeProperty,}
\text{owl:FunctionalProperty;}
\text{rdfs:domain P:Preference;}
\text{rdfs:range xsd:string}.
\]

• **P:Target**: represents a particular preference item. It is composed of three properties. (1) **P:hasName** represents the name of a data item, e.g. age, name, address, etc. (2) **P:hasDecision** represents the data-owner choice (consent) associated with that target, e.g. Yes to indicate that the data-owner agrees to disclose the value of that target to (resp. for) the corresponding recipient (resp. purpose) of the preference, No otherwise. (3) **P:hasAccuracy** represents the accuracy that will be applied to that target in the case of positive consent (decision=Yes), e.g. Anonymization, Nullification, etc.

\[
P:\text{Target} \text{ rdf:type owl:Class};
P:\text{hasName} \text{ rdf:type owl:DatatypeProperty,}
\text{owl:FunctionalProperty;}
\text{rdfs:domain P:Target;}
\text{rdfs:range xsd:string}.
P:\text{hasAccuracy} \text{ rdf:type owl:DatatypeProperty;}
\text{rdfs:domain P:Target;}
\text{rdfs:range xsd:string}.
P:\text{hasDecision} \text{ rdf:type owl:DatatypeProperty,}
\text{owl:FunctionalProperty;}
\text{rdfs:domain P:Target;}
\text{rdfs:range xsd:string}.
\]
Figure 6.3: Privacy Preferences Ontology

Figure 6.4 shows an example of privacy preferences defined by the data-owner Safaa for the purpose _purpose_1 and the recipient _Bob_. She decided to disclose her name in clear (without accuracy) and her age after k-anonymization \[94\] with \( k = 15 \).

Let \( OWL_{data} \) be the data ontology or the ontology on which the initial query is expressed. Properties of \( OWL_{data} \) are associated with the PrivOrBAC ontology \( OWL_{privOrBAC} \) defined above, via a mapping table \( M \). It is defined as follows: each property of \( OWL_{data} \) (i.e. \( dt:name \), \( dt:age \)) is mapped to a string value (‘name’, ‘age’ resp.) which is the value of the property \( P:hasName \) of an instance of the class \( P:Target \). We denote this mapping table as \( M \). For instance, in figure 6.4, ‘\_:t1’ is an instance of the class \( P:Target \). It corresponds to the preference defined by the data-owner for the property \( dt:age \).

The mapping table \( M \) represents also a mapping between values of the Target attribute defined in the Consent_preference view, and \( OWL_{data} \) properties.
6.4 The correctness criteria

Before presenting our rewriting algorithm, we will first introduce some criteria that should be satisfied by any rewriting algorithms.

Wang et al. [68] proposed a formal notion of correctness for fine-grained access control in relational databases. They presented three correctness criteria (sound, secure and maximum) that should be satisfied by any query processing algorithm in order to be “correct”.

Figure 6.4: Privacy Preferences Example
Soundness: An algorithm is sound if and only if its rewritten query $Q_{rw}$ returns only correct answers i.e. answers to the initial query.

Maximality: An algorithm is maximum if and only if $Q_{rw}$ returns as much information as possible.

Security: An algorithm is secure if and only if the result of $Q_{rw}$ respects the security and privacy policy of the queried system.

In the rest of this section, we present a condition that should be satisfied by our algorithm in order to satisfy the soundness, maximality and security criteria [68].

Let $A$ be a query processing algorithm that enforces privacy policies. Let $D$ be a database, $P$ a disclosure policy and $Q$ a query. Let $Q' = A(P, Q)$ be the rewriting query of $Q$ using the policy $P$. We denote $R = A(D, P, Q)$ the output result of $Q'$ on $D$.

Definition 1: Given two tuples $t_1 = (x_1, x_2, ..., x_n)$ and $t_2 = (y_1, y_2, ..., y_n)$, we say that $t_1$ is subsumed by $t_2$, denoted $t_1 \sqsubseteq t_2$, if and only if $\forall i \in [1..n]: (x_i = y_i) \vee (x_i = \text{unauthorized})$.

Definition 2 [68]: Given two relations $R_1$ and $R_2$, we say that $R_1$ is subsumed by $R_2$, denoted $R_1 \sqsubseteq R_2$, if and only if:

$$(\forall t_1 \in R_1)(\exists t_2 \in R_2)|t_1 \sqsubseteq t_2$$

Definition 3 [68]: Two databases states $D$ and $D'$ are “equivalent” with respect to policy $P$ (denoted as $(D \equiv_p D')$ if the information allowed by $P$ in $D$ is the same as that allowed by $P$ in $D'$.

Let $S$ denote the standard query answering procedure and $S(D, Q)$ the result of the query $Q$ in the database state $D$ without any privacy restriction. [68] formalizes the three criteria as follows: a query processing algorithm $A$ is:

- sound if and only if $\forall P \forall Q \forall D \quad A(D, P, Q) \sqsubseteq S(D, Q)$
- secure if and only if $\forall P \forall Q \forall D \forall D' \quad [((D \equiv_p D') \implies (A(D, P, Q) = A(D', P, Q))]
- maximum if and only if $\forall D' \forall R$
  
  if $[(D \equiv_p D') \land (R \sqsubseteq S(D', Q))]$ then we have $R \sqsubseteq A(P, D, Q)$
6.5 Rewriting Algorithm principle

For a given SPARQL query, we assume that we have the associated requestor (recipient) and purpose. We assume that there is only one data-owner for each data item. We do not handle the case where several data-owners share one data item. This issue is not handled by PrivOrBAC.

The principle of our approach is summarized in the following items. For each property of the where clause of the initial query:

1. we normalize the triple pattern corresponding to that property (Algorithm 3)
2. we get its associated choice \((P:\text{hasDecision})\) and accuracy \((P:\text{hasAccuracy})\) for each data-owner.
3. we apply the corresponding choice and accuracy using algorithm 4.

6.5.1 Normalization of triple patterns

The normalization is applied to extract implicit filters in SPARQL queries. It transforms the implicit filter into an explicit one with respect to the semantic of the initial query.

The algorithm [3] aims to normalize a triple pattern \(tp\), i.e. transforms \(tp\) to a new triple pattern \(tp_{\text{norm}}\) where its object is a SPARQL variable. For example the normalization of the triple pattern \({?p \ dt:age\ 25}\) is the pattern \({?p \ dt:age\ ?age.\ Filter(?age=25)}\).

**Algorithm 3** Normalize a triple pattern

**Require:** Triple pattern \(tp \leftarrow (s, p, o)\)

1: Let \(tp_{\text{norm}}\) be a triple pattern such that \(tp_{\text{norm}} \leftarrow tp\)
2: if the object \(o\) of \(tp\) is not a variable then
3: Let \(var\) be a SPARQL variable
4: \(tp_{\text{norm}} \leftarrow (s, p, ?var)\)
5: \(tp_{\text{norm}} \leftarrow tp_{\text{norm}}.\ Filter(?var = o)\)
6: end if
7: return \(tp_{\text{norm}}\)

For a given property \(prop\) of a data-owner \(do\), algorithm 4 returns the new value of \(prop\) that \(do\) wants to return to the requester (recipient), by applying its corresponding
choice and accuracy. It checks if the choice (decision) is negative then it returns a null value. Otherwise, if the accuracy is defined then it returns the value of prop by applying the corresponding accuracy, using the function apply(value, accuracy). If the accuracy is not defined, it returns the original value of prop.

Algorithm 4 Apply the choice and its accuracy on a property value

Require: choice, accuracy, value
1: if choice = 'No' then
2:    return null
3: end if
4: if accuracy is bound to a value then
5:    return apply(value, accuracy)
6: end if
7: return value

6.5.2 Preferences acquisition

Before defining the general algorithm we will first introduce some useful methods that are used by the general rewriting algorithm.

The second step of our algorithm, after the normalization step, is to insert a call to the privacy service in order to get preferences of each data-owner. The service block corresponding to that call, denoted as ServiceBlock($purpose, $recipient), depends on the purpose $purpose and recipient $recipient of the initial query. This service block is defined as follow.

ServiceBlock($purpose, $recipient) = SERVICE ps:preferences {
    ?dp rdf:type P:DataOwner;
    P:hasId ?id;
    P:hasPreference ?pref.
    ?pref P:hasPurpose $purpose;
    P:hasRecipient $recipient;
}

This block of service SB returns all data-owner’s preferences defined for the given purpose $purpose and recipient $recipient. To filter preferences for specific targets, we add corresponding triples to that service block. For that, we define a method, denoted addTarget(SB:ServiceBlock, tn:Value), that takes as parameter a service block
6.5. REWRITING ALGORITHM PRINCIPLE

SB and the name of the target tn. The set of triples added to the block SB are to get
the decision and accuracy of the given target name tn. The algorithm 5 presents the
definition of the addTarget method. For example, In order to get preferences of the

target ‘name’ defined for the purpose $purpose and the recipient $recipient, we call the
method addTarget(SB, ‘name’). The new value of SB is as follows:

```sparql
SERVICE ps:preferences {
?dp rdf:type P:DataOwner;
  P:hasId ?id;
  P:hasPreference ?pref.
?pref P:hasPurpose $purpose;
  P:hasRecipient $recipient;
  P:hasTarget ?tname.
?tname P:hasName ‘name’;
  P:hasDecision ?nameDecision.
OPTIONAL {?tname P:hasAccuracy ?nameAcc}
}
```

6.5.3 Preferences enforcement

The third step is to apply the data-owner preferences (decision and accuracy), for
each target specified in the query, by using algorithm 4. This step aims to nullify
unauthorized data and to apply the corresponding accuracy, if defined, for authorized
ones. In our algorithm we handle a Basic Graph Pattern $Bgp$ of the initial
SPARQL query.

Let $Bgp$ be a basic group of pattern, tn be a target name and $tp = (s, p, o)$ be
the triple pattern of $Bgp$ that corresponds to the target tn. The method addBind
(algorithm 6) aims to assign the authorized value to the variable o of $tp = (s, o, p)$
after applying the corresponding privacy preferences. The IF keyword is one of the
SPARQL 1.1 operators [96]. It is defined as follows:

---

3A simple set of SPARQL triples

\textbf{Require:} BGP $Bgp$, target name $x$ and normalized triple pattern $tp$

1. Let $o$ be the object variable of the triple $tp = (s, p, o)$
2. Let $o'$ be a SPARQL variable such that $o \neq o'$
3. We rename the variable $o$ of $tp$ by $o'$
4. Let $C_1$ and $C_2$ be the two conditions defined as follow:
   5. $C_1 = (?x\text{Decision} = \text{"No"})$
   6. $C_2 = \text{bound}(?x\text{Accuracy})$
5. Let $V_1$, $V_2$, $V_{\text{accuracy}}$ and $V_{\text{bind}}$ be the values defined as follow:
   6. $V_1 = \text{null}$ /*null to design unauthorized value*/
   7. $V_{\text{accuracy}} = \text{eval}(o', ?x\text{Accuracy})$
   8. $V_2 = \text{IF}(C_2, V_{\text{accuracy}}, o')$
   9. $V_{\text{bind}} = \text{IF}(C_1, V_1, V_2)$
10. Let $B = \text{Bind}(V_{\text{bind}} \ AS \ o)$
11. add the bind $B$ to the BGP $Bgp$

\texttt{rdfTerm IF(expr1, expr2, expr3)}

It evaluates the first argument $expr1$, interprets it as a boolean value, then returns the value of $expr2$ if the boolean value is true, otherwise it returns the value of $expr3$. Only one of $expr2$ and $expr3$ is evaluated. For instance, the value of $V_{\text{bind}}$ (line 11) is $V_1$ if the condition $C_1$ is satisfied, otherwise $V_{\text{bind}} = V_2$.

The \texttt{Bind} keyword is an explicit assignment of variables. It is included in SPARQL1.1 with the syntax:

\texttt{BIND(expr AS ?var)}

The value of the expression ‘$expr$’ will be assigned to the SPARQL variable ‘?\text{var}’.

The SPARQL function \texttt{eval(value, accuracy)} is an implementation of the function \texttt{apply} used in algorithm 4.

The rewriting algorithm 7 aims to build the service block $SB$ of the initial query, based on algorithm 5. Then, it transforms each basic group of pattern, of the initial query, based on algorithm 6.

In the rest of this section we will start by analyzing the case of SPARQL query without filters after normalization. Then we will see the case of SPARQL query with filters.
Algorithm 7 Rewriting Query Algorithm

 Require: purpose Pur, recipient Rec, query Q
 1: Let $Q'$ be the normalized query of $Q$
 2: Let $SB = ServiceBlock(Pur, Rec)$
 3: for each basic group of pattern $Bgp$ do
 4: Let $Bgp'$ be its equivalent in $Q'$
 5: for each triple pattern $tp = (s, p, o)$ of $Bgp$ do
 6: Let $tp' = (s, p', o')$ be the equivalent triple of $tp$ in $Bgp'$
 7: if the property $p$ has mapping in $M$ then
 8: Let $x$ be the mapping of $p$ in $M$
 9: addTarget($SB$, $x$)
 10: addBind($Bgp'$, $x$, $tp'$)
 11: end if
 12: end for
 13: end for
 14: add the service bloc $SB$ to $Q'$
 15: return $Q'$

6.5.4 SPARQL query without filter

Let us take a simple example to illustrate our approach. Bob tries to select the name and the age of all patients for the purpose $purpose_1$. He issues the following query:

```
PREFIX dt:<http://hospital.fr/patients/>
SELECT ?name ?age
FROM dt:infos
WHERE {
  ?p rdf:type dt:Patient;
  dt:name ?name;
  dt:age ?age.
}
```

We assume that in the mapping table $\mathcal{M}$, the property $dt:name$ (resp. $dt:age$) corresponds to the string value ‘name’ (resp. ‘age’). So, The transformed query is as follows:

```
SELECT ?name ?age FROM dt:infos WHERE {
  ?p rdf:type dt:Patient;
  dt:id ?id;
SERVICE ps:preferences {
  ?dp rdf:type P:DataOwner; P:hasId ?id;
  P:hasPreference ?pref.
  pref P:hasPurpose 'purpose_1';
  P:hasRecipient 'Bob';
  P:hasTarget ?tp1, ?tp2.
}
```
Lines 5 to 17 correspond to a call to the service of privacy preferences. In this clause we get the choice and its accuracy of properties $dt:name$ and $dt:age$ corresponding respectively to 'name' and 'age' (lines 11 to 16) for the triple (data-owner, purpose_1, Bob) (lines 6, 8, 9).

Lines 18 to 20 (resp. 21 to 23) correspond to algorithm 6 for the property $dt:name$ (resp. $dt:age$). The function $udf:eval(x,y)$ takes two parameters $x$ and $y$. The first parameter $x$ is a value and the second parameter is the accuracy type to be applied to the value $x$. This function returns the result of applying the accuracy $y$ to the value $x$.

Table 6.1 shows an example of privacy policy preferences of the properties name and age for the purpose purpose_1 and the recipient Bob. The value 'anonym' (resp. 'k_anonym, k=15') of the accuracy means that the value of the corresponding target must be anonymized (resp. k-anonymized where k=15). Table 6.2 shows an example of values of the properties name and age before and after applying the algorithm 4 based on the preferences given in table 6.1. For instance, the value 'axfd14' (resp. [28,43]) is the anonymization (resp. k-anonymization with k=15) of the value 33 (resp. 30). Finally, table 6.3 shows the result of the user query, presented in the example above, before and after query transformation.

In the rest of this section we will study the case of a query with one triple, then the case of a query with many triples. Let $D$ be a database state, $P$ be a disclosure policy, $tp = (s,pred,o)$ be a triple pattern where o is a variable, and $var(tp)$ be all variables defined in tp. We denote $Q^D_{tp}$ the query defined as follow:

$$\text{SELECT } var(tp) \text{ FROM } D \text{ WHERE } \{s \ pred \ o\}$$

The result of $Q^D_{tp}$ represents all triples that match with $tp$.

