Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Elettronica, Informatica e delle Telecomunicazioni

Ciclo XXI

Settore/i scientifico-disciplinare/i di afferenza: ING-INF/05

Security control brought back to the user

Presentata da: Juri Luca De Coi

Coordinatore Dottorato	Relatori
Chiar.ma prof. P. Mello	Chiar.mo prof. A. Natali
	Chiar.mo prof. W. Nejdl

Esame finale anno 2010

Ringraziamenti – Danksagung – Acknowledgments

A joint Ph.D. is a challenge not only for the Ph.D. candidate, who has both the privilege and the burden to get in touch with different ways of thinking, but also for people assisting her, who sometimes have to face non-trivial (bureaucratic) problems. For this reason, I owe my gratitude to

- my supervisors Antonio Natali and Wolfgang Nejdl
- my further referees Evelina Lamma and Matthew Smith
- the Ph.D. student managers Paolo Bassi and Paola Mello

I would also like to thank Daniel Olmedilla (resp. Peter Fankhauser), who led an inexperienced (resp. not completely inexperienced) Ph.D. student along his way, as well as the colleagues with whom I shared ideas, visions and work, especially Philipp Kärger, Arne Wolf Kösling and Sergej Zerr.

Dass in mir der Wunsch überhaupt entstanden ist ins Ausland zu ziehen, verdanke ich Julija Samsonova. Einen unentbehrlichen Beitrag zur Erfüllung dieses Wunsches trug Doris Anita Höhmann. Per l'assistenza nei primi tempi del mio soggiorno in terra straniera il mio ringraziamento va a Katia Cappelli.

Ein ruhiges Privatleben ist eine notwendige Voraussetzung um eine Promotion durchzuführen. Dafür bin ich vor allem Yuliyana Dimitrova und Anna Schulze dankbar. Se il significato dell'espressione "vita sociale" ancora mi è noto, lo devo in gran parte a Carla Conte e Gian Luca Volpato.

Meine analytische Denkweise hat es mir nicht ermöglicht Giorgia Cremonese zu entschlüsseln, wobei Karen Stöckers synthetische Denkweise deren geistliche Verwandtschaft unverzüglich erkannt hat. Es liegt vielleicht daran, dass ich mich mit beiden so wohl gefühlt habe. Il mio approccio analitico non mi ha permesso di venire a capo di Giorgia Cremonese, mentre l'approccio sintetico di Karen Stöcker non ha esitato a ravvisarne l'affinità spirituale. Sarà per questo che mi sono trovato così bene con ambedue.

CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

Riassunto ix

Zusammenfassung x

Abstract xi

List of publications xii

1. Introduction 1

    1.1 The concept of policy 2

    1.2 From uid/psw-based authentication to trust negotiation 6

2. A Review of the State-of-the-art 8

    2.1 Related work 8

    2.2 Considered policy languages 9

    2.3 Comparison criteria 10

        2.3.1 Core policy properties 11

        2.3.2 Contextual properties 12

    2.4 Comparison 14

    2.5 Discussion 21

3. The Protune Policy Language 24

    3.1 Protune's conceptual space 24

        3.1.1 The user interface 26

        3.1.2 The communication channel 26

        3.1.3 Protune policies 27

        3.1.4 Evaluation strategies 28

    3.2 A gentle introduction to Protune's syntax 29

    3.3 A gentle introduction to Protune's semantics 34

        3.3.1 Provisional atoms 36

        3.3.2 Comparison atoms 38

        3.3.3 Value-assignment atoms 41

3.4 Protune's syntax . . . 42
3.4.1 Terms . . . 42
3.4.2 Atoms . . . 43
3.4.3 Goals and rules . . . 44
3.4.4 Metaatoms . . . 46
3.4.5 Metarules . . . 47
3.4.6 Policy . . . 48
3.5 (Toward) Protune's semantics . . . 48
3.5.1 Basic definitions . . . 48
3.5.2 Terms . . . 49
3.5.3 Atoms . . . 50
3.5.4 Goals and rules . . . 51
3.5.5 Metaatoms . . . 52
3.5.6 Metarules . . . 52
3.5.7 Policy . . . 52
3.6 What's new . . . 53
4. Using Policies in a Natural Way . . . 58
4.1 Controlled natural languages . . . 60
4.1.1 Attempto Controlled English . . . 61
4.1.2 ACE as a natural language . . . 62
4.1.3 Discourse Representation Structure . . . 63
4.2 Mapping DRS to Protune . . . 66
4.2.1 Mapping requirements . . . 67
4.2.2 Mapping . . . 67
4.3 Usability issues . . . 72
4.3.1 A predictive authoring tool . . . 73
4.3.2 Editor's features . . . 74
4.3.3 Evaluation . . . 75
4.4 Related work . . . 79
5. Access Control across Desktops . . . 81
5.1 The general-purpose strategy . . . 82
5.1.1 The conceptual framework . . . 82
5.1.2 A naïve general-purpose strategy . . . 84

5.1.3 The general-purpose strategy revisited . . . 87
5.2 The special-purpose strategy . . . 89
5.2.1 The Datalog case . . . 89
5.2.2 The Protune case . . . 100
5.3 Experimental results . . . 101
5.4 Related work . . . 108
6. Access Control in RDF Stores . . . 110
6.1 Related work . . . 111
6.2 Policy-based query expansion . . . 113
6.2.1 RDF queries . . . 113
6.2.2 Specifying policies over RDF data . . . 116
6.2.3 Policy evaluation and query expansion . . . 119
6.3 Implementation . . . 129
6.3.1 Architecture . . . 129
6.3.2 Experiments and evaluation . . . 132
7. Conclusions and Outlook . . . 135
BIBLIOGRAPHY . . . 136
Acknowledgments . . . 145
Lebenslauf . . . 146

LIST OF TABLES

1.1 Example policies 4

2.1 Policy language comparison 22

3.1 Comparisons supported by default 39

4.1 A standard NLP tool's processing steps vs. APE's ones 62

4.2 An example DRS 64

4.3 A pretty-printed DRS 66

4.4 DRSs representing PROTUNE rules 69

4.5 DRS→PROTUNE mapping examples 70

4.6 Sentences exploited in the user study 76

5.1 Relaxation of condition $P_1 \models p(\vec{x}) \dot{\vee} P_2 \models p(\vec{x})$ 93

5.2 Relaxation of condition $P_1 \models \theta B_R \dot{\vee} P_2 \models \theta B_R$ 95

6.1 Policies for RDF statements 118

6.2 Application of policies to triple patterns 122

6.3 Application of function *policyFiltering* 127

LIST OF FIGURES

3.1 Example PROTUNE policy 30

4.1 Example PROTUNE policy 59

4.2 A predictive authoring tool 76

5.1 A basic diagram of a generic authorization framework 85

5.2 The diagram of our authorization framework 86

5.3 Algorithm to compute the *modified change function* 97

5.4 Beagle^{++} architecture and semantic data sharing 102

5.5 Schema of the database used in our evaluation 103

5.6 Overhead of adding/removing resources 104

5.7 Overhead of adding/removing policies 105

5.8 Optimized vs. non-optimized policy evaluation 106

5.9 Algorithm to find the policy applicable to a given resource 107

6.1 Example RDF query 115

6.2 Expanded RDF query 130

6.3 Architecture of AC4RDF 131

6.4 Response time when increasing the number of literals 134

Riassunto

L'attività di ricerca dell'Ing. Juri Luca De Coi può essere suddivisa in tre parti. La prima parte ha riguardato l'indagine dello stato dell'arte in *policy language* ed ha prodotto i seguenti contributi:

- Identificazione dei requisiti di un moderno *policy language*;
- Definizione di un linguaggio che soddisfa tali requisiti (PROTUNE);
- Implementazione di un *framework* per l'attuazione di PROTUNE *policy*.

La seconda parte si è focalizzata sulla semplificazione del processo di definizione di *policy* ed ha prodotto i seguenti contributi:

- Identificazione di un sottoinsieme del linguaggio naturale controllato ACE[1] adatto ad esprimere PROTUNE *policy*;
- Implementazione del *mapping* tra ACE *policy* e PROTUNE *policy*;
- Creazione di un *editor* in grado di guidare gli utenti passo-passo nella definizione di PROTUNE *policy*.

La terza parte ha testato la fattibilità dell'approccio scelto mediante l'applicazione a scenari di concreta rilevanza, tra cui:

- Controllo d'accesso basato su *policy* per *metadata store*;
- Sviluppo di una sovrastruttura di sicurezza per RDF *store*.

L'attività di ricerca è stata svolta in forte collaborazione con la Leibniz Universität Hannover e altri partner europei all'interno dei progetti REWERSE[2], TENCompetence[3] ed OKKAM[4].

[1] `http://attempto.ifi.uzh.ch/site/`
[2] `http://rewerse.net/`
[3] `http://www.tencompetence.org/`
[4] `http://www.okkam.org/`

Zusammenfassung

Die im Laufe der Promotion von Herrn Juri Luca De Coi durchgeführten Forschungsarbeiten befassten sich mit dem Gebiet der *policy languages* und lassen sich in drei Teile gliedern. Im ersten Teil wurde der Stand der Technik im Bereich *policy languages* untersucht und erweitert, mit den folgenden Ergebnissen:

- Identifizierung der Anforderungen, die von modernen Policy-Sprachen erfüllt werden sollten;
- Definition der Policy-Sprache PROTUNE, die diese Anforderungen erfüllt;
- Implementierung einer Infrastruktur, die PROTUNE Policies interpretiert.

Der zweite Teil der Arbeit fokussierte die Vereinfachung des Definitionsverfahrens von PROTUNE Policies, mit den folgenden Ergebnissen:

- Identifizierung einer zur Formulierung von PROTUNE-Policies geeigneten Untermenge der kontrollierten natürlichen Sprache ACE[1];
- Implementierung einer automatischen Übersetzung von ACE-Policies in PROTUNE-Policies;
- Erweiterung des ACE-Editors für die automatisierte Unterstützung von Benutzern bei der Definition von ACE-Policies.

Im dritten Teil wurde die Anwendung der in den ersten beiden Teilen vorgestellten Ansätze auf realistische Szenarien untersucht. Dazu zählt

- der leistungsfähige und skalierbare Policy-basierte Zugriff auf Metadaten und
- die Entwicklung einer Sicherheitebene für RDF-Datenbanken.

Die Forschungen wurden in enger Zusammenarbeit mit der Alma Mater Studiorum – Università di Bologna als auch mit weiteren europäischen Partnern innerhalb der Projekte REWERSE[2], TENCompetence[3] und OKKAM[4] durchgeführt.

[1] `http://attempto.ifi.uzh.ch/site/`
[2] `http://rewerse.net/`
[3] `http://www.tencompetence.org/`
[4] `http://www.okkam.org/`

Abstract

The activity of the Ph.D. student Juri Luca De Coi involved the research field of policy languages and can be divided in three parts. The first part of the Ph.D. work investigated the state of the art in policy languages, ending up with

- identifying the requirements up-to-date policy languages have to fulfill
- defining a policy language able to fulfill such requirements (namely, the PROTUNE policy language)
- implementing an infrastructure able to enforce policies expressed in the PROTUNE policy language

The second part of the Ph.D. work focused on simplifying the activity of defining policies and ended up with

- identifying a subset of the controlled natural language ACE[1] to express PROTUNE policies
- implementing a mapping between ACE policies and PROTUNE policies
- adapting the ACE Editor to guide users step by step when defining ACE policies

The third part of the Ph.D. work tested the feasibility of the chosen approach by applying it to meaningful real-world problems, among which

- efficient policy-aware access to metadata stores
- development of a security layer on top of RDF stores

The research activity has been performed in tight collaboration with European partners within the projects REWERSE[2], TENCompetence[3] and OKKAM[4].

[1] http://attempto.ifi.uzh.ch/site/
[2] http://rewerse.net/
[3] http://www.tencompetence.org/
[4] http://www.okkam.org/

List of publications

[1] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. Enabling advanced and context-dependent access control in RDF stores. In *ISWC/ASWC*, pages 1–14, 2007.

[2] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. A user interface to define and adjust policies for dynamic user models. In *WEBIST*, pages 184–191, 2009.

[3] Piero A. Bonatti, Juri Luca De Coi, Daniel Olmedilla, and Luigi Sauro. A rule-based trust negotiation system. *Accepted for TDKE*.

[4] Piero A. Bonatti, Juri Luca De Coi, Daniel Olmedilla, and Luigi Sauro. Policy-driven negotiations and explanations: Exploiting logic-programming for trust management, privacy & security. In *ICLP*, pages 779–784, 2008.

[5] Piero A. Bonatti, Juri Luca De Coi, Daniel Olmedilla, and Luigi Sauro. Rule-based policy representations and reasoning. In *REWERSE*, pages 201–232. 2009.

[6] Juri Luca De Coi, Peter Fankhauser, Tobias Kuhn, Wolfgang Nejdl, and Daniel Olmedilla. Controlled natural language policies. In *CCS*, 2009.

[7] Juri Luca De Coi, Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Controlled English for reasoning on the Semantic Web. In *REWERSE*, pages 276–308. 2009.

[8] Juri Luca De Coi, Eelco Herder, Arne Kösling, Christoph Lofi, Daniel Olmedilla, Odysseas Papapetrou, and Wolf Siberski. A model for competence gap analysis. In *WEBIST*, 2007.

[9] Juri Luca De Coi, Philipp Kärger, Arne Wolf Koesling, and Daniel Olmedilla. Exploiting policies in an open infrastructure for lifelong learning. In *EC-TEL*, pages 26–40, 2007.

[10] Juri Luca De Coi, Philipp Kärger, Arne Wolf Koesling, and Daniel Olmedilla. Control your eLearning environment: Exploiting policies in an open infrastructure for lifelong learning. *TLT*, 1(1):88–102, 2008.

[11] Juri Luca De Coi, Philipp Kärger, Daniel Olmedilla, and Sergej Zerr. Semantic Web policies for security, trust management and privacy in social networks. In *ICAIL*, pages 112–119, 2009.

[12] Juri Luca De Coi, Philipp Kärger, Daniel Olmedilla, and Sergej Zerr. Using natural language policies for privacy control in social platforms. In *SPOT*, pages 112–119, 2009.

[13] Juri Luca De Coi and Daniel Olmedilla. A flexible policy-driven trust negotiation model. In *IAT*, pages 450–453, 2007.

[14] Juri Luca De Coi and Daniel Olmedilla. A review of trust management, security and privacy policy languages. In *SECRYPT*, pages 483–490, 2008.

[15] Juri Luca De Coi, Daniel Olmedilla, Piero A. Bonatti, and Luigi Sauro. Protune: A framework for Semantic Web policies. In *International Semantic Web Conference (Posters & Demos)*, 2008.

[16] Juri Luca De Coi, Daniel Olmedilla, Sergej Zerr, Piero A. Bonatti, and Luigi Sauro. A trust management package for policy-driven protection & personalization of web content. In *POLICY*, pages 228–230, 2008.

[17] Ekaterini Ioannou, Juri Luca De Coi, Arne Wolf Koesling, Daniel Olmedilla, and Wolfgang Nejdl. Access control for sharing Semantic Data across desktops. In *PEAS*, 2007.

[18] Arne Wolf Koesling, Eelco Herder, Juri Luca De Coi, and Fabian Abel. Making legacy LMS adaptable using policy and policy templates. In *LWA*, pages 35–40, 2008.

[19] Piero A. Bonatti, Juri Luca De Coi, Wolfgang Nejdl, Daniel Olmedilla, Luigi Sauro, and Sergej Zerr. Policy based protection and personalized generation of web content. In *LA-WEB*, 2009.

CHAPTER 1
Introduction

Security management is a foremost issue in large-scale networks like the World Wide Web. In such a scenario, traditional assumptions for establishing and enforcing access-control regulations do not hold anymore. In particular, identity-based access-control mechanisms have proved to be ineffective, since in decentralized and multicentric environments the requester and the service provider are often unknown to each other.

Policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web: policies specify who is allowed to perform which action on which resource depending on properties of the requester and of the resource as well as parameters of the action and environmental factors (e.g., time). In the last years many languages have been proposed which allow to express policies in a formal way (*policy languages*). Some of them came together with software components (*policy engines*) able to enforce policies written in the corresponding policy language.

The potential policies have proved to own is not fully exploited yet, since nowadays their usage is mainly restricted to specific application areas and the policy languages proposed so far can be expected to be used only by security experts or other computer scientists: common users cannot profit from them, since almost no policy framework offers facilities or tools to meet the needs of users without a strong background in computer science.

Being usability a major issue in moving toward a policy-aware Web, this work tackles it by providing the following contributions.

1. The first complete description of the semantics of the policy language PROTUNE [19, 67] is provided

2. An extension of the PROTUNE language toward the Semantic Web vision is presented: a PROTUNE program is not self-contained anymore but points to

resources available on the Semantic Web which are automatically identified, located, deployed and finally exploited at policy evaluation time

3. The natural language front-end for the PROTUNE policy framework is described

4. A description how the PROTUNE engine can be used in order to create a security level on top of metadata stores and RDF repositories is provided. Such use cases have been addressed in order to test feasibility, efficacy and efficiency of the PROTUNE engine w.r.t. real-world scenarios

This Ph.D. thesis is organized as follows. Chapter 1 presents an overall picture of the research field: in particular, Section 1.1 introduces the concept of policy, whereas Section 1.2 outlines the historical evolution of policy languages. Chapter 2 presents the state-of-the-art in policy languages. Chapter 3 describes syntax and semantics of the last version of the PROTUNE language and especially focuses on its differences and improvements w.r.t. the previous version. Chapter 4 presents PROTUNE's ACE front-end which allows to define PROTUNE policies in natural language. Chapters 5 and 6 describe applications of the PROTUNE engine to real-world scenarios, namely how it can be used in order to create a security level on top of metadata stores and RDF repositories respectively. Finally, Chapter 7 presents open issues and further work, and concludes.

1.1 The concept of policy

According to [77]'s well-known definition, policies are "rules governing the choices in the behavior of a system", i.e., statements which describe which decision the system must take or which actions it must perform according to specific circumstances. *Policy languages* are special-purpose programming languages which allow to specify policies, whereas *policy engines* are software components able to enforce policies expressed in some policy language.

Policies are encountered in many situations of our daily life: the following example is an extract of a return policy of an on-line shop[1].

[1] `http://www.fancydressking.co.uk/returns-policy/info_5.html`

> Any item for return must be received back in its original shipped condition and original packing. The item must be without damage or use and in a suitable condition for resale. All original packaging should accompany any returned item. We cannot accept returns for exchange or refund if such items have been opened from a sealed package.

With the digital era, the specification of policies has emerged in many Web-related contexts and software systems. E-mail client filters like the following one are a typical example of policy.

> If the header of an incoming message contains a field `X-Spam-Flag` whose value is `YES` then move the message into the folder `INBOX.Spam`. Moreover, if this rule matches, do not check any other one.

Some of the main application areas where policies have been lately used are security and privacy as well as specific business domains, where they take on the name of "business rules" (but cf. [62, 63, 80] for other application areas).

A security policy defines security restrictions for a system, organization or any other entity. It may define which resources a system should regard as security relevant, in which way the resources should be protected and how the system should proceed, if access to those resources is requested by a third party. A privacy policy is a declaration made by an organization regarding its use of customers' personal information (e.g., whether third parties may have access to customer data and how that data will be used). Finally, business rules describe the operations, definitions and constraints that apply to an organization in achieving its goals. Tab. 1.1 presents excerpts of a security policy[2], a privacy policy[3] and a business rule[4] (in the business domain of a tax collection agency).

The use of formal policies yields many advantages compared to other approaches: formal policies are usually dynamic, declarative, have a well-defined semantics and allow to be reasoned over. In the following, all above-mentioned properties will be thoroughly described.

[2] `http://www.sans.org/resources/policies/DMZ_Lab_Security_Policy.doc`
[3] `http://www.siteprocentral.com/contracts/privacy_policy_sample.html`
[4] `http://law.onecle.com/texas/tax/11.145.00.html`

1.	Original firewall configurations and any changes thereto must be reviewed and approved by InfoSec (including both general configurations and rule sets). InfoSec may require additional security measures as needed.
2.	Our site users can choose to electronically forward a link, page, or documents to someone else by clicking "e-mail this to a friend". The user must provide their email address, as well as that of the recipient. This information is used only in the case of transmission errors and, of course, to let the recipient know who sent the email. The information is not used for any other purpose.
3.	A person is entitled to an exemption from taxation of the tangible personal property the person owns that is held or used for the production of income if that property has a taxable value of less than $500.

Table 1.1: Example policies: (1) a security policy; (2) a privacy policy; (3) a business rule

Dynamicity If the description of the behavior of an agent or other software component is built in the component itself, whenever the need for a different behavior arises, new code for that behavior must be written, compiled and deployed. A more reusable design choice provides the component with the ability of adapting itself according to some dynamically configurable description of the desired behavior (i.e., a policy). This way, as soon as the need for a different behavior arises, only the policy and not the whole component must be replaced.

Declarativeness The traditional (*imperative*) programming paradigm requires programmers to explicitly specify an algorithm to achieve a goal. On the other hand, the *declarative* approach simply requires programmers to specify the goal, whereas the implementation of the algorithm is left to the support engine. This difference is commonly expressed by resorting to the sentence "declarative programs specify *what* to do, whereas imperative programs specify *how* to do it". For this reason, declarative languages are commonly considered a step closer to the final user than imperative ones. Policy languages are typically declarative and policies are typically declarative statements: as such, common users should be able to define them in an easier way.

Well-defined semantics A language's semantics is well-defined if the meaning of a program written in that language is independent of the particular implementation of the language interpreter. Programs written in a language provided with well-defined semantics can be easily exchanged among different parties since each party understands them in the same way. Policy languages are usually based on some mathematical formalism (like Logic Programming [64] or Description Logic [9]), which ensures policies expressed in such languages to have a well-defined semantics. Formal policies have hence advantages over policies expressed by means of natural language sentences, which tend to be ambiguous and lend themselves to different interpretations.

Reasoning The term "reasoning" refers to the possibility of combining known information in order to infer new one, like in the following example.

> If it is known that "all humans are mortal" and that "Socrates is human" one can infer that "Socrates is mortal".

On the one hand it is true that the sentence "Socrates is mortal" is different than the ones preceding it, but on the other one it is clear that, according to the common sense, one can deduce (i.e., *infer*) the third sentence out of the first two. The inferred information is referred to as *implicit* knowledge, since it was not explicitly available before. In the context of declarative programs (and in particular of policy languages), statements a program consists of can be reasoned over in order to infer new statements. For instance, reasoning applied to the third policy in Tab. 1.1 allows to infer that, if

- John is a person
- John owns a car
- the car is a tangible personal property
- John uses the car in his daily work (and therefore he uses it for the production of income) and
- the car has a taxable value of less than $500

then John is entitled to an exemption from taxation of the car.

1.2 From uid/psw-based authentication to trust negotiation

Traditional access-control mechanisms (like the ones exploited in usual operating systems) make authorization decisions based on the identity of the requester: the user must provide a pair $(username, password)$ and, if this pair matches any of the entries in some static table kept by the system (e.g., the file `/etc/pwd` in Unix), the user is granted with some privileges. However, in decentralized and multicentric environments, peers are often unknown to each other and access control based on identities may be ineffective.

In order to address this scenario, role-based access-control mechanisms have been developed. In a role-based access-control system every user is assigned with one or more roles which are in turn exploited in order to take authorization decisions. Since the number of roles is typically much smaller than the number of users, role-based access-control systems reduce the number of access-control decisions. A thorough description of role-based access control can be found in [46].

In a role-based access-control system the authorization process is split in two steps, namely assignment of one or more roles and check whether a member of the assigned role(s) is allowed to perform the requested action. Role-based languages usually provide support only to one of the two steps, moreover role-based authentication mechanisms require that the requester provides some information in order to map her to some role(s). In the easiest case this information can be once again a (uid, pwd) pair, but systems which need a stronger authentication usually exploit credentials, i.e., digital certificates representing statements certified by given entities (*certification authorities*) which can be used in order to establish properties of their holder.

More modern approaches (cf. Chapter 2) directly exploit the properties of the requester in order to make authorization decisions, i.e., they do not split the authorization process in two steps like role-based languages. Nevertheless, they do not use credentials in order to certificate the properties of the requester.

Credentials, as well as declarations (i.e., non-signed statements about proper-

ties of the holder), are however building blocks of languages designed to support the trust negotiation [60] vision: trust between peers is established by exchanging sets of credentials between them in a negotiation which may consist of several steps.

CHAPTER 2
A Review of the State-of-the-art in Policy Languages

In the last years many policy languages have been proposed, targeting different application scenarios and provided with different features and expressiveness: goal of this chapter is to compare most of such languages in order to outline the features an up-to-date policy language should provide. Such features have been used as a guideline when defining the PROTUNE policy language we will describe in Section 3.

Comparisons among policy languages have been already provided in the literature. However existing comparisons either do not consider a relevant number of available solutions or mainly focused on the application scenarios their authors worked with (e.g., trust negotiation in [75] or ontology-based systems in [81]). Moreover, policy-based security management is a rapidly evolving field and most of this comparison work is now out-of-date.

Currently, a broad and up-to-date overview covering most of the relevant available policy languages is lacking. In this chapter we intend to fill this gap by providing an extensive comparison covering eleven policy languages. Such a comparison will be carried out on the strength of ten criteria, partly already known in the literature and partly introduced in our work for the first time.

This chapter is organized as follows. In Section 2.1, related work is accounted for. Sections 2.2 and 2.3 introduce the languages which will be compared and the criteria according to which the comparison will be carried out respectively. The actual comparison takes place in Section 2.4, whereas Section 2.5 presents overall results and draws some conclusions.

2.1 Related work

The paper of Seamons et al. [75] is the basis of our work: some of the insights they provided have proved to be still valuable right now and as such they are recalled in our work as well. Nevertheless, in over seven years the research field has considerably changed and nowadays many aspects of [75] are out of date: new languages

have been developed and new design paradigms have been taken into account, what makes the comparison performed in [75] obsolete and many criteria according to which they were evaluated not suitable anymore.

The pioneer paper of Seamons et al. paved the way to further research on policy language comparisons like Tonti et al. [81], Anderson [7] and Duma et al. [38]. Although [81] actually presents a comparison of two ontology-based languages (namely, KAoS and Rei) with the Object-oriented language Ponder, the work is rather an argument for ontology-based systems, since it clearly shows the advantages of ontologies.

Because of the impressive amount of details it provides, [7] restricts the comparison to only two (privacy) policy languages, namely EPAL and XACML, therefore a comprehensive overview of the research field is not provided, and features which neither EPAL nor XACML support are not taken into account at all among the comparison criteria.

Finally, [38] provides a comparison specifically targeted to giving insights and suggestions to policy authors (*designers*). Therefore, the criteria according to which the comparison is carried out are mainly practical ones and scenario-oriented, whereas more abstract issues are considered out of scope and hence not addressed.

2.2 Considered policy languages

To date, a bunch of policy languages have been developed and are currently available: we have chosen those which at present seem to be the most popular ones, namely Cassandra [13], EPAL [8, 10], KAoS [82, 83], PeerTrust [42], Ponder [35], PSPL [21], Rei [48], *RT* [61], TPL [46], WSPL [6] and XACML [65, 43]. The information we will provide about the aforementioned languages is based on the referenced documents. Whenever a feature is not addressed in the considered literature nor is it known to the author in other way, that feature will be supposed not to be provided by the language.

The number and variety of policy languages proposed so far is justified by the different requirements they had to accomplish and the different use cases they were designed to support. Ponder is meant to help local security policy specification and

security management activities, therefore typical addressed application scenarios include registration of users or logging and audit events, whereas firewalls, operating systems and databases belong to the applications targeted by the language. WSPL's name itself (namely, Web Services Policy Language) suggests its goal: supporting the description and control of various aspects and features of a Web Service. Web Services are addressed by KAoS too, as well as general-purpose grid computing, although it was originally oriented to software agent applications where dynamic runtime policy changes need to be supported. Rei's design was primarily concerned with support to pervasive computing applications in which people and devices are mobile and use wireless networking technologies to discover and access services and devices. EPAL (Enterprise Privacy Authorization Language) was proposed by IBM to support enterprise privacy policies. Some years before, IBM had already introduced the pioneer role-based policy language TPL (Trust Policy Language), which paved the way to other role-assignment policy languages like Cassandra and *RT* (Role-based Trust-management framework), both of which aim to address access-control and authorization problems which arise in large-scale decentralized systems when independent organizations enter into coalitions whose membership and very existence change rapidly. The main goal of PSPL (Portfolio and Service Protection Language) was to provide a uniform formal framework for regulating service access and information disclosure in an open, distributed network system like the Web; support to negotiations and private policies were among the basic reasons which led to its definition. PeerTrust is a simple yet powerful language for trust negotiation on the Semantic Web based on a distributed request evaluation. Finally, XACML (eXtensible Access-Control Markup Language) was meant to be a standard general-purpose access-control policy language, ideally suitable to the needs of most authorization systems.

2.3 Comparison criteria

We acknowledge the remark made by [38], according to which a comparison among policy languages on the basis of the criteria presented in [75] is only partially satisfactory for a designer, since general features do not help in understanding

which kind of policies can be practically expressed with the constructs available in a language. Therefore, in our comparison we selected a good deal of criteria having a concrete relevance (e.g., whether actions can be defined within a policy and executed during its evaluation, how the result of a request looks like, whether the language provides extensibility mechanisms and to which extent). On the other hand, since we did not want to come short on theoretical issues, we selected four additional criteria, basically taken from [75], and somehow reworked and updated them. We call these more theoretical criteria *core policy properties*, whereas more practical issues have been grouped under the label *contextual properties*.

2.3.1 Core policy properties

Well-defined semantics As we already mentioned in Section 1.1, we consider a policy language's semantics to be well-defined if the meaning of a policy written in that language is independent of the particular implementation of the language. Logic Programs and Description Logic knowledge bases have a mathematically defined semantics, therefore we assume policy languages based on either of the formalisms to have well-defined semantics.

Monotonicity In the sense of logic, a system is monotonic if the set of conclusions which can be drawn from the current knowledge base does not decrease by adding new information to the knowledge base. In the sense of [75], a policy language is considered to be monotonic if an accomplished request would also be accomplished if accompanied by additional disclosure of information by the actors involved in the transaction: in other words, disclosure of additional evidences and policies should result in granting additional privileges. Policy languages may be non-monotonic in the sense of logic (as it happens with Logic Programming-based languages) but still be monotonic in the sense of [75].

Condition expressiveness A policy language must allow to specify under which conditions the request of the user (e.g., to perform an action or to disclose a credential) should be accomplished. Policy languages differ in the expressiveness of such conditions: some of them allow to set constraints on properties of the requester

but not on parameters of the requested action. Moreover, constraints on environmental factors (e.g., time) are not always supported. Cassandra's expressiveness can be even tuned by varying the constraint domain it is equipped with. This criterion subsumes "credential combinations", "constraints on attribute values" and "inter-credential constraints" in [75].

Underlying formalism A good deal of policy languages base on some well-known formalism: PSPL bases on Logic Programming, whereas Cassandra, Peer-Trust and *RT* on a subset of it (namely, Constrained Datalog) and KAoS on Description Logic. Knowledge about the formalism a language bases upon can be useful in order to understand some features of the language itself: e.g., the fact that a language is based on Logic Programming with negation (as failure) entails consequences regarding the monotonicity of the language (in the sense of logic), whereas knowing that Description Logic knowledge bases may contain contradictory statements could induce to infer that Description Logic-based languages needs to deal with such contradictions.

2.3.2 Contextual properties

Action execution During the evaluation of a policy, some actions may have to be performed: one may want to retrieve the current system time (e.g., in case authorization should be allowed only in a specific time frame), to send a query to a database or to record some information in a log file. Cassandra, equipped with a suitable constraint domain, allows to specify side-effect free actions, whereas Ponder and XACML support particular kinds of actions. It is worth noticing that this criterion evaluates whether a language enables the *policy author* to specify actions within a policy: during the evaluation of a policy the engine may carry out non-trivial actions on its own (e.g., both *RT* and TPL engines provide automatic resolution of credential chains) but such actions are not considered in our investigation.

Delegation Delegation is often used in access-control systems to cater for temporary transfer of access rights to agents acting on behalf of other ones (e.g., passing write rights to a printer spooler in order to print a file). The right of delegating is a right as well and as such can be delegated, too. Some languages provide a means for

cascaded delegations up to a certain length, whereas others allow unbounded delegation chains. In order to support delegation, many languages provide a specific built-in construct, whereas others exploit finer-grained features of the language in order to simulate high-level constructs. The latter approach allows to support more flexible delegation policies and is hence better suited for expressing the subtle but significant semantic differences which appear in real-world applications.

Evidences During the evaluation of authentication policies, the requester might have to provide *credentials* (cf. Section 1.2), i.e., digital certificates representing statements certified by given entities (*certification authorities*) which can be used in order to establish properties of their holder. Credentials are not supported by languages not targeting authentication policies (e.g., Ponder) nor by e.g., EPAL or KAoS. PeerTrust and PSPL provide another kind of evidence, namely *declarations* which are non-signed statements about properties of the holder (e.g., credit-card numbers).

Negotiation support [6] adopts a broad notion of "negotiation". A negotiation is supposed to happen between two actors whenever: (i) both actors are enabled to define a policy; and (ii) both policies are taken into account when processing a request. According to this definition, WSPL supports negotiations as well. In this work we adopt a narrower definition of negotiation by adding a third prerequisite stating that: (iii) the evaluation of the request must be distributed, i.e., both actors must evaluate the request locally and either decide to terminate the negotiation or send a partial result to the other actor who will go on with the evaluation. Whether the evaluation is local or distributed may be considered an implementation issue as long as policies can be freely disclosed. Distributed evaluation is required under a conceptual point of view as soon as the need for keeping policies private arises: indeed, if policies were not private, simply merging the actors' policies would reveal possible compatibilities between them.

