Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

A Formal Approach to Specification, Analysis and Implementation of Policy-based Systems

Andrea Margheri

Supervisor
Prof. Rosario Pugliese

Ph.D. Coordinator
Prof. Pierpaolo Degano

August, 2016

To my family

Abstract

The design of modern computing systems largely exploits structured sets of declarative rules called *policies*. Their principled use permits controlling a wide variety of system aspects and achieving separation of concerns between the managing and functional parts of systems.

These so-called *policy-based* systems are utilised within different application domains, from network management and autonomic computing to access control and emergency handling. The various policy-based proposals from the literature lack however a comprehensive methodology supporting the whole life-cycle of system development: specification, analysis and implementation. In this thesis we propose formally-defined tool-assisted methodologies for supporting the development of policy-based access control and autonomic computing systems.

We first present FACPL, a formal language that defines a core, yet expressive syntax for the specification of attribute-based access control policies. On the base of its denotational semantics, we devise a constraint-based analysis approach that enables the automatic verification of different properties of interest on policies.

We then present PSCEL, a FACPL-based formal language for the specification of autonomic computing systems. FACPL policies are employed to enforce authorisation controls and context-dependent adaptation strategies. To statically point out the effects of policies on system behaviours, we rely again on a constraint-based analysis approach and reason on progress properties of PSCEL systems.

The implementation of the languages and their analyses provides us some practical software tools. The effectiveness of the proposed solutions is illustrated through real-world case studies from the e-Health and autonomic computing domains.

Preface

This thesis has been prepared at the Department of Computer Science of the University of Pisa and at the Department of Statistics, Computer Science and Applications of the University of Firenze. The thesis partially fulfils the requirements for acquiring the Ph.D. degree in Computer Science.

The Ph.D. research activities have been conducted under the supervision of Professor Rosario Pugliese from University of Firenze between November 2012 and April 2016. A first version of the thesis was submitted for external review in April 2016.

Most of the work underlying this thesis is the result of my research activities and I take full responsibility for its originality and authenticity. The contents of the thesis are based on the scientific works [MMPT13a, MMPT13b, MPT13, LMPT14, ACH+15, CBT+15, DLL+15, MPT15, MMPT16, MRNNP16a, MRNNP16b], some of which are currently under submission for publication.

The research activities carried out in [MMPT13a, MMPT13b, MMPT16] benefited from the collaboration with Tiani Spirit GmbH, an Austrian IT company working on access control and electronic healthcare system. The research activities carried out in [MPT13, LMPT14, ACH+15, CBT+15, DLL+15, MPT15] were conducted within the European FP7 research project ASCENS [FP715] and the Italian PRIN research project CINA [PRI16]. The research activities carried out in [MRNNP16a, MRNNP16b] resulted from a collaboration with Hanne Riis Nielson and Flemming Nielson from Technical University of Denmark. The role and the relevance of each work is further clarified within the thesis.

In addition to the above, the work reported in the thesis benefited from multiple fruitful interactions with some researchers from University of Firenze, IMT Lucca and University of Camerino, and from the opportunity I had to stay as a visiting Ph.D. student in the group (at that time called *Language-based Technology*) lead by Hanne Riis Nielson at the Technical University of Denmark.

Pisa, August 2016
Andrea Margheri

Acknowledgements

I would like to express my deep gratitude to all the people who consciously and unconsciously contributed in shaping this thesis during all the years of my PhD.

My first deep and sincere thanks are for my supervisor Rosario Pugliese who encouraged and supported me during the whole PhD. His dedication and passion in research were always an incentive and motivation for improving my efforts. I appreciated all the discussions and advice more than I can say.

My special thanks are for all the people I had the privilege and pleasure to work with. A special thanks must go to Francesco Tiezzi who was a reliable, trusted collaborator and mentor during all these years. I have also to thank Massimiliano Masi for his precious points of view and opinions, and Lorenzo Bettini for plenty of inestimable advice and admonitions in my using of Xtext.

I would like to thank Emilio Tuosto and Luís Caires who accepted to read this thesis and gave me priceless comments and suggestions.

A particular thanks is for Hanne Riis Nielson and Flemming Nielson from Technical University of Denmark for their hospitality during my visit in their research group. A special thanks is for Alberto Lluch Lafuente and his family that helped and supported me during all my stay in Copenhagen. I am really grateful to all the members of the group and all the nice people I had the pleasure to meet. Alessandro, Elena and Erisa for the enjoyable moments together. All the '*container's mates*' for their company and seamless entertainment. And Giulia for her sincerity and all the moments spent together supporting '*la Viola*'.

I am really grateful to all the people I met at University of Pisa, University of Firenze, IMT Lucca and during the ASCENS and CINA projects. Particular thanks are for Rocco De Nicola and Michele Loreti for their suggestions and great help. Thanks to my roommates in Firenze for all the chats, lunches and coffee breaks together. Thanks to Betti and Piluc for many lunches together.

I am really grateful to all my friends for the time we spent together. Thanks to all '*I ritardatari*' for our holidays and memorable moments together. Thanks to all '*I Galluzzini*' for their enjoyable company and friendships. Thanks to all '*I Bibbona remember*' for our weekends and dinners in Bibbona and Firenze. Thanks to all my English(-ish) friends and special people for their fancy company, holidays and moments together.

Last but not least, my most deep thanks are for my family and for their boundless support during all these years. Their honest, heartfelt advice profoundly proved their trust and faith in me.

Contents

Abstract **i**

Preface **v**

Acknowledgements **vii**

List of Acronyms **xiii**

Chapter 1 Introduction **1**
1.1 About this Thesis 4

Chapter 2 Setting the Scene **7**
2.1 Access Control 8
2.1.1 Terminology 9
2.1.2 Models 9
2.1.3 Policy-based Evaluation Process 10
2.1.4 An Attribute-based Specification Language: XACML 11
2.1.5 Research Objectives 14
2.2 Autonomic Computing 15
2.2.1 The SCEL Specification Language 15
2.2.2 Research Objectives 16
2.3 Case Studies 17
2.3.1 An e-Health Provisioning Service 18
2.3.2 A Robot-Swarm Disaster Scenario 19
2.3.3 An Autonomic Cloud Platform 20

Chapter 3 The FACPL Language **23**
3.1 Evaluation Process 24
3.2 Syntax 25
3.3 Informal Semantics 27
3.4 FACPL at work on the e-Health Case Study 29
3.5 Formal Semantics 32
3.5.1 Requests 33
3.5.2 Policy Decision Process 34
3.5.3 Combining Algorithms 37
3.5.4 Policy Enforcement Process 39
3.5.5 Policy Authorisation System 40
3.5.6 Properties of the Semantics 40