If the predicate $pred$ does not have a mapping in $M$ then the rewritten query of $Q^D_{tp}$ is the same as $Q^D_{tp}$ (line 7 of algorithm 7). Otherwise, the rewritten query is as follow:
### 6.5. Rewriting Algorithm Principle

<table>
<thead>
<tr>
<th>ID</th>
<th>Target</th>
<th>Choice</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>age</td>
<td>Yes</td>
<td>anonym</td>
</tr>
<tr>
<td>2</td>
<td>age</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>age</td>
<td>Yes</td>
<td>k_anonym, k=15</td>
</tr>
<tr>
<td>4</td>
<td>age</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>name</td>
<td>Yes</td>
<td>anonym</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td>Yes</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: Preferences of name and age properties for (purpose_1, Bob)

<table>
<thead>
<tr>
<th>ID</th>
<th>age</th>
<th>Algo. 4</th>
<th>ID</th>
<th>name</th>
<th>Algo. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33</td>
<td>axfd14</td>
<td>1</td>
<td>Alice</td>
<td>bob15s</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>null</td>
<td>2</td>
<td>Charlie</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>[28,43]</td>
<td>3</td>
<td>Safaa</td>
<td>Safaa</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>27</td>
<td>4</td>
<td>Said</td>
<td>Said</td>
</tr>
</tbody>
</table>

Table 6.2: Values of name and age properties after and before applying algorithm 4

<table>
<thead>
<tr>
<th>Before Transformation</th>
<th>After Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>age</td>
</tr>
<tr>
<td>Alice</td>
<td>33</td>
</tr>
<tr>
<td>Charlie</td>
<td>42</td>
</tr>
<tr>
<td>Safaa</td>
<td>30</td>
</tr>
<tr>
<td>Said</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 6.3: Result of the query before and after transformation

```
SELECT var(tp) FROM D
WHERE { SB.
s pred o’. BIND(f_P,pred(o’) AS o)}
```

$SB$ corresponds to the service block and $f_P,pred(o')$ corresponds to $V_{bind}$ which was defined in algorithm 6. $V_{bind}$ can take three possible values: (i) null for negative decision, (ii) accuracy$(o')$ for positive decision with the accuracy accuracy and (ii) $o'$ otherwise. The result $A(D, P, Q^D)$ of the rewritten query corresponds to $S(D, Q^D)$ by replacing values of $o$ by $f_P,pred(o)$. So, for all $t_1 = (x_1,..., x_n)$ of $A(D, P, Q^D)$ there exists an element $t_2 = (y_1,..., y_n)$ of $S(D, Q^D)$ such that $\forall i \in [1..n] x_i = y_i$ or $x_i = unauthorized$ where unauthorized denotes null or accuracy$(o')$. We deduce that in that case the soundness property is satisfied.
Let $Q^D_2$ be a SPARQL query with two triples $tp_1 = (s_1, pred_1, o_1)$ and $tp_2 = (s_2, pred_2, o_2)$. If $pred_1$ and $pred_2$ do not have a mapping in $M$ then the rewritten query of $Q^D_2$ is the same as $Q^D_2$. Otherwise, we can write $Q^D_2$ as a join of two SPARQL queries with one triple, as follow:

$$Q^D_2 = Q^D_{tp_1} \bowtie Q^D_{tp_2}$$

If the join is based on the subject part of triples $tp_1$ and $tp_2$ i.e. $s_1 = s_2 = s$ then there is no impact for the join of the result of the rewriting queries of $Q^D_{tp_1}$ and $Q^D_{tp_2}$. If the join is based on the object part of $tp_1$ and subject part of $tp_2$ i.e. $o_1 = s_2 = o$ then the join of the result of the rewritten queries is based on $f_{P,pred_1}(o)$ and $o$. Since $f_{P,pred_1}(o_1)$ is not generally equal to $s_2$ then $A(D, P, Q^D_2)$ may contain unsound result.

The problem of join occurs, in the case of SPARQL query $Q$, when the where clause of $Q$ contains an object property that has a mapping in $M$.

By induction we can generalize the result bellow for SPARQL query composed of a set of triples.

### 6.5.5 SPARQL query with filters

SPARQL query may contain filters. Filters are restrictions on solutions over the whole group in which they appear. For instance, a query that gets name and age of patients who are more than 25 years old, is expressed as follows:

```sparql
SELECT ?name ?age 
FROM dt:infos 
WHERE {
  ?p rdf:type dt:Patient;
  dt:name ?name;
  dt:age ?age. 
FILTER(?age >= 25)
}
```

As explained in section 6.5, the result of the query could be heterogeneous (nullified, anonymized, etc.) after applying the policy preferences. There are two different strategies for query rewriting. The first one is to execute filters before applying the privacy policy preferences. It is obvious that this case is not secure. For instance, a malicious user could issue a query that returns the name and age of patients who are 25 years old. The returned result corresponds exactly to patients who are 25 years old even if the value of the age is hidden.

The second strategy is to execute filters after applying the privacy policy preferences. In this case, filters are executed on heterogeneous blinded data. Thus the returned result could not be sound. For instance, Bob tries to select patients who live in Rennes. We assume that Alice lives in Paris and she chooses to disclose her city, to Bob, after anonymization to ‘Rennes’. The result of the query issued by Bob will contain Alice
who is living in Paris. The returned result could also be not maximum. For example, based on the data given in tables 6.1 and 6.2, the result of the query above after transformation is obtained by applying the SPARQL filter $\text{FILTER}(\text{?age}\geq 25)$ on the data of table 6.3 after transformation. Table 6.5 shows the result of the query after transformation and table 6.4 shows the expected result. As we can see the result is not maximum. The value of the age of Safaa is in the interval $[28, 43]$ (a $k$-anonymization of 30 with precision $k = 15$) which is always greater than 25. One way to correct this problem, is to extend the semantic of SPARQL filters. For example, in the case of comparison operators, we have to take into account the comparison between numbers and intervals. It depends also on the types of anonymization. Another way is to categorize those types and see which ones are applicable for preserving and implementing the privacy requirements.

As explained above, in the case of normalized query with filters, the result of the rewritten query may not be sound.

### 6.6 Conclusion

In this chapter we introduced an approach that preserves the privacy preferences by query modification in the case of SPARQL query. We take into account various dimensions of privacy preferences through the concepts of consent, accuracy, purpose and recipient. A possible extension will be to handle provisional obligations.

Our approach satisfies the security criteria. However, the maximality and soundness criteria may not be satisfied in the case of query with filters as illustrated in section 6.5.5. They depend on the semantic of SPARQL filters and how they interpret obfuscated and anonymized data. An interesting future work is to extend the semantic

\[ \text{age} \geq 25 \text{ is interpreted as false for each value of age that is not a number} \]
of SPARQL filters, in the case of obfuscated data, in order to preserve the soundness and maximality criteria.
7 Privacy query rewriting algorithm instrumented by a privacy-aware access control model

7.1 Introduction

In the literature, several approaches have been proposed for enforcing privacy requirements. The most significant effort to make privacy policies more readable and enforceable, is the Platform for Privacy Preferences (P3P) [97] which enables Web sites to encode their data collection and data-use practices in a machine-readable XML format. Some work (see [98] for instance) reported how different aspects of data protection can be handled by an extension of access control models. In point of fact, several formal privacy aware models have been defined like [88], but formalization of their enforcement process is rarely defined. Moreover, there are few research works related to web services privacy policies specification and application. Enforcing privacy policy is still thus an open issue.

Recently, several research papers on privacy enforcement focused on query rewriting. For instance [67, 68, 69] propose approaches based on SQL query rewriting. Each of these rewriting algorithms uses a new dedicated privacy model unlike the approach proposed in chapter 6 which is based on an existing privacy-aware model (PrivOrBAC [9], see section 7.2).

In this chapter we present an approach that instruments a SPARQL query rewriting algorithm using a privacy aware access control model. The verb instrument is used to mean supplying appropriate constraints. We propose a lightweight vocabulary, on top of the privacy aware access control model PrivOrBAC, that enables defining fine-grained privacy preferences for each data element. Our approach is designed to view
privacy preferences as RDF data based on an ontology \( OWL_{privacy} \). We take into account various dimensions of privacy preferences through the concepts of consent, accuracy, purpose and recipient. We implement and evaluate our process of privacy enforcement by query rewriting.

We go further in this chapter, by transforming the hypothesis into a real and effective instrumentation process of the rewriting algorithm using an existing privacy aware access control model like PrivOrBAC. PrivOrBAC has been chosen because it considers all the privacy requirements appearing in the standards and recommendations like the European Commission directives [85, 86, 99] or the OECD recommendations [83], but of course it can be substituted with another privacy access model without upsetting the whole process.

The rest of this chapter is organized as follow. Section 7.2 gives an overview of the PrivOrBAC model. Section 7.3 presents our approach. Finally Section 7.4 concludes this chapter.

### 7.2 The privacy-aware OrBAC model (PrivOrBAC)

In the literature there are several models of privacy used to model and integrate privacy requirements into a security policy. These security policies are generally specified according to an access control model. This permits an easy upgrade of existing information systems, which already implemented access control policies. For instance, models such as P-RBAC [88], Purpose-BAC [89], Pu-RBAC [90] focused on purpose entity and on some other privacy requirements [83], but this does not provide a complete set of concepts to specify privacy policies.

PrivOrBac [9] is also a privacy policy model. It extends the OrBAC model [8]. It reuses most of existing mechanisms implemented in OrBAC unlike other models that propose major changes in existing models like the definition of new languages to express access contexts. PrivOrBAC takes into account the main privacy requirements [85] [86] [83] through the concepts of consent, accuracy, purposes of the access, provisional obligation and other concepts.

PrivOrBAC models (see figure 7.1) respectively the subject’s consent as a context, view hierarchy based on the accuracy of objects, purpose as a user declared context, provisional obligation following the access to sensitive information, the current state context and the enhanced spatial context.
7.2. THE PRIVACY-AWARE ORBAC MODEL (PRIVORBAC)

Figure 7.1: The Privacy-aware OrBAC model

7.2.1 Consent

Consent_preference view: Users store their consent preferences in the consent_preference view. Each object in this view corresponds to a particular data owner preference and has four attributes: Data-owner, who is the subscriber that the object or the view referred to, Requester, who is the subject who requests the access to the object, Target, which is the requested object, and NeedConsent, which is a Boolean parameter whose value is true when the consent is needed.

Example 22 We assume that the user Alice permits Bob to see her name without her consent. For this purpose, Alice defines the object Alice_preference_1:

- Requester: Bob
- Target: name
- Data-owner: Alice
- NeedConsent: false

This object must be inserted to the consent_preference to be effective:

Use(hospital, Alice_preference_1, consent_preference)

Consent context: The user consent is modelled in the OrBAC model as a new context type. The consent context is a relevant parameter in the privacy preference. It takes into account the data-owner preferences and/or notifies him when his personal information is accessed. The subscribers can define their abstracted policy by specifying that context preference should be checked before granting the access. Two cases
are identified. The first case is when the consent is needed \((\text{NeedConsent} = \text{true})\). The data-owner response is modelled by a built-in predicate \(\text{Consent}\_\text{response}\). If \(\text{org}\) is an organization, \(s\) is a subject, \(\text{do}\) is a data-owner, \(\text{resp} \in \{\text{accept}, \text{deny}\}\), then \(\text{Consent}\_\text{response}(\text{org}, \text{do}, s, \text{resp})\) is the response returned by the data owner to the organization. The second case is when the data owner does not require his consent before revealing his private data to the requester. In this case, the \(\text{NeedConsent}(\text{cp})\) attribute is false. The access decision can be made without waiting for the \(\text{Consent}\_\text{response}\).

The user consent context is specified as follows:

\[
\text{Rule_consent} : \forall \text{org} \in \text{organization}, \forall s \in S, \forall \alpha \in A, \forall o \in O, \forall \text{cp} \in O, \\
\text{Hold(}\text{org}, s, \alpha, o, \text{Consent}\_\text{context}) \leftarrow \text{Use(}\text{org, cp, Consent}\_\text{preference}) \\
\quad \land \text{Requester(}\text{cp}, s) \\
\quad \land \text{Target(}\text{cp}, o) \\
\quad \land \text{Data}\_\text{owner(}\text{cp, do}) \\
\quad \land (\neg \text{NeedConsent(}\text{cp}) \lor \text{Consent}\_\text{response(}\text{Org, do, s, accept}))
\]

The above formula means that if \(\text{org}\) is an organization, \(s\) a subject, \(\alpha\) an action, \(v\) a view, \(\text{cp}\) is an object belonging to the \(\text{Consent}\_\text{preference}\) view, and \(\text{Consent}\_\text{response}\) is the built-in predicate detailed above then the \(\text{Consent}\_\text{context}\) holds if there is an object \(\text{cp}\), which has the attributes \(s, v\) and \(\text{NeedConsent}(\text{cp})\). When the latter is \(\text{false}\) we do not need the consent of the data owner of the object \(o\), which belongs to the view \(v\), else the predicate \(\text{Consent}\_\text{response}\) is needed. By this means, the data owner can choose which view the subject can access.

**Example 23** The OrBAC rule expressing that the consent of patients is needed when nurses try to read patients information is expressed as follows:

\[
\text{Rule}_{\text{nurse}} : \text{Permission(}\text{org, Nurse, Read, PatientView, Consent}\_\text{context}) \\
\text{such that the context definition of Consent}_\text{context is expressed by Rule}_{\text{consent}}
\]

### 7.2.2 Purpose

The purpose is modeled as a user-declared context. Each data owner can create purpose objects to specify the purposes for which access to private objects are allowed. The purpose objects are grouped in a \(\text{Purpose}\) view. Purpose values range over the purpose value domain \(PV\).

Each purpose object has two attributes \([78]\).
\begin{itemize}
  \item \textit{Recipient}: defines who takes advantage of the declared purpose,
  \item \textit{declared\_purpose}: associates a purpose value with the declared purpose object
\end{itemize}

Each data owner defines objects belonging to his purpose sub-view, say \textit{do\_purpose} (data owner purpose). Purpose objects form a finite set denoted \textit{PO}. They are used to describe the user-declared context activated by some data owners. So, data owners have to define these \textit{po} objects, which have two attributes \textit{Recipient} and \textit{Declared\_purpose}.

The purpose context is specified as follows:

\begin{align*}
  \text{Rule}_{\text{purpose}} : & \forall \textit{org} \in \textit{Org}, \forall \textit{s} \in \textit{S}, \forall \alpha \in \textit{A}, \forall \textit{o} \in \textit{O}, \forall \textit{pv} \in \textit{PV}, \forall \textit{po} \in \textit{PO}, \\
  & \textit{Hold}(\textit{org}, \textit{s}, \alpha, \textit{o}, \textit{user\_declared}(\textit{pv})) \leftarrow \textit{data\_owner}(\textit{o}, \textit{do}) \\
  & \quad \wedge \textit{Use}(\textit{org}, \textit{po}, \textit{do\_purpose}) \\
  & \quad \wedge \textit{Recipient}(\textit{po}, \textit{s}) \\
  & \quad \wedge \textit{Declared\_purpose}(\textit{po}, \textit{pv})
\end{align*}

That is, in organization \textit{org}, subject \textit{s} performs action \textit{\alpha} on object \textit{o}, which has \textit{do} as data owner, in the user declared context \textit{user\_declared(pv)}, if there is a purpose object \textit{po} used in the subview view \textit{do\_purpose} by organization \textit{org} such that \textit{s} is the \textit{recipient} associated with \textit{po} and \textit{pv} is the declared purpose associated with \textit{po}. Each data owner, say \textit{do}, has his own view \textit{do\_purpose}. The latter is a sub-view of the Purpose view. We denote \textit{purpose\_view\_of(\textit{org}, \textit{do}, \textit{dop})} as the predicate that associates a data-owner \textit{do} to his purpose view \textit{dop} in the organization \textit{org}. So the \textit{Rule}_{\text{purpose}} could be defined as follows:

\begin{align*}
  \text{Rule}_{\text{purpose}} : & \forall \textit{org} \in \textit{Org}, \forall \textit{s} \in \textit{S}, \forall \alpha \in \textit{A}, \forall \textit{o} \in \textit{O}, \forall \textit{pv} \in \textit{PV}, \forall \textit{po} \in \textit{PO}, \\
  & \textit{Hold}(\textit{org}, \textit{s}, \alpha, \textit{o}, \textit{user\_declared}(\textit{pv})) \leftarrow \textit{data\_owner}(\textit{o}, \textit{do}) \\
  & \quad \wedge \textit{purpose\_view\_of(\textit{org}, \textit{do}, \textit{dop})} \\
  & \quad \wedge \textit{Use}(\textit{org}, \textit{po}, \textit{dop}) \\
  & \quad \wedge \textit{Recipient}(\textit{po}, \textit{s}) \\
  & \quad \wedge \textit{Declared\_purpose}(\textit{po}, \textit{pv})
\end{align*}

\textbf{Example 24} The OrBAC rule expressing that the patient's consent is needed when nurses try to read patients information for some medical purposes (e.g., to ensure that patients receive the right medical dosages corresponding to their ages, etc.), is expressed as follows:

\begin{align*}
  \text{Rule}_{\text{nurse2}} : & \textit{Permission(\textit{Hospital}, \textit{Nurse}, \textit{Read}, \textit{PatientView}, \textit{Consent} \wedge \textit{Medical\_Analysis})}
\end{align*}
such that the context definition of the Consent context is expressed in Rule\textsubscript{consent} and the context Medical\_Analysis is associated with Rule\textsubscript{purpose} by replacing the value \(pv\) by the value Medical\_Treatment, i.e.:

\[
\text{Rule}_{\text{Medical\_Analysis}} : \forall s \in S, \forall \alpha \in A, \forall o \in O, \forall po \in PO,
Hold(\text{Hospital}, s, \alpha, o, \text{Medical\_Analysis}) \leftarrow data\_owner(o, do) \\
\land \text{purpose}\_view\_of(\text{Hospital}, do, dop) \\
\land Use(\text{Hospital}, po, dop) \\
\land Recipient(po, s) \\
\land Declared\_purpose(po, Medical\_Treatment)
\]

### 7.2.3 Accuracy

Privacy enforcement requires the use of different levels of accuracy depending on the purpose and the subject requesting the access to the private data. That principle is consistent with the privacy directive of collection limitation since service providers cannot access more accurate objects than user preference and needed accuracy for the service.

\[9\] suggests that private objects, of each data owner, may have different levels of accuracy. The authors consider a hierarchy between the root view of one data owner that groups the initially collected objects, and sub-views of that data owner. These sub-views group the derived objects that have different accuracies. We recall that Sub\_view is a relation over domains \(\text{Org} \times V \times V\), if \(\text{org}\) is an organisation, and \(v_1\) and \(v_2\) are views, then \(\text{Sub\_view}(\text{org}, v_1, v_2)\) means that in organisation \(\text{org}\), view \(v_1\) is a sub-view of \(v_2\). So, data owner can define several access policies depending on accuracies of his private data objects.