Policy engine decision The result of the evaluation of a policy must be notified to the requester. The result sent back by the policy engine may carry information

to different extents: in the easiest case a boolean answer may be sent (allowed vs. denied). Some languages (e.g., EPAL, WSPL and XACML) support error messages, whereas Cassandra returns a set of constraints which is a subset of the one in the requester's query.

Extensibility Since experience shows that each system needs to be updated and extended with new features, a good programming practice requires to keep things as general as possible in order to support future extensions. Almost every language provides some support to extensibility. *RT* may be regarded as an exception since, as pointed out by [13], the need for more advanced features was handled by releasing a new flavor of the language (available *RT* flavors can be obtained by combining RT_0 and RT_1 on the one hand with RT^T and/or RT^D on the other one). In the following we provide a description of the mechanisms languages adopt in order to support extensibility.

2.4 Comparison

In this section the considered policy languages will be compared according to the criteria outlined in Section 2.3. The overall result of the comparison is summarized in Tab. 2.4.

Well-defined semantics As mentioned in Section 2.3.1, a policy language's semantics is well-defined if the meaning of a policy written in that language is independent of the particular implementation of the language. We assume policy languages based on Logic Programming or Description Logic to have well-defined semantics. The formalisms underlying the considered policy languages will be accounted for in the following. So far we restrict ourselves to listing the languages provided with a well-defined semantics, namely, Cassandra, EPAL, KAoS, PeerTrust, PSPL, Rei and *RT*.

Monotonicity In the sense of [75], a policy language is considered to be monotonic if disclosure of additional evidences and policies only results in granting additional privileges. Therefore, the concept of "monotonicity" does not apply to languages

which do not provide support for evidences, namely EPAL, Ponder, WSPL and XACML. All other languages are monotonic, with the exception of TPL, which explicitly supports *negative certificates*, stating that a user can be assigned a role R if there exists no credential of some type claiming something about her. The authors of TPL acknowledge that it is almost impossible proving that there does not exist such a credential somewhere. Therefore, they interpret their statement in a restrictive way, i.e., they assume that such a credential does not exist if it is not present in the local repository. Despite this restrictive definition, the language is still non-monotonic since, as soon as such a credential is released and stored in the local repository, consequences which could be drawn previously cannot be drawn anymore.

Condition expressiveness A role-based policy language maps requesters to roles. The assigned role is afterwards exploited in order (not) to authorize the requester to execute some action. The mapping to a role might in principle be performed according to the identity or other properties of the requester (to be stated by some evidence) and possibly environmental factors (e.g., current time). Cassandra, equipped with a suitable constraint domain, supports both scenarios.

Environmental factors are not taken into account by TPL, where the mapping to a role is only performed according to the properties of the requester. Such properties can be combined by using boolean operators, moreover a set of built-in operators (e.g., $>$, $=$) is provided in order to set constraints on their values.

Environmental factors are not taken into account by RT_0 either, where role membership is identity-based, meaning that a role must explicitly list its members. Nevertheless, since: (i) roles are allowed to express sets of entities having a certain property; and (ii) conjunctions and disjunctions can be applied to existing roles in order to create new ones; then role membership is eventually based on properties of the requester. RT_1 goes a step beyond and, by adding the notion of *parametrized role*, allows to set constraints not only on properties of the requester but also on the ones of the resource on which the requested action should be performed. This feature makes the second step traditional role-based policy languages consist of unnecessary.

For this reason, RT_1, as well as the other RT flavors basing on it, may be considered to lay on the border between role-based and non role-based policy languages.

A non role-based policy language does not split the authentication process in two different steps but directly provides an answer to the question whether the requester should be allowed to execute some action. In this case, the authorization decision can in principle be made not only according to properties of the requester or the environment, but also according to the ones of the resource on which the action would be performed as well as parameters of the action itself. EPAL introduces the further notion of "purpose" for which a request was issued and allows to set conditions on it. Some non role-based languages make a distinction between conditions which must be fulfilled in order for the request to be at all taken into consideration (which we call *prerequisites*, according to the terminology introduced by [21]) and conditions which must be fulfilled in order for the request to be satisfied (*requisites*, according to [21]): not always both kinds of conditions have the same expressiveness.

Let start checking whether and to which extent the non role-based policy languages we consider support prerequisites: WSPL and XACML only support a simple set of criteria to determine a policy's applicability to a request, whereas Ponder provides a complete solution which allows to set prerequisites involving properties of requester, resource, environment and parameters of the action. Prerequisites can be set in EPAL and PSPL as well: the expressiveness of PSPL prerequisites is the same as the one of its requisites, which we will discuss later.

With the exception of Ponder, which allows to set restrictions on environmental properties only for delegation policies, each other language supports requisites (Rei is even redundant in this respect): KAoS allows to set constraints on properties of the requester and the environment, Rei also on action parameters and PSPL, WSPL and XACML also on properties of the resource. EPAL supports conditions on the purpose for which a request was issued but not on environmental properties. Attributes must be typed in EPAL, WSPL, XACML and typing can be considered a constraint on the values the attribute can assume, anyway the definition of the semantics of such attributes is outside WSPL's scope. Finally, in PeerTrust

conditions can be expressed by setting guards on policies: each policy consists of a guard and a body, the body is not evaluated until the guard is satisfied.

Underlying formalism The most part of languages provided with a well-defined semantics rely on some kind of Logic Programming or Description Logic. Logic Programming is the semantic foundation of PSPL, whereas a subset of it, namely Constraint Datalog, is the basis for Cassandra, PeerTrust and *RT*. KAoS relies on Description Logic, whereas Rei combines features of Description Logic (ontologies are used in order to define domain classes and properties associated with the classes), Logic Programming (Rei policies are actually particular Logic Programs) and Deontic Logic (in order to express concepts like rights, prohibitions, obligations and dispensations). EPAL exploits Predicate Logic without quantifiers. Finally, no formalisms underly Ponder (which only bases on the Object-oriented paradigm), TPL, WSPL and XACML.

Action execution Ponder allows to access system properties (e.g., time) from within a policy. Moreover, it supports obligation policies, asserting which actions should be executed if some event happens: examples of such actions are printing a file, tracking some data in a log file and enabling/disabling user accounts. XACML allows to specify actions within a policy: these actions are collected during the policy evaluation and executed before sending a response back to the requester. A similar mechanism is provided by EPAL and of course by WSPL, which is indeed a specific profile of XACML. The only actions the policy author can specify in PeerTrust and PSPL are related to the sending of evidences. Cassandra (equipped with a suitable constraint domain) allows to call side-effect free functions (e.g., to access the current time). It is worth noticing that languages which allow to specify actions within policies can to some extent simulate obligation policies, as long as the triggering event is the reception of a request, although in such languages the flexibility provided by Ponder is not met. Finally, KAoS, Rei, *RT* and TPL do not support the execution of actions.

Delegation Ponder defines a specific kind of policies in order to deal with delegation: the field `valid` allows *positive* delegation policies to specify constraints (e.g., time restrictions) to limit the validity of the delegated access rights. Rei allows to define not only policies delegating rights but also policies delegating the right to delegate (some other right). Delegation is supported by RT^D ("D" stands indeed for "delegation"): being RT a role-based language, the right which can be delegated is the one of activating a role, i.e., the possibility of acting as a member of such a role. Ponder delegation chains have length 1, whereas RT delegation chains always have unbounded length. Cassandra provides a more flexible mechanism which allows to explicitly set the desired length of a delegation chain (as well as other properties of the delegation): in order to obtain such a flexibility, Cassandra does not provide high-level constructs to deal with delegation but simulates them by exploiting finer-grained features of the language. Delegation (of authority) can be expressed in PeerTrust by exploiting the operator `@`. Finally, EPAL, KAoS, PSPL, TPL, WSPL and XACML do not support delegation.

Type of evaluation The most part of the considered languages require that all policies to be evaluated are collected in some place before starting the evaluation, which is hence performed locally: this is the way EPAL, KAoS, Ponder, RT and TPL work. Cassandra, Rei, WSPL and XACML perform policy evaluation locally, nevertheless they provide some facility in order to collect policies or policy fragments which are spread across the network. For instance, in XACML combining algorithms define how to collect results of the evaluation of multiple policies and derive a single result, whereas Cassandra allows policies to refer to policies of other entities, so that policy evaluation may trigger requests for remote policies (possibly the requester's one) over the network. Policies can be collected in a single place if they can be freely disclosed, therefore the languages mentioned so far do not address the possibility that policies may have to be kept private themselves. Protection of sensitive policies can be granted only by providing support to distributed policy evaluation, like the one carried out by PeerTrust or PSPL.

Evidences The result of a policy's evaluation may depend on the identities or other properties of the actor who requested its evaluation: a means needs hence to be provided in order for the actors involved in the transaction to communicate such properties to each other. Such information is usually provided in the form of digital certificates signed by trusted entities (*certification authorities*) and called *credentials*. Credentials are a key element in Cassandra, *RT* and TPL, whereas they are unnecessary in Ponder, whose policies are concerned with limiting the activity of users who have already been successfully authenticated. The authors of PSPL were the first ones advocating for the need of non-signed statements (e.g., credit card numbers), which they called *declarations*. Declarations are supported by PeerTrust as well. Finally, EPAL, KAoS, Rei, WSPL and XACML do not support evidences.

Negotiation support As stated in Section 2.3.2, we use a narrower definition of negotiation than the one provided in [6], into which WSPL does not fit. Therefore, only pretty few languages support negotiation in the sense we specified above, namely Cassandra, PeerTrust and PSPL.

Policy engine decision The evaluation of a policy should end up with a result to be sent back to the requester. In the easiest case such result is a boolean stating whether the request was (not) fulfilled: KAoS, PeerTrust, Ponder, PSPL, *RT* and TPL conform to this pattern. Beside `permit` and `deny`, WSPL and XACML provide two other result values to cater for particular situations: `not_applicable` is returned whenever no applicable policies or rules could be found, whereas `indeterminate` accounts for some error which occurred during the processing. In the latter case, optional information is made available to explain the error.

A boolean value, stating whether the request was (not) fulfilled, does not make sense in the case of an obligation policy, which simply describes the actions which must be executed as soon as an event (e.g., the reception of a request) happens. Therefore, beside the so-called *rulings* `allow` and `deny`, EPAL defines a third value (`don't care`) to be returned by obligation policies. One of the elements an EPAL policy consists of is a global condition which is checked at the very begin-

ning of the policy evaluation: not fulfilling such a condition is considered an error and a corresponding error message (`policy_error`) is returned. A further message (`scope_error`) is returned in case no applicable policies were found.

Cassandra's request format contains (among other things) a set of constraints c belonging to some constraint domain: the response consists of a subset c' of c which satisfies the policy. In case $c' = c$ (resp. $c' = \emptyset$) `true` (resp. `false`) is returned. Rei obligation policies enable the requester to decide whether to complete the obligation by comparing the effects of meeting the obligation (`MetEffects`) and the effects of not meeting it (`NotMetEffects`).

Extensibility Extensibility is a fuzzy concept: almost all languages provide some extension points to let the user adapt it to her specific needs. Nevertheless, the extension mechanisms greatly vary from language to language: here we briefly summarize the means the various languages provide in order to address extensibility.

Extensibility is described as one of the criteria taken into account when designing Ponder: in order to smoothly provide support to new types of policies which may arise in the future, inheritance was considered a suitable solution and Ponder was hence implemented as an Object-oriented language.

XACML's support to extensibility is two-fold: (i) on the one hand, new datatypes, as well as functions for dealing with them, may be defined in addition to the built-in ones. Datatypes and functions must be specified in XACML requests, which indeed consist of typed attributes associated with the requester, the resource acted upon, the action being performed and the environment; (ii) as we mentioned above, XACML policies can consist of any number of distributed rules. XACML already provides a number of combining algorithms which define how to collect results of the evaluation of multiple policies and derive a single result. Nevertheless, a standard extension mechanism is available to define new algorithms.

Using non-standard user-defined datatypes would lead to wasting one of the strong points of WSPL, namely the standard algorithm for merging two policies into a single one which subsumes both of them (assuming that such a policy exists), since there can be no standard algorithm for merging policies which exploit user-defined

attributes (except in case the only comparison operators supported are = and ≠). The use of non-standard algorithms would in turn mean that WSPL policies could not be supported by a standard policy engine. Being standardization the main goal of WSPL, no wonder that it comes short on the topic "extensibility", which is not necessarily a drawback, if the assertion of [6] holds: "most Web Services will probably use fairly simple policies in their service definitions".

Ontologies are the means to cater for extensibility in KAoS and Rei: the use of ontologies facilitates a dynamic adaptation of the policy framework by specifying the ontology of a given environment and linking it with the generic framework ontology. Both KAoS and Rei define basic built-in ontologies, which are supposed to be further extended for a given application.

Extensibility was the main issue taken into account in the design of Cassandra. Its authors realized that standard policy idioms (e.g., role hierarchy or role delegation) occur in real-world policies in many subtle variants. Instead of embedding such variants in an *ad hoc* way, they decided to define a policy language able to express this variety of features smoothly. In order to achieve this goal, the key element is the notion of *constraint domain*, an independent module which is plugged into the policy engine in order to adjust the expressiveness of the language. The advantage of this approach is that the expressiveness (and hence the computational complexity) of the language can be chosen depending on the requirements of the application and can be easily changed without having to change the language semantics.

Finally, PeerTrust, PSPL, *RT* and TPL do not provide extension mechanisms.

2.5 Discussion

By carrying out the task of comparing a considerable amount of policy languages, we came to believe that they can be classified in two main groups collecting, so to say, *standard-oriented* and *research-oriented* languages respectively.

EPAL, WSPL and XACML can be considered standard-oriented languages since they provide a well-supported but restricted set of features. Although it is likely that this set will be extended as long as the standardization process proceeds,

	Cassandra	EPAL	KAoS	PeerTrust	Ponder	PSPL	Rei	RT	TPL	WSPL	XACML
Well-defined semantics	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No
Monotonicity	Yes	–	Yes	Yes	–	Yes	Yes	Yes	No	–	–
Underlying formalism	Constraint Datalog	Predicate Logic without quantifiers	Description Logic	Constraint Datalog	Object-oriented paradigm	Logic Programming	Deontic logic, Logic Programming, Description Logic	Constraint Datalog	–	–	–
Action execution	Yes (side-effect free)	Yes	No	Yes (only sending evidences)	Yes (access to system properties)	Yes (only sending evidences)	No	No	No	Yes	Yes
Delegation	Yes	No	No	Yes	Yes	No	Yes	Yes (RT^D)	No	No	No
Type of evaluation	Distributed policies, Local evaluation	Local	Local	Distributed	Local	Distributed	Distributed policies, Local evaluation	Local	Local	Distributed policies, Local evaluation	Distributed policies, Local evaluation
Evidences	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations	–	Credentials	Credentials	No	No
Negotiation	Yes	No	No	Yes	No	Yes	No	No	Yes	No (policy matching supported)	No
Result format	A/D and a set of constraints	A/D, `scope error`, `policy error`	A/D	A/D	A/D	A/D	A/D[a]	A/D	A/D	A/D, `not applicable`, `indeterminate`	A/D, `not applicable`, `indeterminate`
Extensibility	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes

Table 2.1: Policy language comparison ("–" = not applicable)

[a]Cf. Section 2.4 for Rei's obligation policies.

so far the burden of providing advanced features is charged on the user who needs them. Standard-oriented languages are hence a good choice for users who do not need advanced features but for whom compatibility with standards is a foremost issue.

Ponder, *RT* and TPL lay somehow in between. On the one hand, Ponder provides a complete authorization solution, which however takes place after a previously overcome authentication step. Therefore, Ponder cannot be applied to contexts (like pervasive environments) where users cannot be accurately identified. On the other hand, *RT* and TPL do not provide a complete authorization solution, since they can only map requesters to roles and need to rely on some external component to perform the actual authentication (although parametrized roles available in RT_1 and the other *RT* flavors basing on it make the previous statement no longer true).

Finally research-oriented languages strive toward generality and extensibility and provide a number of more advanced features in comparison with standard-oriented languages (e.g., conflict harmonization in KAoS and Rei, negotiations in Cassandra, PeerTrust and PSPL). Therefore, they should be the preferred choice for users who do not mind about standardization issues but require the advanced functionalities which research-oriented languages provide.

CHAPTER 3
The Protune Policy Language

This chapter describes the syntax and semantics of the PROTUNE policy language. An engine able to enforce PROTUNE policies has been implemented and can be freely downloaded from `http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/?pagename=Evaluation`. The PROTUNE language specifications are partially an original contribution of this work (cf. Section 3.6 for a description of the improvements w.r.t. the previous version) as is the reasoning-related part of the PROTUNE engine. Language specifications and engine are just two building blocks of the PROTUNE framework, which also comprises a policy-filtering module [19], an explanation-generation module [20], a natural language front-end [30], a wrapper for Web applications [17, 32] and an editor (available at `http://policy.l3s.uni-hannover.de:9080/policyFramework/protune/demo.html`). None of these further modules (but the natural language front-end described in Chapter 4) are original contributions of and will be described in this work.

This chapter is organized as follows: Section 3.1 introduces PROTUNE's conceptual space. PROTUNE's syntax and semantics are informally introduced in Sections 3.2 and 3.3 respectively and formalized in Sections 3.4 and 3.5 respectively. Finally, Section 3.6 outlines original contributions and improvements of the version of the PROTUNE language described in this work w.r.t. the previous version.

3.1 Protune's conceptual space

A typical access-control scenario involves two *actors*: the one which issues a request to access some resource or service and the one which accepts or rejects such request.

Note 1 *With "actors" we mean "logical actors", which do not need to be physically separated entities: access requests can be issued by one component of a concentrated system to another component of the same system. However, it is often the case*

that actors are independent entities residing on different nodes of a network which seldom come across each other for the purpose of accessing resources or services.

Note 2 *The handling of access requests might involve more than two actors. In a typical scenario (cf. Section 2.3.2), the actor to which the access request has been issued cannot or does not want to take the access decision and therefore delegates a third entity to take it. In another typical scenario, certification authorities (cf. Section 1.2) are consulted in order to check the authenticity of a certificate provided by some actor. However, except in case the opposite is explicitly stated, we will assume in the following that the handling of access requests involves exactly two actors.*

For this reason, PROTUNE assumes that two actors are available whenever an access request takes place. We will call *client* the one issuing the access request and *server* the one taking the access decision.

Note 3 *As we will see in the following, the communication between actors can evolve to a negotiation (cf. Section 2.3.2) in which, when handling an access request, the server can issue counter-requests to the client and therefore, in some sense, it becomes a client as well. For this reason, in* PROTUNE *the meaning of the terms "client" and "server" is relative to the most deeply nested counter-request. In* PROTUNE *counter-requests can nest to arbitrary depth.*

In order to issue a request, the client must know

1. which request it wants to issue
2. whom it wants to issue the request to

In order to handle a request, the server must know

3. the policy in force
4. according to which strategy the policy should be evaluated

Items 1., 2., 3., and 4. will be discussed in Sections 3.1.1, 3.1.2, 3.1.3 and 3.1.4 respectively.

3.1.1 The user interface

As described in Section 2.4, many policy languages base on some mathematically defined logical formalism in order to provide a well-defined semantics. PROTUNE is no exception, since it bases on (a subset of) Logic Programming [64] (LP in the following). In particular, an access request is modeled as a (PROTUNE) *goal*. For this reason, PROTUNE natively supports two kinds of requests

- yes/no requests (e.g., *am I allowed to access resource* `res1`*?*)
- multiple-answer requests (e.g., *which resources am I allowed to access?*)

Like usual LP-engines (e.g., SWI-Prolog[1], tuProlog[2], YAP-Prolog[3]), the PROTUNE engine implements the answer to a goal as an iterator [41]: the iterator is empty if the requesting actor is not allowed to access the resource (for yes/no requests) or if there is no resource it is allowed to access (for multiple-answer requests). Otherwise, the different elements of the iterator can be retrieved one by one.

Note 4 *The* PROTUNE *engine's user interface also provides a commodity to retrieve all accessable resources at once. Because of the possibly considerable increment of computational resources required as well as of sensitive information disclosed, users are strongly encouraged to rely on the iterator-based interface and to resort to the above-mentioned commodity only if all accessable resources have to be retrieved.*

3.1.2 The communication channel

A communication channel must be provided between the actors of an access-control scenario which, as mentioned in Note 1, often reside on different nodes of a network. The PROTUNE engine provides an abstraction layer over the communication medium, so that users willing to issue a request to an actor are only required to provide the name or address of the host on which the actor resides. The PROTUNE engine will check automatically and in a transparent way through which medium the communication with the actor can be established.

[1] `http://www.swi-prolog.org/`
[2] `http://alice.unibo.it/xwiki/bin/view/Tuprolog/`
[3] `http://www.dcc.fc.up.pt/~vsc/Yap/`

Note 5 *In case of a negotiation (cf. Note 3), the medium through which counter-requests are issued and answered does not have to be the one through which the original request is issued and answered.*

This design choice has both advantages and drawbacks. Up to now, sockets[4] are the only communication medium supported. As soon as other media (like, e.g., secure sockets[5], RMI[6] or Web Services[7]) are supported, the required changes to the PROTUNE engine will not affect the network interface. However, as it is often the case in computer science, as soon as the abstraction level is risen, a bigger user-friendliness is obtained to the detriment of flexibility. Our approach is user-friendly in the sense that nothing but name or address of the actor's host has to be provided by the user. On the other hand, only one single PROTUNE engine (and hence one single policy) per host is possible. Since we did not find any reasonable scenario yet for which this restriction could be a limitation, we do not plan to change our approach yet.

3.1.3 Protune policies

Trust is the top of the well-known Semantic Web stack [76] and PROTUNE has been proposed within the Semantic Web community in order to become a building block of the Semantic Web vision. As such, PROTUNE fosters interoperability among resources spread across different nodes of the Semantic Web. For this reason, a PROTUNE policy is a distributed program: the policy residing on the host of either of the communicating actors is supposed to be only an entry point which refers to other resources, namely, further policies and executable files (specifically, `jar` files). The policies which are referred to can in turn refer to other ones, so that in general the evaluation of a PROTUNE policy requires to evaluate a graph of resources, all of which are automatically identified, located, deployed and finally exploited by the PROTUNE engine.

[4] `http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`

[5] `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`

[6] `http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html`

[7] `http://java.sun.com/webservices/technologies/index.jsp`

A policy can refer to an external resource by exploiting a predicate (cf. following sections) which is not defined within the policy itself but in a different *package*. The concept of package has been inspired by Java packages, which indeed share some similarities with PROTUNE packages: the pair $(package, predicateName)$ univocally identifies a PROTUNE predicate, just like the pair $(package, className)$ univocally identifies a Java class. However, the two concepts do not overlap: in the following we account for the main differences.

- Java packages are simply namespaces introduced in order to avoid conflicts when defining classes with the same name. PROTUNE packages are URIs[8] pointing to actual information (stored in a *manifest* file) about the type and the location of the external resource
- The Java Virtual Machine is able to understand whether a required class is missing, but it cannot automatically retrieve it. The PROTUNE engine is able to understand whether a required predicate is missing and to identify, locate, deploy and exploit in an automatic way the external resource defining it

Note 6 *A* PROTUNE *manifest file contains exactly two entries, namely, the type and the location of the external resource. The type is either* `logical` *(for* PROTUNE *policies) or* `provisional` *(for executable files), whereas the location is a URI.*

Note 7 *Undiscriminated import of external policies can lead to security leakage. Therefore, special care was taken when designing the importing process. Importing an allow rule allow*$(\langle atom \rangle) \leftarrow \langle body \rangle$*. (cf. following sections) would add to the importing policy a further condition enabling the requesting actors which fulfill* $\langle body \rangle$ *to know whether* $\langle atom \rangle$ *holds, and it might be the case that this behavior does not match the intention of the author of the importing policy. For this reason, the* PROTUNE *engine automatically filters allow rules out of the policies to be imported.*

3.1.4 Evaluation strategies

It is often the case that there is more than one single way to fulfill a request. For instance, in order to access some resource an identity card or a student id might

[8] `http://tools.ietf.org/html/rfc3986`

have to be disclosed. Moreover, it might be the case that some actor possesses both an identity card and a student id but prefers to disclose one of them over the other.

Different alternatives can be followed at the same time. This might be meaningful in case of negotiations (cf. Note 3), where the disclosure of sensitive information does not guarantee *per se* access to the requested resource, which also depends on the successful evaluation of counter-requests. If time is an issue, following different alternatives with the hope that at least one of them leads to a successful negotiation might be more meaningful than trying an alternative, waiting until it fails and trying the next one, even if more sensitive information must be disclosed.

Identifying in an automatic way the best set of sensitive resources to be disclosed according to some optimality criterion as well as enabling the user to define herself which resources should be disclosed by means of a *preference language* (e.g., [34]) are research fields on their own (cf. [84, 85, 86, 87] for a general account) and are orthogonal to the topic of this work. The PROTUNE engine provides support to evaluation strategies by supplying the well-known *eager* and *parsimonious* strategies [84] as well as a general interface which can be extended by third parties in order to define customized evaluation strategies.

3.2 A gentle introduction to Protune's syntax

Fig. 3.1 shows an excerpt of the PROTUNE policy of the fictitious on-line bookstore "Happy Book". We will make use of such excerpt in order to introduce PROTUNE's syntax in an informal way: the formal specification of PROTUNE's syntax is provided in Section 3.4. Fig. 3.1 shows keywords and other tokens of the PROTUNE language in *italics*. The policy should be self-explanatory because of the comments which exemplify PROTUNE's conventions for single-line (cf. lines 5, 8, 12, 14 and 36) and multiple-line (cf. lines 21-27) comments.

PROTUNE is based on Datalog [2] and, as such, it is an LP-based policy language (cf. Section 2.3.1). A PROTUNE program is basically a set of normal Logic Program *rules* [64] $A \leftarrow L_1, \ldots L_n$ where $n \geq 0$, A is an *atom* (called the *head* of the rule) and $L_1, \ldots L_n$ (the *body* of the rule) are *literals*, i.e., $\forall i : 1 \leq i \leq n$ L_i equals either B_i or $\sim B_i$ (where the symbol $\sim$ denotes the negation-as-failure [64]

```
1:  import <www.happy-book.com/policies/module.mf!/retrieve/2>
2:  import <www.protune.com/modules/basics.mf!/date/1>
3:  import <www.protune.com/modules/credentials.mf!/send/2>
4:  import <www.visa.com/policies/creditCard.mf!/chargedOn/2>

5:  % The client can retrieve a resource if it is public.
6:  allow(retrieve(Id,Resource)) ←
7:      public(Id).
8:  // The client can retrieve a non-public resource if it buys it.
9:  allow(retrieve(Id,Resource)) ←
10:     not public(Id),
11:     buy(Resource).

12: % Resource res1 is public.
13: public(res1).
...

14: // The following lines define the properties type, ... date of resource res1.
15: res1["type"] = book.
16: res1["title"] = "De Agri Cultura".
17: res1["author"] = "M. Porcius Cato".
18: res1["author"] = "A. Mazzarino".
19: res1["isbn"] = 3598711328.
20: res1["date"] = 'July 1998'.
...

21: /*
22: The client buys a resource if it
23:     * sends a credit card and
24:     * (assuming that CurrentDate denotes the current date)
25:     * the credit card has not expired yet and
26:     * the price of the resource has been charged on the credit card
27: */
28: buy(Resource) ←
29:     [1] send(
30:         "SELECT id FROM credentials WHERE type = 'creditCard'",
31:         CreditCard
32:     ),
33:     date(CurrentDate),
34:     CreditCard["expirationDate"] ≥ CurrentDate,
35:     chargedOn(Resource["price"], CreditCard).

36: % The literal labeled [1] (lines 29-32) must be evaluated by the client.
37: [1] → actor : peer.
```

Figure 3.1: Example Protune policy

operator) for some atom B_i. Lines 6-7, 9-11 and 28-35 show rules whose body is not empty, whereas lines 13 and 15-20 show *facts*, i.e., rules whose body is empty.

The vocabulary of the atoms occurring in PROTUNE rules is partitioned into the following categories.

Allow Have the form `allow(`*atom*`)`, where *atom* is a non-allow atom (cf. lines 6, 9). Allow atoms are the entry points of PROTUNE policies, in the sense that a goal atom can be successfully evaluated only if it matches the non-allow atom in the head of some allow rule (cf. Note 8)

Classical Have the form p`(`t_1`, ...` t_a`)`, where $a \geq 0$ (cf. lines 7, 10-11, 13, 28-33, 35). p and a are called the *name* and the *arity* respectively of the *predicate* exploited in the atom, whereas $\forall i : 1 \leq i \leq a$ t_i is a *term* (cf. below). Classical atoms are further partitioned into *logical* and *provisional atoms*, according to whether they exploit *logical* or *provisional predicates* (cf. Note 6 and below)

Comparison Have the form $t_1 \circ t_2$, where t_1 and t_2 are terms and $\circ$ is a comparison operator (cf. line 34). PROTUNE supports the following comparison operators: $=, \neq, >, \geq, <, \leq$

Value-assignment Have the form ct `=` t, where ct is a *complex term* (cf. below) and t is a term (cf. lines 15-20). Value-assignment atoms owe their name to the fact that, when appearing in the head of a rule whose body is fulfilled, they assign a value to a given attribute of an *object* (cf. below)

Note 8 *All of allow, ... value-assignment atoms (but provisional atoms) can appear both in rule heads and bodies. Rules in whose head an allow, ... value-assignment atom appears are called allow, ... value-assignment rules respectively.*

By definition, provisional atoms cannot appear in rule heads since they are not defined in a policy but in an executable file (cf. Note 6). Value-assignment atoms in rule bodies check whether an object possesses a given $(attribute, value)$ *pair. Examples 2, 3 and 4 show meaningful applications of comparison rules. Finally, allow atoms can be exploited in rule bodies whenever the conditions to be fulfilled in*

order to evaluate an atom $atom_2$ are a superset of the conditions to be fulfilled in order to evaluate an atom $atom_1$, like in the following example.

$$\textbf{\textit{allow}}(atom_1) \leftarrow$$
$$condition_1, \ldots condition_m.$$
$$\textbf{\textit{allow}}(atom_2) \leftarrow$$
$$condition_1, \ldots condition_m, furtherCondition_1, \ldots furtherCondition_n.$$

The second rule can be replaced by the more compact one

$$\textbf{\textit{allow}}(atom_2) \leftarrow$$
$$\textbf{\textit{allow}}(atom_1), furtherCondition_1, \ldots furtherCondition_n.$$

Logical predicates are defined in (i.e., appear in the head of some rule of) a (possibly imported) policy (cf. the predicates *public*/1 and *buy*/1 at lines 7, 10-11, 13, 28). The evaluation of provisional predicates (like *retrieve*/2, *send*/2, *date*/1 and *chargedOn*/2 at lines 6, 9, 29-33, 35) requires to perform an action. Provisional packages (cf. Note 6) define the mapping between provisional predicates and actions, evaluate the formers and execute the latters. For this reason, the provisional predicates appearing in Fig. 3.1 are not part of the PROTUNE language and the following explanation of their interface does not add anything to its description but is only meant to provide for a better understanding of the policy presented in Fig. 3.1.

retrieve/2 Predicate defined by Happy Book (cf. line 1) which allows to retrieve the resource corresponding to a given identifier from a local database. The identifier is modeled as an *object identifier* (cf. below), whereas the resource is modeled as an object having at least the attribute `price` whose value is modeled as a *number* (cf. below). The evaluation of (an atom exploiting) the predicate can be successful only if its first (resp. second) parameter is (resp. is not) instantiated

send/2 Predicate defined in the *credentials* package of the PROTUNE framework (cf. line 3) which allows to retrieve the credentials fulfilling given constraints from a local credential database. The specification of the constraints is modeled as

a(n SQL) *string* (cf. below), whereas the credentials are modeled as objects. The evaluation of the predicate can be successful only if its first (resp. second) parameter is (resp. is not) instantiated

date/1 Predicate defined in the *basics* package of the PROTUNE framework (cf. line 2) which allows to retrieve the current date. The current date is modeled as a *date* (cf. below). The evaluation of the predicate can be successful only if its parameter is not instantiated

chargedOn/2 Predicate defined by VISA (cf. line 4) which allows to charge a given amount on a given credit card. The amount is modeled as a number, whereas the credit card is modeled as an object. The evaluation of the predicate can be successful only if both parameters are instantiated

PROTUNE is a typed language and provides four scalar datatypes (namely, *booleans*, *numbers*, *strings* and *dates*) and one vectorial datatype (namely, *objects*). Booleans are represented by means of the keywords `false` and `true`, whereas the right-hand side of the equality at line 19 (resp. 18, 20) shows that numbers (resp. strings and dates) are represented as lists of digits (resp. double-quoted character strings and single-quoted character strings whose content can be interpreted as a date—more on this in Section 3.4). Objects are sets of $(attribute, value)$ pairs linked to an identifier and people familiar with Java, JavaScript or PHP can think of them as maps, objects or associative arrays respectively. However, whilst in such languages each attribute has one single value, PROTUNE attributes can be multi-valued (cf. lines 17-18). Notice that objects whose attributes are integers can simulate arrays and lists.

PROTUNE terms can be variables, scalars, object identifiers (cf. line 13 and the right-hand side of the equality at line 15) or *complex terms* (cf. the left-hand sides of the equalities at lines 15-20, the left-hand side of the inequality at line 34 and the first parameter at line 35). Complex terms have the form t_1`[`t_2`]` where t_1 is a variable, an object identifier or a complex term in turn, whereas t_2 is a term.

The heads of rules contain atoms, whereas the heads of *metarules* contain *metaatoms*. In this work we only focus on *actor metaatoms/rules*. *Sensitivity* and

explanation metaatoms/rules, exploited by the policy-filtering [19] and explanation-generation module [20] respectively (cf. the beginning of this chapter), are beyond the scope of this work.