3.6 Supporting Tools . . . 43
3.6.1 The FACPL Library . . . 44
3.6.2 The FACPL IDE . . . 45
3.7 Concluding Remarks . . . 46

Chapter 4 Analysis of FACPL Policies 49
4.1 Attribute-based Formalisation of Security Policies . . . 50
4.1.1 Attribute-based Characterisation . . . 50
4.1.2 Semantic-based Formalisation . . . 52
4.2 Verification of Security Policy Enforcement . . . 54
4.2.1 Towards an Automated Verification Approach . . . 55
4.3 A Constraint-based Representation for FACPL . . . 56
4.3.1 A Constraint Formalism . . . 56
4.3.2 From FACPL Policies to Constraints . . . 58
4.3.3 Properties of the Representation . . . 61
4.3.4 Constraint-based Representation of the e-Health Case Study . . . 67
4.4 Formalisation of Properties . . . 68
4.4.1 Authorisation Properties . . . 68
4.4.2 Structural Properties . . . 69
4.4.3 Properties on the e-Health Case Study . . . 70
4.5 Automated Property Verification . . . 71
4.5.1 Expressing Constraints with SMT-LIB . . . 72
4.5.2 SMT-LIB-based Property Verification . . . 74
4.5.3 Supporting Tools . . . 76
4.6 Concluding Remarks . . . 77

Chapter 5 The PSCEL Language 79
5.1 PSCEL: a FACPL-based Instantiation of SCEL . . . 80
5.2 Syntax . . . 81
5.3 Formal Semantics . . . 84
5.3.1 Policy Constructs . . . 85
5.3.2 Programming Constructs . . . 88
5.4 PSCEL at work on the Robot-Swarm Case Study . . . 97
5.4.1 A PSCEL Extension: Adaptive Policies . . . 97
5.4.2 PSCEL Specification . . . 98
5.5 Supporting Tools . . . 102
5.5.1 The PSCEL Java Runtime Environment . . . 102
5.5.2 The PSCEL IDE . . . 105
5.6 Concluding Remarks . . . 106

Chapter 6 Analysis of PSCEL Specifications 109
6.1 Towards the Analysis of PSCEL Specifications . . . 110
6.2 PSCEL at work on the Autonomic Cloud Case Study . . . 111
6.2.1 Interplay between Policies and Processes . . . 113
6.3 Policy-Flow: Definition and Constraint-based Analysis . . . 114
6.3.1 A PSCEL-Oriented Constraint Formalism: Syntax and Exploitation . . . 115
6.3.2 From PSCEL Policies To Constraints . . . 117

6.4 Policy-Flow Graph . . . 121
6.4.1 Policy-Flow Graph at work on the Autonomic Cloud Case Study . . . 123
6.4.2 Automated Policy-Flow Graph Construction . . . 125
6.5 Progress Analysis of PSCEL Specifications . . . 127
6.5.1 Context-stable Policies . . . 128
6.5.2 Context-stable Policies with respect to a Process . . . 130
6.5.3 Context-Stability: Syntactic Check . . . 131
6.6 Concluding Remarks . . . 131

Chapter 7 Related Works **133**
7.1 FACPL vs XACML . . . 134
7.2 Languages for Access Control Policies . . . 135
7.3 Analysis of Access Control Policies . . . 137
7.4 Design and Analysis of Autonomic Computing Systems . . . 139
7.5 Supporting Tools . . . 140

Chapter 8 Concluding Remarks **143**

Bibliography **145**

List of Tables **155**

List of Figures **157**

List of Acronyms

ACM Access Control Matrix 8

ABAC Attribute-Based Access Control 7

DAC Discretionary Access Control 10

ECA Event-Condition-Action 79

FACPL Formal Access Control Policy Language 3

IDE Integrated Development Environment 23

MAC Mandatory Access Control 10

PAS Policy Authorisation System 25

PBAC Policy-Based Access Control 10

PDP Policy Decision Point 10

PEP Policy Enforcement Point 10

PR Policy Repository 10

PSCEL Policed-SCEL 3

RBAC Role-Based Access Control 8

SCEL Software Component Ensemble Language 3

SoD Separation of Duty 50

XACML eXtensible Access Control Markup Language 2

Chapter 1

Introduction

The secret of getting ahead is getting started.

Mark Twain

Computing systems are nowadays pervading our daily activities, fostering new services and carrying out critical applications, like, e.g., the management of cyber physical infrastructures and of sensitive datasets. In their design, the major challenges to address come from the highly dynamic operating environment and the massive number of the involved heterogeneous entities, featuring distributed control, unpredictable interactions and huge amounts of data to manage. Given their importance and societal impact, it is of paramount importance to ensure that modern computing systems are

- *secure*, that is they cannot enter into a non-secure configuration and their misuses cannot affect trustworthiness of the managed data; and
- *autonomic*, that is they are able to react and adapt themselves to changes of the operating environment.

To systematically ensure these characteristics, the use of linguistic approaches based on *policies*, i.e. structured sets of declarative rules, is largely advocated in the literature. Policies are employed to control a wide variety of system aspects and to achieve separation of concerns between the managing and functional parts of a system. These so-called *policy-based systems* have been recently exploited within different application domains. In this thesis, we address policy-based solutions for computer security, in particular concerning access control, and for autonomic computing.

In the rest of this chapter, we briefly introduce first the computer security and autonomic computing application domains and their more relevant open issues, then we outline the main contributions of the thesis. Section 1.1 concludes by commenting on the structure of the thesis and its related publications.

Computer security

Computer security is a broad field, covering several different approaches, using different technologies and involving various degrees of complexity. To define a system secure, a *security policy* expressing the allowed or forbidden behaviours needs to be provided. The enforcement of such a policy relies on a combination of various approaches, ranging, e.g., from cryptography to access control, and depends on the security aspects it addresses. A crucial one is managing the accesses to system resources, which is commonly addressed by identifying two different stages: *authentication* and *authorisation*. Authentication is the process of verifying if subjects are who they claim to be, while authorisation is the process of establishing if (already authenticated) subjects are allowed to access to a resource. Our focus is on authorisation and, in particular, on how it is enforced in term of an *access control* system.