When a data owner defines his anonymity preferences, it creates a view for each preference. This means that each view contains the sensitive objects of the data owner with specified anonymity preferences. Figure 7.2 shows an example of accuracy-based object hierarchy of location data.

Each data-owner defines his own private data hierarchy, composed of different views. Each view is described by the anonymity level field anonymity\_level. The data-owner can then specify a different privacy policy and access control for each view. “Permission privilege” is granted to the authorized service providers by data-owners.
7.3 OUR APPROACH: PRIVORBAC QUERY REWRITING ALGORITHM

Our approach is to instrument the privacy rewriting algorithm [100] by the privacy-aware model PrivOrBAC. Figure 7.3 illustrates the architecture of the approach. It is composed of three main components: (i) **Queried System**, (ii) **PrivOrBAC SPARQL Service** and (iii) **PrivOrBAC component**. (i) is the component that implements the privacy rewriting algorithm [100]. It transforms the received query $Q$ to $Q'$ such that the execution of $Q'$ invokes the preferences service (ii) via the subquery $Q_{pref}$. (ii) is the SPARQL endpoint of privacy preferences policy expressed by the PrivOrBAC model in the component (iii). (ii) is a server that receives the SPARQL query $Q_{pref}$ expressed in OWLprivacy. It decomposes this SPARQL query into a set of PrivOrBAC web services that covers $Q_{pref}$ (see section 7.3.2). (iii) represents the privacy preferences expressed in the PrivOrBAC model and are accessed via web services $S_1$, $S_2$ and $S_3$ (see section 7.3.1).

7.3.1 PrivOrBAC services

PrivOrBAC has two categories of services: (i) administration services and (ii) consultation services. (i) represents services that add, remove or update preferences. (ii) allow access to preferences of each data-owner. In this section we are interested in some services of the second category, consultation services.

We use ‘$'$ to denote input variables and '?' to denote output variables. We are interested in the following services:

- $S_1(?do)$: is a service that returns list of all data-owners $?do$. 

---

Figure 7.2: Accuracy levels of location data
• $S_2(do, t, purp, recip, ?c)$: is the service that returns the consent $?c$ of the data-owner $do$ of the target $t$, defined for the recipient $recip$ and purpose $purp$.

• $S_3(do, t, purp, recip, ?a)$: is the service that returns the accuracy $?a$ defined by the data-owner $do$ for the target $t$.

For each service presented above we define an RDF view, based on $OWL_{privacy}$, that represents its semantic. Figure 7.4 illustrates those RDF views.

Let $Pred_{-}S_1$, $Pred_{-}S_2$ and $Pred_{-}S_3$ be respectively application predicates of services $S_1$, $S_2$ and $S_3$.

**Service $S_1$**

According to section 7.2.1 preferences of data-owners are stored in the view $Consent_{-}preference$. The predicate $Pred_{-}S_1$, that allows to get all data-owners, is defined as follows:

$$Pred_{-}S_1(do) \leftarrow Use(org, cp, Consent_{-}preference) \land Data_{-}owner(cp, do)$$

For example, the fact $Pred_{-}S_1(Alice)$ states that the user $Alice$ defines at least one preference on the current privacy policy.
7.3. OUR APPROACH: PRIVORBAC QUERY REWRITING ALGORITHM

Service $S_2$

Let $do$ be a data-owner, $tg$ be a target (name of an object of $do$), $purp$ be a purpose and $recip$ be a recipient. The consent, denoted $\text{consent}(do, tg, purp, recip)$, given by the data-owner $do$ to the recipient $recip$ for the target $tg$ and purpose $purp$ corresponds to $\text{Is\_permitted}(recip, read, tg)$ such that:

$$\text{Data\_owner}(tg, do) \land \text{User\_declared\_context}(recip, purp) \leftarrow true$$

where $\text{Data\_owner}$ and $\text{User\_declared\_context}$ are application predicates. $\text{Data\_owner}(tg, do)$ means that $do$ is the data owner of the object $tg$. $\text{User\_declared\_context}(recip, purp)$ means that the subject $recip$ declares the context value $purp$. So the consent is modelled as follows:

$$\text{consent}(do, tg, purp, recip) \leftarrow \text{Is\_permitted}(recip, read, tg) \land \text{Data\_owner}(tg, do) \land \text{User\_declared\_context}(recip, purp)$$
CHAPTER 7. PRIVACY QUERY REWRITING ALGORITHM INSTRUM...

The predicate \( \text{Pred}_S2 \) is defined as follows:

\[
\text{Pred}_S2(\text{do}, \text{tg}, \text{purp}, \text{recip}) \leftarrow \text{consent}(\text{do}, \text{tg}, \text{purp}, \text{recip})
\]

For example, the fact \( \text{Pred}_S2(Alice, Age, Medical\_Treatment, Bob) \) states that the privacy policy allows the subject \( Bob \) to consult the \textit{age} of the patient \( Alice \) for the purpose \( Medical\_Treatment \).

Service \( S_3 \)

Service \( S_3 \) returns the accuracy \( acc \) defined by a data owner \( do \), for his target \( tg \), to the recipient \( recip \) for the purpose \( purp \). As explained in section 7.2.3, the data-owner assigns his data object \( o \) to a specific view \( V_{\text{accuracy}} \) which is a subview of the root view of the private object \( o \). Then the data owner \( do \) defines a permission privilege on \( V_{\text{accuracy}} \). The predicate \( \text{Pred}_S3 \) is defined as follows:

\[
\text{Pred}_S3(\text{do}, \text{tg}, \text{purp}, \text{recip}, \text{acc}) \leftarrow \text{Empower}(\text{Org}, \text{recip}, R) \land \text{Use}(\text{Org}, \text{tg}, V) \\
\land \text{Consider}(\text{Org}, \text{read}, \text{Read}) \\
\land \text{Permission}(\text{Org}, R, \text{Read}, V, C) \\
\land \text{Hold}(\text{Org}, \text{recip}, \text{read}, \text{tg}, C) \\
\land \text{Data\_owner}(\text{tg}, \text{do}) \land \text{Anonymity\_level}(V, acc)
\]

\( \text{Anonymity\_level}(V, acc) \) means that \( acc \) is the anonymity level of the given view \( V \).

For example, the fact \( \text{Pred}_S3(Alice, SSN, Medical\_Treatment, Bob, Anonymous) \) states that the privacy policy allows the subject \( Bob \) to consult the ananmized version of the social security number \( SSN \) of the patient \( Alice \) for the purpose \( Medical\_Treatment \).

7.3.2 PrivOrBAC SPARQL Service

PrivOrBAC SPARQL service is a SPARQL endpoint of privacy preferences. It receives a SPARQL query \( Q_{\text{pref}} \) expressed in term of the privacy ontology \( OWL_{\text{privacy}} \). Then it decomposes \( Q_{\text{pref}} \) into a set of PrivOrBAC web services that will be invoked later. The collected result is correctly merged, filtered and formatted, then transferred to the requester (a SPARQL Engine in our case).

The PrivOrBAC SPARQL service relies on an RDF query rewriting algorithm that is proposed in [101] to find the necessary services that cover the received query. The algorithm assumes that all services are described by RDF views and includes two phases:
7.3. OUR APPROACH: PRIVORBAC QUERY REWRITING ALGORITHM

- **Finding the relevant services**: in this phase the algorithm compares the preferences SPARQL query $Q_{pref}$ to the RDF views of available services and determines the parts of $Q_{pref}$ that are covered by these views.

- **Combining the relevant services**: in the second phase the algorithm combines the different parts to cover the whole preferences query $Q_{pref}$.

Let us take an example.

**Example 25**  We assume that Bob tries to select the name and the age of all patients for the purpose $\text{purpose}_1$. He issues the following query:

1. SELECT ?name ?age
2. FROM dt:infos WHERE {
3.    ?p rdf:type dt:Patient;
4.       dt:id ?id;
5.       dt:name ?name;
7.}

We assume that in the mapping table $M$, the property $dt:name$ (resp. $dt:age$) corresponds to the string value ‘name’ (resp. ‘age’). Figure 7.5 shows the transformed query associated with Bob’s SPARQL query.
$Q_{pref}$ corresponding to the service block of the rewritten query $Q'$, presented in figure 7.5, is defined as follow.

```sql
SELECT * FROM ps:preferences
WHERE {
  ?dp rdf:type P:DataOwner;
  P:hasId ?id;
  P:hasPreference ?pref.
  ?pref P:hasPurpose 'purpose_1';
  P:hasRecipient 'Bob';
  P:hasTarget ?tp1, ?tp2.
  ?tp1 P:hasName 'name';
  P:hasDecision ?nameDecision.
  OPTIONAL {?tp1 P:hasAccuracy ?nameAccuracy}
  ?tp2 P:hasName 'age';
  P:hasDecision ?ageDecision.
  OPTIONAL {?tp2 P:hasAccuracy ?ageAccuracy}
}
```

We denote $V$ as the RDF view corresponding to $Q_{pref}$. We denote respectively variables $?nameDecision, ?nameAccuracy, ?ageDecision$ and $?ageAccuracy$ as $?c_1, ?a_1, ?c_2$ and $?a_2$.

After applying the algorithm defined in [101] we get the following composition:


Figure 7.6 shows the execution plan of the service composition of the RDF view $V$ associated with $Q_{pref}$. 

![Figure 7.6: Composition execution plan of $V$](image-url)
In this chapter we present an approach for “instrumenting” a query rewriting algorithm that enforces privacy preferences based on SPARQL query rewriting.

Privacy preferences are defined using the PrivOrBAC model and are accessed via SPARQL Services. However, our approach is independent of the privacy-aware model. It could be integrated with any other privacy-aware access control models. We only have to define new SPARQL services for the given privacy-aware model. If the later proposes a web service then the approach is exactly the same as in PrivOrBAC. Otherwise, we have to define adapters that allow us to query the preferences of the new privacy-aware model using SPARQL queries based on $OWL_{privacy}$ ontology. For instance, [93] and [92] could be used when policies are stored in XML format.

In addition, we implemented our approach and tested its performance (see section 9.2 for more details).
8 Secure and Privacy-preserving Execution Model for Data Services

8.1 Introduction

Data services have almost become a standard way for data publishing and sharing on top of the Web. In this chapter, we present a secure and privacy-preserving execution model for data services [102]. Our model controls the information returned during service execution based on the identity of the data consumer and the purpose of the invocation. We implemented and evaluated the proposed model in the healthcare application domain and the obtained results are promising (see chapter 9).

Recently, Web services have started to be a popular medium for data publishing and sharing on the Web. Modern enterprises are moving towards service-oriented architectures for data sharing on the Web by developing Web service frontends on top of their databases, thereby providing a well-documented, interoperable method for interacting with their data [103, 104, 105, 106, 107]. We refer to this class of services as data services in the rest of the chapter.

Data services are software components that encapsulate a wide range of data-centric operations over “business objects” in underlying data sources. They abstract data consumers from the details of where data pieces are located and how they should be accessed. They allow data providers to restrain the way their business objects are manipulated and enforce their own business rules and logic. Data services are used in many contexts, for instance: data publishing [103, 108], data exchange and integration [109], service-oriented architectures (SOA) [104], data as a service (DaaS) [107], and recently, cloud computing [110].
Most of the time data services are used to access privacy-sensitive information. For example, in the healthcare domain, data services are widely used to access and manipulate the electronic healthcare records [109]. Given the sensitive nature of the accessed information and the social and legal implications for its disclosure [111, 112], security and privacy are considered among the key challenging issues that still impede the widespread adoption of data services [113].

A considerable body of recent research works have been devoted to security and privacy in the area of Web services [114, 115, 116, 117]. Their focus was on providing mechanisms for ensuring that services act only on the authorized requests and for ensuring SOAP message confidentiality and integrity. However, this is not sufficient as control over who can invoke which service is just one aspect of the security and the privacy problem for data services. A fine-grained control over the information disclosed by data service calls is required, where the same service call, depending on the call issuer and the purpose of the invocation, can return more or less information to the caller. Portions of the information returned by a data service call can be encrypted, substituted, or altogether removed from the call’s results. We explain the privacy and the security challenges for data services based on a concrete example.

### 8.1.1 Motivating Scenario

Let us consider a healthcare scenario in which a nurse Alice needs to consult the personal information (e.g., name, date of birth, etc.) of patients admitted into her healthcare organization ‘NetCare’ for some medical purposes (e.g., to ensure that patients receive the right medical dosages corresponding to their ages, etc.). The NetCare organization involves many specialized departments (cardiology, nephrology, etc.) and laboratories, and follows a data service based approach [103, 105, 109] to overcome the heterogeneity of its data sources at their various locations. We assume that Alice works in the cardiology department, and that she issued the following query: “Q: return the names and dates of birth DoB for all patients”.

Obviously, the query Q can be resolved by simply invoking $S_1$ with the value center = NetCare. However, executing the service $S_1$ involves handling security and privacy concerns that could be associated with the service’s accessed data. For example, nurses may be only allowed to access the information of patients from their own departments; physicians may be only allowed to access the information of their own patients, etc. These are security concerns that are typically defined in security policies.
the patients should also be allowed to control who can access their data, for what purposes and under what conditions. For example, two patients Bob and Sue whose data are accessed by $S_1$ may have different preferences regarding the disclosure of their ages to a nurse for medical treatment purposes. These are privacy concerns that relate to individuals and their requirements about their data. They are typically defined in privacy policies.

### 8.1.2 Challenges

Based on our scenario, we identify the following two challenges which are addressed in this chapter. The first challenge is how to enable the service providers (e.g., NetCare) to handle the cited security and privacy constraints. A common approach in the database field to handle such constraints is to push them to the underlying DBMS by rewriting the query to include these constraints [69]. However, this may not be applicable to data services as the same service may access a multitude of heterogeneous data sources that may not necessarily have a DBMS (e.g., XML files, flat files, silos of legacy applications, external Web services, etc.). An alternative approach is to enforce privacy and security policies at the application level [118], by modifying, in our case, the source code of data services. However, this also may not always be applicable nor advisable as most of current data service creation platforms (e.g., AquaLogic [119]) provide data services as black boxes that cannot be modified. Even if the code was modifiable, this solution often leads to privacy leaks [69], as the dropped programming code may contain flaws; i.e., its correctness is hard to be proven (especially for complex queries), compared to declarative rewritten queries in the first approach. The second challenge is how to specify and model the security and privacy concerns associated with data services. There is a need for a model that provides explicit description of these concerns to ensure the correct execution of services and the proper usage of their returned data.

### 8.1.3 Contributions

In this chapter, we propose a secure, privacy-preserving execution model for data services allowing service providers to enforce their privacy and security policies without changing the implementation of their data services. Our model is inspired by the database approach to enforce privacy and security policies. It relies on a declarative modeling of data services using RDF Views. When a data service is invoked, our model modifies the RDF view of the corresponding service to take into account pertaining security and privacy constraints. Our model uses our query rewriting techniques de-
We propose a semantic modeling for data services, privacy and security policies. The modeling is based on RDF views and domain ontologies.

- We propose a secure and privacy-preserving execution model for data services. Our model exploits our previous work on the areas of query rewriting and modification, and defines new filtering semantics to protect the service’s accessed data.