The head of the actor metarule at line 37 contains the actor metaatom `[1]→ actor:peer`. Actor metarules owe their name to the actor metaatoms their heads contain. Actor metaatoms owe their name to the token `actor` they contain. Actor metaatoms are the mechanism used by PROTUNE to support counter-requests (cf. Note 3): literals whose labels are referred to by actor metaatoms are supposed to be evaluated by the client. Finally, the bodies of metarules do not necessarily have to be empty: metarule bodies do not substantially differ from rule bodies, beside the fact that they can contain (unlabeled) metaatoms, too.

We conclude this section by describing the format of the importing statements at lines 1-4. Whenever a provisional predicate or a logical predicate not defined in the current policy is exploited, a corresponding importing statement must be added to the current policy. Importing statements have the form `import <`*uri*`!/`*name*`/`*arity*`>`, where *name* (resp. *arity*) is the name (resp. arity) of the imported predicate, whereas *uri* is the URI of the manifest of the package defining the predicate (cf. Section 3.1.3).

3.3 A gentle introduction to Protune's semantics

Usual LP-engines (like the ones mentioned in Section 3.1.1) evaluate in a similar way goals and the bodies of rules which have to be evaluated, namely, according to the SLDNF-resolution mechanism [64]. The evaluation of a goal/body $L_1, \ldots\ L_n$ (where $n > 0$) is successful iff the evaluation of each of the literals L_i (where $1 \leq i \leq n$) is successful.

The PROTUNE engine evaluates PROTUNE goals and bodies in a similar way. However, unlike usual LP-engines, it evaluates differently goal and body literals because of the following reasons.

- A goal literal is supposed to be evaluated by the server, whereas a body literal might have to be evaluated by the client (cf. Note 3)

- Before evaluating a goal literal, the server must check whether the policy in force allows to evaluate it. Such a check is not needed for body literals

The evaluation of a goal literal of the form *atom* is successful iff both of the following conditions are fulfilled.

- *atom* can be evaluated according to the server's policy, i.e., the server's policy contains some rule whose head matches `allow(`*atom*`)` and the evaluation of whose body is successful
- The evaluation of *atom* is successful

The evaluation of a goal literal of the form $\sim$ *atom* is successful iff the evaluation of the goal literal *atom* is not (*negation-as-failure* [64] inference rule).

Note 9 *The evaluation of a goal fails if either of the atoms it consists of cannot be evaluated or if the evaluation of either of the literals it consists of is unsuccessful. In both cases, the client will receive the same failure message, so that it has no means to understand which case applies. Providing the client with information about which kind of failure occurred would lead to unnecessarily disclosing (possibly sensitive) information about the server's policy. Being our goal to minimize unnecessary information disclosure, we designed the* PROTUNE *engine's interface so that it does not distinguish between the two kinds of failure.*

The evaluation of a body literal L is successful iff either of the following conditions is fulfilled.

- L has to be evaluated by the server and
  - either L is of the form *atom* and the server's policy contains some rule whose head matches *atom* and the evaluation of whose body is successful
  - or L is of the form $\sim$ *atom* and the evaluation of the body literal *atom* is unsuccessful (*negation-as-failure* [64] inference rule)
- L has to be evaluated by the client and the client evaluates it successfully

A literal L has to be evaluated by the client iff the server's policy contains an actor metarule (cf. Section 3.2) whose head matches [*id*]→`actor:peer` (where [*id*] is the label identifying L) and the evaluation of whose body is successful.

The evaluation of an atom varies according to the type of the atom. Sections 3.3.1, 3.3.2 and 3.3.3 thoroughly describe how provisional, comparison and value-assignment atoms respectively are evaluated. Logical atoms are not considered since their evaluation takes place according to the well-known SLDNF-resolution mechanism [64].

3.3.1 Provisional atoms

As mentioned in Section 3.1.3, the evaluation of (atoms exploiting) provisional predicates requires to perform an action. The execution of such an action is delegated to an external resource (*executor*), which has to provide a given interface. The input parameter of an executor is the atom to be evaluated, whereas the return value is similar to the one of the whole PROTUNE engine (cf. Section 3.1.1), namely an iterator whose different elements can be retrieved one by one.

Note 10 *The return value of executors has been designed to be an iterator instead of an atomic value because such interface models some scenarios in the most natural way: against an SQL query, RDBMSs return sets of tuples, each of which can be modeled most naturally as an element of an iterator.*

Note 11 *The interface executors must provide is a syntactic constraint, but not a semantic one: in particular, the* PROTUNE *specifications do not describe when a (non-)empty iterator must be returned. It is reasonable that an empty iterator is returned whenever no result is available. However, some executor developer might want to return an empty iterator in order to signal that some error occurred (e.g., some variable was not instantiated). We do not encourage this practice whenever it leads to ambiguities (i.e., whenever an empty iterator could be interpreted as signaling both that no result is available and that some error occurred). However, it can be considered a reasonable design choice for actions which, if successfully executed, always return at least one result (e.g., retrieving the current date—cf.*

Fig. 3.1, line 33), if the executor developer has strong motivations for not providing the executor user with any information about the occurred error.

Provisional atoms are recognized as such at policy-load time (cf. Section 3.1.3) and track of the executors in charge of evaluating them is kept. Whenever a provisional atom must be evaluated, the corresponding executor is invoked.

We conclude this section by describing how objects are handled throughout the control flow from the PROTUNE engine to an executor and back. As described in Section 3.2, objects are not atomic entities but sets of $(attribute, value)$ pairs linked to an identifier. In order to be able to operate on an object, an executor must be provided with all its $(attribute, value)$ pairs. For this reason, whenever a provisional atom must be evaluated, if some term of its is an object, the PROTUNE engine collects all such pairs before handing them over to the executor.

Note 12 *Notice that such pairs might be spread throughout the policy as different facts or might even have to be inferred (as it happens if they appear in the head of some value-assignment rule). Moreover, since the attributes and values pairs consist of can be objects in turn, their* $(attribute, value)$ *pairs must be collected as well.*

Conversely, if it happens that some result returned by the executor contains an object, the PROTUNE engine keeps track of its $(attribute, value)$ pairs in the current scope.

Note 13 *Beside containing* PROTUNE *objects, represented as sets of* $(attribute, value)$ *pairs linked to an identifier, the current scope can contain their representation as serializable*[9] *Java objects linked to the same identifier. The Java representation is not used by the* PROTUNE *engine during the reasoning process but executors might find it handier than the pair-based one. For this reason, whenever a provisional atom must be evaluated, if some term of its is an object, the* PROTUNE *engine hands its Java representation over to the executor and, if it happens that some result returned by the executor contains an object, it adds its Java representation to the current scope as well. Finally, the Java representation is required to be a serializable object since*

[9] `http://java.sun.com/developer/technicalArticles/Programming/serialization/`

it might have to be transmitted across the network and the PROTUNE *engine makes extensive use of the Java serialization mechanism for its network-related tasks.*

Example 1 shows that the support to objects and actions reduces the declarativeness of the PROTUNE language.

Example 1 *Assuming that*

- *the evaluation of the provisional atom* $a(X)$ *has as a result the binding of* X *to an object having the* $(attribute, value)$ *pair* (*"a"*, *"v"*)
- *no value-assignment atoms of the form* $Y[$ *"a"*$] =$ *"v" can be inferred from the policy in force or the current scope before the evaluation of* $a(X)$

the evaluation of the atom $Y[$ *"a"*$] =$ *"v" will be successful only if performed after the evaluation of* $a(X)$*.*

As shown in Example 1, in order to be able to write policies which are interpreted as intended, a policy author must be aware of both: (i) the side-effects of the evaluation of a provisional atom; and (ii) the order according to which the PROTUNE engine evaluates the literals of a goal/body. Although this behavior of the PROTUNE engine might reduce the user-friendliness of the PROTUNE language, it is the price to pay in order to integrate objects and actions into a policy language and, all in all, it does not differ from the behavior of a usual Prolog engine: some knowledge of the way a Prolog program is interpreted is required in order to write effective and efficient Prolog programs as well.

3.3.2 Comparison atoms

The evaluation of a comparison atom is unsuccessful if either of the terms to be compared is not instantiated or if they are not of the same type. In particular, implicit casting is not supported (e.g., when comparing the string `"1"` and the number `0`, the former does not get automatically converted into the number `1`, nor is the number `0` automatically converted into the string `"0"` when comparing it with the string `"a"`). If both terms are instantiated and of the same type, the outcome of the evaluation depends on the type itself as well as on the comparison operator.

	$=$	$\neq$	$>$	$\geq$	$<$	$\leq$
boolean	$\times$	$\times$				
number	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
string	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
date	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
object	$\times$	$\times$				

Table 3.1: Comparisons supported by default

Tab. 3.1 summarizes the comparisons supported by default: the evaluation of comparison atoms involving types and operators corresponding to cells not marked with a cross is always unsuccessful (e.g., the evaluation of the comparison atom `true > false` is unsuccessful). The outcome of the evaluation of comparison atoms involving types and operators corresponding to cells marked with a cross depends on the actual terms to be compared: the PROTUNE ordering of numbers is the intuitive one, strings are ordered alphabetically and dates temporally. The boolean `false` is equal to itself and different than the boolean `true` (the same holds if `false` and `true` are swapped). Finally, two objects are equal if and only if they are the same one (i.e., they have the same identifier).

The default semantics of the operators has been defined in order to fit the needs of most users. However, users can also modify (but not override) it by defining comparison rules (cf. Note 8). Such rules extend the semantics of an operator by stating further conditions which, if fulfilled, allow a comparison atom to be successfully evaluated. Examples 2, 3 and 4 show some cases in which comparison rules can be useful.

Example 2 *In some scenarios it might be convenient setting an ordering between booleans so that, e.g.,* ***`false`*** *precedes* ***`true`****. This behavior can be obtained by defining the following comparison facts.*

```
false < true.
true > false.
false <= false.
false <= true.
true <= true.
```

```
false >= false.
true >= false.
true >= true.
```

Example 3 *As mentioned above, by default two objects are considered to be equal if and only if they have the same identifier. However, in some scenarios it might be convenient considering equal two objects o_1 and o_2 if they have the same set of $(attribute, value)$ pairs (or, more formally, if $subset(o_1, o_2) \wedge subset(o_2, o_1)$ holds, where $subset(o_1, o_2)$ holds iff $\forall(a_1, v_1) \in o_1 \exists(a_2, v_2) \in o_2$ such that a_1 equals a_2 and v_1 equals v_2). This behavior can be obtained by defining one or more comparison rules.*

Example 4 *As mentioned above, by default the evaluation of a comparison atom is unsuccessful if either of the terms to be compared is not instantiated. However, in some scenarios it might be convenient having an equality operator which behaves similarly as the `=/2` Prolog predicate*[10]*, i.e., which unifies the terms to be compared whenever possible so that, e.g., the evaluation of the comparison atom `X = 2` succeeds (and, as a side-effect, unifies `X` and `2`) instead of failing. This behavior can be obtained by defining one or more comparison rules.*

Comparison rules conform to PROTUNE's namespace mechanism (cf. Section 3.1.3) as well, i.e., the extension of the semantics of an operator by means of a rule within a given policy does not affect the semantics of the same operator within a different policy. On the other hand, such extension can be imported into a different policy by exploiting the namespace of the imported policy together with the operator within the importing one, like in the following example (where `www.protune.com/modules/basics.mf` is the namespace of the imported policy).

```
X <www.protune.com/modules/basics.mf> = Y
```

Note 14 *Whenever the user extends the semantics of an operator, the burden of ensuring the consistency of the extension lays on the user herself. For instance,*

[10] `http://www.iso.org/iso/catalogue_detail.htm?csnumber=21413`

although users are allowed to define the fact `true = false.`*, they must be aware of the consequences of such a definition (in particular, the boolean* `true` *would be both equal to and different than the boolean* `false` *– because of the user-defined fact and the default semantics respectively).*

Similarly, users are allowed to define both facts `true >= false.` *and* `true <= false.`*, although this would lead to losing the antisymmetric property equality usually possesses.*

Note 15 *W.r.t. the first example presented in Note 14, the definition of the fact* `true = false.` *implies that the evaluation of the atom* `true = false` *is successful, but it does not imply per se that the evaluation of the atom* `false = true` *is successful as well. In order to obtain such a result, a further fact* `false = true.` *must be defined. For a similar reason, none of the facts listed in Example 2 may be omitted, if an ordering with the usual properties is desired.*

Finally, during the evaluation of a comparison atom user-defined rules are evaluated (according to the well-known SLDNF-resolution mechanism [64]) before resorting to the default semantics.

3.3.3 Value-assignment atoms

As described in Section 3.3.1, the evaluation of a provisional atom might end up with the addition of one or more objects (and hence one or more sets of $(attribute, value)$ pairs) to the current scope. On the other hand, as described in Section 3.2, users can create objects within a policy by defining value-assignment rules. For this reason, during the evaluation of a value-assignment atom, both user-defined rules and objects available in the current scope must be inspected. The evaluation of a value-assignment atom is successful iff it is unifiable with

- the head of some (value-assignment) rule, the evaluation of whose body is successful or
- some $(attribute, value)$ pair of some object available in the current scope

Two value-assignment atoms $t_1^1[t_2^1] = t_3^1$ and $t_1^2[t_2^2] = t_3^2$ (where $\forall 1 \leq i \leq 2, 1 \leq j \leq 3$ t_j^i is not a complex term—cf. Section 3.2) are unifiable iff the triples (t_1^1, t_2^1, t_3^1) and (t_1^2, t_2^2, t_3^2) are unifiable. Cf. Section 3.5 for the case in which some t_j^i is a complex term.

The value-assignment atom $t_1[t_2] = t_3$ (where $\forall 1 \leq i \leq 3$ t_i is not a complex term) is unifiable with the pair (a, v) of the object whose identifier is id iff the triples (t_1, t_2, t_3) and (id, a, v) are unifiable. Cf. Section 3.5 for the case in which some t_i is a complex term.

Finally, the evaluation of user-defined rules (according to the well-known SLDNF-resolution mechanism [64]) takes place before objects available in the current scope are inspected.

3.4 Protune's syntax

This section formally describes the syntax of the PROTUNE policy language. For the sake of completeness, we will also formalize sensitivity and explanation metaatoms, although we did not describe them thoroughly in the previous sections (cf. the beginning of this chapter).

3.4.1 Terms

Definition 1 (Term) *A term is either a variable, a boolean, a number, a string, a date, an object identifier or a complex term (cf. Definition 2).*

We will represent variables as Prolog variables, booleans (resp. strings) as Java booleans (resp. strings), numbers as Java integers, longs, floats or doubles[11], dates as single-quoted strings whose content is considered a valid date by the `parse(String)` method of the `java.text.DateFormat` Java class[12] and object identifiers as Prolog non-quoted identifiers. Cf. below for the representation of complex terms.

Example 5 *All of the following are well-formed terms: the variables* `X` *and* `_`*, the boolean* `false`*, the numbers* `0`*,* `0L`*,* `1e1f` *and* `1e-9d`*, the string* `""`*, the date* `'24.06.09 10:50'` *and the object identifier* `a`*.*

[11] `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`
[12] `http://java.sun.com/javase/7/docs/api/java/text/DateFormat.html`

Definition 2 (Complex term) *A complex term is a pair* $(id, term)$*, where* id *is either a variable, an object identifier or a complex term and* $term$ *is a term (cf. Definition 1).*

We will represent complex terms as id`[`$term$`]`, where id is the string representation of the variable (resp. object identifier, complex term) and $term$ is the string representation of the term.

Example 6 *All of the following are well-formed complex terms:* `a[true]`*,* `X[a]`*,* `a[a[a]]` *and* `a[X][0]`*.*

Definition 3 (Comparison operator) *A comparison operator is either of* `=`*,* `!=`*,* `>`*,* `>=`*,* `<` *and* `<=`*.*

3.4.2 Atoms

Definition 4 (Classical atom) *A classical atom is a pair* $(name, terms)$*, where* $name$ *is a predicate name and* $terms$ *is a (possibly empty) list of terms (cf. Definition 1).*

We will represent predicate names as Prolog non-quoted identifiers and classical atoms as

- $name$, where $name$ is the string representation of the predicate name, if $terms$ is empty
- $name$`(`$term_1$`, ... `$term_n$`)`, where $n = \#terms$ and $term_i$ is the string representation of the i-th term in $terms$ ($1 \leq i \leq n$), if $terms$ is not empty

Example 7 *All of the following are well-formed classical atoms:* `a` *and* `a(X, false, 0, "", '24.06.09 10:50', a, a[a])`*.*

Definition 5 (Comparison atom) *A comparison atom is a triple* $(operator, left, right)$*, where* $operator$ *is a comparison operator (cf. Definition 3), and* $left$ *and* $right$ *are terms (cf. Definition 1).*

We will represent comparison atoms as $left\ \ operator\ \ right$, where $left$ (resp. *operator*, *right*) is the string representation of the left term (resp. comparison operator, right term).

Example 8 *All of the following are well-formed comparison atoms:* `X = false`*,* `0 < ""` *and* `'24.06.09 10:50' >= a`*.*

Definition 6 (Queriable atom) *A queriable atom is either a classical atom (cf. Definition 4) or a comparison atom (cf. Definition 5).*

Definition 7 (Decision atom) *A decision atom is a singleton* $\{atom\}$*, where atom is a queriable atom (cf. Definition 6).*

We will represent decision atoms as `allow(`*atom*`)`, where *atom* is the string representation of the queriable atom.

Example 9 *All of the following are well-formed decision atoms:* `allow(a)` *and* `allow(X = false)`*.*

Definition 8 (Atom) *An atom is either a queriable atom (cf. Definition 6) or a decision atom (cf. Definition 7).*

3.4.3 Goals and rules

Definition 9 (Queriable literal) *A queriable literal is a pair* $(negated, atom)$*, where negated is either* `false` *or* `true` *and atom is a queriable atom (cf. Definition 6).*

We will represent queriable literals as

- *atom*, where *atom* is the string representation of the queriable atom, if *negated* is `false`
- `not` *atom*, if *negated* is `true`

Example 10 *All of the following are well-formed queriable literals:* `a`*,* `not X = false`*.*

Definition 10 (Goal) *A goal is a non-empty list of queriable literals (cf. Definition 9).*

We will represent goals as $literal_1$`,` ... $literal_n$`.`, where n is the number of queriable literals the goal consists of and $literal_i$ is the string representation of the i-th queriable literal ($1 \leq i \leq n$).

Example 11 *All of the following are well-formed goals:* `a.` *and* `a, X = false.`.

Definition 11 (Literal) *A literal is either a pair* $(negated, atom)$ *or a triple* $(label, negated, qAtom)$*, where negated is either* `false` *or* `true`*, atom is an atom (cf. Definition 8), qAtom is a queriable atom (cf. Definition 6) and label is a label.*

We will represent labels as Prolog non-quoted identifiers and literals as

- *atom*, where *atom* is the string representation of the atom, if the literal is of the form $(negated, atom)$ and *negated* is `false`
- `not` *atom*, if the literal is of the form $(negated, atom)$ and *negated* is `true`
- `[`*label*`]` *qAtom*, where *label* is the string representation of the label and *qAtom* is the string representation of the queriable atom, if the literal is of the form $(label, negated, atom)$ and *negated* is `false`
- `[`*label*`]` `not` *qAtom*, if the literal is of the form $(label, negated, atom)$ and *negated* is `true`

Example 12 *All of the following are well-formed literals:* `a`*,* `not X = false` *and* `[a] allow(a)`*.*

Definition 12 (Rule) *A rule is either a pair* $(head, body)$ *or a triple* $(label, head, body)$*, where head is an atom (cf. Definition 8), body is a (possibly empty) list of literals (cf. Definition 11) and label is a label.*

We will represent labels as Prolog non-quoted identifiers and rules as

- *head*`.`, where *head* is the string representation of the atom, if the rule is of the form (*head*, *body*) and *body* is empty
- *head* `:-` $literal_1$`,` ... $literal_n$`.`, where $n = \#body$ and $literal_i$ is the string representation of the i-th literal in *body* ($1 \leq i \leq n$), if the rule is of the form (*head*, *body*) and *body* is not empty
- `[`*label*`]:` *head*`.`, where *label* is the string representation of the label, if the rule is of the form (*label*, *head*, *body*) and *body* is empty
- `[`*label*`]:` *head* `:-` $literal_1$`,` ... $literal_n$`.`, if the rule is of the form (*label*, *head*, *body*) and *body* is not empty

Example 13 *All of the following are well-formed rules:* `a.`, `X = false :- a.` *and* `[a]: allow(a) :- X = false.`.

3.4.4 Metaatoms

Definition 13 (Actor metaatom) *An actor metaatom is a singleton* {*label*}*, where label is a literal label (cf. Definition 11).*

We will represent actor metaatoms as *label*`->actor:peer`, where *label* is the string representation of the literal label.

Definition 14 (Sensitivity metaatom) *A sensitivity metaatom is a singleton* {*label*}*, where label is a literal label (cf. Definition 11) or a rule label (cf. Definition 12).*

We will represent sensitivity metaatoms as *label*`->sensitivity:private`, where *label* is the string representation of the literal/rule label.

Definition 15 (Explanation metaatom) *An explanation metaatom is a pair* (*label*, *term*)*, where label is a literal label (cf. Definition 11) or a rule label (cf. Definition 12) and term is a term (cf. Definition 1).*

We will represent explanation metaatoms as *label*`->explanation:`*term*, where *label* (resp. *term*) is the string representation of the literal/rule label (resp. term).

Definition 16 (Metaatom) *A metaatom is either an actor metaatom (cf. Definition 13) or a sensitivity metaatom (cf. Definition 14) or an explanation metaatom (cf. Definition 15).*

Example 14 *All of the following are well-formed metaatoms: the actor metaatom* `[a]->actor:peer`*, the sensitivity metaatom* `[a]->sensitivity:private` *and the explanation metaatom* `[a]->explanation:""`*.*

3.4.5 Metarules

Definition 17 (Metaliteral) *A metaliteral is a pair* $(negated, metaatom)$*, where negated is either* `false` *or* `true` *and metaatom is a metaatom (cf. Definition 16).*

We will represent metaliterals as

- *metaatom*, where *metaatom* is the string representation of the metaatom, if *negated* is `false`
- `not` *metaatom*, if *negated* is `true`

Example 15 *All of the following are well-formed metaliterals:* `[a]->actor:peer` *and* `not [a]->sensitivity:private`*.*

Definition 18 (Generic literal) *A generic literal is either a literal (cf. Definition 11) or a metaliteral (cf. Definition 17).*

Definition 19 (Metarule) *A metarule is a pair* $(head, body)$*, where head is a metaatom (cf. Definition 16) and body is a (possibly empty) list of generic literals (cf. Definition 18).*

We will represent metarules as

- *head*`.`, where *head* is the string representation of the metaatom, if *body* is empty
- *head* `:-` $literal_1$`,` ... $literal_n$`.`, where $n = \#body$ and $literal_i$ is the string representation of the i-th generic literal in *body* ($1 \leq i \leq n$), if *body* is not empty

Example 16 *All of the following are well-formed metarules:* `[a]->actor:peer.`, `[a]->sensitivity:private :- X oinden false.` *and* `[a]->explanation:"" :- [a]->actor:peer.`.

3.4.6 Policy

Definition 20 (Generic rule) *A generic rule is either a rule (cf. Definition 12) or a metarule (cf. Definition 19).*

Definition 21 (Policy) *A policy is a (possibly empty) list of generic rules (cf. Definition 20).*

3.5 (Toward) Protune's semantics

At least two (non mutually exclusive) ways are feasible to describe the semantics of the PROTUNE policy language: (i) describing how a generic PROTUNE policy can be mapped to a Logic Program [64]; and (ii) providing an executable semantics (cf. Chapter 7). Whilst we regard both approaches as future work, each of them would profit from a semantic characterization of complex terms (cf. Section 3.2).

This section provides such a characterization by describing how a generic PROTUNE policy can be brought to a canonical form (beside variable renaming) in which complex terms only appear within value-assignment atoms (cf. Section 3.2) of the form t_1`[`t_2`] = `t_3, where none of t_1, t_2 and t_3 is a complex term. In particular, this section describes the effects of function $\mathcal{F}$ on the syntactic constructs a PROTUNE policy consists of.

3.5.1 Basic definitions

In the following we will write $(a_1, \ldots\ a_n)$ to denote a list consisting of the elements a_1, $\ldots\ a_n$ ($n \geq 0$) and $\emptyset$ to denote the empty list. Moreover we will write $A \cup B$ to denote the concatenation of lists A and B.

Example 17 $\emptyset \cup (a) \cup (b, c) = (a, b, c)$.

Definition 22 (Function f) *If x is a term then*

- $f : x \mapsto (x, \emptyset)$ *if* x *is not a complex term*
- $f : (id, term) \mapsto ((id, term), \emptyset)$ *if* $(id, term)$ *is a complex term,* $f : id \mapsto (id, \emptyset)$ *and* $f : term \mapsto (term, \emptyset)$
- $f : (id, term) \mapsto ((id, variable), assignments \cup ((id_1, term_1, variable)))$ *if* $f : id \mapsto (id, \emptyset)$ *and* $f : term \mapsto ((id_1, term_1), assignments)$
- $f : (id, term) \mapsto ((variable, term), assignments \cup ((id_1, term_1, variable)))$ *if* $f : id \mapsto ((id_1, term_1), assignments)$ *and* $f : term \mapsto (term, \emptyset)$
- $f : (id, term) \mapsto ((variable_1, variable_2), assignments_1 \cup assignments_2 \cup ((id_1, term_1, variable_1), (id_2, term_2, variable_2)))$ *if* $f : id \mapsto ((id_1, term_1), assignments_1)$ *and* $f : term \mapsto ((id_2, term_2), assignments_2)$

where variable, variable$_1$ *and variable*$_2$ *are fresh variables.*

Example 18 f : `a` $\mapsto$ (`a`, $\emptyset$),f : `a[b[c]]` $\mapsto$ (`a[X]`, (`b[c] = X`)),f : `a[b][c]` $\mapsto$ (`X[c]`, (`a[b] = X`)),f : `a[b][c[d]]` $\mapsto$ (`X[Y]`, (`a[b] = X`, `c[d] = Y`)).

3.5.2 Terms

Definition 23 (Term) *If* x *is a term different than a complex term then* $\mathcal{F} : x \mapsto (x, \emptyset)$. *Cf. Definition 24 for the output of* $\mathcal{F}$ *in case* x *is a complex term.*

Example 19 $\mathcal{F}$: `a` $\mapsto$ (`a`, $\emptyset$).

Definition 24 (Complex term) *If* $(id, term)$ *is a complex term then*

$$\mathcal{F} : (id, term) \mapsto (variable, assignments \cup ((id_1, term_1, variable)))$$

where $f : (id, term) \mapsto ((id_1, term_1), assignments)$ *and variable is a fresh variable.*

Example 20 $\mathcal{F}$: `a[b[c]]` $\mapsto$ (`Y`, (`b[c] = X`, `a[X] = Y`)),$\mathcal{F}$: `a[b][c]` $\mapsto$ (`Y`, (`a[b] = X`, `X[c] = Y`)),$\mathcal{F}$: `a[b][c[d]]` $\mapsto$ (`Z`, (`a[b] = X`, `c[d] = Y`, `X[Y] = Z`)).

Theorem 1 *For no term term it happens that* $\mathcal{F} : term \mapsto (term_1, assignments)$, *where* $term_1$ *is a complex term.*

3.5.3 Atoms

Definition 25 (Classical atom) *If* $terms = (term_1, \ldots\ term_n)$ *(*$n \geq 0$*) and* $(name, terms)$ *is a classical atom then* $\mathcal{F} : (name, terms) \mapsto ((name, terms^1), assignments)$*, where* $terms^1 = (term_1^1, \ldots\ term_n^1)$*,*$assignments = \bigcup_{1 \leq i \leq n} assignments_i$ *and* $\forall 1 \leq i \leq n\ \mathcal{F} : term_i \mapsto (term_i^1, assignments_i)$*.*

Example 21 $\mathcal{F}$: `a` $\mapsto$ (`a`, $\emptyset$)*,*$\mathcal{F}$: `a(b[c])` $\mapsto$ (`a(X)`, (`b[c] = X`))*.*

Definition 26 (Comparison atom) *If* $(operator, left, right)$ *is a comparison atom then*

- *if operator is* $=$ *and* $left$ *is a complex term,* $\mathcal{F} : (operator, left, right) \mapsto ((operator, left_1, right_1), assignments_l \cup assignments_r)$*, where* $f : left \mapsto (left_1, assignments_l)$ *and* $\mathcal{F} : right \mapsto (right_1, assignments_r)$
- *otherwise,* $\mathcal{F} : (operator, left, right) \mapsto ((operator, left_1, right_1), assignments_l \cup assignments_r)$*, where* $\mathcal{F} : left \mapsto (left_1, assignments_l)$ *and* $\mathcal{F} : right \mapsto (right_1, assignments_r)$

Example 22 $\mathcal{F}$: `a = b` $\mapsto$ (`a = b`, $\emptyset$)*,*$\mathcal{F}$: `a > b` $\mapsto$ (`a > b`, $\emptyset$)*,*$\mathcal{F}$: `a[b] = c[d]` $\mapsto$ (`a[b] = X`, (`c[d] = X`))*,*$\mathcal{F}$: `a[b] > c[d]` $\mapsto$ (`X > Y`, (`a[b] = X`, `c[d] = Y`))*.*

Theorem 2 *For no assignment atom atom it happens that* $\mathcal{F} : atom \mapsto ((operator, left, right), assignments)$*, where right is a complex term. Moreover* $left$ *can be a complex term only if operator is* $=$*.*

Definition 27 (Decision atom) *If* $\{atom\}$ *is a decision atom then* $\mathcal{F} : \{atom\} \mapsto (\{atom_1\}, assignments)$*, where* $\mathcal{F} : atom \mapsto (atom_1, assignments)$*.*

Example 23 $\mathcal{F}$: `allow(a(b[c]))` $\mapsto$ (`allow(a(X))`, (`b[c] = X`)) *and* $\mathcal{F}$: `allow(a[b] = c[d])` $\mapsto$ (`allow(a[b] = X)`, (`c[d] = X`))*.*

3.5.4 Goals and rules

Definition 28 (Queriable literal) *If* $(negated, atom)$ *is a queriable literal then* $\mathcal{F} : (negated, atom) \mapsto \bigcup_{1 \leq i \leq n}((\texttt{true}, assignment_i)) \cup ((negated, atom_1))$*, where* $\mathcal{F} : atom \mapsto (atom_1, assignments)$ *and* $assignments = (assignment_1, \ldots\ assignment_n)$ *(*$n \geq 0$*).*

Example 24 $\mathcal{F}$: `a(b[c])` $\mapsto$ (`b[c] = X`, `a(X)`) *and* $\mathcal{F}$: `not a[b] = c[d]` $\mapsto$ (`c[d] = X`, `not a[b] = X`).

Definition 29 (Goal) *If* $goal = (literal_1, \ldots\ literal_n)$ *(*$n \geq 1$*) is a goal then* $\mathcal{F} : goal \mapsto \bigcup_{1 \leq i \leq n} literals_i$*, where* $\forall 1 \leq i \leq n\ \mathcal{F} : literal_i \mapsto literals_i$

Example 25 $\mathcal{F}$: `a(b[c])` $=$ `b[c] = X, a(X)` *and* $\mathcal{F}$: `a, not a[b] = c[d]` $=$ `a, c[d] = X, not a[b] = X`.

Definition 30 (Literal) *If* $(negated, atom)$ *is a literal then* $\mathcal{F} : (negated, atom) \mapsto \bigcup_{1 \leq i \leq n}((\texttt{true}, assignment_i)) \cup ((negated, atom_1))$*, where* $\mathcal{F} : atom \mapsto (atom_1, assignments)$ *and* $assignments = (assignment_1, \ldots\ assignment_n)$ *(*$n \geq 0$*).*

If $(label, negated, qAtom)$ *is a literal then* $\mathcal{F} : (label, negated, qAtom) \mapsto \bigcup_{1 \leq i \leq n}((\texttt{true}, assignment_i)) \cup ((label, negated, qAtom_1))$*, where* $\mathcal{F} : qAtom \mapsto (qAtom_1, assignments)$ *and* $assignments = (assignment_1, \ldots\ assignment_n)$ *(*$n \geq 0$*).*

Example 26 $\mathcal{F}$: `a(b[c])` $\mapsto$ (`b[c] = X`, `a(X)`) *and* $\mathcal{F}$: `[a] not b[c] = d[e]` $\mapsto$ (`[a] not b[c] = X`,`d[e] = X`).

Definition 31 (Rule) *If* $body = (literal_1, \ldots\ literal_n)$ *(*$n \geq 0$*) and* $(head, body)$ *is a rule then* $\mathcal{F} : (head, body) \mapsto (head_1, assignments \cup \bigcup_{1 \leq i \leq n} \mathcal{F}(literal_i))$*, where* $\mathcal{F} : head \mapsto (head_1, assignments)$.

If $(label, head, body)$ *is a rule then* $\mathcal{F} : (label, head, body) \mapsto (label, head_1, assignments \cup \bigcup_{1 \leq i \leq n} \mathcal{F}(literal_i))$*, where* $\mathcal{F} : head \mapsto (head_1, assignments)$.

Example 27 $\mathcal{F}$: `b[c] = d[e] :- a.` $\mapsto$`b[c] = X :- d[e] = X, a.` *and* $\mathcal{F}$: `[a]: allow(a(b[c])) :- not b[c] = d[e].` $\mapsto$`[a]: allow(a(X)) :- b[c] = X, d[e] = Y, not b[c] = Y.`.

3.5.5 Metaatoms

Definition 32 (Actor metaatom) *If atom is an actor metaatom then* $\mathcal{F} : atom \mapsto (atom, \emptyset)$.

Definition 33 (Sensitivity metaatom) *If atom is a sensitivity metaatom then* $\mathcal{F} : atom \mapsto (atom, \emptyset)$.

Definition 34 (Explanation metaatom) *If* $(label, term)$ *is an explanation metaatom then* $\mathcal{F} : (label, term) \mapsto ((label, term_1), assignments)$, *where* $\mathcal{F} : term \mapsto (term_1, assignments)$.