The expected duties of an access control system amount to

> *control every access to the controlled system and its resources, ensuring that all and only authorised accesses can take place.*

It decides whether an *access request* should be permitted, thus to ensure that the resources are protected against unauthorised disclosure (*confidentiality*) and improper modification due to unauthorised accesses (*integrity*), while the effective use of resources by authorised subjects is ensured (*availability*).

Modern access control systems are defined by sets of policies containing the access rules used to filter out insecure accesses. These rules are based on *attributes*, i.e. security-relevant information exposed by the system, and define flexible, fine-grained controls. To support the specification of such systems, different proposals have been advocated in the recent years. A well-established one is the standard *eXtensible Access Control Markup Language (XACML)* [OAS13], which offers an XML formalism for the specification of access control policies and a structured evaluation process for their enforcement. Unfortunately, the management of XACML and, in general, of attribute-based policies is in practice cumbersome, hence it should be supported by rigorous analysis approaches, like, e.g., those in [ACC14, FKMT05, TdHRZ15]. However, the proposed approaches usually dealt only with a limited set of typical aspects of modern access control policies. Above all, they do not fully support the development of such policies with a comprehensive methodology encompassing specification, analysis and enforcement functionalities

Therefore, even though access control may seem a straightforward concept, its design and enforcement is complex and error-prone in practice. Due to the pervasive exploitation of access control in modern computing systems, a principled approach to the specification, analysis and implementation of access control systems is needed to improve their effectiveness and reliability.

Autonomic Computing

Autonomic computing is a recently devised paradigm for dealing with the difficulties of modern computing systems. This vision was proposed by IBM [Hor01] and consists in enhancing computing systems with self-managing functionalities. Such systems have indeed the ability to

> *manage their behaviours and dynamically adapt to operating changes in accordance with business policies and computational objectives.*

To achieve this type of behaviours, systems must enforce the so-called *self-** properties, that is capabilities of taking the appropriate decisions based on the information sensed from the operating environment. For example, enforcing the *self-configuring* property means that the system dynamically reconfigures itself to react to a sensed information change. However, due to the highly dynamic nature of the behaviours to design and the massive number of components to manage, the specification of autonomic computing systems must adhere to principled approaches supported by analysis functionalities.

Among other specification approaches, the use of high-level linguistic abstractions is a widely exploited solution that also advocates the use of policies to define and enforce adaptation strategies [HM08]. A rigorous framework exploiting policies is the *Software Component Ensemble Language (SCEL)* [DLPT14], which offers a set of linguistic abstractions expressly devised for the specification of autonomic computing systems. However, SCEL abstracts from the specific policy formalism to use and, hence, from the practical effects of policies on system behaviours.

Therefore, the SCEL rigorous framework has to be enriched with an effective policy formalism and appropriate analysis approaches aiming at statically reasoning on the dynamic behaviours enforced by systems.

Contributions

Policies are widely used means for the specification of access control and autonomic computing systems. The declarative nature of policies makes indeed them intuitive and easy to maintain. However, the various policy-based proposals from the literature lack a comprehensive methodology supporting the whole development life-cycle of such policy-based systems, i.e. from specification to analysis and implementation. This thesis aims at filling this gap by proposing formally-defined tool-assisted methodologies to support the whole development life-cycle of policy-based access control and autonomic computing systems. To devise such methodologies, our approach crucially relies on formal methods like, e.g., formal semantics and constraint formalisms.

The main contribution of this thesis is the *Formal Access Control Policy Language (FACPL)*, a language devoted to the specification of attribute-based access control systems. FACPL is equipped with practical formally-defined analysis approaches and supporting tools. Furthermore, the thesis introduces *Policed-SCEL (PSCEL)*, a full-fledged FACPL-based instantiation of SCEL supported by static analysis approaches to reason on the effects of policies on system behaviours.

Thus, the thesis features the specification, analysis approaches and supporting tools of the following languages

Formal Access Control Policy Language (FACPL): a specification language for attribute-based access control systems;

Policed-SCEL (PSCEL): a FACPL-based instantiation of SCEL supporting the specification of context-dependent authorisation controls and adaptation strategies.

The evaluation processes of these languages are formalised by rigorous formalisms that lay the basis for the formal development of analysis approaches. We indeed exploit constraints to uniformly and precisely represent policies and to enable automatic verification of properties of interest. Specifically, we rely on the so-called *satisfiability modulo theories* (SMT) formulae that permit a combined use of formulae from multiple theories, like, e.g., boolean and linear arithmetics. To validate the effectiveness of these languages, as well as of their analysis and enforcement functionalities, we exploit some real-world case studies.

Concerning the scientific contributions, the languages and their functionalities advance the current state-of-the-art towards a twofold direction. On the one hand, FACPL fills the gap of a single, fully-integrated methodology supporting the specification, analysis and implementation of access control policies. On the other hand, PSCEL advances the current specification formalisms for autonomic computing systems by formalising a principled exploitation of policy-based adaptation strategies and, most of all, addresses for the first time a flow analysis of policies that can be used to reason on the dynamic behaviours of autonomic systems. Finally, it is worth noticing that each ingredient of both FACPL and PSCEL is first formally introduced and then implemented via Java-based tools.

1.1 About this Thesis

The rest of the thesis is organised as follows

Chapter 2 sets the scene by introducing the background concepts of access control and autonomic computing, presenting the research objectives of the thesis and outlining the case studies we focus on in the remaining chapters.

Chapter 3 reports the full account of the FACPL language, starting from the evaluation process to the syntax and formal semantics. After presenting some properties of the semantics, it outlines the Java-based toolchain supporting FACPL. This chapter is based on [MMPT13b, MMPT16].

Chapter 4 presents the constraint-based analysis for FACPL, a set of properties of interest on policies, and the automatic tools for property verification. To motivate the analysis, it also introduces a FACPL-based formalisation of traditional security policies. This chapter is based on [MPT15, MMPT16].

Chapter 5 reports the full account of the PSCEL language, starting from the design principles to the syntax and formal semantics. It also outlines a Java runtime environment to practically support the development of PSCEL systems. This chapter is based on [MPT13, LMPT14, ACH+15, DLL+15, MRNNP16a, MRNNP16b].

Chapter 6 presents the constraint-based analysis for PSCEL. It introduces a specific flow graph and its exploitation to reason on the effects of policy evaluations on the progress of PSCEL systems. This chapter is based on [MRNNP16a, MRNNP16b].

Chapter 7 reviews more closely related works to the FACPL and PSCEL languages, their analysis approaches and supporting tools.

Chapter 8 concludes the thesis with some final remarks and touches upon directions for future works.