- We integrated our model in the architecture of the widely used Web services container AXIS 2.0, and carried out a thorough experimental evaluation.

The rest of the chapter is organized as follows. In Section 8.2, we present our secure and privacy-preserving execution model for data services. We present also our modeling to data services, security and privacy policies and then conclude the chapter in Section 8.3.

8.2 A Secure and Privacy-Preserving Execution Model for Data Services

In this section, we describe the proposed model for data service execution. We start by giving an overview of our model. Then, we present our modeling to data services and policies. Finally, we detail the different steps that need to be performed to enforce privacy and security policies.
8.2.1 Model Overview

Our model is inspired by the database approach to “declaratively” handle the security and privacy concerns. Specifically, our model relies on modeling data services as RDF Parameterized Views over domain ontologies to explicitly define their semantics. An RDF view captures the semantics of the service’s inputs and outputs (and their inter-relationships) using concepts and relations whose semantics are formally defined in domain ontologies. Views can be integrated into the service description files WSDL as annotations. Our model, as Figure 8.1 shows, enforces the privacy and the security constraints associated with data services “declaratively” as follows. Upon the reception of a service invocation request for a given service (e.g., $S_i$), it extracts the RDF view of the corresponding service from the service description file and the contextual information (e.g., the recipient of requested data, the purpose, the time and location, etc.) from the invocation request. Then, the RDF view is rewritten to include the security and privacy constraints that pertain to the data items referred in the view. These constraints are defined in the security and privacy policies and have the form of SPARQL expressions (which simplifies their inclusion in the RDF view). The generated extended view may include now additional data items necessary for the evaluation of the constraints (e.g., the consent of patients, the departments of nurses, etc.) that are not covered by the initial service. Therefore, the extended view is rewritten in terms of calls to (i) the initial service $S_i$ and (ii) the services covering the newly added data items. Finally, the obtained composition is executed, and the constraints are evaluated and enforced on the obtained results. The obtained results now respect the different security and privacy concerns, and can be returned safely to the service consumer. We explain and illustrate these steps in details in subsequent sections.

8.2.2 Semantic models for data services and policies

**Semantic model for data services**: The semantics of data services should be explicitly defined to allow service consumers to correctly interpret and use the services’ returned data. In this work, we model data services as RDF Parameterized Views (RPVs) over domain ontologies $\Omega$. RPVs use concepts and relations from $\Omega$ to capture the semantic relationships between input and output sets of a data service.

Formally, a data service $S_i$ is described over a domain ontology $\Omega$ as a predicate: $S_i(\$X_i, ?Y_i) : \neg <RPV_i(\overline{X}_i, \overline{Y}_i, \overline{Z}_i), C_i>$, where:
Figure 8.2: Part-A: the RDF View of $S_1$; Part-B: its graphical representation

- $X_i$ and $Y_i$ are the sets of input and output variables of $S_i$, respectively. Input and output variables are also called as distinguished variables. They are prefixed with the symbols “$” and “?” respectively.

- $RPV_i(X_i, Y_i, Z_i)$ represents the semantic relationship between input and output variables. $Z_i$ is the set of existential variables relating $X_i$ and $Y_i$. $RPV_i(X_i, Y_i, Z_i)$ has the form of RDF triples where each triple is of the form $(subject.property.object)$.

- $C_i$ is a set of data value constraints expressed over the $X_i, Y_i$ or $Z_i$ variables.

RDF views have the advantage of making the implicit assumptions made about the service’s provided data explicit. These assumptions may be disclosed implicitly to service consumers. For example, the names and DoBs returned by $S_1$ are for patients “who have diabetes”; i.e., the service consumer will know -implicitly- in addition to the received names that these patients have diabetes. Modeling and explicitly describing this implicit knowledge is the first step to handle this unwanted implicit information disclosure. Note that RDF views can be integrated to the service description files as annotations (e.g., using the WSDL-S approach (www.w3.org/Submission/WSDL-S/)).

Security and privacy policies: In this work, we suppose the accessed data are modeled using domain ontologies. We express therefore the security and privacy policies over these ontologies. We adopt the OrBAC [8] and its extension PrivOrBAC [9] to express the security and the privacy policies respectively.
The security rules corresponding to the motivating example, i.e. nurses may be only allowed to access the information of patients admitted in the same department, can be expressed in the OrBAC model as follows:

\[
\text{SecRule}_1 = \text{Permission(NetCare, Nurse, Read, NameView, SameDepartment)}
\]

\[
\text{SecRule}_2 = \text{Permission(NetCare, Nurse, Read, DoBView, SameDepartment)}
\]

\[
\text{SecRule}_3 = \text{Permission(NetCare, Nurse, Read, DiseaseView, SameDepartment)},
\]

where the “SameDepartment” is the context defined as follows:

\[
\text{Hold(NetCare, s, } \alpha, o, \text{SameDepartment}) \leftarrow \text{Patient}(p, o) \land \text{TreatedIn}(o, d) \land \text{EmployedIn}(s, d)
\]

The privacy rules of our example are as follows:

\[
\text{PrivRule}_1 = \text{Permission(NetCare,Nurse,Medical_Treatment,Read,NameView,Consent)},
\]

\[
\text{PrivRule}_2 = \text{Permission(NetCare,Nurse,Medical_Treatment,Read,DoBView,Consent)},
\]

\[
\text{PrivRule}_3 = \text{Permission(NetCare,Nurse,Medical_Treatment,Read,DiseaseView,Consent)},
\]

where the “Consent” is the PrivOrBAC consent context. Consent context could be expressed also against domain ontologies. Figure 8.3 shows the Consent expressed as a SPARQL expression as well as its graphical representation.

### 8.2.3 RDF views rewriting to integrate security and privacy constraints

In this step, the proposed model extends the RDF view of the queried service with the applicable security and privacy rules (from the policies) as follows:

Our model extracts the RDF view of the invoked service from the service description file, and consults the associated security and privacy policies to determine the applicable rules for the given couple of (recipient, purpose). With respect to security policies,
our model applies the access rules associated with each of the data items declared in the view to remove unauthorized data items. In some cases, the access to a given data item is granted only under certain conditions. For example, the security rules in our example restrict the access to the patient’s personal information to the nurses working in the department where the patients are treated. These conditions (which have con-
cretely the form of SPARQL expressions) are accommodated in the RDF view. The parts (a) and (b) of Figure 8.4 shows respectively the initial and the extended view; the added RDF triples are marked in red. Similarly, our algorithm rewrites the extended view to integrate the privacy rules. Returning to our example, the condition related to the patient’s consent are added to the view. Figure 8.4 (Part-c) shows the extended view, where the added RDF triples are marked in blue.

8.2.4 Rewriting the extended view in terms of data services

The extended RDF view \( v_{\text{extended}} \) may include additional data items (denoted by \( \Delta v = v_{\text{extended}} - v_{\text{original}} \)) required to enforce security and privacy constraints. These data items may not be necessary covered by the initial service. In our example (Figure 8.4, Part-c), \( \Delta v \) includes the RDF triples ensuring that the patients and the nurse have the same departments, and the RDF triples querying the patient’s consent relative to the disclosure of his personal and medical data.

In this step, we find the data services covering \( \Delta v \) to prepare for the enforcement of privacy and security conditions (in a later step), and rewrites \( v_{\text{extended}} \) in terms of these services along with the initial service. In this work, we assume the data items necessary for the evaluation of the security and privacy constraints (e.g., consent, time, location, etc.) are also provided as data services.

In our running example, the extended view in Figure 8.4 (Part-c) ensures that the query issuer works in the same department as the one the queried patients have been admitted to. It includes as well a schema-level representation of these patients’ preferences regarding the disclosure of their personal and medical information (i.e., names, DoBs and diseases).

Therefore in this step our proposed model finds the data services that provide the additional information to prepare for the evaluation of privacy and security conditions. We assume that this information is provided as services by service providers. Note that, these services may include privacy and security dedicated services (e.g., the services returning the patients’ preferences, the services returning the user’s groups and privileges) as well as business services that could be used apart (e.g., the services returning the department where a patient is treated, etc.).

Our rewriting algorithm that implements this step has two phases:

**Phase 1: Finding the relevant services:** In this phase, the algorithm compares \( v_{\text{extended}} \) to the RDF views of available services and determines the parts of \( v_{\text{extended}} \) that are covered by these views. We illustrate this phase based on our example. We
assume the existence of a service $S_2$ returning the centers and the departments where a given patient is treated, and a service $S_3$ returning the privacy preference of a given patient regarding the disclosure of a given property (e.g., name, DoB, etc.) relative to a given couple of recipient and purpose. The RDF views of these services are shown in Figure 8.5. Table 8.1 shows our sample services and the parts they cover of $v_{\text{extended}}$. The service $S_2$ covers the properties \textit{composedOf}($C,P$) and \textit{treatedIn}($P,D$) and the node $D($\texttt{const2} = "cardiology"), and covers from the nodes $P$ and $C$ the functional properties (i.e., identifiers properties) \texttt{hasName} and \texttt{dName} that could be used to make the connection with the other parts of $v_{\text{extended}}$ that are not covered by $S_2$. The service $S_3$ covers the identical sub graphs involving a node of a \texttt{Preferences} type (e.g., $P_1$, $P_2$, $P_3$, $P_4$), a node of \texttt{Target} type (e.g., $T_1$, $T_2$, $T_3$, $T_4$) and the object properties \texttt{hasPreferences} and \texttt{hasTarget}, hence its insertions in the third, fourth, fifth and sixth rows of the Table 8.1.

**Phase 2: Combining the relevant services:** In the second phase, the algorithm combines the different lines from the generated table (in the first phase) to cover $v_{\text{extended}}$ entirely. In our example we need to combine all of Table-1’s lines to cover $v_{\text{extended}}$: $v_{\text{extended}}(\texttt{$x,y,z,w,q,u,r$}) \leftarrow S_1(\texttt{$x,y,z$}) \land \texttt{const1="Diabetes"}$

\[
\land S_2(\texttt{$y,x$,const2}) \land \texttt{const2="cardiology"}
\land S_3(\texttt{$y$,"HealthCare","Nurse","hasDisease"},?w)
\land S_3(\texttt{$y$,"HealthCare","Nurse","hasDoB"},?q)
\land S_3(\texttt{$y$,"HealthCare","Nurse","dName"},?u)
\land S_3(\texttt{$y$,"HealthCare","Nurse","hasDisease"},?r)
\]

<table>
<thead>
<tr>
<th>Service</th>
<th>Partial Containment Mapping</th>
<th>Covered nodes &amp; object properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1(\texttt{$x,y,z$})$</td>
<td>$V'.P \rightarrow S_1.P$, $V'.C \rightarrow S_1.C$</td>
<td>$P(y,z,\texttt{const1})$, admittedIn($P,C$), $C(x)$</td>
</tr>
<tr>
<td>$S_2(\texttt{$y,x$,?b})$</td>
<td>$V'.P \rightarrow S_2.P$, $V'.D \rightarrow S_2.D$, $V'.C \rightarrow S_2.C$</td>
<td>composedOf($C,D$), treatedIn($P,D$), $C(x),D($\texttt{const2}), $P(y)$</td>
</tr>
<tr>
<td>$S_3(\texttt{$y,b,c,d,w,q,u,r$})$</td>
<td>$V'.P \rightarrow S_3.P_a$, $V'.P \rightarrow S_3.P$, $V'.T_1 \rightarrow S_3.T$, $y \rightarrow a$, $b \rightarrow \text{&quot;HealthCare&quot;}$, $c \rightarrow \text{&quot;Nurse&quot;}$, $d \rightarrow \text{&quot;hasDoB&quot;}$, $q \rightarrow e$</td>
<td>$P(y),$ hasPreferences($P,P_1$), $P_1(\text{&quot;HealthCare&quot;},&quot;Nurse&quot;),$ hasTarget($P_1,T_1$), $T_1(&quot;hasName&quot;,w)$</td>
</tr>
<tr>
<td>$S_3(\texttt{$y,b,c,d,w,q,u,r$})$</td>
<td>$V'.P \rightarrow S_3.P_a$, $V'.P \rightarrow S_3.P$, $V'.T_2 \rightarrow S_3.T$, $y \rightarrow a$, $b \rightarrow \text{&quot;HealthCare&quot;}$, $c \rightarrow \text{&quot;Nurse&quot;}$, $d \rightarrow \text{&quot;hasDoB&quot;}$, $q \rightarrow e$</td>
<td>$P(y),$ hasPreferences($P,P_2$), $P_2(\text{&quot;HealthCare&quot;},&quot;Nurse&quot;),$ hasTarget($P_2,T_2$), $T_2(&quot;hasDoB&quot;,q)$</td>
</tr>
<tr>
<td>$S_3(\texttt{$y,b,c,d,w,q,u,r$})$</td>
<td>$V'.P \rightarrow S_3.P_a$, $V'.P \rightarrow S_3.P$, $V'.T_3 \rightarrow S_3.T$, $y \rightarrow a$, $b \rightarrow \text{&quot;HealthCare&quot;}$, $c \rightarrow \text{&quot;Nurse&quot;}$, $d \rightarrow \text{“hasName”}$, $u \rightarrow e$</td>
<td>$P(y),$ hasPreferences($P,P_3$), $P_3(\text{&quot;HealthCare&quot;},&quot;Nurse&quot;),$ hasTarget($P_3,T_3$), $T_3(&quot;hasName&quot;,u)$</td>
</tr>
</tbody>
</table>

**Table 8.1:** The sample services along with the covered parts of the extended view $V'$.
8.2.5 Enforcing security and privacy constraints

The services selected in the previous step are orchestrated into a composition plan to be executed. The composition plan defines the execution order of services and includes filters to enforce privacy and security conditions. Figure 8.6 shows the execution plan of the running example. The service $S_1$ is first invoked with the name of the healthcare center ($x = \text{"NetCARE"}$); the patient names obtained (denoted by the variable $y$) are then used to invoke the service $S_3$ which returns the patients’ preferences relative to the disclosure of their properties (name, DoB, department, and disease). In parallel, the service $S_2$ is invoked to retrieve the departments where the patients are treated. The results of these services are then joined. Figure 8.7 gives the outputs of the join operator.

After the join operation has been realized, the obtained results are processed by a privacy filter that uses the values of the properties that were added to the initial view to evaluate the privacy constraints of the different properties that are subject to privacy constraints in the view. Null values will be returned for properties whose privacy constraints evaluate to False.

Privacy filters are added to the outputs of services returning privacy sensitive data. The semantics of a privacy filter is defined as follows:

**Definition 1:**

Let $t$ (resp., $t_p$) be a tuple in the output table $T$ (resp., $T_p$) of a service $S$ returning privacy sensitive data, let $n$ be the number of columns in $T$, let $t[i]$ and $t_p[i]$ be the projected datatype properties that are subject to privacy constraints, and let $\text{constraint}(t[i])$ be a boolean function that evaluates the privacy constraints associated with $t[i]$. A tuple $t_p$ is inserted in $T_p$ as follows:
For each tuple \( t \in T \)

For \( i = 1 \) to \( n \)

\[
\text{if } \text{constraint}(t[i]) = \text{true} \text{ Then } t_p[i] = t[i] \\
\text{else } t_p[i] = \text{null}
\]

Discard all tuples that are null in all columns in \( T_p \)

Continuing with our running example, the added filter computes the values of \( y, z, \) 
const1 (i.e, department) and const2 (i.e, disease) as follows:

\[
y = y \text{ if } w = \text{“Yes”}, \text{ otherwise } y = \text{Null} \\
z = z \text{ if } q = \text{“Yes”}, \text{ otherwise } z = \text{Null} \\
\text{const1} = \text{const1} \text{ if } u = \text{“Yes”}, \text{ otherwise } \text{const1} = \text{Null} \\
\text{const2} = \text{const2} \text{ if } r = \text{“Yes”}, \text{ otherwise } \text{const2} = \text{Null}
\]

After applying the privacy filter, the composition execution plan applies the predicates of the extended view (e.g., \( \text{dep} = \text{“cardiology”}, \) and \( \text{di} = \text{“Diabetes”}\)) on the filter’s outputs. This operation is required for two reasons: (i) to remove the tuples that the recipient is not allowed to access according to the security policy, and (ii) to remove the tuples that the recipient has access to, but whose disclosure would lead to a privacy breach.

Figure 8.7 shows the output of the \textit{Select} (\( \text{dep} = \text{“cardiology”} \)) operator. The tuples \( t_4 \) and \( t_5 \) have been removed. \( t_6 \) has been removed in compliance with the security policy which requires the patient and recipient to be in the same department - the patient \textit{Stacy} is treated in the surgery department, whereas the recipient \textit{Alice} works in
the cardiology department). $t_4$ was removed despite the fact that the patient and the recipient are in the same department. Note that if $t_4$ were disclosed, then the recipient Alice would infer that the patient Andy is treated in the cardiology department which violates Andy’s privacy preferences.