Example 28 $\mathcal{F}$: `[a]->explanation:b[c]` $\mapsto$ (`[a]->explana-tion:X`, (`b[c] = X`)).

3.5.6 Metarules

Definition 35 (Metaliteral) *If* $(negated, metaatom)$ *is a metaliteral then* $\mathcal{F} : (negated, metaatom) \mapsto \bigcup_{1 \leq i \leq n}((\texttt{true}, assignment_i)) \cup ((negated, metaatom_1))$, *where* $\mathcal{F} : metaatom \mapsto (metaatom_1, assignments)$ *and* $assignments = (assignment_1, \ldots\ assignment_n)$ *(*$n \geq 0$*).*

Example 29 $\mathcal{F}$: `not [a]->explanation:b[c]` $\mapsto$ (`b[c] = X`, `not [a]->expla-nation:X`)

Definition 36 (Metarule) *If* $body = (literal_1, \ldots\ literal_n)$ *(*$n \geq 0$*) and* $(head, body)$ *is a metarule then* $\mathcal{F} : (head, body) \mapsto (head_1, assignments \cup \bigcup_{1 \leq i \leq n} \mathcal{F}(literal_i))$, *where* $\mathcal{F} : head \mapsto (head_1, assignments)$.

Example 30 $\mathcal{F}$: `[a]->explanation:b[c] :- not b[c] = d[e].` $\mapsto$ `[a]->exp-lanation:X :- b[c] = X, d[e] = Y, not b[c] = Y.`.

3.5.7 Policy

Definition 37 (Policy) *If* $policy = (rule_1, \ldots\ rule_n)$ *(*$n \geq 0$*) is a policy then* $\mathcal{F} : policy \mapsto \bigcup_{1 \leq i \leq n} \mathcal{F}(rule_i)$.

3.6 What's new

This section describes the most important differences between the version of the PROTUNE language presented in this work (PROTUNE2 for short) and the one presented in [19] (PROTUNE1 for short).

External resources Logical external resources (cf. Note 6) were not supported in PROTUNE1 nor were the automatic identification, location, deployment and exploitation of external resources (cf. Section 3.1.3). As a first consequence, the enforcement of PROTUNE1 policies could take place only if: (i) needed (provisional) external resources were manually made available to the PROTUNE1 engine; and (ii) each external resource was manually linked to an identifier within a configuration file. Moreover, the binding between the external resource responsible for a (provisional) predicate appearing in a policy and the predicate itself had to be made explicit within the policy by means of *ontology metarules* like the following one, which binds the (provisional) predicate `retrieve/2` to the external resource identified in the configuration file by the URI `www.happy-book.com/policies/module.mf!/retrieve/2`.

retrieve(Id, Resource) → *ontology* :
<www.happy-book.com/policies/module.mf!/retrieve/2> .

Finally, for each predicate appearing in a policy, the PROTUNE1 engine had to be told about its type by means of *type metarules* like the following one, which states that the predicate `retrieve/2` is a provisional one.

retrieve(Id, Resource) → *type*:provisional.

The following list shows that, beside unnecessarily requiring the policy author to specify information which could be retrieved in an automatic way, type and ontology metarules were sources of possible errors.

- Mixing up `logical` with `provisional` in a type metarule as well as misspelling the identifier of the external resource in an ontology metarule induced the PROTUNE1 engine to behave differently than expected

- Since in PROTUNE1 type and ontology metarules did not involve predicates but atoms, the PROTUNE1 engine had no means (but by resorting to defaults) to retrieve the type of the predicate exploited in the atom $retrieve(id_2, res_2)$, if the only type metarule involving the predicate `retrieve/2` and appearing in the policy was the following one.

$$retrieve(id_1, res_1) \rightarrow type : provisional.$$

- If, beside the one above, the policy had contained the type metarule

$$retrieve(id_2, res_2) \rightarrow type : logical.$$

the type of the predicate `retrieve/2` would have depended on the atom exploiting it. Similarly, the type of the predicate would have varied according to the outcome of the evaluation of $L_1, \ldots\ L_n$ if the policy had contained a metarule whose body was not empty, like the following one.

$$retrieve(Id, Resource) \rightarrow type : logical \leftarrow L_1, \ldots\ L_n.$$

Like type and ontology metarules, in PROTUNE1 also actor metarules (cf. Section 3.2) involved atoms. Beside the issues listed above, this made impossible for a policy author writing a policy containing two literals which exploited the same predicate and such that the first (resp. second) one had to be evaluated by the server (resp. client).

Provisional atoms PROTUNE1 supported two possible representations of provisional atoms: either as classical atoms (cf. Section 3.2) or as *special atoms*. The first representation was meant to be used for atoms whose evaluation had side-effects, whereas the second one for atoms whose evaluation was side-effect free. A special atom looked like the following

$$\texttt{in([}op_1\texttt{, ... }op_m\texttt{], }r\texttt{:}p\texttt{(}ip_1\texttt{, ... }ip_n\texttt{))} \qquad (1)$$

where r is the (identifier of the) external resource, p the predicate name, $ip_1, \ldots\ ip_n$ and $op_1, \ldots\ op_m$ its input and output parameters respectively. Output parameters could only be variables, whereas input parameters only variables or strings, possibly concatenated with each other by means of the operator `&` (`X & " is a string."`). The following list shows some drawbacks of PROTUNE1's provisional atoms.

- As described above, the two representations of PROTUNE1's provisional atoms were meant to be used in different cases. However, since it was the developer of the external resource who ultimately determined the actual representation, there was no guarantee that she complied with PROTUNE1's guidelines and, in practice, the evaluation of special atoms could have side-effects and the one of provisional atoms represented as classical atoms could be side-effect free. Moreover, one could argue that the policy author mostly does not care whether an atom's evaluation has side-effects. Therefore, forcing her to use a different syntax in order to distinguish the two cases reduced PROTUNE1's transparency and abstraction level

- As we will see in the next paragraph, PROTUNE1 strings could be represented in a number of ways, e.g., as double-quoted character strings. However, neither was it possible representing them as double-quoted character strings outside a special atom nor was it possible representing them but as double-quoted character strings within a special atom. The constraint of representing the same entity in different ways according to the part of the policy it appeared in was clearly misleading, error-prone and user-unfriendly, as was the possibility of using the operator `&` only within special atoms

PROTUNE2 normalizes the representation of provisional atoms by getting rid of special atoms without losing any expressiveness: the PROTUNE1 special atom (1) can be represented as the PROTUNE2 (classical) atom

$$p(ip_1, \ldots\ ip_n, op_1, \ldots\ op_m)$$

if m does not depend on the values of the input parameters, or as the atom

$$p(ip_1, \ldots\ ip_n, op)$$

if it does, where *op* is supposed to be an object (cf. Section 3.2) having as many properties as output parameters of p. In both cases, the directive `import <`r`!/`p`/`a`>` must appear at the beginning of the policy, where a is the arity of p (i.e., $n + m$ or $n + 1$ respectively).

Built-in datatypes PROTUNE1 provided fewer built-in datatypes than PROTUNE2, namely, integers and strings. Although with a slightly different syntax, complex terms (cf. Section 3.2) were supported in PROTUNE1 as well – however, objects were not: the grammar of the PROTUNE1 language allowed to define complex terms but, being their semantics not formally specified, the PROTUNE1 engine was not able to handle them.

A PROTUNE1 string could be represented in a number of ways: as single- or double-quoted character string (`'A string'` or `"A string"`), as <>-delimited or non-delimited character string, if it fulfilled some restrictions (`<Astring>` or `aString`), or as the concatenation of a string defined in a `PREFIX` directive and further characters (e.g., `{p:string}` was yet another representation of the string `Astring`, if the directive `PREFIX p : <A>` appeared at the beginning of the policy). Such a proliferation of representations was likely to puzzle the user, especially considering that some representations were only allowed within certain linguistic constructs (cf. previous paragraph). For this reason, only the double-quoted representation is allowed in PROTUNE2.

Miscellany We conclude by mentioning further restrictions of PROTUNE1 which have been removed in PROTUNE2: (i) only provisional atoms could appear in a PROTUNE1 goal (and they had to be wrapped by the `allow` token); (ii) PROTUNE1 did not allow to extend the semantics of built-in operators (cf. Section 3.3.2); and (iii) PROTUNE1 provided up to one single response to a request. Last but not least, the two versions of PROTUNE base on different formalisms: PROTUNE1 was based

on full Logic Programming [64], whereas PROTUNE2 on Datalog [2].

CHAPTER 4
Using Policies in a Natural Way

Recent experiences with Facebook's *beacon* service[1] and Virgin's use of Flickr pictures[2] have shown that the widespread adoption of applications (like Web 2.0 applications and social softwares) which allow users to share data did not come along with an equally widespread use of technologies (such as policy languages) which allow users to fine-grainedly specify which data can be shared with whom. One reason for this is that current policy languages are too complex to be easily exploited by common users in a profitable way, since they typically require a policy author to be a computer expert[3] (see for instance excerpts of policies in different languages in Fig. 4.1 and [12]).

Complex syntax (e.g., unusual tokens not exploited in every-day communication such as `:-`, `->` and `&`), logical formalisms (cf. Section 2.3.1) and the difficulty in general of grasping the meaning of the policy without previous knowledge or being a computer expert are some of the reasons which make formal languages particularly unfriendly to common users. Moreover, real-world policies tend to be complex, making the specification process even harder.

As we mentioned at the beginning of Chapter 3, PROTUNE includes mechanisms to generate natural language descriptions out of already specified policies (e.g., the explanation metarules shown in lines 10–25 of Fig. 4.1 can be used to explain the policy to the end users in a user-friendly manner). However, this approach only helps to understand policies but not to create or modify them.

This chapter describes PROTUNE's natural language front-end which allows users to define policies in the ACE controlled natural language [39]. Such policies are then automatically translated to the PROTUNE policy language. With this

[1] `http://www.washingtonpost.com/wp-dyn/content/article/2007/11/29/AR2007112902503.html?hpid=topnews`

[2] `http://www.smh.com.au/news/technology/virgin-sued-for-using-teens-photo/2007/09/21/1189881735928.html`

[3] "Too often, only the Ph.D. student that designed a policy language or framework can use it effectively", Kent E. Seamons. Semantic Web and Policy Workshop, 2005.

```
(1)  allow(download(User, Resource)) :-
(2)     authenticated(User),
(3)     hasSubscription(User, Subscription),
(4)     availableFor(Subscription, Resource).

(5)  authenticated(User) :-
(6)     send(User, Credential),
(7)     not forged(Credential),
(8)     Credential["name"] = Name,
(9)     User["name"] = Name.

(10) download(User, Resource)->explanation:
(11)    "user \"" & User & "\" downloads resource \"" & Resource
(12)    & "\"".
(13) authenticated(User)->explanation:
(14)    "user \"" & User & "\" is authenticated".
(15) hasSubscription(User, Subscription)->explanation:
(16)    "user \"" & User & "\" has subscription \"" & Subscription
(17)    & "\"".
(18) availableFor(Subscription, Resource)->explanation:
(19)    "subscription \"" & Subscription &
(20)    "\" is available for resource \"" & Resource & "\"".
(21) send(User, Credential)->explanation:
(22)    "user \"" & User & "\" sends credential \"" & Credential
(23)    & "\"".
(24) forged(Credential)->explanation:
(25)    "credential \"" & Credential & "\" is forged".
```

Figure 4.1: Example Protune policy

approach, the policy shown in Fig. 4.1 can be expressed as follows.

1. If a user is authenticated and has a subscription that is available for a resource then she can download the resource.

2. Every user who sends a credential that is not provably forged and whose name is the user's name is authenticated.

Notice that the use of controlled natural languages to express policies makes the explanation metarules mentioned above not needed anymore, since such policies can be understood by common users as they are.

This chapter is organized as follows: Section 4.1 presents the concept of "controlled natural language", introduces the controlled natural language ACE and es-

pecially focuses on the internal format (DRS) the ACE parsing engine translates ACE sentences into. Section 4.2 describes the mapping we defined between DRSs and PROTUNE policies. Section 4.3 presents a predictive editor which guides the user step by step toward defining PROTUNE policies by means of ACE sentences and reports on the results of a preliminary user study evaluating its usability. Finally, Section 4.4 compares our approach with related work.

4.1 Controlled natural languages

Controlled natural languages (CNLs) are subsets of natural languages that are restricted ("controlled") in a way that reduces or removes the ambiguity of the language [72]. The main motivation is to improve the human-computer interaction[4]. The users should be able to express statements in a language that is familiar to them. On the other hand, the restrictions of the language enable the automatic processing by computers.

The controlled natural languages we consider here are completely formal languages that can be mapped automatically and unambiguously to formal representations. Such languages are described by a formal grammar and are usually explained to the users on the basis of construction rules ("which subset of the natural language is covered?") and interpretation rules ("how are the sentences interpreted that would be ambiguous in the natural language?"). In order to clarify the interpretation issue, let us have a look at the following sentence.

Bob buys the book with a card.

In full natural language this sentence has at least two different meanings according to the context it appears in, namely

- Bob buys a book which has a card.
- Bob buys a book by means of a (credit) card.

[4]There exist also some CNLs whose goal is just to improve the understanding for non-native speakers without aiming to be computer-processable. We do not consider such languages here.

In order to disambiguate sentences like the one above, CNLs have to make a number of design choices in order to deterministically select one out of n possible readings. The controlled natural language ACE, for instance, assumes that all prepositional phrases refer to the closest verb (unless the preposition is *of*, in which case the prepositional phrase always refers to the preceding noun phrase). In the example above, ACE would assume that the phrases *the book* and *with a card* refer to the action of *buying*, boiling down to the second one of the readings listed above.

Ambiguity resolution takes place on a purely syntactic basis, so that CNLs typically do not differentiate between *the purchase of the man* and *the purchase of the book*, although in everyday's speech one probably means that it is the man to buy whereas it is the book to be bought.

In the following sections we introduce the controlled natural language ACE that we used for PROTUNE's natural language front-end. Furthermore, we describe the internal format the ACE parsing engine translates ACE sentences into.

4.1.1 Attempto Controlled English

Attempto Controlled English (ACE)[5] [39] is a mature controlled natural language (concretely, a controlled subset of English) that has been developed and constantly extended during the last fifteen years. Initially designed as a specification language, the focus has shifted toward knowledge representation and especially applications for the Semantic Web.

ACE supports a wide range of English constructs: nouns, proper names, verbs, adjectives, singular and plural noun phrases, active and passive voice, pronouns, relative phrases, conjunction and disjunction (*and*, *or*), existential and universal quantifiers (e.g., *a*, *every*), negation, modal verbs (e.g., *can*, *must*), cardinality restrictions (e.g., *at most 3 objects*), anaphoric references (e.g., *the resource*, *she*), questions, commands and much more.

ACE comes along with a number of tools, the most important of which is the Attempto Parsing Engine (APE)[6]. APE translates ACE sentences into DRSs [16, 51], whose expressiveness is equivalent to the one of first-order logic. Various utility

[5] `http://attempto.ifi.uzh.ch/site/`

[6] `http://attempto.ifi.uzh.ch/ape/`

<table>
<tr><th>Standard NLP tool</th><th>APE</th></tr>
<tr><td rowspan="2">segmentation</td><td>tokenization</td></tr>
<tr><td>sentence splitting</td></tr>
<tr><td>classification</td><td rowspan="2">parsing</td></tr>
<tr><td>association</td></tr>
<tr><td colspan="2">normalization/deduplication</td></tr>
</table>

Table 4.1: A standard NLP tool's processing steps vs. APE's ones

tools use such representation and translate DRSs into other languages like e.g., the ontology languages OWL and SWRL [49]. Furthermore, the ACE paraphraser can translate DRSs back to ACE sentences, thereby providing a reformulation of the original ACE text that can help understanding its APE interpretation.

Furthermore, many tools exist that apply ACE to a certain problem area: RACE[7] is a reasoner that can reason on ACE texts, ACE View [50] is a plugin for the popular Protégé ontology editor and AceWiki [58] is a semantic wiki engine that uses ACE for the representation of formal knowledge within a wiki.

Finally, ACE supports user-defined lexica that define proper names, nouns, verbs, adjectives, adverbs, and prepositions. Thus, it is easy to customize ACE by integrating a particular domain-specific vocabulary.

4.1.2 ACE as a natural language

ACE sentences are interpreted by APE. While interpreting an ACE sentence, APE carries out the usual tasks accomplished by NLP tools. Tab. 4.1 lists such tasks and shows the main differences w.r.t. the processing steps carried out by APE.

Segmentation is concerned with finding the boundaries of grammatical elements (be they single words or whole sentences) which will then undergo further processing. W.r.t. the first ACE policy presented at the beginning of the chapter, during the segmentation step *If*, *a*, *user*, ... are recognized as words which form the single sentence the whole policy consists of. APE breaks down the segmentation step into two sub-tasks, namely tokenization and sentence splitting, concerned with identifying the individual words and sentences of the input text respectively.

[7] `http://attempto.ifi.uzh.ch/race/`

During the classification step, each token is assigned a lexical category, whereas the association step establishes relationships between lexical elements, thereby grouping them into higher-level clusters. W.r.t. the first ACE policy presented at the beginning of the chapter, during the classification step the word *a* (resp. *user*) is recognized as an article (resp. a common noun). During the association step, the words *a* and *user* (resp. *is* and *authenticated*) are recognized as forming a noun (resp. verb) phrase. APE performs classification and association in one single parsing step.

Finally, both in usual NLP tools and in APE, the normalization/deduplication step identifies and resolves yet higher-level relationships between linguistic elements. W.r.t. the first ACE policy presented at the beginning of the chapter, during this step the pronoun *she* is recognized as referring to the common noun *user*.

Beside the differences we just outlined, the overall approach distinguishes APE from NLP tools. Whilst NLP tools aim at processing and making sense out of (in principle) whichever input sentence, APE insists on well-formedness: an input sentence not complying with ACE's grammar is simply rejected. Moreover, whilst an NLP tool is expected to interpret an input sentence the way a native speaker would do, the burden of expressing a content the way APE interprets it relies on the author of the input sentence.

4.1.3 Discourse Representation Structure

As a formal language, ACE has a deterministic and well-defined meaning. Therefore, a developer may in principle use ACE sentences directly as the input format of her application. This choice may turn out to be quite challenging since, being ACE designed to be human-friendly, the translation of ACE sentences into a logical representation is not trivial.

For this reason, the output of APE is a semantically equivalent representation of the input ACE sentence which can be processed more easily in an automatic way. This representation is called Discourse Representation Structure (DRS) and can be thought of as a sort of abstract syntax tree of the input sentence. A DRS encodes the whole information of the original ACE sentence and can be used "as it is" by

```
drs([A, B, C, D, E, F, G, H], [
   object(A, user, countable, na, eq, 1)-1,
   property(B, authenticated, pos)-1,
   predicate(C, be, A, B)-1,
   object(D, subscription, countable, na, eq, 1)-1,
   object(E, resource, countable, na, eq, 1)-1,
   property(F, available, pos)-1,
   predicate(G, be, D, F)-1,
   modifier_pp(G, for, E)-1,
   predicate(H, have, A, D)-1
])
```

Table 4.2: An example DRS

Prolog programs.

Tab. 4.2, shows the DRS corresponding to the *if*-part of the first ACE policy presented at the beginning of the chapter. In the following we will explain this example, focusing on the elements relevant for the remainder of the chapter. A thorough description of the DRS language is provided in [40].

For every noun (*user*, *subscription* and *resource* in the example), the DRS introduces an `object` predicate, which is identified by a variable (`A`, `D` and `E` respectively). All of *user*, *subscription* and *resource* are countable common nouns, therefore the third argument of their `object` predicate is `countable`. Proper names are also represented by `object` predicates, but their third argument is `named`.

For every adjective (*authenticated* and *available*), the DRS introduces a `property` predicate, which is also identified by a variable (`B` and `F` respectively).

For every verb (namely *is (authenticated)*, *is (available)* and *has*), the DRS introduces a `predicate` predicate, which is identified by a variable (`C`, `G` and `H` respectively). The third argument represents the subject, the fourth argument represents the object for transitive verbs and the predicate for the verb *be*, if it is used in a copular way (i.e., to state a property of the subject). For example, the subject of *is (authenticated)* is *user*, therefore the third argument of its `predicate` predicate is the variable identifying *user*—namely `A`. On the other hand, its fourth argument is the variable identifying the predicate *authenticated*—namely `B`.

Beside transitive verbs (like *have*), ACE also supports intransitive and di-

transitive verbs (like *register* and *give* respectively): intransitive verbs only have a subject (e.g., someone *registers*), whereas ditransitive verbs, beside having a subject and a (direct) object, also have an indirect object (e.g., *someone gives something* to someone). Intransitive (resp. ditransitive) verbs are represented in the DRS as `predicate` predicates with three (resp. five) arguments.

The predicate *is (available)* is modified by a prepositional phrase (*for a resource*). For this reason, the DRS contains a `modifier_pp` predicate. Its first argument is the variable identifying *is (available)* (namely `G`), its second argument is the preposition contained in the prepositional phrase (namely *for*) and its third argument is the variable identifying the noun (*resource*) contained in the prepositional phrase (namely `E`).

Tab. 4.3 shows a pretty-printed version of the DRS that corresponds to the the first ACE policy presented at the beginning of the chapter. It shows further features of the DRS language which we will now explain.

Each box in Tab. 4.3 corresponds to a `drs` predicate. Nested boxes reflect the possibility that `drs` predicates are nested within other `drs` predicates. This happens whenever the ACE sentence contains linguistic constructs like implication or possibility.

Our example sentence is a conditional sentence with the protasis *a user is authenticated and has a subscription that is available for a resource* and the apodosis *she can download the resource*. The surrounding box in the DRS connects the boxes representing protasis and apodosis by the implication symbol ($\Rightarrow$).

The apodosis itself contains a possibility construct (*can download the resource*). The DRS represents it by a nested box, where the inner box for *download the resource* is preceded by the deontic operator denoting possibility ($\Diamond$).

Each box in Tab. 4.3 consists of two parts. The bottom part contains predicates (in particular, nested `drs` predicates) within the scope of the `drs` predicate represented by the box, whereas the upper part lists the variables directly introduced by predicates within the scope of the `drs`, i.e., not introduced by nested `drs`'s.

We conclude our introduction to the DRS language by mentioning that ACE allows to express formulae involving (among others) integers, reals, strings, variables

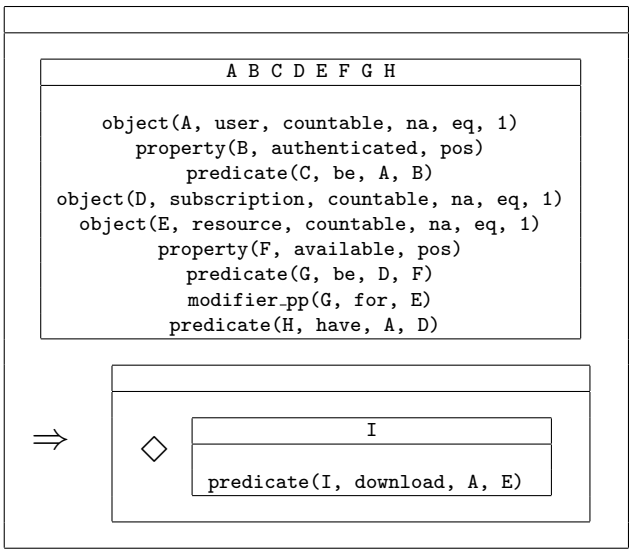


Table 4.3: A pretty-printed DRS

and the usual (in)equality operators. Such formulae are translated into DRSs by means of the `formula` predicate. For instance, the ACE fragment $X > 1.0$ is translated into the DRS predicates

```
object(A, something, dom, na, na, na)
formula(A, >, real(1.0))
```

Notice that the variable occurring in the ACE fragment (namely X) has been replaced by another one (namely A) and that the datatype of 1.0 is explicitly mentioned. Finally, the fact that the ACE fragment contains a variable X is interpreted by APE as asserting that some entity identified by X exists. The existence of such an entity is explicitly stated in the DRS by the `object` predicate.

4.2 Mapping DRS to Protune

Defining a mapping between DRSs and PROTUNE policies involves two main tasks: (i) characterizing the subset of the DRS language that expresses PROTUNE policies; and (ii) mapping this subset to PROTUNE. In both steps, it makes sense trying to translate linguistic constructs of the DRS language into PROTUNE linguistic constructs which have a comparable semantics.

In this section we present the DRS Parsing Engine (DPE), which maps a subset of all possible DRSs into PROTUNE policies. We first outline the requirements for the mapping. The actual mapping is presented in Section 4.2.2.

4.2.1 Mapping requirements

Two main requirements guided the ACE→DRS mapping, namely

Unambiguity The mapping is unambiguous, in the sense that for each ACE sentence there is only one DRS it is translated into (i.e., the input ACE sentence univocally determines the DRS it is translated into)

Syntax-based As described in Section 4.1, only syntactic information is exploited by APE when translating ACE sentences into DRSs. Contextual information, which can help humans to disambiguate ambiguous sentences, is not exploited by APE which resorts to a set of built-in disambiguation rules

Since DPE was meant to be an extension of APE, we designed our DRS→PROTUNE mapping to be unambiguous (i.e., the input DRS must univocally determine the PROTUNE policy it is translated into) and syntax-based. As a third requirement, the mapping should cover PROTUNE as much as possible, i.e., one must be able to define as many PROTUNE policies as possible through DRSs. As we will see in Section 4.2.2, all PROTUNE policies not containing metarules (cf. Section 3.2) can be expressed by means of suitable DRSs. On the other hand, not all possible DRSs can be translated into PROTUNE policies. The reason is that many linguistic constructs of the DRS language (e.g., classical negation, necessity and sentence subordination) do not have an immediate correspondence in PROTUNE.

4.2.2 Mapping

This section introduces the mapping DPE implements. Only a subset of the issues considered when defining the DRS→PROTUNE mapping will be discussed. A more thorough overview is provided by [36].

We first outline the features of the PROTUNE language we chose to express through DRSs and provide a rational for our choices. We then describe the overall

structure a DRS must have in order to represent a PROTUNE policy. Finally, we describe how to represent PROTUNE atoms by means of DRSs.

Which policies? First of all, our mapping does not cover metarules (cf. Section 3.2). Although the PROTUNE language in principle allows to create metarules as complex as rules can be, in almost all cases real-world metarules conform to one out of a set of few predefined patterns. For this reason, providing users with the expressiveness of (an extensive subset of) ACE in order to define metarules would be confusing and hence error-prone.

Mapping `drs` predicates As described in Section 4.1.3, nested DRSs can represent (among else): (i) implication; (ii) negation-as-failure (NAF); and (iii) possibility. These linguistic constructs can be directly translated into PROTUNE: (i) rules; (ii) negated literals; and (iii) decision atoms (cf. Tab. 4.4).

As described in Section 3.2, a generic PROTUNE rule has the form $A \leftarrow L_1, \ldots L_n$ where $n \geq 0$, A is an atom and $L_1, \ldots L_n$ are literals. The meaning of the rule is

> A holds if each and all of $L_1, \ldots L_n$ hold.

$\forall i : 1 \leq i \leq n$ L_i equals either B_i or $\sim B_i$ for some atom B_i. The meaning of $\sim B_i$ is

> B_i does not provably hold.

The semantics of atoms A and B_i varies according to their type. In particular, the meaning of a decision atom $allow(C)$ (for some atom C) is

> C can be evaluated upon request of some external entity.

Tab. 4.4 shows the two formats a DRS must possess in order to represent a PROTUNE rule. Box (a.1) represents a rule's body as a set of literals, whereas box (a.2) represents a rule's head. Box (b) represents the special case of a PROTUNE rule without body. More formally, this leads to the following definition.

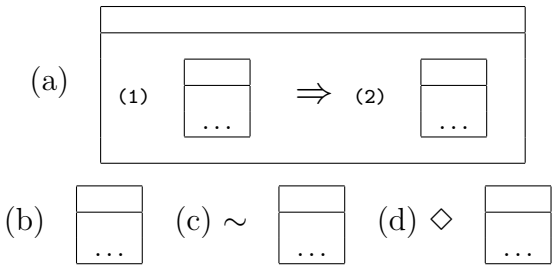


Table 4.4: Generic forms of DRSs representing Protune rules (a-b), negated literals (c) and decision atoms (d)

Definition 38 (DRS rule) *A DRS rule has either of the formats (a) or (b) in Tab. 4.4, where*

- *the content of box (a.1) is a set of DRS literals and (a.2) is a DRS head*
- *(b) is a DRS atom*

DRSs can express NAF, which we can exploit in order to represent PROTUNE negated literals.

Definition 39 (DRS negated literal) *A DRS negated literal has the format (c) in Tab. 4.4, where the content of the box is a DRS atom.*

In order to represent PROTUNE decision atoms, we use the possibility construct of the DRS language, which is supposed to represent the possibility operator of deontic logic but can be nicely reused for PROTUNE decision atoms.

Definition 40 (DRS decision atom) *A DRS decision atom has the format (d) in Tab. 4.4, where the content of the box is a DRS atom.*

Mapping atomic DRS predicates In order to finalize the DRS→PROTUNE mapping, we still have to define how to express comparison, atoms and value-assignment atoms (cf. Section 3.2) by means of constructs of the DRS language. This is accomplished by syntactically characterizing patterns of atomic DRS predicates.

N°	ACE	DRS	PROTUNE
1	X > 1.0	`object(A, something, dom, na, na, na)` `formula(A, >, real(1.0))`	$A > 1.0$

2	(If) Bob sends a credential	`object(A, Bob, named, na, eq, 1)`
		`object(B, credential, countable, na, eq, 1)`
		`predicate(C, send, A, B)`
			$send(\text{“}Bob\text{”}, Credential)$

3	Bob negotiates with Alice	`object(A, Bob, named, na, eq, 1)`
		`object(B, Alice, named, na, eq, 1)`
		`predicate(C, negotiate, A)`
		`modifier_pp(C, with, B)`
			$\text{“}negotiate\#with\text{”}(\text{“}Bob\text{”}, \text{“}Alice\text{”})$

4	Bob is a user	`object(A, Bob, named, na, eq, 1)`
		`object(B, user, countable, na, eq, 1)`
		`predicate(C, be, A, B)`
			$user(\text{“}Bob\text{”})$

5	“subscription123” is available for Bob	`object(A, Bob, named, na, eq, 1)`
		`property(B, available, pos)`
		`predicate(C, be, string(subscription123), B)`
		`modifier_pp(C, for, A)`
			$\text{“}available\#for\text{”}($ $\text{“}subscription123\text{”}, \text{“}Bob\text{”}$ $)$

6	“Guernica.jpg” is in “pictures”	`predicate(A, be, string(Guernica.jpg))` `modifier_pp(A, in, string(pictures))`	$\text{“}be\#in\text{”}(\text{“}Guernica.jpg\text{”}, \text{“}pictures\text{”})$

7	(If) the name of the credential	`relation(C, of, D)`
		`object(C, name, countable, na, eq, 1)`
		`object(D, credential, countable, na, eq, 1)`
			$Credential[\text{“}name\text{”}] = Name$

8	Bob’s name	`object(A, Bob, named, na, eq, 1)`
		`relation(B, of, A)`
		`object(B, name, countable, na, eq, 1)`
			$\text{“}Bob\text{”}[\text{“}name\text{”}] = Name$

Table 4.5: DRS→Protune mapping examples

We already saw in Section 4.1.3 that DRSs can express formulae by means of the predicate `formula`, therefore we can exploit this possibility in order to represent PROTUNE comparison atoms.

Definition 41 (DRS comparison atom) *A DRS comparison atom is a set of DRS predicates containing one single* `formula` *predicate and zero or more* `object` *predicates.*

As the first example in Tab. 4.5 shows, only the DRS `formula` predicate is actually used for the translation.

Different patterns of DRS linguistic constructs can be mapped to PROTUNE classical atoms. We distinguish the following cases.

DRS `predicate` predicates are translated into PROTUNE classical atoms using the original verb lemma as the name of the atom. DRS `object` predicates are dereferenced and directly included into the atom as constants (for proper names) or variables (for common nouns). Example 2 in Tab. 4.5 shows how the DRS for *Bob sends a credential* is translated. *send* becomes the name of a PROTUNE classical atom with a constant argument for the proper name *Bob* and variable argument for the common noun *credential*.

DRS `modifier_pp` predicates are translated by extending the name of the corresponding PROTUNE classical atom with the preposition of the prepositional phrase and again including the dereferenced DRS `object` predicate as an argument (cf. Example 3 in Tab. 4.5).

The translation pattern of the DRS `predicate` predicate differs if: (i) it represents the verb *be*; and (ii) the verb *be* is used in a copular way (cf. Section 4.1.3). Example 4 in Tab. 4.5 illustrates the simple case, where the DRS predicate for the complement (*user*) is translated into the name of a PROTUNE classical atom with one argument for the subject *Bob*. Example 5 illustrates the more complex case, where *be* is further modified by the prepositional phrase *for Bob*. Finally, Example 6 shows that *be*, when not used in a copular way, is translated like other verbs (cf. Example 3).

In summary, PROTUNE classical atoms are generated from the following patterns of DRS predicates.

Definition 42 (DRS classical atom) *A DRS classical atom is a set of DRS predicates containing one single* `predicate` *predicate and zero or more* `modifier_pp`*,* `object` *and* `property` *predicates.*

PROTUNE value-assignment atoms express properties possessed by objects. The DRS language represents possession relationships between two objects X_1 and X_2 by means of the predicate `relation(`X_1`,of,`X_2`)`. For this reason, we exploit the DRS `relation` predicate in order to represent PROTUNE value-assignment atoms. Examples 7 and 8 in Tab. 4.5 illustrate how DRS `relation` predicates are translated into PROTUNE value-assignment atoms.

Definition 43 (DRS value-assignment atom) *A DRS value-assignment atom is a set of DRS predicates containing one single* `relation` *predicate and zero or more* `object` *predicates.*

Together, Definitions 39-43 define all necessary DRS patterns to generate PROTUNE atoms.