Publications

The main publications at the basis of this thesis can be divided between those referring to FACPL and those to PSCEL. In the case of FACPL we have:

- [MMPT16]: it contains the complete presentation of the FACPL language, its analysis functionalities and supporting tools;
- [MPT15]: it contains a preliminary version of the analysis of FACPL policies;
- [MMPT13b]: it contains the application of a preliminary version of FACPL to model an e-Health case study;
- [MMPT13a, CBT+15]: they contain an application of the FACPL supporting tools for the definition of a Cloud manager. The contents of these works are not reported in the thesis for the sake of presentation.

While, in the case of PSCEL we have:

- [MRNNP16a]: it contains the complete presentation of the PSCEL language, its analysis functionalities and supporting tools;
- [MRNNP16b]: it contains a preliminary version of the analysis of PSCEL specifications;
- [MPT13, DLL+15]: they contain a preliminary version of PSCEL;
- [LMPT14, ACH+15]: they contain a preliminary description of the PSCEL supporting tools.

Chapter 2

Setting the Scene

> Computer science is no more about computers than astronomy is about telescopes.
>
> Edsger Dijkstra

In this introductory chapter, we set the scene of the whole thesis by presenting the background concepts of *access control* and *autonomic computing*, and by outlining the main research objectives of the thesis.

We present first access control, one of the key ingredient of computer security. After a general overview of the main access control models proposed in the literature, we focus on the newer *Attribute-Based Access Control (ABAC)* [HKF15] model. Specifically, we introduce the policy-based evaluation process commonly exploited by ABAC systems and then *eXtensible Access Control Markup Language (XACML)* [OAS13], an XML-based standard by OASIS that is largely used in real-world applications to realise ABAC systems.

We then introduce autonomic computing, a recent paradigm envisioned by IBM [Hor01] for the design of self-managing computing systems. Its main objective is enhancing computing systems with self-managing functionalities that permit autonomous and continuous adaptation to changing operating conditions. Besides a brief introduction to autonomic computing, we present *Software Component Ensemble Language (SCEL)* [DLPT14], a formal language expressly devised to design and program autonomic systems.

We conclude by presenting the case studies we use in the rest of the thesis to illustrate our approach and its effectiveness.

Structure of the chapter. The rest of this chapter is organised as follows. Section 2.1 presents access control, its models and the standard XACML. Section 2.2 presents the autonomic computing paradigm and the SCEL language. Section 2.3 introduces the case studies we will use in the rest of the thesis.

2.1 Access Control

Access control systems are the first line of defence for the protection of computing systems. They are defined by *rules* that establish under which conditions a subject's *request* aiming at accessing a resource has to be permitted or denied. In practice, this amounts to restrict physical and logical access rights of subjects to system resources. It is worth noticing that we focus on access control and abstract from the issues of authenticating access requests. Indeed, we assume that requests have been already authenticated by means of one of the various well-established authentication technologies that are nowadays available (see, e.g., OAuth [Har13] or SAML [OAS05]), so that the information included within requests trustwhorty describes the identity of the subjects issuing them.

Over the years, different models for the definition of access controls have been proposed and exploited. Traditional models assign access rights on the base of the identity of subjects and resources, either directly — e.g. *Access Control Matrix (ACM)* [GD72, Lam74] — or through predefined features, such as roles or groups — e.g. *Role-Based Access Control (RBAC)* [FK92]. However, they are inadequate to deal with modern distributed systems, where the population of subjects can be unknown at the assignment time of access rights. Moreover, they cannot easily encompass information representing the evaluation context, as e.g. system status or current time. An alternative model that permits overcoming these issues is ABAC [HKF15]. Here, the rules are based on arbitrary information from the system and organised in structured collections called *policies*. Their evaluation, which is based on the evaluation process introduced in [YPG00], ensures the enforcement of flexible, context-aware controls.

In the context of access control-related topics, the enforcement of security requirements in programming languages is worth to mention due to the large effort it has attracted and the wide literature available. In this area, security policies commonly concern information flow, i.e. asserting that secret input data cannot be inferred by observing system outputs. The main approaches in this area concern language-based solutions like, e.g., type systems and other static analysis techniques (see, e.g., [SM03] for a survey). However, the frame of reference of this thesis does not concern information flow. Instead, it refers to the methodologies used to define, analyse and implement access control systems being part of more complex computing systems, rather than as controls defined on a top of a programming language. In the rest of the thesis, we therefore do not take into account information flow and its related issues.

In the following, first we comment on the access control-related terminology we use throughout the thesis (Section 2.1.1). Then, after a brief introduction to the main access control models (Section 2.1.2), we present the policy-based evaluation process of ABAC systems (Section 2.1.3) and an XML-based standard for their specification (Section 2.1.4).

2.1.1 Terminology

Access control and, in general, computer security are wide and deep fields whose terminology is used in the literature, from time to time, with slightly different meanings. Here, we report our intended meanings of the main access control-related terminology we use.

An **authorisation** is the decision taken by the system regarding a request for access performed by a *subject*. The term **subject** is used to denote any principal willing to perform an access request. For the sake of presentation, the terms **access** and **decision** will be sometimes used in place of authorisation.

An **access control policy** is a structured collection of rules defining the precise conditions leading to positive or negative authorisations. When it is clear from the context, the term **policy** is used in isolation to mean *access control policy*.

An **access control system** is a set of access control policies mediating all the accesses to the resources of the controlled system. The development of access control systems requires to adhere to the specificities of the application domain and the used technology. To this aim, it is usually carried out in terms of a multi-step process that concerns different levels of abstractions: *security policy*, *security model*, *security mechanism*; their descriptions follow.

- **Security policy** (or *high-level policy*): it describes the behaviours an access control system has to enforce. It is usually defined as a set of (structured) descriptions written in *natural language*, like, e.g., "*Doctors can write e-Prescription documents*".
- **Security model**: it provides a *formal* representation of the security policy. It enables the formalisation and proof of properties on the enforced authorisations. We thus consider access control policies be part of the security model.
- **Security mechanism**: it defines the low level (software and hardware) machinery implementing the controls imposed by the security policy and formalised in the security model.

This conceptualisation of access control systems is borrowed from [SdV00] and mainly coincides with the others from the literature, like, e.g., those in [Bis02, NIS09, Gol11].

An **access control model** refers to the design and distinguishing concepts on which an access control system is built up on. For instance, *attribute* is the peculiar concept of the ABAC model, as we will see below.