The $\text{Select}(di= \text{“Diabetes”})$ operator removes the tuple $t_3$ by comparing the value “Null” with the constant “Diabetes”. Note that if $t_3$ was disclosed, then the recipient Alice would infer that the patient Sue has Diabetes which violates Sue’s privacy preferences.

8.3 Conclusion and Perspectives

In this chapter, we tackle the problem of privacy and security management for data services. We identify the following challenges. First, there is a need for service providers to locally handle privacy and security concerns without affecting the code of their business services. Second, there is a need for a model to describe and make explicit the privacy and security concerns attached to data during the invocation process of business services.

To address these challenges, we propose a secure, privacy-preserving execution model for data services allowing service providers to enforce their privacy and security policies without changing the implementation of their data services (i.e., data services are considered as black boxes). Our contribution consists of 1) a semantic model to describe DaaS services as RDF views over domain ontologies, 2) a secure and privacy-preserving execution model for data services that exploits our previous work on the area of query rewriting, and 3) a prototype that demonstrates the applicability of our model according to a scenario inspired from the healthcare domain. We provide an implementation of our prototype for the Axis 2.0 Web service engine and discuss our experimental results (chapter 9, section 9.3). Future work includes extending our model to support privacy and security protection in data service composition, which raises new problems related to relational operators such as joins and projections, and related to composition constructs such as loops and conditions.
In this chapter we present concrete implementation of our approach $f$Query. Then we present some use cases.

This chapter is organized as follow. Section 9.1 presents the implementation of the $f$Query approach. Section 9.2 presents concrete implementation of the approach presented in chapter 7 and shows experimental results. Section 9.3 presents the implementation of the approach presented in chapter 8 which uses the $f$Query module to enforce the security and privacy policies in the execution model for data services. Section 9.4 presents some use cases of our approach.

9.1 Implementation of $f$Query

9.1.1 MotOrBAC tool

Before we present the implementation of the $f$Query approach, we need first to introduce the MotOrBAC tool. The MotOrBAC tool is used to implement the OrBAC model and its administration model AdOrBAC [120]. It is developed and maintained at Telecom Bretagne, by the SERES team. It is distributed under Mozilla licence. It includes all the features supported by OrBAC and provides a user-friendly interface (GUI) to specify and manage the security policy and also the administration policy (more details about this tool are given in [121] and also in the MotOrBAC website [122]). Figure 9.1 shows a screenshot of the MotOrBAC tool.

As shown in figure 9.2, the MotOrBAC tool is composed of two separate modules (1) OrBAC API and (2) GUI, and can be extended through the use of the plug-in system. The OrBAC application programming interface (API) manages and instantiates the security policies. The GUI displays these policies with the associated entities.
9.1.2 Implementation of \textit{fQuery-AC}

\textit{fQuery-AC} is the implementation of the \textit{fQuery} approach that enforces the access control rules (confidentiality policy). \textit{fQuery-AC} is developed in Java using the Jena-ARQ API \cite{123}. We developed two versions of \textit{fQuery-AC}. The first one is based on the pattern AOP (Aspect Oriented Programming). The second one uses the design pattern \textit{Visitor} provided with the Jena-ARQ API.

**Version based AOP**

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns \cite{124}. There are many implementation of that paradigm. In our case we are using the extension \textit{AspectJ}. \textit{AspectJ} is an aspect-oriented extension for the Java programming language. Its aspects can contain several entities like \textit{point-cuts}, \textit{advice} ... \textit{Point-cuts} allow to specify join points \textit{i.e} well-defined moments in the execution of a program, for instance method call, object instantiation, or variable access. A point-cut determines whether a given join
point matches. For example, this point-cut matches the call of the instance method `insert`:

```java
pointcut insertCall():
    call (void SPARQLParser10.insert(TripleCollector, int, Node, Node, Path, Node));
```

A pointcut could be defined based on another point-cut. For example the following point-cut is based on `insertCall` point-cut. It matches the call of the instance method `insert` in an object of type `SPARQLParser10`:

```java
pointcut insertTriple (SPARQLParser10 parser) : insertCall() && target(parser);
```

Advice is used to specify code to run at a join point matched by a pointcut. The actions can be performed before, after, or around the specified join point. The following example controls the execution of the method `insert` defined by the point-cut `insertTriple`. If the condition `omega` is not a simple condition then it executes the `insert` method normally. Otherwise, it does another treatment (see appendix C for more details).
 void around(SPARQLParser10 parser) : insertTriple(parser){
    if(!(omega instanceof SimpleCondition)){
        proceed(parser);
        return;
    }
    SimpleCondition scondition = (SimpleCondition) omega;
    ...
}

Version based Visitor

The *visitor* design pattern is a way of separating an algorithm from an object structure on which it operates [125]. It offers the ability to add new operations to existing object structures without modifying those structures. Jena-ARQ API implements this mechanism at the algebra and syntax levels.

In our case we are interested in the syntax visitor. It provides a simple way to get to the right parts of the query. It helps walking through the query elements and defining specific behavior for each specific element. The appendix D shows in details the implementation of our visitor *RWElementVisitor*. The following code illustrates an example of using the visitor *RWElementVisitor*:

```
public static Query transform(Query originalQuery, Condition omega, boolean type){
    Query rwQuery = originalQuery.clone();
    RWElementVisitor visitor = new RWElementVisitor(omega, type);
    Element el = rwQuery.getQueryPattern();
    visitor.visitAsGroup(el);
    return rwQuery;
}
```

*fQuery-AC MotOrBAC plugin*

We developed a MotOrBAC plugin that (i) manages RDF conditions, (ii) assigns a composition of RDF conditions to an OrBAC context as a definition context, (iii) manages the abstract properties mapping and (iv) manages the alias of prefixes.

*fQuery plugin* is composed of four tabs. The first one (RDF conditions tab) is used to add and manage RDF conditions as shown in figure 9.3.

The second tab “fQuery Contexts” (figure 9.4), loads OrBAC contexts that are instance of *fQueryContext*. Then, it allows us to assign a composition of selected RDF conditions to the selected instance of OrBAC context.

The third tab “Property mapping” (figure 9.5), allows defining the mapping associated with abstract property in case of representing complex condition by an involved
9.1. IMPLEMENTATION OF FQUERY

Figure 9.3: Management of RDF conditions

![Figure 9.3: Management of RDF conditions](image1)

Figure 9.4: OrBAC Context and RDF condition assignment

![Figure 9.4: OrBAC Context and RDF condition assignment](image2)

one (as defined in chapter 3). For instance, figure 9.5 shows an example of mapping defined for the abstract property ‘fap:myDoctorId’.

The fourth tab “Prefixes” presented in figure 9.6 is used to define an alias for each given prefix. This alias will be used as shortcut of the associated prefix. For example instead of writing the following simple condition:

```sparql
?o = $requestorName
?o >= 30
```
$\omega(s, p, o) = (p \neq <http://tb.eu/patient/0.1/#dept>)$

we can write simply $\omega(s, p, o) = (p \neq o:dept)$.
9.1.3 Implementation of \textit{fQuery-Privacy}

\textit{fQuery-Privacy} is the implementation of the \textit{fQuery} approach that enforces the privacy preferences policy of each data-owner. It is developed also in Java based on the design pattern \textit{Visitor} (element visitor) provided with the Jena-ARQ API.

We also developed a \textit{REST} (REpresentational State Transfer) service for the PrivOrBAC policy, based on the RESTful API “JBoss RestEasy v2.3.5”. Then we developed a SPARQL service that allows accessing users’ preferences modelled in the PrivOrBAC policy (see section 9.2). Finally we developed a MotOrBAC plugin that manages users’ preferences, PrivOrBAC SPARQL service, the mapping table \(M\) (define in chapter 6 section 6.3) and allows testing our \textit{fQuery-Privacy} algorithm.

\textit{fQuery-Privacy} MotOrBAC plugin

\textit{fQuery-Privacy} plugin is composed of three tabs. The first one entitled “Process” (figure 9.7) allows to simulate and test our \textit{fQuery-Privacy} API. It is used to rewrite a SPARQL query then execute the initial and the rewritten query.

The second tab “Mapping”, shown in figure 9.8, is used to define and declare private properties of given ontology. Each property declared as private will be taken into account by the rewritten query algorithm.

The third tab “PrivOrBAC Service” (figure 9.9) is used to start/stop the PrivOrBAC SPARQL Service and to manage users’ preferences. It refreshes preferences (load preferences from MotOrBAC current policy), modify and/or delete selected preferences, or adding new ones.

9.2 Performance of \textit{fQuery-Privacy} instrumented by PrivOrBAC

In this section, we present concrete implementation of the approach presented in chapter 7 and show experimental results. Section 9.2.1 shows the architecture of our implementation and used technology. Section 9.2.2 presents a use case of our implementation. Section 9.2.3 shows experimental results.
CHAPTER 9. ARCHITECTURES AND IMPLEMENTATIONS

Figure 9.7: Test and simulation screen of fQuery-Privacy algorithm

Figure 9.8: Managing private properties mapping $M$
9.2. PERFS: FQUERY-PRIVACY INSTRUMENTED BY PRIVORBAC

9.2.1 Architecture

Figure 9.10 shows the architecture of the implementation of our approach. It is composed of three servers.

- **Server 1**: a server that intercepts a user’s SPARQL query\(^1\). It is implemented using “the Embedded jetty Web Server 9.0”. Each query sent to the address “http:localhost:8081/query/” is processed by the *Query Handler* component that handles queries sent to the path “/query”. Each intercepted query is rewritten by the “fQuery-Privacy” API \(^{100}\). Then the rewritten query is executed by the SPARQL engine “Jena-ARQ” \(^{123}\).

- **Server 2**: a server that proposes SPARQL services. It is implemented using the “Embedded jetty Web Server 9.0”. The PrivOrBAC SPARQL service is accessible at the address “http:localhost:8082/privorbac/” and is based on the ontology OWL\textsubscript{privacy}. Each query sent to the path “/privorbac” is handled by the component “PrivOrBAC Handler”. The later splits the received query into a composition of PrivOrBAC services which will be executed by the “Service Execution” component.

- **Server 3**: a server that stores the privOrBAC policy. User’s preferences are accessed via web services based on the REST (REpresentational State Transfer) architecture. It is implemented using *JBoss RestEasy 2.3.5* integrated with “jetty Web Server 9.0”.

\(^{1}\text{SOH: SPARQL Over HTTP}\)
The characteristics of the machine used for test are shown in Table 9.1. Table 9.2 shows technologies and APIs used in our implementation.

### 9.2.2 Use case

This section illustrates a scenario used for test in an organization *Hospital*. We consider a healthcare scenario in which a nurse Bob needs to consult the personal information (name, age, place of birth, etc.) of patients admitted into her healthcare organization for some medical purposes, *e.g.* to ensure that patients receive the right medical dosages...
9.2. PERFS: FQUERY-PRIVACY INSTRUMENTED BY PRIVORBAC

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM</td>
<td>Java 7</td>
</tr>
<tr>
<td>Server</td>
<td>Embedded jetty Web Server 9.0</td>
</tr>
<tr>
<td>OrBAC API</td>
<td>v1.2.2</td>
</tr>
<tr>
<td>RESTful API</td>
<td>JBoss RestEasy v2.3.5</td>
</tr>
<tr>
<td>SPARQL Engine</td>
<td>Jena-ARQ v2.9.4</td>
</tr>
</tbody>
</table>

Table 9.2: Technology used for the implementation

corresponding to their ages, etc. We assume that the privacy policy defined in that healthcare organization for nurses is as follows:

- Nurses are allowed to see the admitted department of all patients.
- Nurses are allowed to see the age of patients who agree to disclose their age to the requestor.
- Nurses are allowed to see the place of birth of patient only for medical treatment purposes.
- Nurses are allowed to see the name of patients who agree to disclose their name to the requestor and only for medical treatment purposes.

To define the OrBAC organizational policy of *Hospital*, we consider the following simple entities:

- **Roles**: Nurse
- **Activities**: Read_activity
- **Views**: Name_view, Age_view, Dept_view and Birthplace_view
- **Contexts**: Consent, Medical_Analysis, Nominal

Then we consider the OrBAC policy corresponding to the permissions assigned to ‘Nurse’:

- Permission(*Hospital*, Nurse, Read_activity, Dept_view, Nominal)
- Permission(*Hospital*, Nurse, Read_activity, Age_view, Consent)
- Permission(*Hospital*, Nurse, Read_activity, Birthplace_view, Medical_Analysis)
Finally we filled PrivOrBAC policy with 2000 preferences of 1000 patients. The scenario that we consider to check our approach, is as follows. Bob is empowered to the role Doctor. We consider three cases.

- (i) Bob tries to get age, department and birthplace of the patient Alice.
- (ii) Bob tries to get name, age, department and birthplace of patients who are less than 20 years old.
- (iii) Bob tries to get name, age, department and birthplace of all patients.

We assume that in all cases, Bob declares the purpose Medical_Treatment.

In the Case (i), the SPARQL query $Q_1$ corresponding to Bob request is as follows:

```
SELECT ?age ?dept ?bp WHERE {
  ?d o:deptName ?dept.
}
```

The query $Q_1$ will be normalized before being rewritten. The normalized query $Q_{N1}$ is as follows:

```
SELECT ?age ?dept ?bp WHERE {
  ?d o:deptName ?dept.
  FILTER(?name='Alice')
}
```

In the case (ii), the SPARQL query $Q_2$ corresponding to Bob request in this case is as follows:

```
SELECT ?name ?age ?dept ?bp WHERE {
  ?d o:deptName ?dept.
  FILTER(?age <= 20)
}
```

The normalized query $Q_{N2}$ of $Q_2$ is the same ($Q_{N2} = Q_2$).

In the case (iii), the SPARQL query $Q_3$ issued by Bob in this case is as follows:

```
SELECT ?name ?age ?dept ?bp WHERE {
  ?d o:deptName ?dept.
}
```
The normalized query $Q_{N3}$ of $Q_3$ is the same ($Q_{N3} = Q_3$).

The rewritten query $Q_{rw3}$ of $Q_{N3}$ contains a block of service that allows remote access to privacy preferences (see figure 9.11). In our case we are using the Jena-ARQ API as SPARQL engine to evaluate our queries. In Jena-ARQ implementation, the algebra operation is executed while disregarding how selective the pattern is. So the order of the query will affect the speed of execution. Because it involves HTTP operations, asking the query in the right order matters a lot [123]. So inserting the block service at the beginning of the query means that the SPARQL engine will first execute the remote query before proceeding the local data. That means that the Engine will get preferences defined for targets name, age, birthplace and department of all patients. So the execution time of the rewritten query depends on the privacy server response time (privacy SPARQL endpoint) and the time of applying inserted bindings.

As we can see, queries $Q_{N1}$, $Q_{N2}$ and $Q_{N3}$ have the same triples pattern in the where clause portion. In the rest of this section we will study the case of the query $Q_{N3}$. The same approach is valid for $Q_{N1}$ and $Q_{N2}$.

### 9.2.3 Experimental results

For each case illustrated in the above section, firstly we execute Bob’s initial queries directly over RDF databases i.e without rewriting process. Secondly we rerun the same test using our transformation approach. Then we show the execution time between the two executions. We fixed the number of privacy preferences defined on PrivOrBAC to 2000 preferences and 1000 data-owners.

Figure 9.12 shows the execution time of initial queries $Q_1$, $Q_2$ and $Q_3$. The execution time varies from 3 to 50 milliseconds. It depends on the filters of queries. The execution time of the query $Q_3$ is greater than the one of other queries, because the query $Q_3$ selects the information (name, age, birthplace and department) of all patients of the database.

Figure 9.13 show the execution time of the rewritten queries $Q_{rw1}$, $Q_{rw2}$ and $Q_{rw3}$ of $Q_1$, $Q_2$ and $Q_3$ respectively. This time includes the server 2 response time (SPARQL Privacy Service) where the average is about 1100 ms. The execution time of the rewritten queries varies from 1200 to 2900 milliseconds.