Definition 44 (DRS atom/literal) *A DRS atom is either a DRS decision atom or a DRS comparison atom or a DRS classical atom or a DRS value-assignment atom. A DRS literal is either a DRS atom or a DRS negated literal.*

4.3 Usability issues

APE's functionalities are available through a number of interfaces. In particular, a stand-alone distribution is available[8] which can be downloaded and installed on one's own computer.

Whenever a non well-formed sentence is input to APE's stand-alone distribution, an error message containing debug information as well as suggestions to fix the problem occurred is shown (e.g., *Every ACE text must end with . or ? or !.*).

DPE provides a similar interface, with the difference that error messages (and suggestions for fixes) are provided not only if the input is not an ACE sentence, but

[8] `http://attempto.ifi.uzh.ch/site/downloads/files/ape-6.5-100128.zip`

also if it is an ACE sentence which does not translate into a DRS rule (as defined in Section 4.2.2), and hence cannot be translated into a PROTUNE policy.

Although meaningful error messages can help by driving users toward well-formed input sentences, the approach provided by APE's stand-alone distribution and DPE suffers from a big drawback in terms of usability: even if the user fixes the problems occurred in her sentence according to the suggestions contained in the error message, there is no guarantee that the corrected sentence will be well-formed, since it might still contain (different) errors. For this reason, the process of inputing a sentence can result in a (possibly long) sequence of steps, in each of which new errors appear which need to be fixed in the following attempt.

Furthermore, the detection of the cause of a syntax error is not trivial at all. For instance, in the case of the incorrect sentence *a customer provides a card pays*, the user probably forgot a word somewhere in the sentence. But at least three different correct sentences can be constructed by the introduction of one word: *a customer provides a card that pays*, *a customer that provides a card pays* and *a customer provides a card and pays*. Thus, without further information it is impossible to find out what the actual mistake was.

W.r.t. usability, a much better approach is to constrain users to only create well-formed input sentences. Section 4.3.1 describes a possible way to enforce this constraint.

4.3.1 A predictive authoring tool

Even though CNLs are much easier to use than other computer languages, they still require a minimum learning process of their restrictions and interpretation rules. In order to ease and speed up this learning process, we suggest to use predictive editors [59], i.e., editors that are aware of the grammar of the used CNL and can guide the user step by step through the creation of a sentence. Such an editor forces the user to continue the sentence in a way that corresponds to the respective grammar and the currently loaded lexicon. Therefore, a complete sentence is always syntactically correct and no error messages are needed.

Several such editors have been introduced, for example ECOLE[9] or GINO [15]. AceWiki and the ACE Editor make use of the same predictive editor that has been developed to support subsets of ACE. This predictive editor is very flexible, which enabled us to adapt it to PROTUNE's requirements and to use it for the evaluation described in Section 4.3.3.

Fig. 4.2 is a screenshot of the predictive editor that we used for the evaluation. The numbered components are explained below.

(1) is a read-only text area that shows the beginning of a sentence. This fragment has been entered by a user and it has been accepted by the editor as a correct sentence beginning. Thus, there is at least one possible completion that leads to a correct sentence. The button `Delete` can be used to delete the last token inserted, whereas pressing the `Clear` button resets the content of the text area.

The text field (2) can be used to enter the next words of the sentence. If they are a correct continuation of the sentence then they are moved to the text field (1) as soon as the `RETURN` key is hit.

Clicking on the entries of the menu boxes (3) is an alternative way to construct a sentence. There is a menu box for each word class that is allowed at the current position. In this case, only function words, intransitive adjectives and singular countable nouns representing persons or objects are allowed. The menu box for verbs, for example, is not shown because verbs are not allowed at this position.

4.3.2 Editor's features

The predictive editor described in Section 4.3.1 is implemented as a Web application and can be easily deployed on a servlet container and accessed remotely. The default editor can be personalized by providing a set of application-specific components, the most important of which is the grammar of the language the editor is meant to support.

As described in Section 4.2.1, not all features of ACE are used to express PROTUNE policies. For this reason, *ACE policies* (i.e., the set of all ACE sentences expressing PROTUNE policies) is a proper subset of the set of all ACE sentences.

[9] `http://www.ics.mq.edu.au/~rolfs/peng/writing-peng.html`

Our editor (shown in Fig. 4.2) currently supports a proper subset of ACE policies, which we called *ProACE*. This set is sound and possesses a remarkable property which we call *half-coverage*.

Soundness Each sentence belonging to ProACE can be translated to a PROTUNE rule. This property ensures that it is never the case that APE translates a ProACE sentence into a DRS which is not a DRS rule as defined in Section 4.2.2

Half-coverage For each ACE sentence which: (i) does not belong to ProACE; and (ii) could be in principle translated to a PROTUNE policy P; there is a(n ACE) sentence belonging to ProACE which can be translated to P. This property ensures that, if there is at all a way to express a PROTUNE policy in ACE, then there is a way to express the same policy in ProACE. In other words, this property guarantees that, although ProACE is a proper subset of ACE, it does not lower ACE's expressiveness w.r.t. the capability of expressing PROTUNE policies

4.3.3 Evaluation

We conducted a preliminary user study in order to evaluate efficacy and efficiency of our framework. The user study makes use of the editor described in Section 4.3.1 and is available at `http://policy.l3s.uni-hannover.de:9080/proace`. Its fine-grained results are reported at `http://policy.l3s.uni-hannover.de:9080/proace/report.xml` (Internet Explorer is required).

We asked the participants to reformulate the (full) natural language sentences listed in Tab. 4.6 so that the editor accepted them. We did not provide any further information to the participants beside the one available on the user study's Web pages. In particular, we did not provide any information about ProACE's syntax and interpretation rules. We only supplied few example ProACE sentences which the users were supposed to mimic when reformulating the sentences in Tab. 4.6.

In order to evaluate our framework, we tracked the user's interaction with it, namely: (i) the time elapsed before the user undertook an action (e.g., pressing a

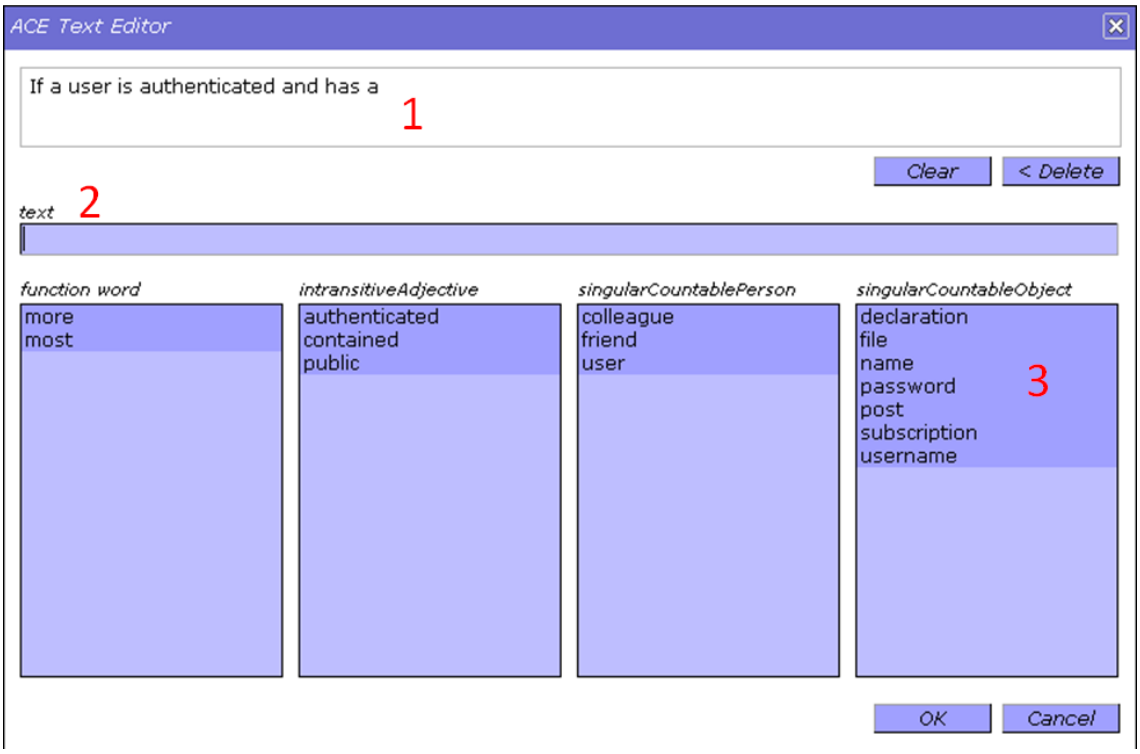


Figure 4.2: A predictive authoring tool

N°	Sentence	Average attempt number	Average time (s)	Success rate (%)
1	Colleagues can access files related to work.	2	194 (5.86)	11.11 (22.22)
2	Directory "work" contains files related to work.	1.79	162 (6.18)	18.18
3	Friends can see everything which is not related to work.	1.64	151 (4.86)	20
4	Friends of friends can see public files.	1.5	125 (4.82)	9.09 (81.81)
5	"Un chien andalou.mpg" is public.	1.38	51 (5.39)	100
6	"Guernica.jpg" is in directory "work".	1.08	42 (7.08)	100
7	John is a friend.	1	17 (3.66)	92.31

Table 4.6: Sentences exploited in the user study

button or hitting the `RETURN` key); (ii) how many times the user reworked the same sentence and the overall time spent with it; and (iii) the final version of a sentence.

The first set of statistics can be used to evaluate the efficiency of our framework: if much time elapses between two following actions it might mean that the user is unsure about what she should do and possibly does not find the interface user-friendly. The third set of statistics can be used to check whether the sentence reformulated by the user, as it is interpreted by APE and DPE, is semantically equivalent to the original (full natural language) sentence, thereby evaluating the efficacy of our framework. Finally, the second set of statistics can be used to estimate the difficulty degree of a sentence: if the user spent much time with a sentence and reworked it many times, most likely she found that sentence more difficult than another one she did not spend so much time with and she did not rework so many times. The estimate of the difficulty degree of a sentence can then be used to normalize the other sets of statistics.

The entries in Tab. 4.6 are ordered according to a descending difficulty level. Tab. 4.6 also shows how many times a sentence has been reworked (in average) as well as the overall time spent in average with it and to start an action (between parenthesis). The time required to reformulate a sentence varies from 17s to a bit more than 3min: taking into account that the users had no previous knowledge about ProACE's syntactical constraints, we consider these values to be satisfactory. Finally, the time required to undertake an action varies from less than 4s to a bit more than 7s, being in average 5.48s: taking into account the time needed to search all menu boxes (cf. Fig. 4.2) and to estimate whether a token can express the intended meaning, we also consider these values to be satisfactory.

In order to evaluate the efficacy of our framework, we classified the sentences reformulated by the users as: (i) ProACE-correct; (ii) NL-correct; (iii) wrong; and (iv) off-topic.

ProACE-correct sentences are semantically equivalent to the original ones both in natural language and in ProACE. NL-correct sentences are semantically equivalent to the original ones in natural language but not in ProACE. For instance, sentence

Every colleague accesses everything in “work”.

can be considered a correct reformulation in natural language of the first sentence in Tab. 4.6. However, intended as a ProACE reformulation of the same sentence, it presents three errors.

1. A periphrastic form must be used to point out that only colleagues are allowed to access everything in “work”: *every colleague* must be replaced by *everyone who is a colleague*

2. The verb *can* must be used to express the permission: *accesses* must be replaced by *can access*

3. A new subordinate sentence must be used to point out the location of the resources: *in “work”* must be replaced by *which is in “work”*

Wrong sentences are semantically equivalent to the original ones neither in natural language nor in ProACE. Finally, off-topic sentences were not meant to reformulate the original ones: sometimes users did not reformulate some of the sentences but combined them with other ones in order to infer new sentences which they then reformulated. For instance, many users reformulated the first sentence in Tab. 4.6 as follows.

Everybody who is a colleague can access everything which is in “work”.

Sentence 2 in Tab. 4.6 indeed asserts that directory “work” contains files related to work. Since no other directories are mentioned which contain files related to work, the users probably considered sentence 2 equivalent to the following one by applying a sort of “closed-world assumption” inference rule.

All files related to work are contained in directory “work”.

Combining this sentence with the first one in Tab. 4.6 leads to the reformulation above. Off-topic sentences are not considered in the last column of Tab. 4.6

The last column of Tab. 4.6 shows the percentage of reformulated sentences which are ProACE-correct. Between parenthesis it is also shown how such percentage would increase if the users had been aware of DPE's interpretation rules which led to the errors described above. The results show that users find ProACE quite intuitive as long as they have to define simple policies (like sentences 5-7 in Tab. 4.6), that a basic knowledge of APE and DPE's interpretation rules already allows to specify more complex policies (like sentence 4) and that a deeper knowledge of such rules is a prerequisite to define yet more complex policies (like sentences 1-3).

4.4 Related work

To the best of our knowledge, there is no previous work to exploit controlled natural languages in order to ease the task of defining formal policies. Still, the need of a user-friendly interface to policy specification is perceived in the community and different approaches have been proposed, some of which also exploit natural language.

[78] suggests to use UML[10] sequence diagrams as a notation for policy specification. The authors evaluate to which extent UML is suitable for the specification of policies and especially focus on expressiveness, utility and human readability. Although sequence diagrams are expressive enough to specify policies, the authors recognize that policy specification by means of sequence diagrams has major readability and understandability problems which, if acceptable for engineers, are unlikely to be overcome by non-technicians without strict guidance.

Among other new features, [83] describes the facilities the KAoS policy management framework (cf. Chapter 2) provides for policy authoring. In particular, the Generic Policy Editor provides a policy author with a generic policy statement in the form of a natural language sentence. More specific policies can be asserted by replacing parts of such statement with terms retrieved from available and relevant ontologies adapted to the current context

Although the Generic Policy Editor does exploit natural language, it simply plays the role of a fancier interface: the freedom of the user is actually constrained

[10] `http://www.uml.org/`

to selecting the values of some fields out of a list, i.e., the approach does not differ from a classical user interface provided with a set of combo boxes.

[23] and [52] describe the policy workbench SPARCLE which enables users to enter policies in natural language. The approach pursued by the authors lays somehow in the middle between a "template-based" and a "full natural language" one. On the one hand, policies must comply with one out of two high-level templates, e.g.,

```
[User category] can [Action] [Data Category] for the purpose
of [Purpose] if [Condition] with [Obligation]
```

On the other hand, users are allowed to define the elements the templates consist of (user categories, actions ...) the way they like. For this reason, there is no guarantee that SPARCLE will correctly interpret the input policy (i.e., that the semantics of the policy, as intended by SPARCLE, corresponds to the one the user wanted to express). In our approach, a misinterpretation of the user's input can only happen if she did not stick to the interpretation rules of ACE and DPE (as described in Section 4.2.1).

Somehow related is also [57], which describes a system able to process privacy policies in full natural language. However, the focus of the paper is not on supporting the users with an easy interface to policy specification but on analyzing user policies (describing under which circumstances users are willing to disclose personal data) and enterprise policies (describing how enterprises handle customers' personal data) in order to find compatibilities between them. NLP techniques are exploited in order to convert policies to sets of $(name, value)$ pairs (where both *name* and *value* belong to a predefined vocabulary), which are then matched.

CHAPTER 5
Access Control for Sharing Semantic Data across Desktops

As we noticed in Section 1.2, the expressiveness of the policy languages proposed by the scientific community increased over time. At the beginning of Chapter 4, we already pointed out a consequence of such increment of expressiveness, namely, the reduction of the user-friendliness of policy languages. This chapter is concerned with another consequence, namely, the increment of the computational overhead required to enforce policies of languages with bigger and bigger expressiveness. This problem is especially serious in all application scenarios which require a big amount of requests to be evaluated against complex policies in a short time (as it happens to, e.g., Web Servers).

A possible way to overcome this problem consists in pre-evaluating the policies available in a system against all possible requests and storing the results in a suitable way (e.g., in a database). This way, as soon as a request comes in, the computationally expensive process of evaluating the system's policies against the request does not need to be performed anymore since the request can be accepted or rejected by looking up the pre-computed result.

This chapter thoroughly describes the issues related to the pre-evaluation of policies in the case of a generic policy language and afterwards focuses on the specific issues of PROTUNE, which we used to test the efficacy and efficiency of our approach. The results of our experiments w.r.t. the Semantic Desktop scenario are also reported.

This chapter is organized as follows. Section 5.1 presents a general-purpose strategy to lower the policy enforcement time by exploiting pre-evaluation. The general-purpose strategy does not provide good performance against policy modifications. In order to overcome this problem, some knowledge of the employed policy language is required. For this reason, Section 5.2 presents a special-purpose strategy applicable to policies defined in the PROTUNE policy language. Section 5.3 introduces the concept of "Semantic Desktop" and presents the experimental results of

the application of the general-purpose strategy to the Semantic Desktop scenario. Finally, we discuss related work in Section 5.4.

5.1 The general-purpose strategy

This section describes our general-purpose strategy to lower the policy enforcement time. We first introduce the conceptual framework our strategy applies to and afterwards we describe the strategy itself.

5.1.1 The conceptual framework

This section: (i) elaborates on some general properties of policy languages described in Chapter 2; and (ii) introduces some general properties of policy engines. Such properties are organized according to the following independent dimensions

- expressiveness
- capability of specifying both positive and negative information explicitly
- interaction with the user at evaluation-time

As mentioned at the beginning of Chapter 1, a policy states conditions under which an *action* can be executed. Such conditions can involve: (i) properties of the entity (human or software agent) requesting its execution (*requester*); (ii) properties of the entity on which the action has to be executed (*resource*); (iii) properties of the action itself; or (iv) *environmental* properties.

As we mentioned in Section 2.3.1, in order to provide the language with a well-defined semantics, policy languages are usually based on some mathematical formalism, typically Description Logic or Logic Programming or some suitable subset of either of them. One of the main differences between Description Logic-based (DL-based) and Logic Programming-based (LP-based) policy languages is the way they deal with negation: Description Logic allows to define negative information explicitly, whereas LP-based systems can only deduce negative information by means of the so-called *negation-as-failure* inference rule (cf. Section 3.3).

Since DL-based languages allow to specify both positive and negative information, consistency problems arise whenever both a statement and its negation are

asserted. This problem is typically addressed by allowing the user to define priorities among statements (this is the strategy adopted, e.g., by the designers of KAoS—cf. Chapter 2).

LP-based languages do not have to deal with consistency issues, since they only allow the user to specify positive information. The other side of the coin is that, as soon as new information is added to Logic Programs, consequences which could be drawn previously cannot be inferred anymore (this is commonly expressed by the sentence "negation-as-failure is a non-monotonic inference rule"—cf. Section 2.3.1).

Since DL-based languages allow to specify both positive and negative information, they can enable the user to define both policies stating under which conditions a request must be accepted (*allow* policies) and policies stating under which conditions a request may not be accepted (*deny* policies). Since LP-based languages allow to specify only positive information, they can only enable the user to define either *allow* or *deny* policies. The first approach automatically rejects incoming requests which do not fulfill the given conditions and is usually preferred, since wrongly accepting a request which should have been rejected is typically a more serious issue than rejecting a request which should have been accepted.

Nevertheless, LP-based languages can be exploited as building blocks of higher-level languages which do allow to explicitly specify both positive and negative information by enforcing a negation semantics on top of them. This can be easily obtained as follows: let L be an LP-based language and L' its extension with negation. A policy of L' is a pair $(P, flag)$, where: (i) P is a policy of L; (ii) $flag \in \{allow, deny\}$; and (iii) P is interpreted as an *allow* policy if $flag = allow$ and as a *deny* policy if $flag = deny$. This approach has been exploited in the definition of the LP-based policy language Rei (cf. Chapter 2) and has been also applied to the LP-based policy language PROTUNE in [1].

Many application scenarios do not require the policy owner to take part in the policy evaluation process. This is e.g., the case of a Web Server which delivers (resp. does not deliver) sensitive information according to the clients' properties as specified in the server policy. In this case, it is reasonable assuming that the policy owner (e.g., the human being who deployed the Web application) is never requested

to provide further information to be used during the evaluation of the policy. Other application scenarios do require that the policy evaluation process involves the policy owner directly. This is often the case of firewalls on desktop computers. As soon as a known application tries to access the computer, the request is (not) accepted automatically according to the available policies. As soon as an unknown application tries to access the computer, the policy owner (i.e., the user) is asked whether the request should (not) be accepted.

In the next section we present a strategy to lower the policy enforcement time which applies to policy languages and engines that

- allow to define policies based on properties of requester, action and resource
- enable to define *allow* and *deny* policies
- may require the policy owner to take part in the policy evaluation process

Cela va sans dire, our solution also applies to policy languages and engines which support only a subset of the features listed above.

On the other hand, our solution does not apply to policies which set conditions on environmental properties: it is clear that it does not make sense pre-evaluating a policy which e.g., grants access to a resource from 8am to 5pm since, if only one out of pre-evaluation time and query time belongs to this interval, access will be wrongly granted/denied.

5.1.2 A naïve general-purpose strategy

The UML[1] diagram presented in Fig. 5.1 shows the entities a generic authorization framework is concerned with, namely requesters, resources, actions and policies: all of requesters, resources and actions are identified by some sort of identifier, whereas a policy, beside the text of the policy itself, possesses a flag indicating whether it applies only to the resources (resp. requesters, actions) available at policy creation-time or also to resources possibly added later on.

Policies are distinct in *allow* and *deny* policies and dominance relationships can be defined among them: an *allow* (resp. *deny*) policy can dominate and be

[1] `http://www.uml.org/`

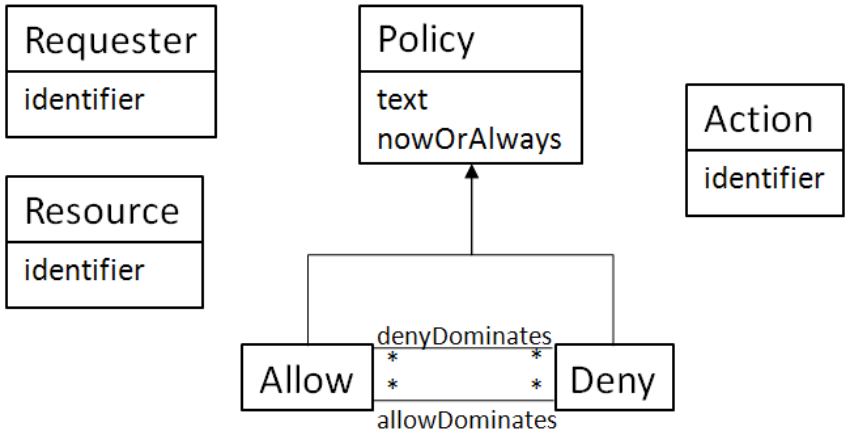


Figure 5.1: A basic UML class diagram of a generic authorization framework

dominated by an arbitrary number of *deny* (resp. *allow*) policies, meaning that in case of conflict the dominating policy will be enforced. In the following we will call such relationships among policies *priority relationships*.

The set of priority relationships induces a graph structure over the set of policies. Special care must be taken in order to avoid that the graph contains loops, since otherwise all policies belonging to a loop would be dominated and none of them could be enforced. We will describe in the following how our strategy avoids that loops are built in the priority relationship graph.

The basic functionality provided by an authorization framework is the capability of (not) authorizing requesters to perform actions on resources. However, real-world authorization frameworks have to provide surrounding facilities for adding and removing requesters, actions, resources and policies. In the following we briefly sketch the internals of such functionalities.

Request evaluation The evaluation of a request for accessing a resource takes place according to the following algorithm. If there is a non-dominated applicable policy P, the request is accepted or rejected, according to whether P is an *allow* or a *deny* policy. Otherwise, the user is asked whether the request should be accepted or rejected. In the latter case, the user's answer is interpreted as the definition of a new policy which dominates every possibly conflicting one. Notice that the addition of such priority relationships

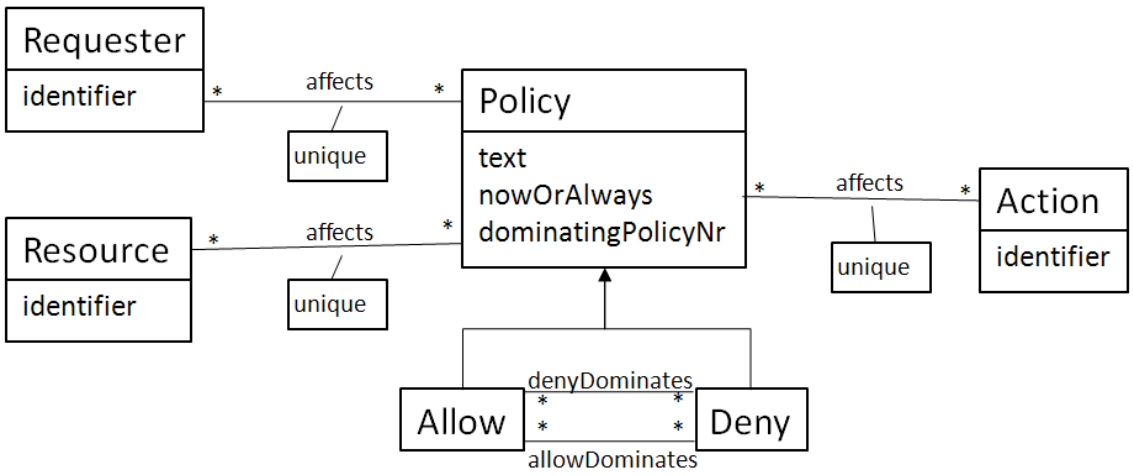


Figure 5.2: The UML class diagram of our authorization framework

can never make the priority relationship graph cyclic, since in such graph the newly added policy does not have incoming arcs

Resource removal The addition of a new resource does not require to perform any further operation, whereas, in order to keep the policy repository as tidy as possible, it makes sense removing all policies affecting one single resource upon removal of that resource. Notice that with our approach it is not infrequent that policies apply to one single resource: as we described above, this is the case whenever the user is asked whether a request should be accepted or rejected. Requester and action addition and removal are handled in a similar way as resource addition and removal

Policy addition Whenever a new policy is added, it is checked whether it conflicts with already defined policies. If this is the case, the user is asked to define priority relationships between the new policy and the conflicting ones. The user is not allowed to define priority relationships which would make the priority relationship graph cyclic

Policy removal Whenever a policy is removed, all relationships it was involved in are removed as well

5.1.3 The general-purpose strategy revisited

When performing the operations listed in Section 5.1.2, some activities must be carried out which are computationally expensive, namely

1. selection of non-dominated policies which apply to a given request (cf. list item 1)
2. identification of policies applicable to one single resource/requester (cf. list item 2)
3. identification of conflicting policies (cf. list item 3)
4. constraining the user to define only priority relationships which keep the priority relationship graph acyclic (cf. list item 3)

The UML diagram presented in Fig. 5.2 is a variant of the one shown in Fig. 5.1 and adds further information which is meant to reduce the computational cost of the activities listed above. A new attribute `dominatingPolicyNr` has been added to policies, indicating how many other policies dominate a given one, and a relationship between policies and resources (resp. requesters, actions) has been added. A policy can affect an arbitrary number of resources (resp. requesters, actions). On the other hand, each resource (resp. requester, action) can be affected by an arbitrary number of policies. These relationships have a `unique` attribute tracking whether the policy affects only that resource (resp. requester, action). In the following we will call relationships between policies and resources (resp. requesters, actions) *influence relationships*.

The addition of redundant information requires some overhead in order to keep such information consistent with the global state of the system. W.r.t. the list presented in Section 5.1.2, the following modifications are needed.

Resource addition Whenever a new resource is added, corresponding relationships with the policies it is affected by are defined. Notice that also relationships with dominated policies are defined, so that they do not need to be computed each time a dominating policy is deleted. Relationships are of

course defined only with policies which apply also to resources added after policy creation-time

Resource removal Whenever a resource is removed, all relationships it was involved in are removed as well

Policy addition Whenever a new policy is added, influence relationships are defined between the new policy and the resources, requesters and actions it affects: if the new policy affects only one single resource (resp. requester, action) the `unique` attribute of the influence relationship is set. Finally, the `dominatingPolicyNr` attribute of the new policy is initialized according to the priority relationships defined and the one of the policies dominated by the new policy is incremented

Policy removal Whenever a policy is removed, all influence relationships it was involved in are removed as well. The `dominatingPolicyNr` attribute of all policies which were dominated by the deleted one is decremented

Against the increase in space needed to store explicitly all information about requesters, actions, resources, policies and the relationships between them, the revisited general-purpose strategy has the potential of dramatically decreasing the time needed by an authorization framework to carry out its activities. For instance, with our strategy the evaluation of a request, which is typically a time-consuming task, simply requires to look up whether a policy exists which applies to the current requester, action and resource and, if this is the case, whether such policy is an *allow* or a *deny* one. Assuming that the information shown in Fig. 5.2 is represented in a reasonable way as a relational database, this task may amount to issuing one single `SELECT` query to the corresponding RDBMS.

However, there is still a task for which our strategy does not provide good performance, namely reacting to policy modifications: against the modification of a policy, all influence relationships involving it have to be recomputed, even if the changes actually involve only few of them. A better approach would be to identify the influence relationships affected by the changes and to update only them. How-

ever, this task essentially depends on the employed policy language, therefore no general solution can be provided.

In the following we describe an efficient strategy to identify the influence relationships affected by PROTUNE policies.

5.2 The special-purpose strategy

This section describes an efficient strategy to identify the influence relationships (cf. Section 5.1.3) affected by PROTUNE policies. This strategy has the potential of dramatically decreasing the time needed to identify the influence relationships affected by modifications of a PROTUNE policy. As described in Section 3.6, PROTUNE is based on Datalog. For this reason, we first introduce our strategy for Datalog programs in Section 5.2.1 and we then extend it to PROTUNE policies in Section 5.2.2. Finally, we only focus on policy modifications consisting in addition and removal of rules.

5.2.1 The Datalog case

Identifying the influence relationships affected by changes to PROTUNE policies is a particular case of identifying the modifications in the extension of a predicate against changes in a Logic Program. This section addresses this more general problem and presents an efficient algorithm which allows to identify a superset of such modifications.

Whenever a PROTUNE policy is modified, all influence relationships it was involved in have to be recomputed. Especially for big sets of requesters, actions and resources, this process can be extremely expensive and time-consuming. A better approach would be to first identify the influence relationships which have to be recomputed (which, at least for small modifications to the PROTUNE policy, are likely to be much fewer than all possible combinations of requesters, actions and resources) and to recompute only those.

However, the identification of each and all influence relationships which have to be recomputed can be an expensive and time-consuming task as well. For this reason, we could relax this requirement and identify a *superset* of the influence

relationships which have to be recomputed. Being the overall time of relationship identification and recomputation to be minimized, an algorithm A_1 identifying more relationships than a slower algorithm A_2 could still be preferable if the time required to recompute the additional relationships is less than the difference of the execution times of A_2 and A_1.

This section introduces an algorithm identifying a superset of the influence relationships to be recomputed and explains the rational behind it. We start by recalling that

1. an atom $p(\vec{x})$ holds according to a Datalog program P iff it matches the head of some rule of P whose body holds. More formally,

$$P \models p(\vec{x}) \Leftrightarrow \exists R \in P, \theta | \theta H_R = p(\vec{x}) \wedge P \models \theta B_R$$

   where θ is a substitution [64] and H_R (resp. B_R) denotes the head (resp. body) of a rule R of P

2. the body B of a rule of a Datalog program P holds iff each of the literals in B holds. More formally,

$$P \models B \Leftrightarrow \exists \theta | \forall l \in B \; P \models \theta l$$

   where θ is a substitution

3. a *ground* negated literal holds iff the atom it exploits does not hold. More formally,

$$P \models\sim p(\vec{x}) \Leftrightarrow \neg P \models p(\vec{x})$$

   where $\neg$ (resp. $\sim$) denotes classical negation (resp. negation-as-failure) and no variables appear in $\vec{x}$

Definition 45 will allow to ease the notation in the following.

Definition 45 (Consequence) *Given a Datalog program P and an atom $p(\vec{x})$, we*

say that $p(\vec{x})$ is a consequence of P by means of a rule $R \in P$ iff

$$\exists\theta | \theta H_R = p(\vec{x}) \wedge P \models \theta B_R$$

where θ is a substitution and H_R (resp. B_R) denotes the head (resp. body) of R.

We will write $P \models_R p(\vec{x})$ to denote the fact that the atom $p(\vec{x})$ is a consequence of a Datalog program P by means of a rule R. We will also write $P \models_{\mathcal{R}} p(\vec{x})$, where $\mathcal{R} \subseteq P$, as a shortcut for $\vee_{R \in \mathcal{R}} P \models_R p(\vec{x})$. Finally, notice that $P \models_P p(\vec{x})$ and $P \models p(\vec{x})$ are equivalent.

We now present the definitions of *extension* and *change functions*. Such definitions, as well as the following ones, rely on the notion of universe $\mathcal{U}$, a generic set which we will assume to be finite and to contain all constants appearing in the given Datalog program(s).

Definition 46 (Extension function) *Given a universe $\mathcal{U}$ and a Datalog program P, the extension function $\mathcal{E}^{\mathcal{U},P}$ associates each predicate p appearing in P with the set*

$$\{\vec{x} \in \mathcal{U}^n | P \models p(\vec{x})\}$$

(extension of p), where n denotes the arity of p.

Intuitively, $\mathcal{E}^{\mathcal{U},P}$ associates each predicate p appearing in P with the set of its ground instances $p(\vec{x})$ which hold in P.

Definition 47 (Change function) *Given a universe $\mathcal{U}$ and two Datalog programs P_1 and P_2, the change function $\mathcal{C}^{\mathcal{U},P_1,P_2}$ associates each predicate p appearing in P_1 or P_2 with the set*

$$\{\vec{x} \in \mathcal{U}^n | P_1 \models p(\vec{x}) \dot{\vee} P_2 \models p(\vec{x})\}$$

(change set of p), where n (resp. $\dot{\vee}$) denotes the arity of p (resp. the XOR boolean operator).