To conclude, we remark that this terminology is intended to be consistent for the topics of this thesis and within the confines of the thesis itself.

2.1.2 Models

Access control systems have the purpose of maintaining the controlled system in a *secure* state. The description of states and the conditions checked by controls characterise each access control model.

Traditional models are based on the identity of subjects and resources, and on their identification by means of unique names. Among others, two well-known examples are *Access Control Matrix (ACM)* and *Role-Based Access Control (RBAC)*.

ACM [GD72, Lam74] is the simplest identity-based model. It came up in the context of operating systems and requires that each action a subject can perform on each system

resource has to be enumerated. The relationship between subjects and resources can thus be detailed by means of the so-called *access control matrix*. Despite its conceptual simplicity, this model suffers from scalability issues, as it is cumbersome to compactly manage a massive number of subjects and resources.

RBAC [FK92] was proposed to overcome scalability issues of ACM. Specifically, it proposes grouping authorisations in terms of conditions on *roles*. A role identifies a collection of authorisations on a set of resources that are associated to a predefined feature identifying a set of subjects. However, despite its multiple variants (see, e.g., those described in [SCFY96]), RBAC hardly permits the specification of fine-grained controls. In fact, to introduce authorisations on single resources, RBAC requires the definition of new roles. This demand leads to a proliferation of roles, whose management, however, has turned out to be a difficult task over time, especially in distributed settings.

An alternative model that permits tackling the weaknesses of the identity-based models is *Attribute-Based Access Control (ABAC)* [HKF15]. Here, the rules are based on *attributes*, i.e. arbitrary security-relevant information exposed by the system, the involved subjects, the action to be performed, or by any other entity of the evaluation context that are relevant to the rules at hand. The use of attributes allows ABAC policies to define both group-based and fine-grained controls.

In recent ABAC proposals, the exploitation of attributes has proceeded in pair with a combined use of positive and negative authorisations. Traditionally, positive and negative authorisations were used in mutual exclusions, but their principled combination permits convenient design strategies for supporting authorisation exceptions. Due to the possible presence of conflictual authorisations, access controls need to be managed according to explicit conflict resolution strategies, the so-called *combining algorithms* [JSSS01]. Attribute-based rules are typically hierarchically structured in the form of specifications called policies; from this name derives the terminology *Policy-Based Access Control (PBAC)* [NIS09], sometimes used in place of ABAC. To sum up, ABAC permits defining hierarchically structured, flexible and context-aware access controls that, as reported in [JKS12], are expressive enough to uniformly represent all the other models.

To complete this outline, we report an additional classification of access control models that is based on the ownership of the controls: *Discretionary Access Control (DAC)* and *Mandatory Access Control (MAC)*. In the former, the specification of controls is left to the discretion of the owner, while in the latter the system controls the access and the resource owner cannot circumvent them. In this respect, ACM better fits the DAC approach, while RBAC and ABAC can be used both for the MAC and DAC approaches. Hybrid approaches are also possible.

2.1.3 Policy-based Evaluation Process

The evaluation process of ABAC systems needs to be flexible and powerful at the same time, thus to uniformly deal with complex, context-dependent controls within systems of different natures. To this aim, ABAC systems typically exploit the policy-based evaluation process introduced in [YPG00], whose aim was to support the evaluation of the general-purpose policy-based specifications of [MESW01]. As distinguishing feature, it decouples the calculation of policy decisions from their subsequent enforcement in the system.

The evaluation process relies on three main architectural components: the *Policy Repository (PR)*, the *Policy Decision Point (PDP)* and the *Policy Enforcement Point (PEP)*. These

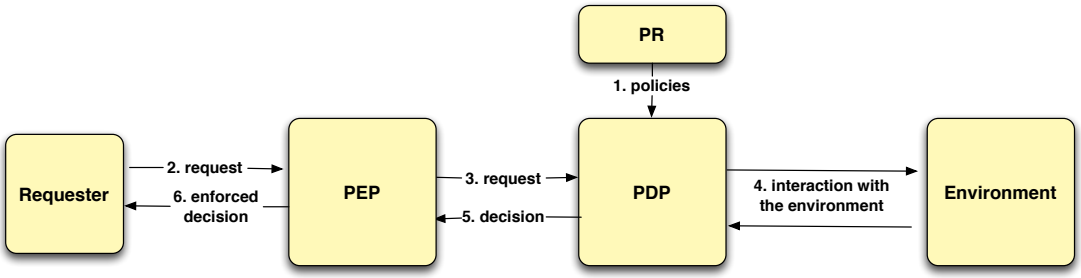


Figure 2.1: Policy-based evaluation process

components can be described as follows

- PR: it stores and provides the policies to the PDP;
- PDP: it calculates the decision for a request on the basis of the available policies;
- PEP: it enforces the decision calculated by the PDP in the controlled system.

The interactions among these components are graphically represented in Figure 2.1. The evaluation process assumes that each system request is managed by some policies that are stored in the PR and made available to the PDP (step 1). When a request to evaluate is received by the PEP (step 2), it is sent for evaluation to the PDP (step 3). The PDP evaluates the request on the base of the available policies and by possibly interacting with the environment (step 4). The calculated decision is then returned to the PEP (step 5), which finally enforces it in the system (step 6).

This structured process for the enforcement of access control systems guarantees separation of concerns among policies, their evaluation and the system itself. Among others, the main advantages it ensures are: (i) different types of requests can be accepted, as the PEP can appropriately encode them in the format required by the PDP; (ii) the PDP can be placed in any part of the system, thus with the PEP acting as gateway or proxy; (iii) the PR can be instantiated to support dynamic, possibly regulated, modifications of policies[1].

This process provides a general workflow that each policy language, as e.g. XACML, can tailor to its specific needs. In Section 3.1 we will present the FACPL-based refinement of this workflow.

2.1.4 An Attribute-based Specification Language: XACML

Many languages have been proposed for the practical specification and enforcement of access control policies (see, e.g., [HL12] for a survey). Among the proposed languages that exploit the ABAC model, the OASIS standard *eXtensible Access Control Markup Language (XACML)* [OAS13] is probably the best-known one.

XACML is a well-established language for the specification of attribute-based access control policies and requests. It relies on an XML-based syntax and on an instantiation of the policy-based evaluation process presented in Section 2.1.3. Besides combining algorithms and complex control functions, it supports the specification of *obligations*[2], that is

[1] When PR provides support for the specification of administration controls on policy modifications, it is usually called *Policy Administration Point* (PAP).