As shown in figure 9.13 the execution time of the rewritten query of $Q_3$ is less than the other two queries. It is due to the normalization of the initial queries before applying the rewriting algorithm. That is, when we normalize a query, we transform all implicit filters into explicit ones. Since the explicit filters are applied at the end of
PREFIX p: <http://orbac.org/privOrBAC/endpoint#>
PREFIX o: <http://tb.eu/patient/0.1/#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX udf: <http://orbac.org/privOrBAC/UserDefinedFunction#>

SELECT ?name ?age ?dept ?bp WHERE {
  #service Block

  SERVICE <http://localhost:8082/privorbac/>
  {
    ?do rdf:type p:DataOwner;
    p:hasId ?name_1;
    p:hasPreference ?pref.
    ?pref p:hasPurpose "Medical_Treatment";
    p:hasRecipient "Bob";
    p:hasTarget ?t0.
    ?t0 p:hasName "name";
    p:hasDecision ?nameDecision.
    OPTIONAL { ?t0 p:hasTarget ?t1.
      ?t1 p:hasName "age";
      p:hasDecision ?ageDecision.
      OPTIONAL { ?t1 p:hasAccuracy ?nameAccuracy}
      ?t1 p:hasTarget ?t2.
      ?t2 p:hasName "dept";
      p:hasDecision ?deptDecision.
      OPTIONAL { ?t2 p:hasAccuracy ?ageAccuracy}
      ?t2 p:hasTarget ?t3.
      ?t3 p:hasName "birthplace";
      p:hasDecision ?birthplaceDecision.
      OPTIONAL { ?t3 p:hasAccuracy ?birthplaceAccuracy}
    }
  }
  { #initial normalized query with new variables
    ?p rdf:type o:Patient;
    o:dept ?d.
    ?d o:deptName ?dept_1.
    ?p o:age ?age_1;
    ?p o:birthplace ?bp_1;
    o:name ?name_1.
    #Applying preferences
    BIND(if(!?nameDecision = "No"), "−");
    if(bound(?nameAccuracy), udf:eval(?name_1,?nameAccuracy), ?name_1) AS ?name)
    BIND(if(!?ageDecision = "No"), "−");
    if(bound(?ageAccuracy), udf:eval(?age_1,?ageAccuracy), ?age_1) AS ?age)
    BIND(if(!?deptDecision = "No"), "−");
    if(bound(?deptAccuracy), udf:eval(?dept_1,?deptAccuracy), ?dept_1) AS ?dept)
    BIND(if(!?birthplaceDecision = "No"), "−");
    if(bound(?birthplaceAccuracy), udf:eval(?bp_1,?birthplaceAccuracy), ?bp_1) AS ?bp)}

Figure 9.11: The rewritten query $Q_{rw3}$ of $Q_{N3}$
9.3 Performance of Secure and Privacy-preserving Execution Model for Data Services

In this section, we present the implementation of the approach presented in chapter 8 which uses the fQuery module to enforce the security and privacy policies in the execution model for data services. The following implementation is developed by SOC Team (Service Oriented Computing) using our fQuery API.

---

\[Q_{rw3}\] has no filter that is why the execution time is less than the other queries.
9.3.1 Implementation

In order to validate and evaluate our proposal, we exploited the extensibility support provided by Axis 2, specifically the ability to deploy user modules, to implement our privacy-preserving service invocation model. As shown in Figure 9.14, we extended the AXIS 2.0 architecture with a privacy module consisting of two handlers: the Input and Output handlers, detailed as follows.

**Input Handler:** This handler intercepts incoming SOAP messages and uses the AXIOM API (http://ws.apache.org/axiom/) to extract context information and the request contents, which are then stored into an XML file. The context information of the request is extracted from the SOAP header and contains the recipient identity and the purpose of the invocation. The business service is then invoked by our Axis2 engine.

**Output Handler:** The output handler intercepts the output SOAP response message before it is sent out of the service engine and makes sure that it complies with the applicable privacy and security policies. To do so, the RDF View Modification component
parses the security and privacy policies associated with the invoked service using the DOM API and extracts the rules that apply to the accessed data items for the recipient and the purpose at hand. It rewrites the RDF view to take into account these extracted rules as explained in the previous sections. Then, the *RDF View Rewriting component* decomposes the obtained extended view into a set of calls to data services that retrieve the different data items requested by the extended view. The obtained composition is then executed. As a final step, the *Result Filtering component* enforces the privacy and the security constraints on the obtained results. The output SOAP message is built and the filtered results are sent to the service consumer.

### 9.3.2 Evaluation

To evaluate the efficiency of our model, we applied it to the healthcare domain. In the context of the PAIRSE Project (http://picoforge.int-evry.fr/cgi-bin/twiki/view/Pairse/), we were provided with a set of /411/ medical data services accessing synthetic medical information (e.g., diseases, medical tests, allergies, etc) of more than /30,000/ patients. The access to these medical data was conditioned by a set of /47/ privacy and security rules. For each patient, we have randomly generated data disclosure preferences with regard to /10/ medical actors (e.g., researcher, physician, nurse, etc.) and different purposes (e.g., scientific research). These preferences are stored in an independent database and accessed via /10/ data services, each giving the preferences relative to a particular type of medical data (e.g., ongoing treatments, allergies, etc.). We deployed all of these services on our extended AXIS server running on a machine with 2.2GHz of CPU and 8GB of memory.
Figure 9.15: The experimental results

We conducted a set of experiments to measure the cost incurred in the enforcement of security and privacy policies during the service invocation. Specifically, we evaluated: (i) the cost \( c_1 \) incurred in computing the extended view and writing it in terms of services, and (ii) the cost \( c_2 \) incurred in enforcing the security and privacy constraints on retrieved data (i.e., the cost incurred in the filters). For that purpose, in the first set of experiments we executed the services to return the medical information about one given patient (e.g., patient Bob). In the second set, we executed the same services to return the medical information for all patients. In the first set of experiments, as the services return the information of one patient only, \( c_2 \) can be neglected and remains only \( c_1 \). In the second set, \( c_2 \) is amplified by the number of processed patients.

The executed services in our experiments were selected such that they have different sizes of RDF views (namely, /1/ class-node, /3/ class-nodes, and /5/ class-nodes). The invocations were made by the same actor (a researcher) and for the same purpose (medical research). Figure 9.15 depicts the results obtained for the invocations in Sets 1 and 2. The results for Set 1 show that security and privacy handling adds only a slight increase in the service invocation time. This could be attributed to the following reasons: (i) the time needed to inject the security and privacy constraints in the service’s RDF view is almost negligible, (ii) rewriting the \( v_{extended} \) in terms of services is not expensive, as most of \( v_{extended} \)’s graph is already covered by \( v_{original} \) and the size of \( \Delta v \) does not exceed generally 20% of the size of \( v_{original} \), and finally (iii) there is no network overhead incurred in invoking the additional services as they are already deployed on the server. The results for Set 2 show that \( c_2 \) is still relatively low if compared to the time required for executing the services without addressing the security and privacy concerns.
9.4 Use cases

9.4.1 AGGREGO Server

AGGREGO [27] is a commercial semantic mediation system. It uses SPARQL as query language. Figure 9.16 shows the AGGREGO architecture.

AGGREGO does not take into account security issues and privacy requirements. Its main goal is the management of heterogeneous sources and database integration. Our goal is to integrate our component fQuery into AGGREGO in order to handle and take into account the security and privacy requirements. fQuery produces queries that are expressed in SPARQL 1.1. However AGGREGO does not support SPARQL 1.1, it supports only queries that are expressed in SPARQL 1.0. fQuery-AC could be configured in order to produce queries that are expressed in SPARQL 1.0 (see chapter 3 for more details). However, fQuery-Privacy output contains a service block which is not supported by SPARQL 1.0. So, the output of the privacy rewriting algorithm could not be used as an input of the AGGREGO system.

To solve this problem, we proposed and developed a component Smart-fQuery, based on fQuery-AC and fQuery-Privacy and SPARQL engine (Jena-ARQ for SPARQL 1.1). Figure 9.17 illustrates the architecture of the component Smart-fQuery.
Smart-fQuery component is located between the mediator and the business applications. It considers the AGGREGO mediator as datasource accessible via a SPARQL service. It intercepts SPARQL query $Q_0$ issued by a business application. Afterwards it rewrites $Q_0$ into $Q_1$ using $f$Query-AC. Then it rewrites $Q_1$ into $Q_2$ using $f$Query-Privacy such that $Q_2$ contains two service blocks. The first one aims to invoke the mediator in order to get the corresponding data ($Q_{data}$) and the second one aims to get corresponding preferences ($Q_{pref}$). Finally $Q_2$ is executed by the SPARQL engine 1.1.

Let us take an example to illustrate the approach. We assume that a doctor Bob is allowed to see information of his patients. Bob is trying to select the name and age of all patients. So the SPARQL query $Q_0$ issued by Bob is as follow:

```
SELECT ?name ?age
WHERE {
  ?p O:hasName ?name.
  ?p O:hasAge ?age.
}
```

Figure 9.17: Smart-fQuery component architecture
We assume that a business application issues the query $Q_0$ on-behalf of Bob. $Q_1$, the rewritten query of $Q_0$ using $f$Query-AC, is as follows:

```
SELECT ?name ?age
WHERE {
?p O:hasName ?name.
?p O:hasAge ?age.
{
?d rdf:type O:Doctor.
?d O:hasId 'Bob'.
}
}
```

$Q_1$ is then rewritten into $Q_2$ using $f$Query-Privacy as follows:

```
SELECT ?name ?age
WHERE {
# Q_data: mediator
SERVICE <http://localhost/mediator/aggrego> {
?p O:hasName ?name_1.
?p O:hasAge ?age_1.
{
?d rdf:type O:Doctor.
?d O:hasId 'Bob'.
}
}
# Q_pref: privOrBAC service
SERVICE <http://localhost:8082(privorbac) {
?do rdf:type p:DataOwner;
?p:hasId ?name_1;
?pref p:hasPurpose 'purpose-1';
?p:hasRecipient 'Bob';
?p:hasTarget ?t0 .
?t0 p:hasName 'name';
?p:hasDecision ?nameDecision .
OPTIONAL { ?t0 p:hasAccuracy ?nameAccuracy}
?pref p:hasTarget ?t1 .
?t1 p:hasName 'age';
?p:hasDecision ?ageDecision .
OPTIONAL { ?t1 p:hasAccuracy ?ageAccuracy}
}
#Applying preferences
BIND( IF( ( ?nameDecision = 'No' ), '−',
IF( bound(?nameAccuracy), udf:eval(?name_1,?nameAccuracy), ?name_1) ) AS ?name)
BIND( IF( ( ?ageDecision = 'No' ), '−',
IF( bound(?ageAccuracy), udf:eval(?age_1,?ageAccuracy), ?age_1) ) AS ?age)
}
```
So the SPARQL query $Q_{\text{data}}$ that will be executed by the mediator is as follows:

```sparql
SELECT * WHERE {
  ?p O:hasName ?name_1 .
  ?p O:hasAge ?age_1 .
  {
    ?d rdf:type O:Doctor .
    ?d O:hasId 'Bob' .
  }
}
```

### 9.4.2 PAIRSE

The ANR PAIRSE project [126] addresses the challenges of heterogeneity and efficient query-processing for the need of data sharing in P2P environments, by advancing a web service-based approach and by taking into account the data privacy dimension. In the PAIRSE framework (see figure 9.18), there are two types of query processing; *Single* and *Multi Peer* query processing.

In Single query processing, the peer handles local queries which are issued against its local ontology by local users. Queries are expressed in the SPARQL query language. Local queries are resolved using local data-providing web services (DP) and a service composition approach based on an efficient RDF query rewriting algorithm. The later takes as input the query $Q$ and the defined RDF views of local services, and produces as an output a composition of DP services that would answer the query.

In the multi peer query processing mode (MPQP), there are some interaction with other peers in the network for the resolution of a query (either a global query or local query that cannot be answered locally). The query has to be splited into sub-queries (SPARQL queries). The component MPQP determines which sub-queries can be solved locally. For parts which cannot be solved locally, they will be relayed on to neighboring peers for processing.

There are two levels of security policy: (i) global policy and (ii) local policy. Global policy is the common policy defined for all PAIRSE peers. Local policy is specific to a PAIRSE peer. Figure 9.19 illustrates the integration of our approach $f$Query into the PAIRSE framework. The global policy is enforced by the smart-$f$Query component which is integrated between a user and PAIRSE peer. The local policy is enforced by the local service provider using the approach presented in chapter 8.
9.4. USE CASES

Figure 9.18: General architecture of the PAIRSE framework
Figure 9.19: $f$Query Component integration in PAIRSE framework
In this thesis we present an approach that enforces the security and privacy requirements based on pre-processing principle for SPARQL query language. It is to rewrite the SPARQL user query such that the execution of the rewritten query returns only authorized data with respect to some confidentiality and privacy preferences policy. Moreover, the rewriting algorithm is instrumented by an access control model (OrBAC) for confidentiality constraints and a privacy-aware model (PrivOrBAC) for privacy constraints. We take into account various dimensions of privacy preferences through the concepts of consent, accuracy, purpose and recipient. However, obligations are not yet taken into account. A possible extension will be to handle provisional obligations.

We show also that the algorithm used in the case of select queries could not be used for update ones. So, we present another approach for update queries, based on query rewriting. This approach aims to rewrite SPARQL update queries without disclosing some other sensitive data whose access would be forbidden through select queries.

Then we present a use case of our approach in the case of web services. We proposed a secure and privacy-preserving execution model for data services. Our model exploits the services’ semantics to allow service providers to enforce locally their privacy and security policies without changing the implementation of their data services i.e., data services are considered as black boxes. We integrated our model to the architecture of Axis 2.0 and evaluated its efficiency in the healthcare application domain. The obtained results are promising. As a future work, we plan to address data privacy concerns when composing autonomous data services with conflicting privacy policies.

We show also that our approach fQuery could be integrated in mediation system, as AGGREGO Server [27], Humboldt Discoverer [25] and Semantic Agreement [26], in order to:
• enforce the *global policy*\(^1\) by adding the \(f\)Query module among the mediator as illustrated in chapter 9.

• enforce the *local policy*\(^2\) by implementing the secure web services model presented in chapter 8.

## Perspectives

### Correctness

Our approach satisfies the security criteria. However, the maximality and soundness criteria may not be satisfied like in the case of query with filters as illustrated in chapter 6. It depends on the semantic of SPARQL filters and how they interpret obfuscated and anonymized data. An interesting future work is to extend the semantic of SPARQL filters, in the case of obfuscated data, in order to preserve the soundness and maximality criteria. For instance, the comparison operators could be extended in order to interpret and compare intervals and numbers.

### RDFS: RDF Schema

The actual version of \(f\)Query does not take into account all RDFS properties. For instance, the inheritance of concepts (classes). Let Person and Employee be two concepts such that Employee inherits from Person. Normally, if a user is not allowed to see properties of individuals of Person concept, then he is not allowed to see properties of individuals of Employee concept. But this inference is not yet implemented by \(f\)Query. There are two possible ways to add this into our approach. The first one is to integrate the inference process into the instrumentation process (dynamic approach). The second one is to infer all possible policy rules form the RDFS ontology of the system (where the SPARQL query is expressed) and the existing policy rules (static approach) without changing the actual version of \(f\)Query.

### Extending Query Normalization

SPARQL queries that contain patterns where predicates are variables (SPARQL variables), are not supported by the current version of \(f\)Query approach. This kind of

---

\(^1\)security policy defined at the mediator level  
\(^2\)security policy defined at each source provider level
queries are rejected from the normalization step, because they have more than one interpretation, i.e. they could match with more than one property or concept of the main ontology. For example the following query returns all the triples stored in databases:

```
SELECT ?s ?p ?o
WHERE{
  ?s ?p ?o
}
```

Actually the query normalization transforms implicit filters to explicit ones. A possible extension is to take into account this kind of queries. For example, in the normalization step we generate possible queries by replacing predicates variables with a concrete value, then rewriting each query.
A.1 Introduction

Les travaux présentés dans cette thèse consistent à développer des algorithmes de réécriture de requêtes d’accès aux ressources basés sur des langages ontologiques. Nous nous intéressons plus particulièrement aux requêtes exprimées dans le langage SPARQL sur une base de données RDF. Il s’agit de montrer que ces algorithmes permettent d’assurer la protection de la vie privée lors de la composition de web services pour satisfaire ces requêtes.

En premier lieu, nous avons défini des algorithmes de réécriture des requêtes SPARQL pour assurer la confidentialité des données reposant sur le langage ontologique RDF. Dans cette approche, que nous avons appelée j\textit{Query}, nous modélisons une politique de confidentialité comme un ensemble de filtres positifs et négatifs (correspon-
dant respectivement à des autorisations et interdictions) qui s’appliquent aux requêtes SPARQL.

En deuxième lieu, nous avons défini d’autres algorithmes de réécriture visant à protéger les requêtes de mise à jour SPARQL/Update en utilisant la transformation de requête. Ils assurent la confidentialité et l’intégrité des documents RDF. Ils consistent à contrôler la cohérence entre les politiques de mise à jour et de consultation des données (entre les opérateurs ‘select’ et ‘update’). L’approche utilisée reste aussi valable pour les autres modèles comme le modèle des vues pour SQL et le modèle de Stonebraker pour les bases de données INGRES. Enfin, nous avons implanté un prototype, pour l’approche fQuery.