Intuitively, $\mathcal{C}^{\mathcal{U},P_1,P_2}$ associates each predicate p appearing in P_1 or P_2 with the set of its ground instances $p(\vec{x})$ which hold either in P_1 or in P_2 but not in both of them.

The following theorem directly derives from the definitions of extension and change functions.

Theorem 3 *Given a universe $\mathcal{U}$ and two Datalog programs P_1 and P_2, for each predicate p appearing in P_1 or P_2*

$$\mathcal{C}^{\mathcal{U},P_1,P_2}(p) = \mathcal{E}^{\mathcal{U},P_1}(p) \triangle \mathcal{E}^{\mathcal{U},P_2}(p)$$

where $\triangle$ denotes the symmetric difference operator.

Unfortunately, Theorem 3 cannot be exploited to compute in an efficient way the change set of predicates after modifications to a Datalog program, since it would require to compute the extension of such predicates before and after the modifications took place and to compare the results with each other.

The remaining of this section is devoted to the description of the *modified change function* $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$. Its definition (cf. below) is pretty similar to the one of *change function*, the main difference being the condition the n-uples $\vec{x}$ have to fulfill in order to belong to the image of a predicate. Since such condition has been relaxed, for each predicate p it holds that $\mathcal{C}^{\mathcal{U},P_1,P_2}(p) \subseteq \hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}(p)$. On the other hand, the relaxation has been carried out in a way such that the computation of $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ is typically less expensive and time-consuming than the one of $\mathcal{C}^{\mathcal{U},P_1,P_2}$.

Tab. 5.1 shows the logical steps we performed to modify the change function into the modified change function. In order to improve readability, we assume throughout the table that conjunctions bind stronger than disjunctions.

Cell 1 is a paraphrase of the condition in Definition 47 which takes into account the logical equivalence $A \dot{\vee} B \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$ and the equivalence presented in Item 1 of the previous list. Only the first part of the paraphrase is shown, since the second part can be obtained from the first one by swapping all occurrences of 1 and 2.

Cell 2 does not differ from the previous one but for a further condition (namely, $R_1^1 \in P_2 \vee R_1^1 \notin P_2$) which has been added to the first element of the conjunction. Being this further condition always true, the addition does not modify the semantics of the whole expression.

1	$(\exists R_1^1 \in P_1, \theta_1^1 \vert \theta_1^1 H_{R_1^1} = p(\vec{x}) \wedge P_1 \models \theta_1^1 B_{R_1^1}) \wedge \neg(\exists R_2^1 \in P_2, \theta_2^1 \vert \theta_2^1 H_{R_2^1} = p(\vec{x}) \wedge P_2 \models \theta_2^1 B_{R_2^1}) \vee$ $\ldots$
2	$(\exists R_1^1 \in P_1, \theta_1^1 \vert \theta_1^1 H_{R_1^1} = p(\vec{x}) \wedge P_1 \models \theta_1^1 B_{R_1^1} \wedge (R_1^1 \in P_2 \vee R_1^1 \notin P_2)) \wedge \neg(\exists R_2^1 \in P_2, \theta_2^1 \vert \theta_2^1 H_{R_2^1} = p(\vec{x}) \wedge P_2 \models \theta_2^1 B_{R_2^1}) \vee$ $\ldots$
3	$(\exists R_1^1 \in P_1 \cap P_2, \theta_1^1 \vert \theta_1^1 H_{R_1^1} = p(\vec{x}) \wedge P_1 \models \theta_1^1 B_{R_1^1}) \wedge \neg(\exists R_2^1 \in P_2, \theta_2^1 \vert \theta_2^1 H_{R_2^1} = p(\vec{x}) \wedge P_2 \models \theta_2^1 B_{R_2^1}) \vee$ $(\exists R_3^1 \in P_1 \backslash P_2, \theta_3^1 \vert \theta_3^1 H_{R_3^1} = p(\vec{x}) \wedge P_1 \models \theta_3^1 B_{R_3^1}) \wedge \neg(\exists R_4^1 \in P_2, \theta_4^1 \vert \theta_4^1 H_{R_4^1} = p(\vec{x}) \wedge P_2 \models \theta_4^1 B_{R_4^1}) \vee$ $\ldots$
4	$(\exists R_1^1 \in P_1 \cap P_2, \theta_1^1 \vert \theta_1^1 H_{R_1^1} = p(\vec{x}) \wedge P_1 \models \theta_1^1 B_{R_1^1} \wedge \neg P_2 \models \theta_1^1 B_{R_1^1}) \vee$ $(\exists R_3^1 \in P_1 \backslash P_2, \theta_3^1 \vert \theta_3^1 H_{R_3^1} = p(\vec{x}) \wedge P_1 \models \theta_3^1 B_{R_3^1}) \wedge \neg(\exists R_4^1 \in P_2, \theta_4^1 \vert \theta_4^1 H_{R_4^1} = p(\vec{x}) \wedge P_2 \models \theta_4^1 B_{R_4^1}) \vee$ $(\exists R_2^2 \in P_2 \cap P_1, \theta_2^2 \vert \theta_2^2 H_{R_2^2} = p(\vec{x}) \wedge P_2 \models \theta_2^2 B_{R_2^2} \wedge \neg P_1 \models \theta_2^2 B_{R_2^2}) \vee$ $(\exists R_4^2 \in P_2 \backslash P_1, \theta_4^2 \vert \theta_4^2 H_{R_4^2} = p(\vec{x}) \wedge P_2 \models \theta_4^2 B_{R_4^2}) \wedge \neg(\exists R_3^2 \in P_1, \theta_3^2 \vert \theta_3^2 H_{R_3^2} = p(\vec{x}) \wedge P_1 \models \theta_3^2 B_{R_3^2}) \vee$
5	$(\exists R \in P_1 \cap P_2, \theta \vert \theta H_R = p(\vec{x}) \wedge (P_1 \models \theta B_R \dot{\vee} P_2 \models \theta B_R)) \vee$ $(P_1 \models_{P_1 \backslash P_2} p(\vec{x}) \wedge \neg P_2 \models p(\vec{x})) \vee$ $(P_2 \models_{P_2 \backslash P_1} p(\vec{x}) \wedge \neg P_1 \models p(\vec{x}))$

Table 5.1: Relaxation of condition $P_1 \models p(\vec{x}) \dot{\vee} P_2 \models p(\vec{x})$

Cell 3 further expands Cell 2 by applying the distributivity of conjunction over disjunction and by recalling that $x \in A \wedge x \in B \Rightarrow x \in A \cap B$ and $x \in A \wedge x \notin B \Rightarrow x \in A \backslash B$, where A and B are generic sets and $\backslash$ denotes the set difference operator. Finally, notice that the second part of the formula can be obtained from the first one by swapping all occurrences of 1 and 2, and of 3 and 4.

Cell 4 presents the first relaxation we introduced: it involves the first line, has been deduced from the first line of Cell 3 and can be explained intuitively as follows. If there are no rules nor substitutions which allow to infer $p(\vec{x})$ from P_2, then not even the rule and substitution which allow to infer $p(\vec{x})$ from P_1 will allow to infer it from P_2. The condition in the first line of Cell 4 is less strong than the one in the first line of Cell 3, since the latter requires that no rule of P_2 allows to infer $p(\vec{x})$, whereas the former simply requires that a particular rule of P_2 does not allow to infer it. As a consequence, all n-uples $\vec{x}$ satisfying the latter condition will satisfy the former as well, i.e., the set of n-uples satisfying the latter condition is a subset of the set of n-uples satisfying the former one.

Cell 5 is a rewriting of Cell 4: the first line of Cell 5 summarizes lines 1 and 3 of Cell 4 by exploiting the logical equivalence $\exists x|A(x) \vee \exists y|B(y) \equiv \exists x|A(x) \vee B(x)$ as well as the one involving the $\dot{\vee}$ operator and the distributivity of conjunction over disjunction we recalled above. Line 2 (resp. 3) of Cell 5 summarizes line 2 (resp. 4) of Cell 4 by exploiting the formalism introduced in Definition 45.

Tab. 5.2 shows the logical steps we performed to relax the condition $P_1 \models \theta B_R \dot{\vee} P_2 \models \theta B_R$ which appears in the first line of the fifth cell of Tab. 5.1. Cell 1 is a paraphrase of such condition which takes into account the well-known definition of the $\dot{\vee}$ operator and the equivalence introduced in Item 2 of the previous list. Only the first part of the paraphrase is shown, since the second part can be obtained from the first one by swapping all occurrences of 1 and 2.

Cell 2 presents the second relaxation we introduced, which does not conceptually differ from the one we presented in Cell 4 of Tab. 5.1: if there are no substitutions which allow to infer every literal of B_R from P_2, then not even the substitution which allows to infer every literal of B_R from P_1 will allow to infer them from P_2.

Cell 3 does not differ from the previous one but for the fact that the logical

1	$(\exists\theta_1^1 \vert \forall l_1^1 \in B_R\ P_1 \models \theta_1^1\theta l_1^1) \wedge \neg(\exists\theta_2^1 \vert \forall l_2^1 \in B_R\ P_2 \models \theta_2^1\theta l_2^1)\vee$ $\ldots$
2	$(\exists\theta_1^1 \vert \forall l_1^1 \in B_R\ P_1 \models \theta_1^1\theta l_1^1 \wedge \neg\forall l_2^1 \in B_R\ P_2 \models \theta_1^1\theta l_2^1)\vee$ $\ldots$
3	$(\exists\theta_1^1, l_2^1 \in B_R \vert \forall l_1^1 \in B_R\ P_1 \models \theta_1^1\theta l_1^1 \wedge \neg P_2 \models \theta_1^1\theta l_2^1)\vee$ $\ldots$
4	$(\exists\theta_1, l_1 \in B_R \vert P_1 \models \theta_1\theta l_1 \wedge \neg P_2 \models \theta_1\theta l_1)\vee$ $(\exists\theta_2, l_2 \in B_R \vert P_2 \models \theta_2\theta l_2 \wedge \neg P_1 \models \theta_2\theta l_2)$
5a	$\exists\theta_1, p(\vec{x}) \in B_R \vert P_1 \models \theta_1\theta p(\vec{x}) \wedge \neg P_2 \models \theta_1\theta p(\vec{x})\vee$ $P_2 \models \theta_1\theta p(\vec{x}) \wedge \neg P_1 \models \theta_1\theta p(\vec{x})$
5b	$\exists\theta_1, \sim p(\vec{x}) \in B_R \vert P_1 \models\sim \theta_1\theta p(\vec{x}) \wedge \neg P_2 \models\sim \theta_1\theta p(\vec{x})$ $P_2 \models\sim \theta_1\theta p(\vec{x}) \wedge \neg P_1 \models\sim \theta_1\theta p(\vec{x})$
6	$\exists\theta_1, p(\vec{x}) \in B_R \vert P_1 \models \theta_1\theta p(\vec{x}) \dot{\vee} P_2 \models \theta_1\theta p(\vec{x})$
7	$\exists p(\vec{x}) \in B_R, \theta_1 \vert \theta_1\theta\vec{x} \in \mathcal{C}^{\mathcal{U},P_1,P_2}(p)$

Table 5.2: Relaxation of condition $P_1 \models \theta B_R \dot{\vee} P_2 \models \theta B_R$

equivalence $\neg\forall x\ A(x) \equiv \exists x \vert \neg A(x)$ has been applied and the resulting existential quantification has been moved to the beginning.

Cell 4 presents the third relaxation we introduced, which can be explained intuitively as follows. If there is a substitution which allows to infer every literal of B_R from P_1, then such substitution allows to infer from P_1 the literal which cannot be inferred from P_2. The condition in the first line of Cell 4 is less strong than the one in the first line of Cell 3, since the latter requires that every literal of B_R can be inferred from P_1, whereas the former simply requires that a particular literal can be inferred.

Cells 5a and 5b are rewritings of Cell 4: they summarize the two lines of Cell 4 by exploiting the logical equivalence involving existential quantification we recalled above. Line 5a (resp. 5b) assumes that the existentially quantified literal has the form $p(\vec{x})$ (resp. $\sim p(\vec{x})$). Notice that, by applying the equivalence presented in Item 3 of the previous list as well as the involutive property of classical negation, the condition presented in line 5b does not differ from the one presented in line 5a, which we can hence assume to represent the general case as long as we interpret the fragment $p(\vec{x}) \in B_R$ as denoting an atom (and not a literal anymore) appearing in the body of a rule R, possibly within a negated literal.

Finally, Cell 6 is a rewriting of Cell 5a by applying the well-known definition of

the $\dot{\vee}$ operator and Cell 7 is a rewriting of Cell 6 by applying backwards Definition 47.

We are now ready to introduce the formal definition of *modified change function.*

Definition 48 (Modified change function) *Given a universe $\mathcal{U}$ and two Datalog programs P_1 and P_2, the modified change function $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ associates each predicate p appearing in P_1 or P_2 with the set of $\vec{x} \in \mathcal{U}^n$ (modified change set) which satisfy either of the following conditions*

- $\exists R \in P_1 \cap P_2, p_1(\vec{x_1}) \in B_R, \theta, \theta_1 | \theta H_R = p(\vec{x}) \wedge \theta_1 \theta \vec{x_1} \in \hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}(p_1)$
- $P_1 \models_{P_1 \setminus P_2} p(\vec{x}) \wedge \neg P_2 \models p(\vec{x})$
- $P_2 \models_{P_2 \setminus P_1} p(\vec{x}) \wedge \neg P_1 \models p(\vec{x})$

where n denotes the arity of p.

The first condition in Definition 48 has been obtained by plugging the Cell 7 of Tab. 5.2 into the first line of Cell 5 of Tab. 5.1 and by moving the existential quantifications to the beginning. The resulting expression has been further relaxed by replacing the set $\mathcal{C}^{\mathcal{U},P_1,P_2}(p)$ with a superset of it, namey $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}(p)$ itself. Such replacement makes Definition 48 recursive and this property will be exploited by the algorithm presented in Fig. 5.3.

Intuitively, $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ associates each predicate p appearing in P_1 or P_2 with the set of its ground instances $p(\vec{x})$ which

- either are a consequence of P_1 by means of a rule not belonging to P_2, but not of P_2
- or are a consequence of P_2 by means of a rule not belonging to P_1, but not of P_1
- or match the head of some rule: (i) belonging to both P_1 and P_2; and (ii) such that some (ground) instance of some predicate p_1 in its body belongs to the set $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ associates p with

Input:
    a universe $\mathcal{U}$
    two Datalog programs P_1 and P_2
Output:
    a map linking predicates with sets of their ground instances

$mcf(\mathcal{U}, P_1, P_2)$:
    p, $p_1 \equiv$ predicates
    $\vec{x} \equiv$ an n-upla of terms
    m_1, $m_2 \equiv$ maps linking predicates with sets of their ground instances
    $R \equiv$ a rule
    θ, $\theta_1 \equiv$ substitutions

(1) $\forall p$ appearing in P_1 or P_2
(2) $\forall \vec{x} \in \mathcal{U}^{arity(p)}$
(3) **if(**
(4) $P_1 \models_{P_1 \setminus P_2} p(\vec{x}) \wedge \neg P_2 \models p(\vec{x}) \vee$
(5) $P_2 \models_{P_2 \setminus P_1} p(\vec{x}) \wedge \neg P_1 \models p(\vec{x})$
(6) **)** $add(m_2, p, \vec{x})$

(7) **do{**
(8) $m_1 ::= m_2$
(9) $\forall p$ appearing in P_1 or P_2
(10) $\forall R \in P_1 \cap P_2$ such that $p(\vec{x})$ appears in H_R
(11) $\forall \theta$ such that $\theta p(\vec{x})$ is ground
(12) **if(**B_R is empty**)**
(13) $add(m_2, p_1, \theta\vec{x})$
(14) **else** $\forall p_1$ such that $p_1(\vec{x_1})$ appears in B_R
(15) **if(**$\exists \theta_1 | isIn(m_1, p_1, \theta_1\theta\vec{x_1})$**)**
(16) $add(m_2, p_1, \theta\vec{x})$
(17)**}** **while(**$m_1 \neq m_2$**)**

(18)**return** m_2

Figure 5.3: Algorithm to compute the *modified change function*

Definition 48 can be directly converted into an algorithm which computes the set $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ associates a generic predicate p with. Such algorithm (shown in Fig. 5.3) relies on three auxiliary functions.

arity Returns the arity of the input predicate

add Adds the input n-upla to the set the input map links the input predicate with

isIn Returns *true* if the input n-upla belongs to the set the input map links the input predicate with, *false* otherwise

Lines (1-6) of the algorithm initialize map m_2 by computing for each predicate appearing in P_1 or P_2 the subset of its extension which fulfills either of the last two conditions listed in Definition 48 and shown in lines (4-5).

The loop shown in lines (7-17) updates m_2 (after keeping a backup copy in m_1) by computing for each predicate appearing in P_1 or P_2 its ground instances which fulfill the first condition listed in Definition 48. The loop ends as soon as m_2 does not differ from m_1, i.e., if m_2 has not been modified during the last iteration: in this case m_2 is returned.

We notice that a loop is needed because the definition of *modified change function* is recursive: the computation of the output of $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ for a given predicate p requires that its output has been already computed for all predicates appearing in the body of rules in whose head p appears. On the other hand, as long as the computation is not over yet, in general only a subset of the output of $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ for a given predicate is available. For this reason, our algorithm computes the output of $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ for a given predicate p assuming that the output of the predicates p depends on has been completely computed, but then it checks this assumption by computing the output of $\hat{\mathcal{C}}^{\mathcal{U},P_1,P_2}$ more and more times and only stops if a fixed point is reached.

We notice that a fixed point is always reached since, for each predicate p, the set m_2 links p with cannot decrease and, in the worst case, it will be equal to $\mathcal{U}^n$ (where n denotes the arity of p), which is a finite set, since we assumed $\mathcal{U}$ to be finite.

We conclude this section by providing some qualitative remarks about the efficiency of the algorithm shown in Fig. 5.3. The performance of such algorithm

greatly varies according to the characteristics of the input programs. Beside the trivial observation that the algorithm performs better: (i) the fewer predicates are exploited in the input programs (line 9); (ii) the fewer rules are shared by them (line 10); and (iii) the smaller the universe is (line 11); it might be worth mentioning that rules whose body is not empty have a bigger impact on the performance than facts, since they require two further inner loops (lines 14-15) to be evaluated.

Lines 1-6 constitute the main difference between our algorithm and the naïve one which can be derived from Theorem 3: our algorithm does not require to compute the extension of predicates w.r.t. the whole input programs but only w.r.t. the rules which are not shared by them. For this reason, the improvement of our algorithm over the naïve one is especially visible whenever the input programs share most rules, like in our application scenario, where the input programs represent the policy in force before and after some modifications took place (with the reasonable assumption that the modifications only involved a small part of the program).

Finally, if either of the input programs uses recursion, the loop shown in lines (7-17) must be performed more than twice. The following example shows that such loop might have to be performed up to $\#\mathcal{U} + 1$ times (where $\#A$ denotes the cardinality of a set A). However, we do not believe that such cases often occurs in practice.

(1) a(X) $\leftarrow$ b(X, Y), a(Y).
(2) b(1, 0).
...
(n) b($n-1$, $n-2$).

($n+1$) a(0).

Let assume that $\mathcal{U}$ is the set of the first n natural numbers (where $n \geq 1$), P_1 is the Datalog program consisting of the first n rules shown above, whereas P_2 consists of all $n+1$ rules. Up to line (6) of the algorithm, m_2 links the predicate a with the set $\{0\}$. At the end of the i-th iteration of the loop (where $1 \leq i \leq n$), the natural number i has to be added to the set m_2 links a with. Since such set has been

modified, a new iteration is needed. After the natural number n has been added to the set, the $(n+1)$-th iteration does not add anything to such set and the loop can be exited.

5.2.2 The Protune case

The algorithm presented in Fig. 5.3 can be easily adapted to PROTUNE policies in order to retrieve the atoms of the form `allow(execute(requester, action, resource))` which have to be re-evaluated against modifications to the policy.

The biggest differences between Datalog and PROTUNE which are relevant in this respect are: (i) PROTUNE is a policy language and not a language for data retrieval; (ii) the evaluation of a PROTUNE literal might require to perform actions; and (iii) PROTUNE supports value-assignment atoms (cf. Section 3.2). In the remaining of this section we discuss whether and how each of these differences requires changes to the algorithm presented in Fig. 5.3.

As described in Section 3.3, the process of evaluating a PROTUNE query involves up to two steps: checking whether the query can be evaluated and, if this is the case, actually evaluating it. This chapter describes how to pre-evaluate triples (rq, a, rs), where rq is a requester allowed to perform the action a on the resource rs. For this reason, we are only concerned with the first one of the two steps we just mentioned, for which the single-step algorithm presented in Fig. 5.3 can still be exploited.

As we just argued, the algorithm presented in Fig. 5.3 can be used to identify the modified change set of PROTUNE *logical* predicates appearing in a policy (cf. Section 3.2). However, there is no standard way to identify the modified change set of PROTUNE *provisional* predicates, since its computation strongly depends on the actions associated to the provisional predicates. At a general level, (at least) two approaches are feasible.

- Assuming that the extension of provisional predicates has changed and recomputing it. This approach would require to modify the algorithm presented in Fig. 5.3 by `OR`-ing a further condition to the ones listed in lines (4-5) stating that p is a provisional predicate. The drawback of this approach is that, in

case the assumption does not hold (i.e., if the extension of some provisional predicate did not change), the modified change set of the other predicates can excessively overestimate their change set

- Requiring that the policy only exploits provisional predicates whose extension does not change. This approach would require to modify the algorithm presented in Fig. 5.3 by adding to line (9) the further constraint that p must be a logical predicate. The drawback of this approach is that it is applicable to a lower number of scenarios

Notice that, since in PROTUNE environmental properties are retrieved by means of provisional predicates, the second approach further constrains the restriction introduced in Section 5.1.1, namely the exclusion from our scenario of policies which set conditions on environmental properties.

Finally, the notation PROTUNE uses to represent value-assignment atoms is nothing but syntactic sugar: each atom of the form id`[`$attr$`] = `val occurring in a PROTUNE policy could be systematically replaced by the Datalog-compatible atom p`(`id`, `$attr$`, `val`)` (where p is the name of a predicate not occurring in the policy) without modifying PROTUNE's semantics nor reducing its overall functionalities and expressive power.

5.3 Experimental results

This section introduces the concept of "Semantic Desktop" and presents the experimental results of the application of our general-purpose strategy (cf. Section 5.1) to the Semantic Desktop scenario.

Personal Information Management (PIM) systems support users in organizing and managing their e-mails, address books, calendars and other information on their desktop. Systems like Google Desktop[2], Beagle[3], Haystack[4] and Gnowsis [71] define semantic data as any content available on the user's personal information space. Their primary goals are to provide convenient access to such semantic data and

[2] `http://desktop.google.com/`
[3] `http://beagle-project.org/`
[4] `http://haystack.lcs.mit.edu/`

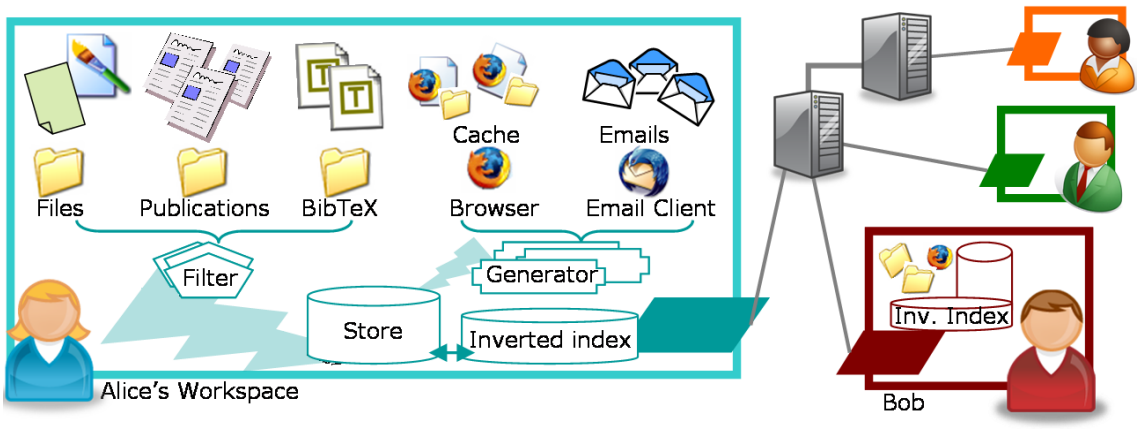


Figure 5.4: Architecture of the Semantic Desktop Beagle^{++} system and semantic data sharing across desktops

to enable sharing of both desktop resources and metadata describing them across PIMs.

Fig. 5.4 illustrates a Semantic Desktop as well as semantic data sharing across desktops. Each user of the system possesses a *Semantic Desktop* corresponding to her personal information space. Each desktop contains a set of resources the user creates, modifies or deletes as well as a set of applications along with their internal resources. A set of *filters* and *generators* process these resources and generate metadata to describe their content. For example, metadata for publications include title and authors, metadata for e-mails include sender, receiver(s), subject and date. Generated metadata are automatically stored in a database and used to maintain an inverted index which maps keywords to actual resources in order to allow for full-text query search.

In order to evaluate our general-purpose strategy against the Semantic Desktop scenario, we simulated the evolution of a user's desktop. At the beginning of the simulation, the user has no resources (and therefore no policies). As resources are added, requests for accessing them are issued. As described in Section 5.1, whenever no policies have been defined for a requested resource, the user is asked to accept or reject the request. The user's feedback is simulated by a random generator of binary values. Resources keep being added until 1.000 policies have been defined, i.e., a number which we consider to be much bigger than the one of current real-world

```
CREATE TABLE POLICY(
  ID INT,
  TYPE BOOLEAN,
  NOW_OR_ALWAYS BOOLEAN,
  DOMINATING_POLICY_NR INT
)
CREATE TABLE DOMINATES(
  DOMINATING INT,
  DOMINATED INT
)
CREATE TABLE AFFECT_RESOURCE(
  POLICY INT,
  RESOURCE INT,
  _UNIQUE BOOLEAN
)
```

Figure 5.5: Schema of the database used in our evaluation

scenarios. At that time, we go the way back, i.e., we remove policies and resources until neither the formers nor the latters are available in the system anymore.

Fig. 5.5 shows the schema of the relational database used in our evaluation as an implementation of (a subset of) the UML diagram shown in Fig. 5.2. Table `POLICY` contains information about the policies defined by the user, namely, their identifier (field `ID`) and type (*allow* or *deny*—field `TYPE`), whether they apply only to the resources available at policy creation-time or also to resources possibly added later on (field `NOW_OR_ALWAYS`) and the number of other policies dominating them (field `DOMINATING_POLICY_NR`). Notice that in our implementation the text of the policies is not contained itself in the database but in a file the `ID` field points to.

Table `DOMINATES` stores the priority relationships (cf. Section 5.1.2) between policies, i.e., it is a set of pairs of policy identifiers whose first element (field `DOMINATING`) dominates the second one (field `DOMINATED`). Finally, table `AFFECT_RESOURCE` stores the influence relationships (cf. Section 5.1.3) between policies and resources, i.e., it is a set of triples containing the identifiers of a resource (field `RESOURCE`) and of a policy affecting it (field `POLICY`) as well as a flag (field `_UNIQUE`) tracking whether the policy affects only that resource.

Notice that no tables storing influence relationships between policies and re-

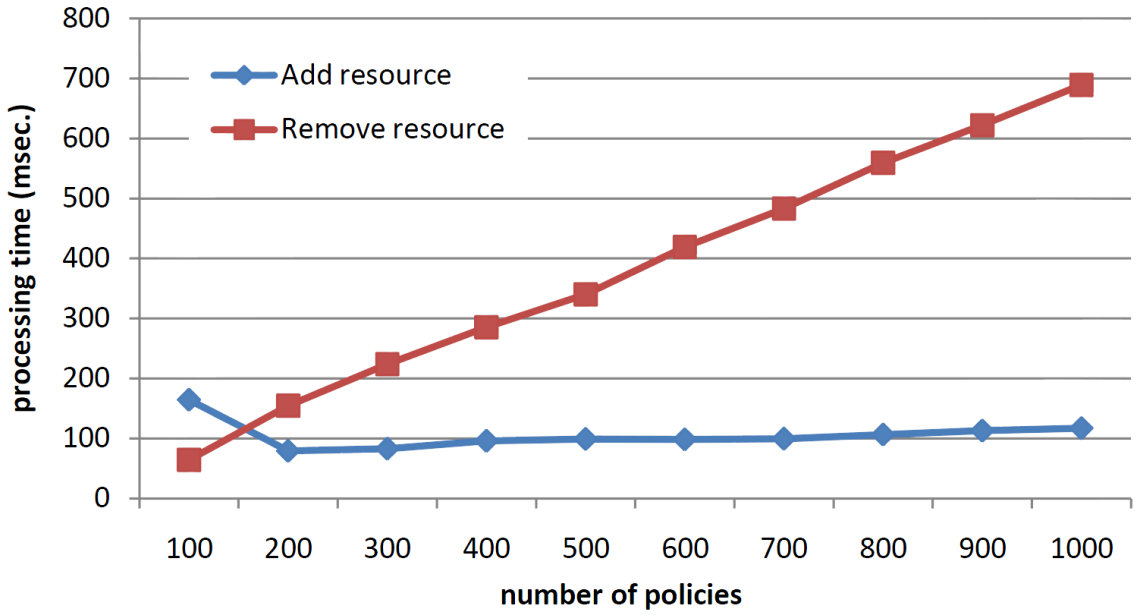


Figure 5.6: Overhead of adding/removing resources

questers/actions are available. Because of symmetry reasons, in our evaluation we restricted ourselves to only considering influence relationships between policies and resources.

Fig. 5.6 shows the overhead of exploiting pre-evaluation when adding/removing resources. As described in Section 5.1, the overhead of adding a resource amounts to defining influence relationships with the policies it is affected by (i.e., in our implementation, to inserting rows into table `AFFECT_RESOURCES`), whereas the overhead of removing a resource amounts to

- removing all policies affecting only that resource (i.e., in our implementation, deleting rows from table `POLICY`)
- deleting all (influence) relationships it was involved in (i.e., in our implementation, deleting rows from table `AFFECT_RESOURCES`)

The higher number of operations needed when removing an existing resource w.r.t. adding a new one explains why in Fig. 5.6 the gradient of the *Remove resource*-curve is higher than the one of the *Add resource*-curve.

Fig. 5.7 shows the overhead of exploiting pre-evaluation when adding/removing policies. As described in Section 5.1, the overhead of adding a policy amounts to

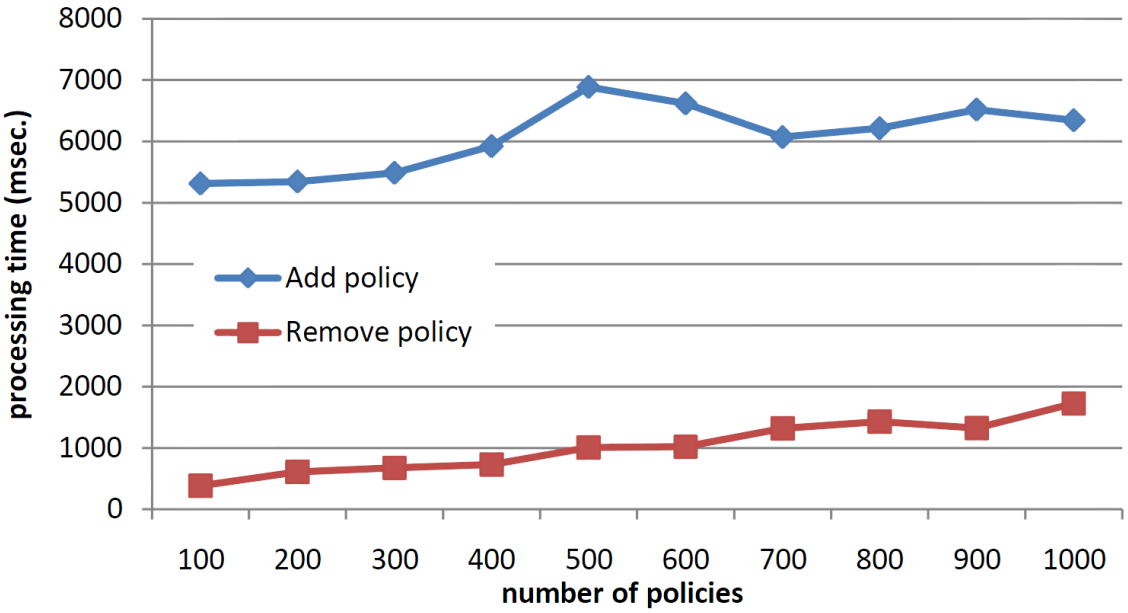


Figure 5.7: Overhead of adding/removing policies

- checking whether it conflicts with already defined policies
- incrementing the `dominatingPolicyNr` attribute of the policies dominated by the new one (i.e., in our implementation, updating rows of table `POLICY`)
- defining influence relationships between the new policy and the resources it affects (i.e., in our implementation, inserting rows into table `AFFECT_RESOURCES`)

whereas the overhead of removing a policy amounts to

- decrementing the `dominatingPolicyNr` attribute of all policies which were dominated by the deleted one (i.e., in our implementation, updating rows of table `POLICY`)
- deleting all influence and priority relationships it was involved in (i.e., in our implementation, deleting rows from tables `AFFECT_RESOURCES` and `DOMINATES`)

Again, the higher number of operations needed when adding a new policy w.r.t. removing an existing one explains why in Fig. 5.7 the *Add policy*-curve presents higher values than the ones of the *Remove policy*-curve.