[2] Obligations were originally introduced in [Slo94] to adapt systems as result of policy evaluations.

additional actions returned as result of policy evaluations (e.g. logging or mailing actions) that have to be discharged by the PEP in order to enforce PDP decisions.

In the rest of this section, we outline the main aspects of the XACML syntax, together with an informal presentation of its evaluation process. As a matter of notation, the words emphasised in sans-serif, e.g. Rule, are XML elements, while attributes of such elements are in italics, e.g. *combining algorithm*.

The access control policies defined by XACML are hierarchically structured as Policys (i.e., collections of Rules) and PolicySets (i.e., collections of Policys and other PolicySets).

A Rule represents the basic element of a policy. It defines a Target, that consists of conditions on request attributes, whose evaluation establishes if the rule applies to the considered request. Additionally, a rule defines a positive or negative *effect* (aka authorisation decision), that is returned when the rule applies. Obligations possibly associated to the effect are instead defined in the form of ObligationExpressions and AdviceExpressions elements. The former corresponds to mandatory enforcement actions, while the latter to optional ones.

Besides their own Target, ObligationExpressions and AdviceExpressions, Policys and PolicySets specify a *combining algorithm* that is used to combine the evaluation results of the enclosed elements.

The AttributeDesignator and AttributeSelector elements are used within Targets to identify and retrieve attribute values from a request. Specifically, an AttributeDesignator specifies an identifier, e.g. “role”, and an attribute category, e.g. “subject”, while an AttributeSelector specifies an XPath [CD15] expression. Each of these elements explicitly defines a value type that must match for the attribute values to be retrieved; the special type *anyUri* matching any value can be used. If no value is retrieved, the evaluation of the enclosing element correspond to either an error or a not-matching comparison, it depends, respectively, on whether the option *MustBePresent* is set to true or false.

A Target has to adhere to a rigid element structure. Indeed, it consists of a (conjunctive) sequence of AnyOf elements, each one containing a (disjunctive) sequence of AllOf elements, each one containing in its turn a (conjunctive) sequence of Match elements. A Match element represents the basic building-block of targets and specifies a control on request attributes in the form of: an AttributeDesignator or an AttributeSelector, a typed literal and a matching function.

The main intricacies of the evaluation of XACML policies are due to the evaluation of Targets. Specifically, it can return true (resp., false), meaning that the parent element is applicable (resp., not applicable) to a request, or indeterminate, meaning that an error occurs. In details, a Target returns true when all enclosed AnyOfs are true, while it returns false when at least one AnyOf is false. An AnyOf returns true if at least an enclosed AllOf is true, while it returns false when all AllOf are false. An AllOf returns true if all enclosed Matchs are true, while it returns false when at least one Match is false. Instead, a Match returns true if at least a value, among those retrieved through the designed attribute, matches the literal according to the matching function; otherwise it returns false. Instead, a Target returns indet if no AnyOf returns false and at least one returns indet. An AnyOf returns indet if no AllOf returns true and at least one returns indet. An AllOf returns indet if no Match returns false and at least one returns indet. A Match returns indet if the requested attribute is missing and the option *MustBePresent* is set to true, or if an error occurs in the evaluation of the matching function.

By way of example, we report in Listing 2.1 an example of an XACML policy that aims

```
<PolicySet
    PolicySetId="AliceEPrescription"
    PolicyCombiningAlgId="policy-combining-algorithm:permit-overrides">
    <Target>
        <AnyOf>
            <AllOf>
                <Match MatchId="string-equal">
                    <AttributeValue DataType="string">
                        Doctor
                    </AttributeValue>
                    <AttributeDesignator
                        DataType="string"   MustBePresent="false"
                        Category="subject"  AttributeId="role" />
                </Match>
            </AllOf>
        </AnyOf>
    </Target>
    <Policy PolicyId="ePrescription"
        PolicyCombiningAlgId="rule-combining-algorithm:permit-overrides">
        <Target>
            <AnyOf>
                <AllOf>
                    <Match MatchId="string-equal">
                        <AttributeValue DataType="string">
                            e-Prescription
                        </AttributeValue>
                        <AttributeDesignator
                            DataType="anyURI" MustBePresent="false"
                            Category="resource" AttributeId="type" />
                    </Match>
                </AllOf>
            </AnyOf>
        </Target>
        <Rule RuleId="permitAll" Effect="Permit"> <Target /> </Rule>
    </Policy>
</PolicySet>
```

Listing 2.1: An XACML Policy

at regulating the access of subjects with role *Doctor* to resources named *e-Prescription*; for presentation's sake, we omit the XML namespaces and many XML elements. Looking at the XML code, it is possible to individuate the rigid structure of targets, the hierarchical organisation of policies and the references to request attributes by means of AttributeDesignator elements.

Furthermore, XACML represents the accesses to authorise as Requests formed by a collection of (possibly multi-valued) attributes. Each attribute is identified by a category and an identifier, and is associated to a (set of) value(s). An example of XACML request is reported in Listing 2.2. The request is defined by two attributes: that with category *subject* and name *role* is assigned to the value "*Doctor*", while that with category *resource* and name *type* is assigned to the value "*e-Prescription*".

The evaluation of XACML policies proceeds according to the policy structure and the matching of targets. When the target matches, the evaluation of rules returns the rule effect, while that of policies and policy sets returns the combined result of the evaluations of the enclosed elements according to the chosen combining algorithm. Notably, all results can be possibly paired with obligations. From this description, it is easy to believe that the policy in Listing 2.1 authorises the request in Listing 2.2 to permit, i.e. it grants the access.

```
<Request>
    <Attributes Category="subject">
        <Attribute IncludeInResult="false" AttributeId="role">
            <AttributeValue DataType="string">
                Doctor
            </AttributeValue>
        </Attribute>
    </Attributes>
    <Attributes Category="resource">
        <Attribute IncludeInResult="false"  AttributeId="type">
            <AttributeValue DataType="string">
                e-Prescription
            </AttributeValue>
        </Attribute>
    </Attributes>
</Request>
```

Listing 2.2: An XACML Request

To sum up, due to its XML-based syntax and the numerous functionalities it provides, XACML is commonly used in many real-world application domains, e.g. e-Health and service-oriented computing, and is supported by several tools[3]. The use of XML clearly ensures wide interoperability, but limits the readability of policies only to trained developers. Furthermore, it is generally acknowledged that XACML lacks a formally-defined semantics (see, e.g., [RLB+09, CM12, RRNN12, ACC14]), thus the specification and realisation of analysis techniques supporting policy developers is cumbersome.