En troisième lieu, nous avons montré la possibilité de représenter et d’exprimer une telle politique de sécurité avec le même langage de spécification utilisé par le modèle de contrôle d’accès OrBAC [8]. Dans l’approche fQuery, la politique de contrôle d’accès est spécifiée par un ensemble de filtres. Ces derniers sont représentés dans le modèle OrBAC sous forme de conditions logiques du premier ordre à l’aide de l’entité “Contexte”. En outre, nous avons défini une approche qui instrumente les algorithmes de réécriture cités auparavant par un modèle de contrôle d’accès tel que OrBAC [8] ou RBAC [57]...

En quatrième lieu, nous avons défini une nouvelle approche qui prend en compte la dimension de “privacy” reposant sur la réécriture des requêtes SPARQL. Nous modélisons la politique de préférences des utilisateurs à l’aide du modèle de privacy PrivOrBAC (basé sur le modèle de contrôle d’accès OrBAC). Afin de simplifier l’accès aux préférences des utilisateurs via le langage SPARQL, nous avons défini une ontologie de préférence qui consiste à représenter le point d’entrée aux services du modèle PrivOrBAC. Cette ontologie est instrumentée par le modèle PrivOrBAC. Le but de cette approche est d’assurer la prise en charge des contraintes de sécurité (préférences) définies par les possesseurs des données.

En cinquième lieu, nous avons intégré nos approches dans le cas de composition des services web. Dans le cas d’une composition normale, lorsqu’un utilisateur exprime sa demande, une liste des algorithmes propose à celui-ci une composition des services qui couvre sa demande. Dans l’autre cas (l’ajout des contraintes de sécurité), la demande de l’utilisateur subira des modifications en lui rajoutant des contraintes de sécurité et de privacy à l’aide des algorithmes de réécriture définis auparavant. Le résultat de cette réécriture est composé d’une liste des filtres F et d’une nouvelle demande D qui sera l’entrée des algorithmes de composition de services. Le résultat de l’invocation de la composition de D sera filtré à l’aide de l’ensemble de filtres F.
A.2. GÉNÉRALITÉS

En dernier lieu, nous avons implanté notre approche fQuery. Ensuite nous l’avons intégré dans plusieurs systèmes comme AGGREGO [27], PAIRSE et dans des systèmes de gestion de composition des services web.

A.2 Généralités

RDF [10] (Resource Definition Framework) est un modèle de graphe pour décrire les données. Il est basé sur l’idée de représenter des ressources (en particulier les ressources Web) sous la forme d’expressions de type sujet-prédicat-objet. Ces expressions sont appelées des triplets dans la terminologie RDF. Le “sujet” représente la ressource à décrire, le “prédicat” représente les propriétés ou les aspects de la ressource et exprime une relation entre le sujet et l’objet. Par exemple, une façon de représenter la proposition “le salaire de Bob est 60K” est le triplet : un sujet noté “Bob”, un prédicat noté “a le salaire” et un objet noté “60K”. Une collection des déclarations RDF représente un multi-graphe orienté. En tant que tel, un modèle de données RDF est plus adapté à certains types de représentations des connaissances que le modèle relationnel et d’autres modèles ontologiques traditionnellement utilisés dans l’informatique d’aujourd’hui.

En pratique, de plus en plus de données sont stockées dans des formats RDF. Ceci a donné naissance au besoin d’un moyen simple d’extraire et localiser des informations spécifiques. SPARQL [13] est un langage de requête puissant qui remplit ce besoin. Il facilite la recherche des données dans un graphe RDF. Il est standardisé par le groupe de travail Data Access du Consortium W3C, et est considéré comme une technologie clé du web sémantique. Une requête SPARQL est composée de motifs de graphe (graph patterns) obligatoires ou optionnels ainsi que de leurs conjonctions ou de leurs disjonctions. Par exemple, la requête ci-dessous retourne les noms et les salaires de tous les employés.

```sql
PREFIX emp:<http://tb.eu/employer/0.1/>
SELECT ?nom ?salaire
WHERE {
}
```

La syntaxe de SPARQL ressemble à celle de SQL. Mais l’avantage de SPARQL est de permettre des requêtes s’appliquant à de multiples sources disparates (locales ou distantes) de données contenant des données hétérogènes semi-structurées. Puisque une requête SPARQL peut accéder à des données confidentielles, il est nécessaire de concevoir des mécanismes de sécurité pour contrôler l’évaluation des requêtes SPARQL.
et empêcher les divulgations non autorisées de données confidentielles et des données relatives à la vie privée.

Notre approche consiste à réécrire une requête SPARQL en appliquant des filtres SPARQL. Lorsqu’un utilisateur envoie sa requête SPARQL au serveur, notre système l’intercepte de manière à vérifier les règles de sécurité s’appliquant à cet utilisateur. Il réécrit ensuite la requête en ajoutant les filtres SPARQL correspondants. Enfin, le résultat de l’exécution de la requête réécrite est envoyé à l’utilisateur.

A.3 *f*Query-AC: Principe de base

*f*Query-AC est le nom de l’approche qui assure la confidentialité des données. Prenons un exemple de transformation de requête pour illustrer notre approche. Nous supposons que l’utilisateur Bob essaie de sélectionner les noms et les salaires des employés. Nous supposons également que Bob n’est pas autorisé à voir les salaires des employés qui gagnent plus que 60K. Le tableau [A.1] montre la requête SPARQL de Bob, avant et après la transformation. La présence de la partie OPTIONAL dans la requête transformée en fait une requête disjonctive. Cela signifie que si la condition à l’intérieur de la clause OPTIONAL est fausse, alors la valeur de la variable salaire est affectée à NULL.

La politique de contrôle d’accès est basée sur des définitions de filtres. Pour chaque utilisateur ou groupe d’utilisateurs, nous définissons un ensemble de filtres. Selon le type de la politique de sécurité, nous considérons deux types de filtres : (1) des filtres positifs correspondant à des permissions et (2) des filtres négatifs correspondant à des interdictions.
Ces filtres peuvent être associés à une condition simple ou une condition composée. Les filtres associés à une condition composée fournissent les moyens de protéger les associations. Dans notre approche, nous supposons que lorsqu’un utilisateur formule une requête, nous pouvons obtenir des informations supplémentaires sur cet utilisateur, comme son identité. Ces informations peuvent être utilisées dans la définition du filtre.

Les filtres fournissent une approche générique pour représenter une politique de contrôle d’accès pour les documents RDF qui n’est pas reliée à un langage spécifique. En outre, nous avons montré la possibilité de représenter et d’exprimer une telle politique de sécurité avec le même langage du modèle de contrôle d’accès OrBAC. Les filtres sont représentés dans le modèle OrBAC sous forme de conditions logiques du premier ordre à l’aide de l’entité “Contexte”.

Dans le cas de mise à jour, la réécriture se fait en deux étapes. La première étape consiste à satisfaire les contraintes de mise à jour. La deuxième est de traiter la cohérence entre les règles de mise à jour et celles de consultation afin d’éviter des violations des règles de confidentialité en effectuant des mises à jour. Le principe de l’approche consiste (i) à restreindre la modification de la requête de l’utilisateur sur l’ensemble des données que l’utilisateur a le droit de modifier. Ensuite (ii) contrôler les conditions de la clause WHERE de la requête de mise à jour afin de limiter la lecture sur l’ensemble des données que l’utilisateur a le droit de consulter.

A.4  *f*Query-Privacy: Principe de base

*f*Query-Privacy est le nom de l’approche que nous proposons pour protéger la vie privée des utilisateurs. Elle consiste à réécrire la requête de l’utilisateur de sorte que les préférences des propriétaires des données soient prises en compte. Nous nous sommes focalisés sur les législations nationales et les conventions internationales [82, 83, 84, 85, 86, 87] en vigueur pour déduire les principes de la vie privée. Nous avons extrait les cinq dimensions qui sont repris par la majorité de ces lois:

- **Le Propriétaire de la donnée:** Le propriétaire est la personne physique que ces données permettent d’identifier directement ou indirectement. Il a un droit d’opposition sur ses données sur une simple demande.
- **La Spécification de l’objectif:** l’objectif de la collecte ou la consultation doit être spécifié avant d’accéder aux données.
- **Le Demandeur:** La personne à qui ces données vont être divulguées.


- **La précision**: La précision à appliquer sur les données après avoir eu l’autorisation d’y accéder.

- **Le contrôle d’usage**: Contrôler l’usage des données après y avoir accédé.

Les étapes de réécriture peuvent être résumées par les points suivants:

- La requête initiale est réécrite en insérant un appel au service de préférences et en insérant les contraintes de sécurité correspondantes.

- La réécriture a pour objectif d’assurer l’exécution de la requête réécrite (i) récupère les données (ii) et leurs préférences correspondantes à partir de service de préférences inséré dans la première étape de réécriture. Ensuite (iii) il applique les filtres de sécurité correspondants ainsi que ceux de la requête initiale.

Notre algorithme de réécriture est instrumenté par un modèle de protection de la vie privée (PrivOrBAC) reposant sur le modèle OrBAC. PrivOrBAC [9] propose une liste de service web permettant de lire et modifier les préférences des utilisateurs. En utilisant ces services, nous avons construit un service SPARQL qui nous permet de récupérer les préférences définies dans le modèle PrivOrBAC à l’aide des requêtes SPARQL.

Prenons un exemple pour illustrer notre approche. Supposons que le docteur Bob veut sélectionner les noms et les âges des patients majeurs pour un traitement médical. La requête SPARQL initiale \( Q_i \) de Bob est comme suit:

```sparql
PREFIX o:<http://tb.eu/patient/0.1/#>
SELECT ?name ?age WHERE {
  ?p rdf:type o:Patient;
  o:name ?name;
  o:age ?age. FILTER(?age >= 18)
}
```

L’approche consiste à insérer l’appel au service des préférences de la vie privée dans la requête initiale. Ce service consiste à récupérer les préférences définies par chaque propriétaire de données (patients) pour le demandeur Bob, pour un objectif de traitement médical, par rapport aux deux propriétés nom et âge. Puis récupérer les données demandées dans la requête initiale (sans les filtres). Ensuite appliquer les préférences de la vie privée associées à chaque donnée. Finalement appliquer les filtres de la requête initiale sur les nouvelles valeurs. La requête réécrite \( Q_{rw} \) de \( Q_i \) est comme suit:

```sparql
PREFIX o:<http://tb.eu/patient/0.1/#>
SELECT ?name ?age WHERE {
  ?p rdf:type o:Patient;
  o:name ?name;
  o:age ?age. FILTER(?age >= 18)
}
```
A.5 Le cas des services de données

Les services de données sont des composants logiciels qui encapsulent une série d'opérations centrées sur les données à travers des objets métiers. Ils rendent les détails sur la localisation des sources de données et comment elles sont accédées, abstraits en
APPENDIX A. FQUERY: RÉÉCRITURE DE REQUÊTES SPARQL

vue de l’utilisateur des données. Les services de données sont utilisés dans plusieurs domaine comme le Cloud Computing, les architectures orientées services SOA ...

Notre challenge est de permettre aux fournisseurs de services de pouvoir prendre en compte les dimensions de la sécurité et de la vie privée sans changer le code source de leurs services. C’est à dire que les services de données sont considérés comme des boites noires.

Notre approche s’appuie sur une modélisation déclarative des services de données en utilisant des vues RDF. Lorsqu’un service de données est invoqué, notre modèle réécrit la vue RDF du service en utilisant nos deux approches précédentes fQuery-Ac et fQuery-Privacy, afin de prendre en compte les contraintes de sécurité et de la vie privée. Ensuite la vue réécrite est transformée en termes d’appels des services de données à l’aide d’un service de composition. Ce dernier prend en entrée une vue RDF et il produit un plan d’exécution de services de données qui couvre la totalité de la vue RDF. Enfin les services sont alors exécutés, et les contraintes sont appliquées sur les résultats retournés.

Prenons un exemple pour illustrer notre approche. Dans un hôpital X, on suppose que les infirmières ont le droit de consulter les informations des patients du département cardiology. Les données des patients sont aussi contrôlées par une politique de la vie privée définie par chaque patient. Supposons que dans l’hôpital X, on dispose des services suivants:

- \( S_1(\text{name}, \text{age}) \): un service qui retourne les noms et les âges des patients
- $S_2($name, ?department): un service qui prend en entrée le nom d’un patient et qui retourne les départements dont lesquels ce patient est traité.

- $S_3($name, $recipient, $purpose, $target, ?content): un service de préférence de la vie privée. Il prend en entrée le nom de patient, le demandeur, l’objectif de la demande, le nom de la propriété demandée et il retourne le consentement de patient.

Alice est une infirmière. Elle veut consulter les noms et les âges des patients de l’hôpital X. Alice va donc invoquer le service $S_1$ qui couvre sa demande. Notre approche consiste à récupérer la vue RDF de service $S_1$ (voir figure A.1) lors de son invocation. Ensuite on réécrit cette vue avec nos deux algorithmes $fQuery$-$AC$ et $fQuery$-$Privacy$. Cette réécriture consiste à rajouter la contrainte du contrôle d’accès associée à l’infirmière Alice, ensuite récupérer les préférences de toutes les propriétés

Figure A.2: La vue réécrite $V'$ du service $S_1$
privées impliquées dans la requête. La figure A.2 illustre le résultat de la réécriture $V'$ de la vue RDF de service $S_1$.

Finalement, la vue réécrite $V'$ sera transformée en appels de services de données à l’aide d’un algorithme de composition des services. La figure A.3 illustre un exemple de plan d’exécution associé à la vue $V'$.

![Diagram](image.png)

Figure A.3: Le plan d’exécution associé à la vue réécrite $V'$

Ce plan d’exécution consiste à invoquer le service $S_1$ afin de récupérer les noms et les âges des patients. Ensuite, pour chaque nom de patient, il invoque en parallèles les services $S_2$ et $S_3$. Le service $S_2$ est invoqué afin de récupérer les départements de chaque patient. Le service $S_3$ est invoqué pour récupérer le consentement de chaque patient concernant la consultation de son nom, âge et département par une infirmière pour un objectif de traitement médical. Après une jointure des résultats d’invocation, on applique les préférences sur les données et on sélectionne les données ayant comme valeur de département “Cardiologie”. Enfin, on projette le résultat sur les propriétés nom et âge.

### A.6 Implémentation

Notre approche fQuery a été développée, intégrée et testée dans plusieurs systèmes, par exemple (1) le système de médiation sémantique commercial AGGREGO [27], (2) la plateforme de médiation sémantique du domaine médical PAIRSE, avec une architecture Pair-à-Pair, et (3) dans le service web Axis 2.0. Voir le chapitre 9 pour plus de détails sur l’implémentation de l’approche fQuery.
In this appendix we prove the theorem 3 of chapter 3. The theorem says that all complex conditions with level $L \geq 2$ could be represented by an involved condition by using abstract properties.

Our proof is based on “mathematical induction”. We start by proving the result for $L = 2$. Then we suppose that this result is valid for $L \geq 2$ and we prove it for $L + 1$.

**Case of $L = 2$**

Let $n \in \mathbb{N}^*$. Let $\omega$ be a complex condition associated with $\{(p_i, \omega_i)\}_{1 \leq i \leq n}$ with level $L = 2$.

We have: $(\forall x = (s, p, o) \in E)$

$$
\omega(x) = \begin{cases}
  \text{True} & \text{if } (\exists(x_1, \ldots, x_n) \in E^n)/(\forall 1 \leq i \leq n)x_i = (s, p_i, o_i) \\
  \text{False} & \text{otherwise}
\end{cases}
$$

such that $\max(\text{level}(\omega_i)) = 1$ i.e. $\text{level}(\omega_i) \in \{0, 1\}$. So $\omega_i$ is a simple condition ($\text{level}(\omega_i) = 0$) or an involved condition ($\text{level}(\omega_i) = 1$).

Let $J$ be a subset of $[1, n]$ such that:

$$
(\forall j \in J) \quad \text{level}(\omega_j) = 1 \quad \text{i.e. } \omega_j \text{ is an involved condition}
$$

Let $x = (s, p, o)$ be an element of $E$ such that $\omega(x) = \text{True}$.

Then $(\exists(x_1, \ldots, x_n) \in E^n)/(\forall i \in [1, n])x_i = (s, p_i, o_i) \quad \text{and} \quad \omega_i(o_i) = \text{True}$

In particular, $(\forall j \in J) \quad x_j = (s, p_j, o_j) \quad \text{and} \quad \omega_j(o_j) = \text{True}$ such that $\omega_j$ is an involved condition.
Let \( j \in J \). Let \( \{ \omega_{j,k} \}_{1 \leq k \leq m_j} \) be a set of simple conditions associated to the involved condition \( \omega_j \).