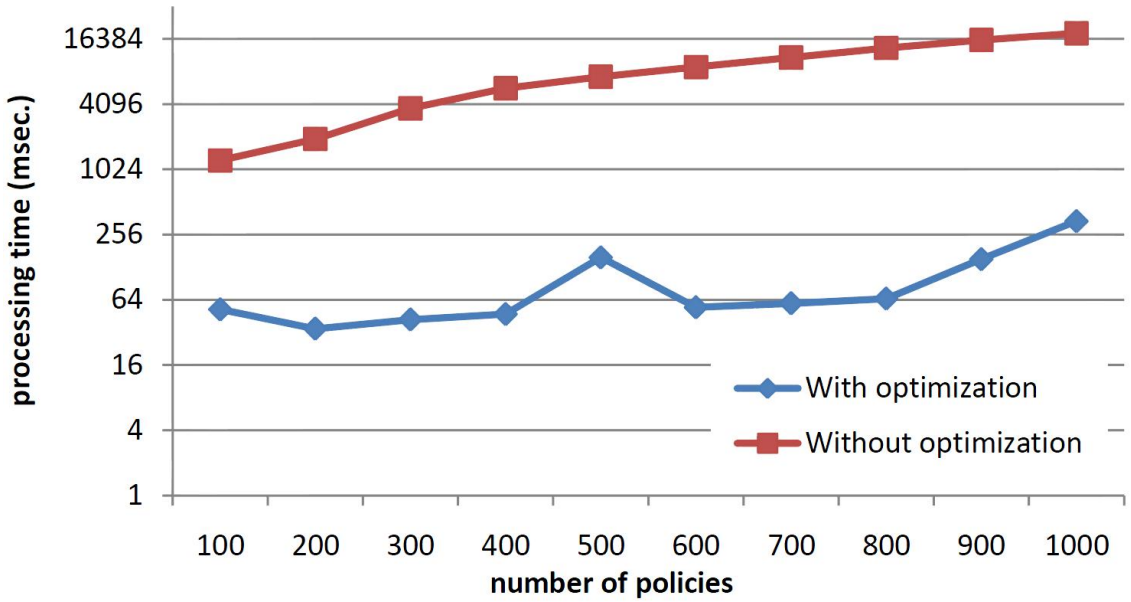


Figure 5.8: Optimized vs. non-optimized policy evaluation

Finally, Fig. 5.8 shows the improvement of exploiting pre-evaluation when evaluating requests. The evaluation of a request by exploiting pre-evaluation amounts to issuing a query to the database whose schema is shown in Fig. 5.5 in order to retrieve the type of the policy affecting the requested resource. If such a policy does not exist then no policies affecting such resource have been defined yet and the user is asked whether the request should be accepted or rejected.

The evaluation of a request without exploiting pre-evaluation is more complex. Since no information is explicitly available about which policy affects which resource, one has to try out all policies defined in the system until: either (i) a non-dominated policy is found which applies to the requested resource; or (ii) it turns out that such a policy does not exist. Notice that the second case, which applies to requests for resources for which no policies have been defined yet, requires to inspect each and all of the policies defined in the system and is hence more and more computationally expensive as the number of policies increases (as it is reflected in Fig. 5.8).

When going through all policies defined in the system, it could be a good idea starting with the non-dominated ones, going down the dominance relationship graph and stopping as soon as the condition above is satisfied. This way, as soon as an applicable policy is found, it is guaranteed that such policy is not dominated by

Input:
    a resource r
Output:
    either of $ACCEPT$, $REJECT$ or $UNDEFINED$

$evaluate(r)$:
    $P \equiv$ a set of policies

    $P = getNonDominatedPolicies()$
    while($P \neq \emptyset$)
        $\forall p \in P$
            if($appliesTo(p, r)$)
                if($isAllowPolicy(p)$)
                    return $ACCEPT$
                else return $REJECT$
        $P = getDominatedPolicies(P)$
    return $UNDEFINED$

Figure 5.9: Algorithm to find the policy applicable to a given resource

any other one w.r.t. the requested resource. This algorithm, which we exploited in our experiments, is formalized by the function *evaluate* in Fig. 5.9. Such function evaluates the request to access a resource r w.r.t. the policies defined in the system and relies on four auxiliary functions.

getNonDominatedPolicies Returns the set of non-dominated policies defined in the system

appliesTo Returns *true* if the input policy p applies to the input resource r, *false* otherwise

isAllowPolicy Returns *true* if the input policy p is an *allow* policy, *false* otherwise

getDominatedPolicies Returns the set of policies defined in the system which are dominated by some policy in the input policy set P

Function *evaluate* iterates over the set of policies defined in the system going down the dominance relationship graph. As soon as an applicable policy is found, the

request is accepted or rejected according to whether the policy is an *allow* or a *deny* one. If no applicable policy is found, $UNDEFINED$ is returned.

To conclude, we notice that the more complex the policies defined in the system are, the more computationally expensive their non-optimized evaluation is and hence the bigger the improvement of exploiting pre-evaluation is. In order not to introduce biases in our experiments, we only exploited the easiest policies one can define with the PROTUNE policy language. Such policies simply state that a resource can(not) be accessed and do not require any condition to be fulfilled. Here is an example how such a(n *allow*) policy looks like

$$allow(access(resourceId)).$$

where $resourceId$ is the identifier of the resource to be accessed.

Because of this design choice, the improvement of the optimized request evaluation over the non-optimized one shown in Fig. 5.8 is underestimated w.r.t. real-world scenarios. Still, our experiments show that our approach is beneficial whenever the ratio between the number of evaluations and the number of other operations is at least 4.5 (for up to 100 policies defined in the system) or 1 (for more than 300 policies). We consider these numbers to be reasonable in most real-world scenarios.

5.4 Related work

In the last years, systems for collaborative work and file sharing gained increasing popularity. The need for effective search in this context despite the increasing amount of information pushed forward further developments of search infrastructures for enterprise data management systems [45]. However, the sometimes private nature of such shared information makes difficult applying traditional document indexing schemes directly: user access levels and access control have to be reflected in the index structures and retrieval algorithms as well as when ranking the search results. The shortcomings of traditional ranking algorithms for search through access-controlled collections is outlined in [26].

In the literature, several solutions have been proposed addressing the problem

of preserving the privacy of data stored on public remote servers, which typically provide a basis for community platforms. For example, cryptographic techniques enable users to store encrypted text files on a remote server and retrieve them by means of keyword search [28, 44, 79].

However, these solutions are not suitable for collaborative multi-user environments. Alternatively, the data shared within a community can be stored locally by the user within an access-controlled collection. In this case, efficient retrieval algorithms to search through access-controlled collections must be provided in order to enable information sharing within the community.

The authors of [11] address the problem of providing privacy-preserving search over distributed access-controlled content. Although this technique enables random provider selection, it does not allow to rank search results obtained from different document collections. On the other hand, our semantically enriched community platform provides a unified view on the whole information set available to the user.

CHAPTER 6
Enabling Advanced and Context-Dependent Access Control in RDF Stores

The Semantic Web vision [76] requires that existing data are provided with machine-understandable annotations. These annotations (commonly referred to as *metadata*) are meant to ease tasks such as data sharing and integration. These metadata are typically represented in RDF/XML [14, 22] or other machine-understandable formats, such as microformats [55] or RDFa [3], which can be conveniently translated into RDF, e.g., by custom GRDDL [66] transformations.

However, it is often the case that unconditional sharing of metadata is undesirable: many Semantic Web applications require to control when, what and to whom metadata are disclosed. Nevertheless, existing RDF stores and standard protocols to access them, such as the SPARQL protocol [29], do not support access control or their support is minimal (e.g., protection only applies to the repository as a whole but not to the data it contains).

A possible solution to this problem would be to embed access control within the RDF store. However, in this case the access-control mechanism would be repository-dependent and not portable across different platforms. Moreover, "hard-wired" protection mechanisms have proved to work well in the context of relational database management systems (RDBMS), where the granularity of access control nicely scales down to the table level. However, such mechanisms do not apply to RDF repositories, since there is no schema underlying the data they store.

A more general solution is to add a new component responsible for access control-related issues on top of the RDF store. However, the problems such component would have to face are not trivial since, e.g., it cannot limit itself to filtering triples which must be kept private out of the results of a query. This depends on the fact that those triples might not be known in advance, as it happens when the outcome of the query consists of triples not previously available in the RDF store but created on-the-fly, e.g., by means of a SPARQL [69] `CONSTRUCT` statement.

In this chapter we describe how policy languages can be exploited to integrate advanced access-control mechanisms based on policies with arbitrary RDF data stores. Since the naïve approach of evaluating the available policies for each triple returned against a query would not scale as soon as the result set exceeds a certain size, our solution requires to rewrite input RDF queries in order to embed into them the constraints set by applicable policies. The modified queries are then sent to the RDF store which executes them like usual RDF queries. Notice that our framework enables access control at triple level, i.e., all triples returned as a response to the query are allowed to be disclosed to the requester according to the policies in force.

The remainder of this chapter is organized as follows: Section 6.1 accounts for alternative approaches and related work. In Section 6.2 we describe how a policy engine can be integrated on top of a generic RDF store in order to enforce access control at triple level. Finally, Section 6.3 presents our current implementation: a set of experiments which estimate the impact of our approach in terms of performance is included as well.

6.1 Related work

In order to fine-grainedly control access to specific RDF statements, two main approaches have been proposed: (i) filtering the set of RDF statements returned by a query as described, e.g., in [33]; and (ii) defining *a priori* a set of RDF statements that are allowed to be accessed by the requester as proposed, e.g., in [37].

As for restricting access to RDF data, filtering query results in a separate post-processing step after query execution as proposed in [33] is not an adequate solution: current RDF query languages allow to arbitrarily structure the results, as shown in the following example[1].

```
CONSTRUCT {CC} newNS:isOwnedBy {User}
  FROM {User} ex:hasCreditCard {CC};
              foaf:name {Name}
  WHERE Name = 'Alice'
```

[1]Our examples use SeRQL [24] syntax (for the sake of simplicity we do not include namespace definitions). However, the ideas behind our solution are language-independent and can be applied to other RDF query languages, such as SPARQL.

Here, filtering the query results is not straightforward, since the result structure is not known in advance. In fact, not the results produced by the query but rather the data accessed in the `FROM` clause should be restricted. Moreover, the approach suggested in [33] is not feasible for large result sets (e.g., suppose an unauthorized requester submits a query asking for all available triples in the store), since it requires to first retrieve all triples and only at that point to filter out the ones which cannot be disclosed.

A different way to address this problem is to define *a priori* which subsets of an RDF database can be accessed by some requester. This approach is pursued in [27] which shows how Named Graphs can be used to evaluate SPARQL queries. Dietzold et Auer [37] propose a framework which first applies all access-control policies to the whole RDF database and afterwards executes the query only on the subset of it containing RDF triples which can be disclosed.

A priori solutions are only applicable to scenarios where access restrictions depend on the data to be accessed. However, in Semantic Web scenarios, where different services might want to access RDF statements, access to data could have to be additionally restricted according to externally checked, contextual conditions. Therefore, pre-computing Named Graphs for each possible combination of environmental factors cannot help, since in the worst case it would lead to a number of graphs to be generated which is exponential in the number of environmental factors. Runtime creation of Named Graphs is infeasible too, since the creation process would excessively slow down the response time.

Reddivari et al. [70] suggest to exploit simple rule-based policies over RDF stores: such policies define graph patterns identifying subgraphs of the database on which actions like `read` and `update` can be executed. The drawback of this approach is performance against query evaluation, since the process of answering a query requires to instantiate the graph patterns, i.e., to generate one graph for each policy and to execute the given query on each graph.

Our approach ensures that access policies are satisfied by rewriting the queries that are sent to an RDF store. Thus, there is no need to instantiate graphs before query execution. Furthermore, our approach enables to exploit RDF Schema entail-

ments similarly as done by the *a priori* approach described in [47]. For instance, if an access-control policy states that no statements about `foaf:OnlineAccount`s should be disclosed, then statements about `foaf:OnlineEcommerceAccount`s would not be disclosed either, assuming that the latter is a subclass of the former.

Finally, none of the policy languages presented in Chapter 2 has been already integrated into RDF databases.

6.2 Policy-based query expansion

As argued in Section 6.1, existing work on RDF data protection does not suit the requirements of dynamic environments. Many available solutions are only applicable to scenarios where access restrictions depend on the data to be accessed but not on environmental factors, i.e., to scenarios where access-control policies include *data-dependent* conditions but no *context-dependent* conditions.

To address this limitation, we decided to enforce an access-control layer on top of RDF stores, which also has the positive side-effect of making our solution store-independent. Our strategy consists of two steps: first, context-dependent conditions stated by the applicable policies are identified and pre-evaluated. If the pre-evaluation succeeds, the query is then modified in order to embed data-dependent conditions, so that only allowed RDF statements are accessed by the underlying RDF store at query-processing time.

By supporting both data- and context-dependent conditions, our solution allows for more expressive policies than the ones supported by the approaches presented in Section 6.1, while at the same time relying on the highly optimized query evaluation of the RDF store for the enforcement of data-dependent conditions.

6.2.1 RDF queries

The following definitions use a similar notation as in [68]: they rely on the sets I, B, L and Var, which are supposed to be infinite and mutually disjoint, and to denote sets of IRIs, blank nodes, literals and variables respectively.

Definition 49 (RDF graph) *An RDF graph is a finite subset of* $(I \cup B) \times I \times (I \cup B \cup L)$.

Definition 50 (Triple pattern) *A triple pattern is a triple of the form* (s, p, o)*, where* $s \in I \cup B \cup Var$*,* $p \in I \cup Var$ *and* $o \in I \cup B \cup L \cup Var$*.*

Definition 51 (Path Expression) *A path expression is a set of triple patterns.*

In the following, we will use $vars(e)$ to denote the set of all variables occurring in a triple pattern or path expression e.

Intuitively, path expressions are templates formed by triple patterns where variables are allowed in any position, and are meant to model conjunctive queries to be matched against RDF graphs.

In the following, we will mean by *atomic constraint* a property of an n-upla of elements of $I \cup B \cup L \cup Var$ (where $n \geq 1$). A property of a pair (e_1, e_2) may be the fact that $e_1 > e_2$. A property of a triple (e_1, e_2, e_3) may be the fact that e_3 is the string concatenation of e_1 and e_2.

The expressiveness of constraints heavily depends on the query language actually exploited. In order to keep the discussion as general as possible, we do not stick to any specific existing language and hence do not further discuss the expressiveness constraints may have.

Atomic constraints can be combined by means of boolean operators in order to form *boolean expressions*.

Definition 52 (Query) *A query is either a pair* (RF, PE) *or a triple* (RF, PE, BE) *where*

- PE *is a path expression (query pattern)*
- RF *is*
  - *either a set of variables* $\subseteq vars(PE)$
  - *or a path expression such that* $vars(RF) \subseteq vars(PE)$

  (result form)
- BE *is a set of boolean expressions*

```
(0) CONSTRUCT * FROM
(1) {Person} foaf:name {Name};
(2)          foaf:phone {Phone};
(3)          foaf:interest {Document};
(4)          foaf:holdsAccount {Account}
```

Figure 6.1: Example RDF query

In the following, we will use $vars(e)$ to denote the set of all variables occurring in a result form, query pattern or boolean expression e.

Intuitively, our definition of "query" is meant to model RDF queries having the following structure (cf. Section 6.19 in [4])[2], where the `WHERE` section can possibly be omitted.

SELECT RF/CONSTRUCT RF
FROM PE
WHERE BE

In `SELECT` queries RF is a set of variables, whereas in `CONSTRUCT` queries it is a path expression. An example query is provided in Fig. 6.1, where the token `*` is exploited, denoting either $vars(PE)$ (in `SELECT` queries) or PE (in `CONSTRUCT` queries). The boolean expressions in BE are supposed to be `AND`-ed. If no access-control policy were defined, this query would return an RDF graph containing all RDF triples matching the graph pattern defined in the `FROM` block, i.e., the query answer would include identifier and name of a person, her phone number(s) and the document(s) she is interested in as well as the account(s) she holds.

The following definition is taken from [64].

Definition 53 (Substitution) *A substitution is a finite set of the form* $\{v_1/t_1, \ldots v_n/t_n\}$*, where* $n \geq 0$ *and* $\forall 1 \leq i, j \leq n$ $v_i \in Var$, $t_i \in I \cup B \cup L \cup Var$, $v_i \neq t_i$ *and* $i \neq j \Rightarrow v_i \neq v_j$.

[2]We focus on common `read` operations which all RDF query languages like SeRQL or SPARQL support. Data manipulation operations, such as `insert` or `delete`, are supported by some extensions such as SPARUL [74] but are not part of any standard yet.

Definition 54 (*disunify* function) *Given a triple pattern $e = (s, p, o)$ and a substitution θ, the function $disunify(e, \theta)$ returns the pair (e', BE), where e' is a triple pattern (s', p, o') and BE is a set of boolean expressions such that*

- $\begin{cases} s' = v_s \text{ and } BE_s = \{v_s = s\} & \text{if } s \notin Var \\ s' = v_s \text{ and } BE_s = \{v_s = X\} & \text{if } s \in Var, s/X \in \theta \\ s' = s \text{ and } BE_s = \emptyset & \text{otherwise} \end{cases}$
- $\begin{cases} o' = v_o \text{ and } BE_o = \{v_o = o\} & \text{if } o \notin Var \\ o' = v_o \text{ and } BE_o = \{v_o = X\} & \text{if } o \in Var, o/X \in \theta \\ o' = o \text{ and } BE_o = \emptyset & \text{otherwise} \end{cases}$

where

- $v_s, v_o \notin vars(e)$ *nor appear they in* θ
- $v_s \neq v_o$
- $BE = BE_s \cup BE_o$

Example 31 *In this example some applications of function disunify are provided.*

- $disunify((a, b, c), \emptyset) = ((X, b, Y), \{X = a, Y = c\})$
- $disunify((X, b, Y), \emptyset) = ((X, b, Y), \emptyset)$
- $disunify((X_1, b, Y_1), \{X_1/X_2, Y_1/Y_2\}) = ((X_3, b, Y_3), \{X_3 = X_2, Y_3 = Y_2\})$

Intuitively, subject and object of the triple pattern are replaced by variables, whereas the boolean expressions keep track of their original values.

6.2.2 Specifying policies over RDF data

In order to restrict access to RDF statements, a policy language must allow to specify graph patterns (i.e., path expressions and boolean expressions), such as one can do in an RDF query. This implies that the policy language must allow to explicitly define variables and to set constraints on the values they can assume. This

requirement by itself excludes policy languages based on Description Logic [9] (e.g., KAoS—cf. Chapter 2), since Description Logic does not allow to explicitly define variables. For this reason, in the following we will focus on the competing track of policy languages which are based on the Logic Programming [64] formalism.

Policies of Logic Programming-based policy languages rely on the notion of *rule*. We consider a policy rule to have the following form

(1) ALLOW/DENY ACCESS TO PE IF
(2) $\quad CP_1$ AND ... CP_l AND
(3) $\quad PE_1$ AND ... PE_m AND
(4) $\quad BE_1$ AND ... BE_n

where $l, m, n \geq 0$, PE and PE_i $(1 \leq i \leq m)$ are triple patterns, CP_j $(1 \leq j \leq l)$ are context-dependent conditions and BE_k $(1 \leq k \leq n)$ are boolean expressions.

We call *head* of the rule the action shown in line (1) (i.e., `ALLOW/DENY ACCESS TO` PE) and *body* of the rule the set of conditions appearing in lines (2-4). In the following, we will use $H(pol)$ (resp. $H_{PE}(pol)$) to denote the head (resp. triple pattern in the head) of a policy rule *pol* and $B(pol)$ to denote the (possibly empty) body of *pol*.

Notice that our policies are expressed in a high-level syntax. This way, we allow them to the LP-based policy languages described in Chapter 2. On the other hand, it is true that the policy language actually chosen will impact the expressiveness of the policies which can be specified, in particular: (i) the set of supported atomic constraints to be exploited in the boolean expressions BE_i; and (ii) the set of supported context-dependent conditions. Finally, notice that, in order for our solution to be applicable to a given query language and a given policy language, the set of atomic constraints supported by the former must be a subset of the one supported by the latter.

Suppose that Alice specified the policies presented in Tab. 6.1, whose intended meaning is as follows.

1. Everyone can access Alice's phone number(s)

N°	Policy
(i)	ALLOW ACCESS TO (#alice, foaf:phone, Z)
(ii)	DENY ACCESS TO (X, foaf:phone, Z) IF (X, foaf:currentProject, #rewerse) AND the requester is #recSer
(iii)	ALLOW ACCESS TO (X, foaf:phone, Z) IF the requester is certified by #bbb AND (#alice, foaf:knows, X)
(iv)	ALLOW ACCESS TO (X, Y, Z) IF the current time is Time AND 09:00 < Time AND Time < 17:00 AND Y = foaf:name AND X != #tom
(v)	ALLOW ACCESS TO (#alice, foaf:interest, Z) IF (Z, rdf:type, foaf:Document) AND (#alice, foaf:currentProject, P) AND (Z, foaf:topic, T) AND (P, foaf:theme, T)
(vi)	ALLOW ACCESS TO (#alice, foaf:holdsAccount, X) IF the requester is Y AND (#alice, foaf:knows, Y)
(vii)	ALLOW ACCESS TO (X, Y, Z) IF (X, rdf:type, foaf:Person) AND the requester sends a credential C AND the issuer of C is X

Table 6.1: Example of high-level policies controlling access to RDF statements

2. The Recommender Service is not allowed to access the phone number(s) of members of project REWERSE

3. Recognized trusted services (which have to provide a credential issued by the Better Business Bureau—BBB) are allowed to access the phone number(s) of people Alice knows

4. RDF statements containing the name of entities different from Alice's boss

Tom can be accessed during work time

5. This policy controls the access to Alice's interests. Only documents related to her current project(s) can be accessed

6. A service can access Alice's on-line eCommerce account only if the service was invoked by a person known to Alice

7. A service can access information about a person if it can provide a credential issued by this person

Context-dependent conditions are expressed in natural language (e.g., *the requester is 'RecommenderService'*), triple patterns according to the triple notation (e.g., $(\#alice, foaf : knows, X)$) and boolean expressions exploit the usual (in)equality operators (e.g., $09 : 00 < Time$).

In order to evaluate policies and to handle conflicts which arise whenever two (different) policies allow and deny access to the same resource, we use a simple policy-evaluation strategy, outlined by the following algorithm, which provides a the *deny by default* strategy.

if a *deny* policy is applicable
    then access to the triple(s) is denied
else if an *allow* policy is applicable
    then access to the triple(s) is allowed
else access to the triple(s) is denied

More advanced algorithms exploiting priorities or default precedences among policies (like in [48]) could be used as well.

6.2.3 Policy evaluation and query expansion

Given an RDF query, each RDF statement matching a triple pattern specified in the FROM block is accessed and possibly returned. Our approach analyzes the set of RDF statements to be accessed and restricts it according to the policies in force. Context-dependent conditions are evaluated by the policy engine, which keeps track of the results in its internal state Σ (cf. [19] – in other words, Σ determines at

each instant the extension of the context-dependent predicates). On the other hand, data-dependent constraints are added to the given query and hence automatically enforced during query processing.

To illustrate the algorithm step by step, we consider the query defined in Fig. 6.1.

Definition 55 (Policy applicability) *Given a triple pattern e, a set of policies P and a time-dependent state Σ, we say that a policy $pol \in P$ is applicable to e according to Σ and P iff $H_{PE}(pol)$ and e are unifiable and there exists a variable substitution σ'' such that*

- $\sigma' = mgu(e, H_{PE}(pol))$, *where mgu denotes the most general unifier [64]*
- $\sigma = \sigma'\sigma''$
- $\forall cp \in B(pol)\ P \cup \Sigma \models \sigma cp$
- $\forall be \in B(pol)$ *such that*
  - $vars(\sigma be) \cap vars(\sigma e) = \emptyset$ *and*
  - $\forall pe \in B(pol)\ vars(\sigma be) \cap vars(\sigma pe) = \emptyset$

  it holds that $P \cup \Sigma \models \sigma be$

and the result of its application to e is a pair (PE, BE) such that

- $PE = \{pe' | pe \in B(pol), disunify(pe, \sigma) = (pe', \cdot)\}$
- $BE_0 = \bigcup_{i \in \{BE | pe \in B(pol), disunify(pe,\sigma) = (\cdot, BE)\}} i$
- $BE_1 = \{\sigma be | be \in B(pol) \wedge \exists pe \in B(pol) : vars(\sigma be) \cap (vars(\sigma pe) \cup vars(\sigma e)) \neq \emptyset\}$
- $BE = BE_0 \cup BE_1 \cup \{X = Y | X/Y \in \sigma \wedge X \in vars(e) \wedge Y \in Const\}$

In the following, we will use $isAppl_{P,\Sigma}(pol, e)$ to refer to the fact that a policy *pol* belonging to a set of policies P is applicable to a triple pattern e according to a state Σ and $appl_{P,\Sigma}(pol, e)$ to refer to the result of such application.

Intuitively, a policy *pol* is applicable to a triple pattern e if the triple the policy is protecting unifies with e and; (i) all context-dependent predicates; and (ii) all boolean expressions which do not share variables with e nor with any of the triple patterns in the body of *pol*; hold according to the available set of policies and state. The result of the application is a pair whose first element is a modified version of the set of triple patterns available in the body of *pol* and whose second element is a set of boolean expressions comprising

- the boolean expressions created when disunifying the triple patterns available in the body of *pol*
- a modified version of the boolean expressions available in the body of *pol* which share variables with e or with some triple pattern in the body of *pol*
- the ground bindings available in the substitution

Example 32 *Tab. 6.2 shows all pairs* (pol, e) *(where pol is applicable to* e*) as well as the result of the application of pol to* e*. The first column of the table shows the policy number* (N_p) *as it has been introduced in Tab. 6.1, whereas the second column shows the number of the triple pattern* (N_t) *as it has been introduced in Fig. 6.1. The further columns exploit the terminology introduced in Definition 55. When computing the applicability of a policy to a triple pattern we made the following assumptions concerning the evaluation of context-dependent predicates.*

- *In policies (ii) and (vi) the requester is #recSer*
- *In policy (iii) the requester is certified by #bbb*
- *In policy (iv) the current time is 15:00*
- *In policy (vii) the requester sent a credential #cred issued by #bbb*

Before we describe the query expansion algorithm we specify the conditions under which a query does not need to be evaluated since the result is empty.

N_p	N_t	σ'	σ''	PE	BE_0	BE_1	$BE \backslash BE_0 \backslash BE_1$
i	2	$\{Person/\#alice, Z/Phone\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{Person = \#alice\}$
ii	2	$\{X/Person, Z/Phone\}$		$\{(X_1, foaf:currentProject, X_2)\}$	$\{X_1 = Person, X_2 = \#rewerse\}$		$\emptyset$
iii	2			$\{(X_3, foaf:knows, X_4)\}$	$\{X_3 = \#alice\}$ $X_4 = Person\}$		
iv	1	$\{X/Person, Y/foaf:name, Z/Name\}$	$\{Time/15:00\}$	$\emptyset$	$\emptyset$	$\{Person \neq \#tom\}$	
v	3	$\{Person/\#alice, Z/Document\}$	$\emptyset$	$\{(X_5, rdf:type, X_6), (X_7, foaf:currentProject, P), (X_8, foaf:topic, T), (P, foaf:theme, T)\}$	$\{X_5 = Document, X_6 = foaf:Document, X_7 = \#alice, X_8 = Document\}$	$\emptyset$	$\{Person = \#alice\}$
vi	4	$\{Person/\#alice, X/Account\}$	$\{Y/\#recSer\}$	$\{(X_9, foaf:knows, X_{10})\}$	$\{X_9 = \#alice, X_{10} = \#recSer\}$		
vii	1	$\{X/Person, Y/foaf:name, Z/Name\}$	$\{C/\#cred, X/\#bbb\}$	$\{(X_{11}, rdf:type, X_{12})\}$	$\{X_{11} = \#bbb, X_{12} = foaf:Person\}$		$\emptyset$
vii	2	$\{X/Person, Y/foaf:phone, Z/Phone\}$		$\{(X_{13}, rdf:type, X_{14})\}$	$\{X_{13} = \#bbb, X_{14} = foaf:Person\}$		
vii	3	$\{X/Person, Y/foaf:interest, Z/Document\}$		$\{(X_{15}, rdf:type, X_{16})\}$	$\{X_{15} = \#bbb, X_{16} = foaf:Person\}$		
vii	4	$\{X/Person, Y/foaf:holdsAccount, Z/Account\}$		$\{(X_{17}, rdf:type, X_{18})\}$	$\{X_{17} = \#bbb, X_{18} = foaf:Person\}$		

Table 6.2: Application of policies to triple patterns

Definition 56 (Query failure) *Given a query $q = (RF, PE, BE)$, a set of policies P and a state Σ, we say that q fails if either of the following two conditions hold.*

- $\exists e \in PE \nexists pol \in P | pol$ *is an allow policy* $\wedge isAppl_{P,\Sigma}(pol, e)$
- $\exists e \in PE, pol \in P | pol$ *is a deny policy* $\wedge isAppl_{P,\Sigma}(pol, e) \wedge appl_{P,\Sigma}(pol, e) = (\emptyset, \emptyset)$

Intuitively, a query fails if there does not exist any triple to be returned according to both the query and the applicable policies, i.e., if the query contains at least one triple pattern for which no matching triples are allowed to be accessed (deny by default) or for which all matching triples are not allowed to be accessed (explicit deny).

The query expansion algorithm is defined as follows.

Input:

Two sets of triple patterns PE_1 and PE_2

A set of boolean expressions BE'

Output:

$PE \equiv$ a set of triple patterns

$BE \equiv$ a set of boolean expressions

$removeDuplicates(PE_1, PE_2, BE')$:

$\theta \equiv$ a substitution

(1) $PE = \theta = \emptyset$
(2) $\forall pe_2 \in PE_2$
(3) if $\exists pe_1 \in PE_1 | pe_2$ and pe_1 are unifiable
(4) $\theta = \theta mgu(pe_2, pe_1)$
(5) else $PE \cup = \{pe_2\}$
(6) $BE = \{\theta be' | be' \in BE'\}$

Input:

a query $q = (RF, PE, BE)$

a set of policies P

a state Σ

Output:

$PE^{+}_{new} \equiv$ a set of triple patterns (from *allow* policies)

$PE^{-}_{new} \equiv$ a set of triple patterns (from *deny* policies)

$BE^{+}_{new} \equiv$ a set of boolean expressions (from *allow* policies)

$BE^{-}_{new} \equiv$ a set of boolean expressions (from *deny* policies)

$policyFiltering(q, P, \Sigma)$:

$BE^{+}_{or} \equiv$ a set of boolean expressions (from *allow* policies)

$BE^{-}_{or} \equiv$ a set of boolean expressions (from *deny* policies)

$P_{app} \equiv$ a set of policies

(1) $PE^{+}_{new} = PE^{-}_{new} = BE^{+}_{new} = BE^{-}_{new} = \emptyset$

(2) $\forall e \in PE$

(3) $BE^{+}_{or} = BE^{-}_{or} = \emptyset$

// check *allow* policies

(4) $P_{app} = \{pol \in P | pol \text{ is an } allow \text{ policy} \wedge isAppl_{P,\Sigma}(pol, e)\}$

(5) if $P_{app} = \emptyset$

// no triples matching e can be accessed

return query failure

(6) $\forall pol \in P_{app}$

(7) $appl_{P,\Sigma}(pol, e) = (PE', BE')$

(8) $removeDuplicates(PE^{+}_{new}, PE', BE') = (PE'', BE'')$

(9) $PE^{+}_{new} \cup = PE''$

(10) $BE^{+}_{or} \cup = \{\wedge_{be \in BE''} be\}$

(11) $BE^{+}_{new} \cup = \{\vee_{be \in BE^{+}_{or}} be\}$

// check *deny* policies

(12) $P_{app} = \{pol \in P | pol \text{ is a } deny \text{ policy} \wedge isAppl_{P,\Sigma}(pol, e)\}$

(13) if $\exists pol \in P_{app} : appl_{P,\Sigma}(pol, e) = (\emptyset, \emptyset)$

// all triples matching e cannot be accessed

return query failure

(14) $\forall pol \in P_{app}$

(15) $appl_{P,\Sigma}(pol, e) = (PE', BE')$

(16) $removeDuplicates(PE^{-}_{new}, PE', BE') = (PE'', BE'')$

(17) $PE^{-}_{new} \cup = PE''$

(18) $BE^{-}_{or} \cup = \{\wedge_{be \in BE''} be\}$

(19) $BE^{-}_{new} \cup = \{\vee_{be \in BE^{-}_{or}} be\}$

Intuitively, function *removeDuplicates*: (i) merges two sets of triple patterns so that triple patterns which are equal beside variable renaming are not taken into account; and (ii) modifies a set of boolean expressions in order to take into account the merging performed during the previous step. First, the output parameter PE as well as the local variable θ are initialized to the empty set (line 1). Afterwards, for each triple pattern pe_2 belonging to the input parameter PE_2 (line 2) it is checked whether pe_2 already belongs to the input parameter PE_1 beside variable renaming. To this goal, it is checked whether PE_1 contains a triple pattern pe_1 which is unifiable with pe_2 (line 3). If this is the case, θ is updated by composing it with the most general unifier between pe_2 and pe_1 (line 4), otherwise pe_2 is added to PE_1 (line 5). Finally, the output parameter BE is set to be equal to the input parameter BE', beside the fact that the substitution θ is applied to all boolean expressions in BE' (line 6).

Intuitively, for each triple pattern available in the `FROM` clause of a query, function *policyFiltering* retrieves the policies applicable to it. The triple patterns resulting from the application (cf. Definition 55) of the policies to the triples are collected. The boolean expressions resulting from the application of a policy to a triple are `AND`-ed, whereas the `AND`-expressions resulting from the application of all (applicable) policies to a triple are `OR`-ed and such `OR`-expressions are collected. During the whole process, *allow* and *deny* policies are kept separate.