2.1.5 Research Objectives

Besides the various proposals like, e.g., XACML, the development of ABAC systems is lacking of a comprehensive methodology encompassing not only a specification language, but also analysis functionalities and tool support.

Therefore, the main objectives of our research activities conducted on access control-related topics have been as follows.

O1 Provide a compact, yet expressive language for the specification of ABAC systems.

O2 Formally and precisely define the evaluation process of ABAC systems.

O3 Formally devise an analysis approach capable of taking into account all the peculiarities of ABAC.

O4 Implement the language, its evaluation process and the analysis approach by means of practical software tools.

The ultimate goal is to allow access control system developers to use formally-defined functionalities without requiring them to be familiar with formal methods.

[3] An excerpt of the tools supporting XACML can be found at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#other.

2.2 Autonomic Computing

Autonomic computing is a paradigm to build computing systems that are able to manage themselves. This vision was introduced by IBM [Hor01] with the aim of dealing with the ever growing complexity of modern software-intensive systems. The key idea is to introduce within systems *self-managing* functionalities that can relieve users and administrators from the burden of low-level activities. The term 'autonomic' was indeed coined by looking at the human autonomic nervous system, where multiple self-managing functionalities are seamlessly embedded in our bodies that we barely notice them.

An autonomic system, from a practical point of view, is intended to be formed by multiple collaborating autonomic entities that adapt their behaviours for reacting to operating changes. Their computational power is usually limited to sensing the evaluation context, reading/updating their (limited) knowledge or exchanging messages. Therefore, to provide complex behaviours, autonomic systems rely on cooperative and collaborative interactions among components that lead to *emergent behaviours*, i.e. collaborative behaviours that single components could not achieve working in isolation. To this aim, it is commonly advocated that components should be able to self-organise themselves into goal-oriented groups, the so-called *ensembles* [Pro07].

The design and deployment of autonomic systems can concern a variety of aspects and pursue even more design choices. Among others, the architectural design of components is of paramount importance. A well-known approach is that called MAPE-K [KC03, IBM06], which defines a control loop that, by relying on component knowledge, aims at continuously monitoring, analysing, planning and executing possible adaptation strategies on the managed element. On the base of MAPE-K, multiple design proposals have been suggested in the literature and exploited in real applications (see, e.g., [HM08] for a survey or [LMD13] for a text book). The use of high-level models and languages is one of the most widely exploited proposal, see, e.g., in [BCL+06, SGP12, HK14]. Our focus is on *Software Component Ensemble Language (SCEL)* [DLPT14], a formal language fostering a component-based approach to design and program autonomic systems. We introduce the main ingredients of SCEL in the next section.

2.2.1 The SCEL Specification Language

The SCEL approach to autonomic systems relies on the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs). ACs are the building blocks of the systems and represent the computational units whose dynamical organisation forms ACEs. The aim of SCEL is thus to define an appropriate set of linguistic abstractions that permit specifying the behaviours of ACs, their interactions and the formation of ACEs.

An AC consists of: *Interface*, *Processes*, *Knowledge* and *Policies*. How these constituent elements are organised to form an AC is graphically illustrated in Figure 2.2; their description follows.

- *Interface*: it publishes information on the component itself in the form of *features*[4].
- *Processes*: they describe how computations may progress and are modelled in the style of process calculi.

[4]Notably, in the SCEL reference paper [DLPT14], the term *attribute* is used in place of *feature*. In this thesis we choose *feature* to not overload *attribute* with different meanings.

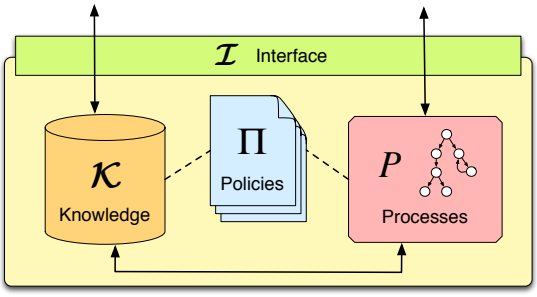


Figure 2.2: SCEL Autonomic Component

- *Knowledge*: it stores and manages component information.
- *Policies*: they control and adapt the processes by enforcing adaptation strategies and guaranteeing the accomplishment of tasks.

The design choice of taking Processes apart from Policies ensures a clear separation of concerns: the normal computational behaviour is defined through processes, while the authorisation and adaptation strategies are defined through policies. At the same time, the use and exploitation of the Knowledge allow components to acquire information about their status (*self-awareness*) and their environment (*context-awareness*) in order to perform *self-adaptation* and to initiate *self-healing* or *self-optimising* actions. All these *self-** properties, as well as *self-configuration*, can be naturally expressed by exploiting SCEL higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes.

Another distinguishing trait of the SCEL approach is the flexibility of communication means. A particular type of communication, called *predicate-based*, allows processes to implicitly select the set of components, i.e. the ensemble, acting as communication partners. Indeed, processes can create predicates defining conditions on component interface features, whose referred values are dynamically checked at runtime to establish if a component can take part in a communication. The use of predicates supports a highly dynamic and flexible formation of ensembles, as it is graphically shown in the example of Figure 2.3. Specifically, according to the features exposed in component interfaces, which are exemplified as the colours green and red, two (abstract) predicates are used to identify two (overlaid) ensembles of components.

2.2.2 Research Objectives

Although SCEL is equipped with a formal operational semantics (see [DLPT14] for a full account), the prediction of the dynamic evolutions of SCEL systems is challenging. Indeed, SCEL abstracts from the formalism used to specify policies and, hence, from how adaptation strategies can be precisely defined and enforced in system components. Furthermore, the use of predicates, their runtime evaluation and the dynamic enforcement of adaptation strategies can cause that behaviours unforeseen at design time may arise at runtime and lead to unexpected consequences.

Therefore, our research activities on autonomic computing-related topics have been as follow.