By definition of an involved condition we have:

\[
\omega_j(o_j) = \text{True} \iff [(\exists(y_1, \ldots, y_{m_j}) \in E^{m_j})/\forall k \in [1, m_j] \quad y_k = (o_j, p_{j,k}, o_{j,k})
\text{and} \quad \omega_{j,k}(o_{j,k}) = \text{True}]
\]

Let \( k \in [1, m_j] \), we denote \( p_{j,abst}^{j,k} \) the abstract property defining the relation between \( s \) (subject of \( x \)) and \( o_{j,k} \). \( p_{j,abst}^{j,k} \) is defined as follows:

\[
\{s \ p_{j,k}^{abst} \ o_{j,k}\} \iff \\
\{s \ p_j \ o_j, \\
o_j \ p_{j,k} \ o_{j,k}\}.
\]

We denote \( z_k = (s, p_{j,k}^{abst}, o_{j,k}) \). We deduce that:

\[(\forall j \in J)(\exists(z_1, \ldots, z_{m_j}) \in E^{m_j})/\forall k \in [1, m_j] \quad z_k = (s, p_{j,k}^{abst}, o_{j,k})
\text{and} \quad \omega_{j,k}(o_{j,k}) = \text{True} \quad \text{and} \quad \omega_{j,k} \text{ is a simple condition.}
\]

Finally:

\[
(\forall x = (s, p, o) \in E)
\omega(x) = \begin{cases} 
\text{True} & \text{if } (\exists(x_1, \ldots, x_N) \in E^N)/\forall i \in [1, N] x_i = (s, p_i, o_i) \\
& \text{where } o_i \in E_{\text{object}} \text{ and } \omega_{N,i}(o_i) = \text{True} \\
\text{False} & \text{Otherwise}
\end{cases}
\]

such that \( \{\omega_{N,i}\}_{1 \leq i \leq N} \) are simple conditions, \( N = (n - \text{card}(J)) + \sum_{j \in J} m_j \) and

\((\forall j \in J) \ p_j \) is an abstract property.

**Case of** \( L \geq 2 \)

Let \( L \) be an integer such that \( L \geq 2 \). We suppose that the result of the theorem is valid for each level \( l \) of \([2, L]\), then we prove that it is valid for the level \( L + 1 \).

Let \( n \in \mathbb{N}^* \). Let \( \omega \) be a complex condition associated with \( \{(p_i, \omega_i)\}_{1 \leq i \leq n} \) with level \( L + 1 \).

We have: \((\forall x = (s, p, o) \in E)\)

\[
\omega(x) = \begin{cases} 
\text{True} & \text{if } (\exists(x_1, \ldots, x_n) \in E^n)/\forall 1 \leq i \leq n x_i = (s, p_i, o_i) \\
& \text{where } o_i \in E_{\text{object}} \text{ and } \omega_i(o_i) = \text{True} \\
\text{False} & \text{Otherwise}
\end{cases}
\]
such that $\max(\text{level}(\omega_i)) = L$, so $(\forall i \in [0, n]) \; \text{level}(\omega_i) \in [0, L]$.

Let $C, I$ and $S$ be subsets of $[1, n]$ such that:

\[
\begin{cases} 
(\forall j \in C) & \text{level}(\omega_j) \in [2, L] \\
(\forall j \in I) & \text{level}(\omega_j) = 1 \\
(\forall j \in S) & \text{level}(\omega_j) = 0 \\
C \cup I \cup S = [1, n]
\end{cases}
\]

Let $j \in [1, n]$. We distinguish three cases:

- if $j \in C$ i.e. level($\omega_j$) $\in [2, L]$, then, following the assumption, $\omega_j$ could be presented as an involved condition, denoted $\psi_j$.

- if $j \in I$ i.e. level($\omega_j$) $= 1$, then, by definition, $\omega_j$ is an involved condition.

- if $j \in S$ i.e. level($\omega_j$) $= 0$, then, by definition, $\omega_j$ is a simple condition.

For $j \in (I \cup S)$ we define $\psi_j$ as $\omega_j$ i.e. $\psi_j = \omega_j$.

We deduce that $\omega$ could be presented by a complex condition $\psi$ associated with $\{(q_i, \psi_i)\}_{1 \leq i \leq n}$ with level $L = 2$, such that:

\[
(\forall i \in [1, n]) \; \text{level}(\psi_i) = \begin{cases} 1 & \text{if } i \in C \cup I \\ 0 & \text{if } i \in S \end{cases}
\]

According to the case $L = 2$, we deduce that $\psi$ could be presented by an involved condition $\phi$.

Finally, $\omega$, a complex condition with level $L + 1$, could be presented by the involved condition $\phi$. 
package fr.enstb.lussi.fquery.algo.aspect;

import java.util.Iterator;
...

/**
 * Aspect of SPARQL Query Rewriting Algorithm for security reasons.
 * @author Said OULMAKHZOUNE
 */
public aspect Algorithm {
    private Condition omega;
    private boolean permission;
    private List<ElementFilter> filters;

    /**
     * @param omega
     * @param permission
     */
    public Algorithm() {
        omega = new DefaultCondition();
        permission = false;
        filters = new LinkedList<ElementFilter>();
    }

    /**
     * This point cut is used in order to insert corresponding filter of each TriplesBlock
     */
    pointcut eachGroupElement():
        call(Element SPARQLParser10.GroupGraphPatternSub()) && target(SPARQLParser10);

    /**
     * This point cut is used in order to handle each parsed triple pattern.
     * Check if it respects the security policy
     */
    pointcut insertCall():
call(void SPARQLParser10.insert(TripleCollector, int, Node, Node, Path, Node));

/**
 * @param parser
 */
pointcut insertTriple(SPARQLParser10 parser) : insertCall() && target(parser);

/**
 * @param parser
 */
void around(SPARQLParser10 parser) : insertTriple(parser){
    if(!(omega instanceof SimpleCondition)){
        proceed(parser);
        return;
    }
    SimpleCondition scondition = (SimpleCondition) omega;
    Object[] args = thisJoinPoint.getArgs();
    //TripleCollector acc = (TripleCollector) args[0];
    //int index = (Integer) args[1];
    Node s = (Node) args[2];
    Node p = (Node) args[3];
    //Path path=(Path) args[4];
    Node o = (Node) args[5];
    if(p==null){
        proceed(parser);
        return;
    }
    Triple triple = new Triple(s, p, o);
    OmegaResult v = OmegaHelpers.eval(scondition, triple);
    if(EValue.TRUE.equals(v.getEValue())) {
        if(!permission){
            proceed(parser); //nothing todo
        }
    } else if(EValue.FALSE.equals(v.getEValue())){
        if(!permission){
            proceed(parser); //nothing todo
        }
    } else if(EValue.EXPR.equals(v.getEValue())){
        //Omega(triple) is expressed on terms of triples variable
        proceed(parser);
        String expr ;
        if(permission)
            expr = v.getExpression();
        else
            expr = '!'+'v.getExpression()+'';
        ElementFilter el = Helpers.createSPARQLFilter(expr);
        filters.add(el);
    } else{
        System.out.println('Error ');
    }
Element around():

```java
ElementGroup el = (ElementGroup) proceed();
if (omega instanceof DefaultCondition)
    return el; // normal case: nothing to do

if (omega instanceof InvolvedCondition) {
    List<Element> elements = el.getElements();
    for (Iterator<Element> iterator = elements.iterator(); iterator.hasNext();)
        if (element instanceof ElementTriplesBlock) {
            ElementTriplesBlock etb = (ElementTriplesBlock) element;
            InvolvedCondition ic = (InvolvedCondition) omega
            algorithmInvolvedCondition(ic, etb.getPattern().getList());
        }
}
```

/******* The following code is available for each kind of condition ******/
/* add filters */
for (ElementFilter filter : filters) {
    el.addElement(filter);
}
/* then clean the filter list */
filters.clear();
return el;
```

```java
public void algorithmInvolvedCondition(InvolvedCondition omega,
List<Triple> triples) {
    ConditionResult scResult =
    AlgorithmTools.algoInvolvedCondition(omega, permission, triples);
    triples.addAll(scResult.getTriplesToAdd());
    filters.addAll(scResult.getFilters());
}
```

/******* @return the permission *******
```
public boolean isPermission() {
    return permission;
}
```

/******* @param permission the permission to set *******
```
public void setPermission(boolean permission) {
    this.permission = permission;
}
```
/**
 * @return the omega
 */
public Condition getOmega() {
    return omega;
}

/**
 * @param omega the omega to set
 */
public void setOmega(Condition omega) {
    this.omega = omega;
}


```java
package fr.swid.fquery.condition.sparql.visitor;
import java.util.Iterator;
...

/**
 * fQuery-AC: Rewriting algorithm as visitor
 *
 * @author Said OULMAKHZOUNE
 */
public class RWElementVisitor implements ElementVisitor {
    private Condition omega;
    private boolean permission;
    private LinkedList<List<ElementFilter>> filters;

    /**
     * @param omega
     * @param permission
     */
    public RWElementVisitor(Condition omega, boolean permission) {
        super();
        this.omega = omega;
        this.permission = permission;
        this.filters = new LinkedList<List<ElementFilter>>() {
            super();
            this.omega = omega;
            this.permission = permission;
            this.filters = new LinkedList<List<ElementFilter>>() {
                super();
                if (omega instanceof SimpleCondition) {
                    BasicPattern pattern = el.getPattern();
                    SimpleCondition sc = (SimpleCondition)omega;
                    ConditionResult scResult =
```

APPENDIX

fQuery-AC Visitor
AlgorithmTools.algoSimpleCondition(sc, permission, pattern);

for (Triple triple : scResult.getTriples()) {
    pattern.remove(triple);
}

// filters
this.filters.addLast(scResult.getFilters());
return;
}

if(omega instanceof InvolvedCondition){
    BasicPattern pattern = el.getPattern();
    InvolvedCondition ic = (InvolvedCondition)omega;
    ConditionResult scResult =
        AlgorithmTools.algoInvolvedCondition(ic, permission, pattern.getList());
    for (Triple triple : scResult.getTriples()) {
        pattern.add(triple);
    }

    // filters
    this.filters.addLast(scResult.getFilters());
    return;
}

/**
 * When visiting an union block
 * @see com.hp.hpl.jena.sparql.syntax.ElementVisitor#visit(
 * com.hp.hpl.jena.sparql.syntax.ElementUnion)
 *
 */
@Override
public void visit(ElementUnion el) {
    for (Iterator<Element> iter=el.getElements().listIterator(); iter.hasNext();){
        Element subElement = iter.next();
        subElement.visit(this);
    }
}

/**
 * When visiting an optional block
 * @see com.hp.hpl.jena.sparql.syntax.ElementVisitor#visit(
 * com.hp.hpl.jena.sparql.syntax.ElementOptional)
 *
 */
@Override
public void visit(ElementOptional el) {
    visitAsGroup(el.getOptionalElement());
}

/**
 * When visiting a group element block
 * @see com.hp.hpl.jena.sparql.syntax.ElementVisitor#visit(
 * com.hp.hpl.jena.sparql.syntax.ElementGroup)
 *
 */
@Override
public void visit(ElementGroup el) {
    for (Iterator<Element> iter=el.getElements().listIterator(); iter.hasNext();){
        Element subElement = iter.next();
        subElement.visit(this);
    }

    // adding filters here
List<ElementFilter> groupfilters = filters.removeLast();
for (ElementFilter filter : groupfilters) {
    el.addElement(filter);
}

/*
 * @param el
 */
public void visitAsGroup(Element el){
    el.visit(this);
}

@Override
public void visit(ElementFilter el) {}
@Override
public void visit(ElementDataset el) {}
@Override
public void visit(ElementAssign el) {}
@Override
public void visit(ElementPathBlock el) {}
@Override
public void visit(ElementNamedGraph el) {}
@Override
public void visit(ElementService el) {}
@Override
public void visit(ElementFetch el) {}
@Override
public void visit(ElementSubQuery el) {}
@Override
public void visit(ElementExists el) {}
@Override
public void visit(ElementNotExists el) {}
List of Publications

International Conferences


International Journals


National Conferences


National Journals


[83] OECD. Organisation for economic co-operation and development. ‘protection of privacy and transborder flows of personal data’. September 1980. 69, 70, 90, 149


[104] Michael J. Carey. Declarative data services: This is your data on soa. In *SOCA*, page 4, 2007. 103


List of Figures

2.1 SPARQL Query with Graph clause ........................................ 8
2.2 SPARQL Federated Query .................................................. 8
2.3 View-based approach: (a) Combination of building blocks, (b) Illustration, (c) Processing flow ............................................. 13
2.4 Pre-Processing approach: (a) Combination of building blocks, (b) Illustration, (c) Processing flow ............................................. 15
2.5 Post-Processing approach: (a) Combination of building blocks, (b) Illustration, (c) Processing flow ............................................. 17
3.1 Query approach ............................................................... 20
3.2 (A) Permission case. (B) Prohibition case .............................. 37
3.3 (A) and (B) Permission case. (C) and (D) Prohibition case ...... 39
4.1 Consistency between select and update operators .................... 46
5.1 Example from the employee ontology ................................... 64
5.2 Example of a generated views from the employee ontology ........ 64
5.3 Relation between $V_{\text{name}}, V_{\text{salary}}$ and generated views ........ 65
5.4 Relation between $V_{\text{complex}}$ and generated views .................. 66
6.1 Our approach principle ..................................................... 72
6.2 Our approach using SPARQL Service ................................... 74
6.3 Privacy Preferences Ontology .............................................. 76
6.4 Privacy Preferences Example ........................................ 77

7.1 The Privacy-aware OrBAC model .................................. 91

7.2 Accuracy levels of location data .................................. 95

7.3 The instrumentation approach of privacy rewriting .......... 96

7.4 RDF views of $S_1$, $S_2$ and $S_3$ ............................. 97

7.5 Transformation of Bob’s SPARQL Query ....................... 99

7.6 Composition execution plan of $V$ ............................... 100

8.1 Overview of the Privacy and Security aware Execution Model .. 106

8.2 Part-A: the RDF View of $S_1$; Part-B: its graphical representation ... 108

8.3 The SPARQL and the graphical representations of the patient’s consent 109

8.4 (a) The original view of $S_1$; (b) The extended view after applying the security policy; (c) The extended view after applying the privacy policy 110

8.5 A graphical representation of the services $S_2$ and $S_3$ .................. 110

8.6 The Obtained Composition ....................................... 113

8.7 The intermediate and final results ................................ 114

9.1 MotOrBAC Tool .................................................. 118

9.2 MotOrBAC Architecture ......................................... 119

9.3 Management of RDF conditions ................................ 121

9.4 OrBAC Context and RDF condition assignment .............. 121

9.5 Predicates Management: Mapping and definitions ............ 122

9.6 Definition of prefixes: URI shortcut .......................... 122

9.7 Test and simulation screen of $f$Query-Privacy algorithm ........ 124

9.8 Managing private properties mapping $\mathcal{M}$ .................. 124

9.9 Managing users’ preferences and PrivOrBAC SPARQL Service .......... 125

9.10 Implementation architecture of our approach .................. 126

9.11 The rewritten query $Q_{rw3}$ of $Q_{N3}$ ......................... 130
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.12</td>
<td>Execution time of initial queries $Q_1$, $Q_2$ and $Q_3$</td>
<td>131</td>
</tr>
<tr>
<td>9.13</td>
<td>Execution time of rewritten queries $Q_{rw1}$, $Q_{rw2}$ and $Q_{rw3}$</td>
<td>132</td>
</tr>
<tr>
<td>9.14</td>
<td>The extended architecture of AXIS 2.0</td>
<td>133</td>
</tr>
<tr>
<td>9.15</td>
<td>The experimental results</td>
<td>134</td>
</tr>
<tr>
<td>9.16</td>
<td>AGGREGO Server architecture</td>
<td>135</td>
</tr>
<tr>
<td>9.17</td>
<td>Smart-fQuery component architecture</td>
<td>136</td>
</tr>
<tr>
<td>9.18</td>
<td>General architecture of the PAIRSE framework</td>
<td>139</td>
</tr>
<tr>
<td>9.19</td>
<td>$f$Query Component integration in PAIRSE framework</td>
<td>140</td>
</tr>
<tr>
<td>A.1</td>
<td>La vue RDF du service $S_1$</td>
<td>152</td>
</tr>
<tr>
<td>A.2</td>
<td>La vue réécrite $V'$ du service $S_1$</td>
<td>153</td>
</tr>
<tr>
<td>A.3</td>
<td>Le plan d'exécution associé à la vue réécrite $V''$</td>
<td>154</td>
</tr>
</tbody>
</table>