First, the output parameters PE^{+}_{new}, PE^{-}_{new}, BE^{+}_{new} and BE^{-}_{new} are initialized to the empty set (line 1). Afterwards, for each triple pattern e available in the `FROM` clause of the input query (line 2) the local variables BE^{+}_{or} and BE^{-}_{or} are initialized to the empty set (line 3) and *allow* (line 4) and *deny* (line 12) policies applicable to e are subsequently inspected. According to Definition 56, if no applicable *allow* policies are found (line 5) or some *deny* policy applies without any restriction (line 13) the query fails. Otherwise, for each applicable policy (lines 6, 14) functions *appl* (cf. Definition 55—lines 7, 15) and *removeDuplicates* (lines 8, 16) are subsequently invoked in order to retrieve a set of triple patterns PE'' and a set of boolean expressions BE''. The triple patterns belonging to PE'' are added to PE^{+}_{new} (line 9) and PE^{-}_{new} (line 17), whereas the boolean expressions belonging to BE'' are `AND`-ed

and the result expression is added to BE^{+}_{or} (line 10) and BE^{-}_{or} (line 18). After all applicable policies have been checked, the boolean expressions belonging to BE^{+}_{or} (resp. BE^{-}_{or}) are `OR`-ed and the result expression is added to BE^{+}_{new} (line 10) (resp. BE^{-}_{new}—line 18).

Example 33 *Tab. 6.3 exemplifies the application of function policyFiltering to the query shown in Fig. 6.1 and the policies shown in Tab. 6.1. We made the following assumptions concerning the evaluation of context-dependent predicates.*

- *The requester is #recSer (therefore policies (ii) and (vi) apply to the query)*
- *The requester is certified by #bbb (therefore policy (iii) applies to the query)*
- *The current time is 18:00 (therefore policy (iv) does not apply to the query)*
- *The requester sent a credential #cred issued by #bbb (therefore policy (vii) applies to the query)*

Because of these assumptions, all policies but (iv) are applicable to the query. The first two columns of Tab. 6.3 do not differ from the corresponding ones of Tab. 6.2, whereas the further columns show the values returned by function removeDuplicates (PE and BE) as well as the last value of its local variable θ upon subsequent calls within function policyFiltering, assuming that both the triple patterns in the `WHERE` *clause of the query and the policies in Tab. 6.1 are processed in a top-down fashion. Finally, notice that allow (resp. deny) policies are shown in the upper (resp. lower) part of Tab. 6.3.*

Tab. 6.3 allows to retrieve the output parameters of function policyFiltering: $PE^{-}_{new} = \{(X_1, foaf : currentProject, X_2)\}$ and $BE^{-}_{new} = \{X_1 = Person \wedge X_2 = \#rewerse\}$. PE^{+}_{new} is the union of the sets appearing in column PE (beside the last line), whereas BE^{+}_{new} can be computed as follows.

1. *All equalities appearing in each cell of column BE must be* `AND`*-ed*
2. *The conjunctions built during the previous step which refer to the same triple pattern must be* `OR`*-ed*

N_t	N_p	PE	BE	θ
1	vii	$\{(X_{11}, rdf:type, X_{12})\}$	$\{X_{11} = \#bbb, X_{12} = foaf:Person\}$	$\emptyset$
2	i	$\emptyset$	$\{Person = \#alice\}$	
2	iii	$\{(X_3, foaf:knows, X_4)\}$	$\{X_3 = \#alice, X_4 = Person\}$	
2	vii	$\emptyset$	$\{X_{11} = \#bbb, X_{12} = foaf:Person\}$	$\{X_{13}/X_{11}, X_{14}/X_{12}\}$
3	v	$\{(X_7, foaf:currentProject, P),$ $(X_8, foaf:topic, T), (P, foaf:thema, T)\}$	$\{X_{11} = Document, X_{12} = foaf:Document,$ $X_7 = \#alice, X_8 = Document, Person = \#alice\}$	$\{X_5/X_{11}, X_6/X_{12}\}$
3	vii	$\emptyset$	$\{X_{11} = \#bbb, X_{12} = foaf:Person\}$	$\{X_{15}/X_{11}, X_{16}/X_{12}\}$
4	vi		$\{X_3 = \#alice, X_4 = \#recSer, Person = \#alice\}$	$\{X_9/X_3, X_{10}/X_4\}$
4	vii		$\{X_{11} = \#bbb, X_{12} = foaf:Person\}$	$\{X_{17}/X_{11}, X_{18}/X_{12}\}$
2	ii	$\{(X_1, foaf:currentProject, X_2)\}$	$\{X_1 = Person, X_2 = \#rewerse\}$	$\emptyset$

Table 6.3: Application of function *policyFiltering* to the query in Fig. 6.1 and the policies in Tab. 6.1

3. BE^+_{new} *is the set of the disjunctions built during the previous step*

Definition 57 (Expanded query) *An expanded query is a pair* $((RF^+, (PE^+, PE^+_O), BE^+), (RF^-, (PE^-, PE^-_O), BE^-))$ *where*

- (RF^+, PE^+, BE^+) *and* (RF^-, PE^-, BE^-) *are (usual) queries*
- PE^+_O *and* PE^-_O *are path expressions*

Intuitively, an expanded query is meant to model RDF queries having the following structure.

CONSTRUCT RF^+
FROM PE^+ [PE^+_O]
WHERE BE^+
MINUS
CONSTRUCT RF^-
FROM PE^- [PE^-_O]
WHERE BE^-

where

- `MINUS` denotes the set difference operator: the result set of the query above is the set of results which are returned by the first subquery but not by the second one
- `[` and `]` denote the optional path expression modifier: path expressions contained within the brackets do not have to be matched to find query results

Input:
a query $q = (RF, PE, BE)$
a set of policies P
a state Σ
Output:
an expanded query $q = (RF^+, (PE^+, PE^+_O), BE^+), (RF^-, (PE^-, PE^-_O), BE^-))$

$expandQuery(q, P, \Sigma)$

1) $policyFiltering(q, P, \Sigma) = (PE^+_{new}, PE^-_{new}, BE^+_{new}, BE^-_{new})$
2) $RF^+ = RF^- = RF$
3) $PE^+ = PE^- = PE$
4) $PE^+_O = PE^+_{new}$
5) $PE^-_O = PE^-_{new}$
6) $BE^+ = BE \cup BE^+_{new}$
7) $BE^- = BE \cup BE^-_{new}$

Intuitively, function *expandQuery* modifies a query having the structure shown in Section 6.2.1 into a query having the structure shown above. Optional path expressions as well as further boolean expressions integrating the ones available in the original query are provided by the *policyFiltering* function.

Example 34 *Fig. 6.2 shows the result of applying function expandQuery to the query shown in Fig. 6.1 and the policies shown in Tab. 6.1, with the same assumptions as in Example 33.*

6.3 Implementation

In this section we present the implementation of the algorithm we described in Section 6.2.3. We first introduce a generic architecture (which we call *Access Control for RDF Stores*—AC4RDF) and describe our concrete implementation. Afterwards, we demonstrate how we successfully integrated AC4RDF into existing infrastructures. Finally, we evaluate the performance of our approach.

6.3.1 Architecture

A key goal of our approach is to be applicable to and reusable in different settings in which access to RDF data should be controlled. Our RDF query rewriting framework is based on three external component, which can be autonomously configured and adapted to a specific setting.

```
  CONSTRUCT *
  FROM {Person} foaf:name {Name};
                foaf:phone {Phone};
                foaf:interest {Document};
                foaf:holdsAccount {Account}
     [ {X11} rdf:type {X12},
       {X3} foaf:knows {X4},
       {X7} foaf:currentProject {P},
       {X8} foaf:topic {T},
       {P} foaf:theme {T} ]
  WHERE ( X11 = #bbb AND X12 = foaf:Person ) AND
        ( Person = #alice OR
          ( X3 = #alice AND X4 = Person ) OR
          ( X11 =#bbb AND X12 = foaf:Person )
        ) AND (
          ( X11 = Document AND X12 = foaf:Document AND
            X7 = #alice AND X8 = Document AND
            Person = #alice
          ) OR
          ( X11 = #bbb AND X12 = foaf:Person )
        ) AND (
          ( X3 = #alice AND X4 = #recSer AND
            Person = #alice
          ) OR
          ( X11 = #bbb AND X12 = foaf:Person )
        )
MINUS
  CONSTRUCT *
  FROM {Person} foaf:name {Name};
                foaf:phone {Phone};
                foaf:interest {Document};
                foaf:holdsAccount {Account}
     [ {X1} foaf:currentProject {X2} ]
  WHERE X1 = Person AND X2 = #rewerse
```

Figure 6.2: Expanded RDF query

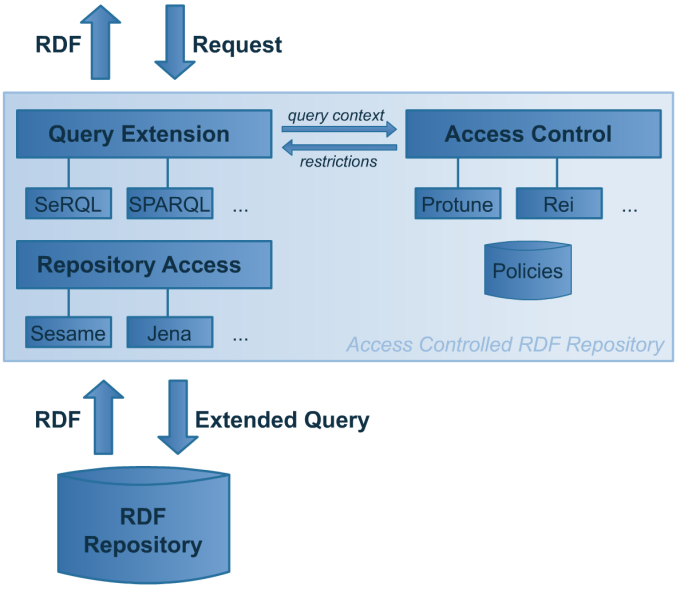


Figure 6.3: Architecture of AC4RDF

RDF Query Language Nowadays, many RDF query languages exist. Although SPARQL has recently become a W3C Recommendation, SeRQL and RDQL [73] are still being used. Therefore, we designed our framework in order to be able to support different RDF query languages

Policy language As mentioned in Section 6.1, different policy languages and engines can be exploited in order to specify and enforce RDF access-control policies. Our framework is able to deal with different policy languages and engines, as long as they provide the required expressiveness as described in Section 6.2.2

RDF store Finally, our framework is agnostic on the way RDF data are stored in order to support application scenarios which require different stores (like Sesame[3] [25], Kowari[4] or Jena[5])

The interface to such external components is provided by the *Query Extension*, *Access Control* and *Repository Access* modules of AC4RDF, whose generic architecture is shown in Fig. 6.3.

[3] `http://www.openrdf.org/`
[4] `http://kowari.sourceforge.net/`
[5] `http://jena.sourceforge.net/`

Query Extension The main task of this module is to rewrite a given query in such a way that only allowed RDF statements are accessed and returned. To this goal, it has to query the Access Control module for each `FROM` clause of the input query in order to retrieve further path expressions and constraints (cf. Section 6.2.3) which will be used to expand the query. Currently, an interface for the SeRQL query language is available

Access Control This module is responsible for evaluating the policies available in the system against the requests issued by the Query Extension module. The evaluation may take into account contextual information provided by the Query Extension module, such as properties of the requester (possibly to be certified by credentials) or environmental factors (e.g., time of the request). Currently, an interface for the PROTUNEpolicy language and framework is available

Repository Access When the expanded query is sent to the underlying RDF repository, it must be ensured that the latter is able to interpret it. For this reason, the main task of this module is to translate the incoming query in a language the repository is able to understand. Since our implementation exploits Sesame, which natively supports SeRQL, we could skip the translation step. Since the returned result set only contains allowed statements, it can be directly handed over to the requester

6.3.2 Experiments and evaluation

We set up a Sesame database with more than 3,000,000 RDF statements about persons and mails into a Quad CPU AMD Opteron 2.4GHz with 32GB memory and issued the queries from a Dual Pentium 3.00GHz with 2GB memory. We tested our approach in the worst case by issuing a query which would return a very large number of results (namely 1,280,000) if access control were disabled.

Fig. 6.4 shows the time needed to evaluate the query if access control is enabled: the query has been issued many times with different policies, which led to different expanded queries. The results are ordered according to the number of further `FROM` and `WHERE` literals of the expanded query w.r.t. the original one. Graph

(a) shows experiments performed when only *allow* policies were defined, whereas the experiments reported in graph (b) used both *allow* and *deny* policies.

Both graphs show that the addition of `WHERE` literals linearly increases the evaluation time. The reasons for this increment are: (i) each `WHERE` literal specifies new triples that are allowed to be accessed, therefore the number of triples that will be returned increases; and (ii) the further literals require time for their evaluation.

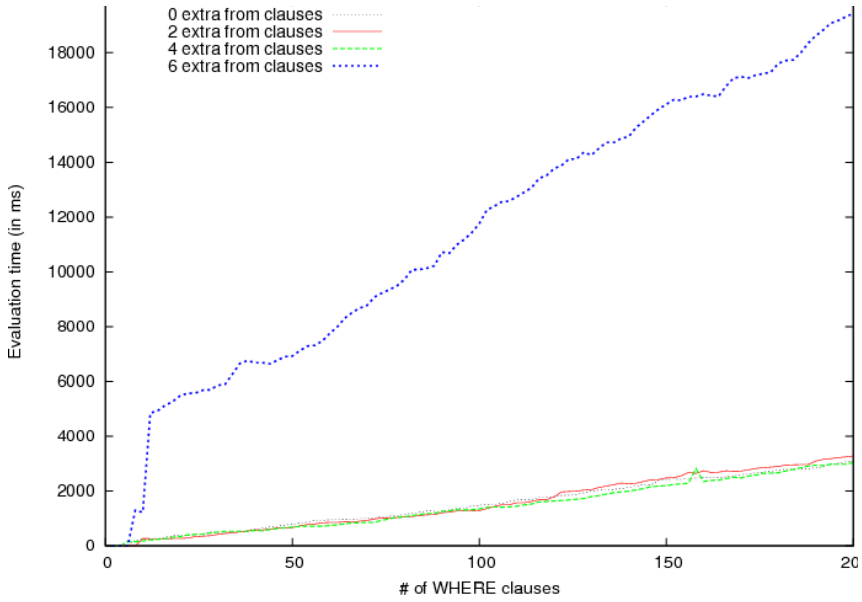
As the reader can notice, both graphs show a huge increment of the evaluation time when six `FROM` literals are added. Upon closer inspection of the behavior of the repository, we observed that the new literals produced a triplication of the number of triples to be considered, even though none of the new ones was to be returned (for this reason we believe that appropriate optimizations in the repository would help to increment the performance).

We also made other experiments (not reported here) with more selective queries and we noticed that the addition of `FROM` literals only produced linear increment of the evaluation time.

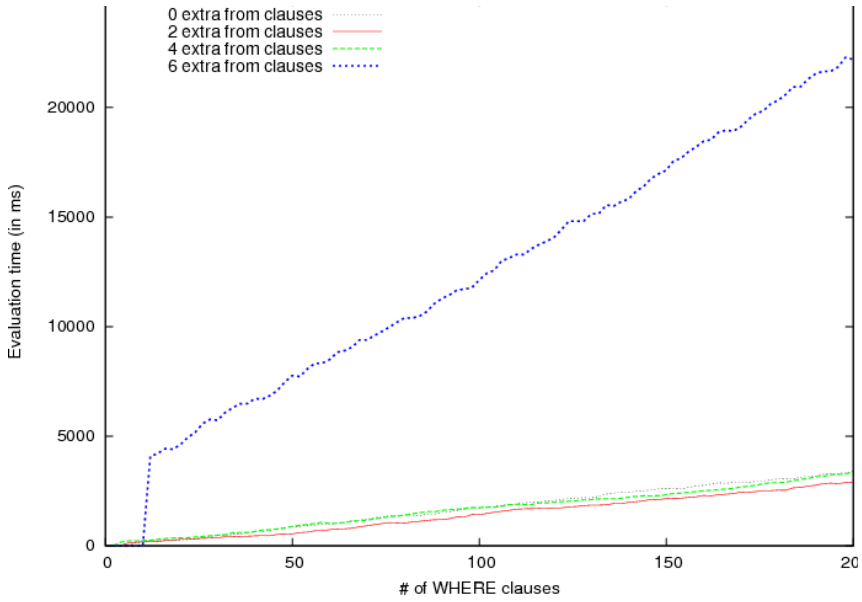
These results demonstrate that the approach described in this chapter scales to a large number of policies since, even if thousands of policies are defined in the system, not all of them will contribute to expand the original query with new literals. Only those: (i) protecting the triples appearing in the `FROM` clause of the original query; and (ii) whose context-dependent conditions are fulfilled; will be taken into account.

Our approach is especially appealing if the policies available in the system induce the expanded query to only contain boolean constraints and selective path expressions. In such a setting, longer evaluation times could be a reasonable price for the advantages provided by fine-grained access control. In particular, this cost may be acceptable for Semantic Web applications and services that must deal with sensitive data and which use highly selective queries.

On the other hand, further optimizations are required in order to reduce the evaluation time for non-selective queries: such optimizations include improvements in the query expansion process, reordering of constraints as well as native optimizations in the RDF repository.



(a)



(b)

Figure 6.4: Response time when increasing the number of `FROM` and `WHERE` literals: (a) with *allow* policies; and (b) with *allow* and *deny* policies

CHAPTER 7
Conclusions and Outlook

We have illustrated the PROTUNE policy framework and described how it can be used in order to create a security level on top of metadata stores and RDF repositories. The positive performance evaluation experiments we reported showed PROTUNE's feasibility w.r.t. real-world scenarios. According to [38, 31], PROTUNE is one of the most complete policy frameworks available to date w.r.t the desiderata laid out in the literature. More information about PROTUNE and the vision behind it can be found on the Web site of REWERSE's working group on policies: `http://cs.na.infn.it/rewerse/`. There, on the software page, the interested reader can find links to PROTUNE's software and some on-line demos and videos.

The main challenges for PROTUNE are related to usability: PROTUNE tackles usability issues by (partially or totally) automating the information exchange operations related to access control and information release control, and by providing a natural language front-end for policy authoring.

We plan to continue the development of PROTUNE by adding new features and improving the prototype. In particular, we plan to support reliable forms of evidences not based on standard certification authorities by exploiting services such as OpenId and enabling user-centric credential creation. Support to obligation policies is another foreseen extension (preliminary work is reported in [5, 54, 53, 18]).

Another important line of research concerns standardization: we are investigating how PROTUNE's policies and messages can be encoded by adapting and combining existing standards such as XACML (for decision rules—cf. Chapter 2), RuleML[1] or RIF[2] (for rule-based ontologies), WS-Security[3] (for message exchange) and so on. Concerning W3C's RIF initiative, our working group has contributed with a use case about policy and ontology sharing in trust negotiation.

Finally, we plan to complete some preliminary work which has been accounted

[1] `http://ruleml.org/`
[2] `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`
[3] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

for in the previous chapters, namely: (i) specification of the semantics of the PROTUNE policy language; (ii) more thorough user evaluation of feasibility, efficacy and efficiency of PROTUNE's natural language front-end; and (iii) experimental evaluation of the special-purpose strategy to create a security level on top of metadata stores by protecting them by means of PROTUNE policies.

1. Section 3.5 described how PROTUNE policies can be brought to a canonical form (beside variable renaming), thereby providing the first step toward the definition of the semantics of the PROTUNE policy language. However, the semantics of a generic PROTUNE policy in canonical form has not been described in this work. Two (non mutually exclusive) ways appear to be feasible: (i) describing how a PROTUNE policy can be mapped to a Logic Program (whose semantics has been extensively described in the literature—cf. [64]); and (ii) providing an executable semantics. The latter strategy bases on the *Model-driven architecture* [56] approach to software design, originally pursued by the Object Management Group[4]. By means of automatic generators, MDA-based tools (most noticeably, the Eclipse Modeling Framework[5]) allow to create reference implementations of a system out of a description of its model

2. Section 4.3.3 reported on the results of a preliminary unsupervised user study evaluating the usability of PROTUNE's natural language front-end. However, it is desirable confirming the results of such user study through a more thorough one, conducted in a supervised fashion and involving more participants as well as a higher number of policies

3. Chapter 5 presented a general-purpose strategy and a special-purpose one to lower the policy enforcement time by exploiting pre-evaluation. However, only the general-purpose strategy has been evaluated (cf. Section 5.3), whereas solely the theoretical properties of the special-purpose one have been investigated

[4] `http://www.omg.org/`
[5] `http://www.eclipse.org/modeling/emf/`

BIBLIOGRAPHY

[1] Fabian Abel, Juri Luca De Coi, Nicola Henze, Arne Wolf Koesling, Daniel Krause, and Daniel Olmedilla. A user interface to define and adjust policies for dynamic user models. In *WEBIST*, pages 184–191, 2009.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. RDFa in XHTML: Syntax and processing – second edition. Technical report, W3C, 2008. `http://www.w3.org/TR/rdfa-syntax`.

[4] B.V. Aduna. The SeRQL query language (revision 1.2). Technical report, Sirma AI Ltd., 2006. `http://www.openrdf.org/doc/sesame/users/`.

[5] José Júlio Alferes, Ricardo Amador, Philipp Kärger, and Daniel Olmedilla. Towards reactive Semantic Web policies: Advanced agent control for the Semantic Web. In *International Semantic Web Conference (Posters & Demos)*, 2008.

[6] Anne H. Anderson. An introduction to the Web Services Policy Language (WSPL). In *POLICY*, pages 189–192, 2004.

[7] Anne H. Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *SWS*, pages 53–60, 2006.

[8] Paul Ashley, Satoshi Hada, Gnter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). Technical report, IBM, November 2003.

[9] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[10] Michael Backes, Günter Karjoth, Walid Bagga, and Matthias Schunter. Efficient comparison of enterprise privacy policies. In *SAC*, pages 375–382, 2004.

[11] Mayank Bawa, Roberto J. Bayardo Jr., and Rakesh Agrawal. Privacy-preserving indexing of documents on the network. In *VLDB*, pages 922–933, 2003.

[12] Moritz Y. Becker. *Cassandra: Flexible Trust Management and its Application to Electronic Health Records*. PhD thesis, University of Cambridge, 2005.

[13] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY*, pages 159–168, 2004.

[14] Dave Beckett. RDF/XML syntax specification. Technical report, W3C, 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[15] Abraham Bernstein and Esther Kaufmann. GINO – a guided input natural language ontology editor. In *International Semantic Web Conference*, pages 144–157, 2006.

[16] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics*. Center for the Study of Language and Information, 2005.

[17] Piero A. Bonatti, Juri Luca De Coi, Wolfgang Nejdl, Daniel Olmedilla, Luigi Sauro, and Sergej Zerr. Policy based protection and personalized generation of web content. In *LA-WEB*, 2009.

[18] Piero A. Bonatti, Philipp Kärger, and Daniel Olmedilla. Reactive policies for the Semantic Web. In *ESWC*, 2010.

[19] Piero A. Bonatti and Daniel Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *POLICY*, pages 14–23, 2005.

[20] Piero A. Bonatti, Daniel Olmedilla, and Joachim Peer. Advanced policy explanations on the web. In *ECAI*, pages 200–204, 2006.

[21] Piero A. Bonatti and Pierangela Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.

[22] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. Technical report, W3C, 2004. `http://www.w3.org/TR/rdf-schema/`.

[23] Carolyn Brodie, Clare-Marie Karat, and John Karat. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *SOUPS*, pages 8–19, 2006.

[24] Jeen Broekstra and Arjohn Kampman. *Semantic Web and Peer-to-Peer*, chapter An RDF query and transformation language, pages 23–39. Springer, 2006.

[25] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[26] Ingo Brunkhorst, Paul Alexandru Chirita, Stefania Costache, Julien Gaugaz, Ekaterini Ioannou, Tereza Iofciu, Enrico Minack, Wolfgang Nejdl, and Raluca Paiu. The Beagle^{++} toolbox: Towards an extendable desktop search architecture. In *Semantic Desktop Workshop 2006*, November 2006.

[27] Jeremy J. Carroll, Christian Bizer, Patrick J. Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *WWW*, pages 613–622, 2005.

[28] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455, 2005.

[29] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL protocol for RDF. Technical report, W3C, 2008. `http://www.w3.org/TR/rdf-sparql-protocol/`.

[30] Juri Luca De Coi, Peter Fankhauser, Tobias Kuhn, Wolfgang Nejdl, and Daniel Olmedilla. Controlled natural language policies. In *CCS*, 2009.

[31] Juri Luca De Coi and Daniel Olmedilla. A review of trust management, security and privacy policy languages. In *SECRYPT*, pages 483–490, 2008.

[32] Juri Luca De Coi, Daniel Olmedilla, Sergej Zerr, Piero A. Bonatti, and Luigi Sauro. A trust management package for policy-driven protection & personalization of web content. In *POLICY*, pages 228–230, 2008.

[33] Alex Cozzi, Stephen Farrell, Tessa A. Lau, Barton A. Smith, Clemens Drews, James Lin, Bob Stachel, and Thomas P. Moran. Activity management as a Web Service. *IBM Systems Journal*, 45(4):695–712, 2006.

[34] Lorrie Cranor, Marc Langheinrich, and Massimo Marchiori. A P3P preference exchange language 1.0 (APPEL1.0). Technical report, W3C, 2002. `http://www.w3.org/TR/P3P-preferences/`.

[35] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[36] Juri Luca De Coi. Mapping natural language to formal policies: an ACE→ PROTUNE mapping. Technical report, Forschungszentrum L3S, July 2008.

[37] Sebastian Dietzold and Sren Auer. Access control on RDF triple stores from a Semantic Wiki perspective. In *European Semantic Web Conference*, 2006.

[38] Claudiu Duma, Almut Herzog, and Nahid Shahmehri. Privacy in the Semantic Web: What policy languages have to offer. In *POLICY*, pages 109–118, 2007.

[39] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for knowledge representation. In *Reasoning Web*, pages 104–124, 2008.

[40] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Discourse representation structures for ACE 6.5. Technical report, Department of Informatics, University of Zurich, February 2008.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[42] Rita Gavriloaie, Wolfgang Nejdl, Daniel Olmedilla, Kent E. Seamons, and Marianne Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the Semantic Web. In *ESWS*, pages 342–356, 2004.

[43] Simon Godik and Tim Moses. OASIS eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, February 2003.

[44] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD Conference*, pages 216–227, 2002.

[45] David Hawking. Challenges in enterprise search. In *ADC*, pages 15–24, 2004.

[46] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, pages 2–14, 2000.

[47] Amit Jain and Csilla Farkas. Secure resource description framework: an access control model. In *SACMAT*, pages 121–129, 2006.

[48] Lalana Kagal, Timothy W. Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *POLICY*, pages 63–, 2003.

[49] Kaarel Kaljurand. *Attempto Controlled English as a Semantic Web Language.* PhD thesis, Faculty of Mathematics and Computer Science, University of Tartu, 2007.

[50] Kaarel Kaljurand. ACE View – an ontology and rule editor based on Attempto Controlled English. In *OWLED*, 2008.

[51] Hans Kamp and Uwe Reyle. *From Discourse to Logic. Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Springer, 1993.

[52] Clare-Marie Karat, John Karat, Carolyn Brodie, and Jinjuan Feng. Evaluating interfaces for privacy policy rule authoring. In *CHI*, pages 83–92, 2006.

[53] Philipp Kärger, Emily Kigel, and VenkatRam Yadav Jaltar. SPoX: combining reactive Semantic Web policies and Social Semantic Data to control the behaviour of Skype. In *International Semantic Web Conference (Posters & Demos)*, 2009.

[54] Philipp Kärger, Emily Kigel, and Daniel Olmedilla. Reactivity and Social Data: Keys to drive decisions in social network applications. In *SDoW*, 2009.

[55] Rohit Khare and Tantek Çelik. Microformats: a pragmatic path to the Semantic Web. In *WWW*, pages 865–866, 2006.

[56] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[57] Takao Kojima and Yukio Itakura. Proposal of privacy policy matching engine. In *Digital Identity Management*, pages 9–14, 2008.

[58] Tobias Kuhn. AceWiki: Collaborative ontology management in controlled natural language. In *SemWiki*, 2008.

[59] Tobias Kuhn and Rolf Schwitter. Writing support for Controlled Natural Languages. In *ALTA*, 2008.

[60] Jiangtao Li, Ninghui Li, and William H. Winsborough. Automated trust negotiation using cryptographic credentials. In *ACM Conference on Computer and Communications Security*, pages 46–57, 2005.

[61] Ninghui Li and John C. Mitchell. A role-based trust-management framework. In *DISCEX (1)*, pages 201–, 2003.

[62] Ninghui Li, Mahesh V. Tripunitara, and Qihua Wang. Resiliency policies in access control. In *ACM Conference on Computer and Communications Security*, pages 113–123, 2006.

[63] Ninghui Li and Qihua Wang. Beyond separation of duty: an algebra for specifying high-level security policies. In *ACM Conference on Computer and Communications Security*, pages 356–369, 2006.

[64] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[65] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis G. Kafura, and Sumit Shah. First experiences using XACML for access control in distributed systems. In *XML Security*, pages 25–37, 2003.

[66] L. Ian Lumb and Keith D. Aldridge. Grid – enabling the global geodynamics project: Automatic RDF extraction from the ESML data description and representation via GRDDL. In *HPCS*, page 29, 2006.

[67] Daniel Olmedilla. *Policy Representation and Reasoning for Security and Trust Management in Distributed Environments*. PhD thesis, Leibniz Universität Hannover, 2009.

[68] Axel Polleres. From SPARQL to rules (and back). In *WWW*, pages 787–796, 2007.

[69] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. Technical report, W3C, 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[70] Pavan Reddivari, Tim Finin, and Anupam Joshi. Policy based access control for a RDF store. In *Proceedings of the Policy Management for the Web Workshop*, A WWW 2005 Workshop, pages 78–83. W3C, May 2005.

[71] Leo Sauermann, Gunnar Aastrand Grimnes, Malte Kiesel, Christiaan Fluit, Heiko Maus, Dominik Heim, Danish Nadeem, Benjamin Horak, and Andreas Dengel. Semantic desktop 2.0: The Gnowsis experience. In *International Semantic Web Conference*, pages 887–900, 2006.

[72] Rolf Schwitter, Kaarel Kaljurand, Anne Cregan, Catherine Dolbear, and Glen Hart. A comparison of three Controlled Natural Languages for OWL 1.1. In *OWLED 2008 DC*, 2008.

[73] Andy Seaborne. RDQL – a query language for RDF. Technical report, W3C, 2004. `http://www.w3.org/Submission/RDQL/`.

[74] Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. A language for updating RDF graphs. Technical report, W3C, 2008. `http://www.w3.org/Submission/SPARQL-Update/`.

[75] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobson, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *POLICY*, pages 68–79, 2002.

[76] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[77] Morris Sloman. Policy driven management for distributed systems. *J. Network Syst. Manage.*, 2(4), 1994.

[78] Bjørnar Solhaug, Dag Elgesem, and Ketil Stølen. Specifying policies using UML sequence diagrams – an evaluation based on a case study. In *POLICY*, pages 19–28, 2007.

[79] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[80] Scott D. Stoller, Ping Yang, C. R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *ACM Conference on Computer and Communications Security*, pages 445–455, 2007.

[81] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In *International Semantic Web Conference*, pages 419–437, 2003.

[82] Andrzej Uszok, Jeffrey M. Bradshaw, Renia Jeffers, Niranjan Suri, Patrick J. Hayes, Maggie R. Breedy, Larry Bunch, Matt Johnson, Shriniwas Kulkarni, and James Lott. KAoS policy and domain services: Toward a Description-Logic approach to policy representation, deconfliction, and enforcement. In *POLICY*, pages 93–, 2003.

[83] Andrzej Uszok, Jeffrey M. Bradshaw, James Lott, Maggie R. Breedy, Larry Bunch, Paul J. Feltovich, Matthew Johnson, and Hyuckchul Jung. New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS. In *POLICY*, pages 145–152, 2008.

[84] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DISCEX (1)*, pages 88–102, 2000.

[85] Ting Yu, Xiaosong Ma, and Marianne Winslett. PRUNES: an efficient and complete strategy for automated trust negotiation over the Internet. In *ACM Conference on Computer and Communications Security*, pages 210–219, 2000.

[86] Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable strategies in automated trust negotiation. In *ACM Conference on Computer and Communications Security*, pages 146–155, 2001.

[87] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Trans. Inf. Syst. Secur.*, 6(1):1–42, 2003.

Acknowledgments

The author's efforts were funded by the European Commission in the

- Network of Excellence REWERSE[1] (IST-2004-506779)
- Project TENCompetence[2] (IST-2004-02787)

[1] http://rewerse.net/
[2] http://www.tencompetence.org/

Lebenslauf

Persönliche Daten

Juri Luca De Coi
geboren am 1. November 1979 in Hagen
ledig

Ausbildung

1985-1993	Grundschule in Castel di Casio bei Bologna
1993-1998	Gymnasium in Porretta Terme bei Bologna
1991-1998	Studium des Fachs Klavier an der Musikhochschule Bologna
1998-2005	Studium des Fachs Komposition an der Musikhochschule Bologna
1998-2004	Studium des Informatikingenieurwesens an der Universität Bologna
seit 2006	Promotion an der Leibniz Universität Hannover

Tätigkeiten

1998-2001	Mitarbeit an der Montessori-Schule in Porretta Terme bei Bologna als einer der verantwortlichen Leiter des Musical-Laboratoriums
2004-2005	Wissenschaftlicher Mitarbeiter am D.E.I.S., dem Forschungsinstitut für Informatik-, Elektro- und Systemwissenschaften der Universität Bologna
2006-2010	Forscher am L3S (Learning Laboratories of Lower Saxony), Leibniz Universität Hannover

Sprachkenntnisse

Italienisch	Muttersprache
Deutsch	gut
Englisch	gut
Französisch	gut

Hannover, 9.3.2010 Juri Luca De Coi