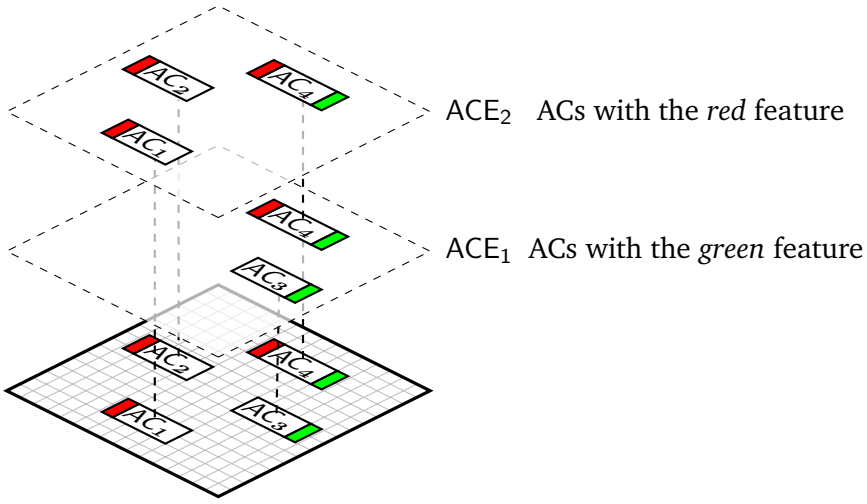


Figure 2.3: Abstract representation of SCEL Autonomic Component Ensembles

O5 Instantiate SCEL with an appropriate policy language capable of regulating interactions among components and enforcing adaptation strategies.

O6 Devise a static analysis approach for the language aiming at pointing out the effects on system behaviours of policy-based adaptation strategies.

O7 Implement the language, its evaluation process and analysis approach by means of practical software tools.

The ultimate goal is to deploy a full-fledged policy-based instantiation of SCEL both as a formal language and as a runtime environment, and to support its analysis.

2.3 Case Studies

The languages, analysis techniques and tools we propose for the development of secure or autonomic computing systems have been extensively validated by means of real-world case studies covering a large spectrum of aspects of interest. The case studies we consider in the thesis are as follows.

- *E-Health*: it concerns the provision of e-Health (aka electronic healthcare) services for the exchange of private health data among European countries. Chapters 3 and 4 specify and analyse, respectively, an access control system for securing these services.
- *Robot-swarm*: it concerns the modelling of a collaborative, goal-oriented swarm of rescuing robots. Chapter 5 presents the modelling of the swarm and a runnable simulation of it.
- *Autonomic Cloud*: it concerns the modelling of an illustrative autonomic cloud platform whose nodes cooperate to ensure certain requirements. Chapter 6 models the platform and uses it as a motivating example for the static analysis approach.

The last two case studies are taken from the ASCENS project [FP715], where the research activities concerning this thesis were partially conducted.

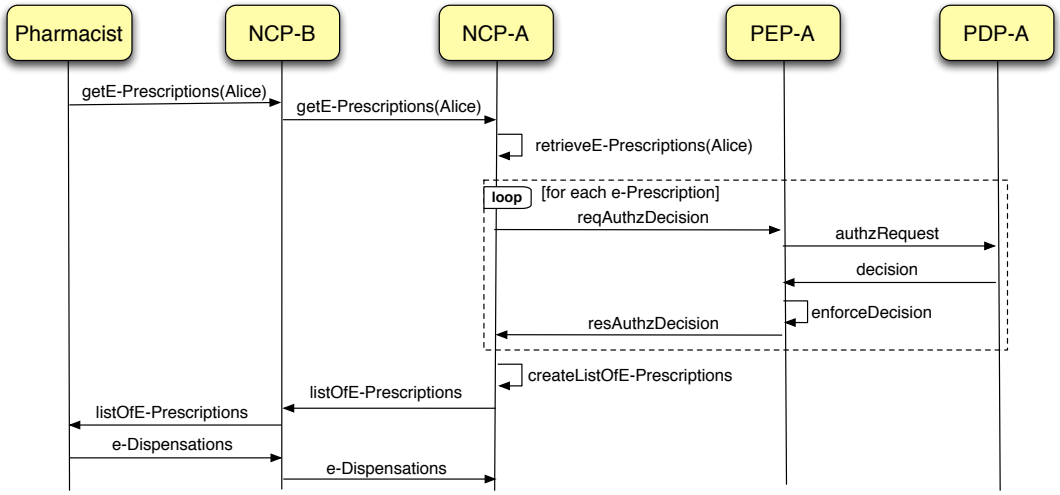


Figure 2.4: E-Health case study: e-Prescription service protocol

In the following, we first outline the e-Health case study (Section 2.3.1), then we introduce the robot-swarm (Section 2.3.2) and the autonomic cloud (Section 2.3.3) ones.

2.3.1 An e-Health Provisioning Service

To improve the effectiveness of healthcare systems, e-Health services aim at allowing healthcare professionals (such as doctors, nurses, pharmacists, etc.) to remotely access and exchange patients data. We outline here a standardised European service solution.

The exchange of patients health data among European points of care (such as clinics, hospitals, pharmacies, etc.) has been pursued by the EU through the large scale pilot project epSOS[5], with the goal of improving healthcare treatments to EU citizens that are abroad. This exchange must respect a set of requirements in order to fulfil country-specific legislations [Eur95, The13] and to enforce the *patient informed consent*, i.e. the patients informed indications pertaining to personal data processing.

These data exchanging services, standardised by epSOS, are currently used by many European countries to facilitate the cross-board interoperability of their healthcare systems [Kov14]. As a case study, we take into account the *electronic prescription* (e-Prescription) service. This service allows EU patients, while staying in a foreign country B participating to the project, to have dispensed a medicine prescribed by a doctor in the country A where the patient is insured. The protocol implemented by this service is illustrated in the message sequence diagram in Figure 2.4. The e-Prescription service helps pharmacists in country B to retrieve (and properly convert) e-Prescriptions from country A; this relies on trusted actors named National Contact Points (NCPs). Therefore, once a pharmacist has identified the patient (Alice), the remote access is requested to the local NCP (NCP-B), which in its own turn contacts the remote NCP (NCP-A). The latter one retrieves the e-Prescriptions of the patient from the national infrastructure and, for each e-Prescription, performs through PEP-A an authorisation check against the patient informed consent. In details, PEP-A asks PDP-A to evaluate the pharmacist request with respect to the e-Prescription and the policies expressing the patient consent. Once all

[5]epSOS (Smart Open Services for European Patients) - http://www.epsos.eu

Eh- #	*Description*
Eh-1	Doctors can write e-Prescriptions
Eh-2	Doctors can read e-Prescriptions
Eh-3	Pharmacists can read e-Prescriptions
Eh-4	Authorised user accesses must be recorded by the system
Eh-5	Patients must be informed of unauthorised access attempts
Eh-6	Data exchanged should be compressed

Table 2.1: Requirements for the e-Prescription service

decisions are enforced by PEP-A, NCP-A creates the list of e-Prescriptions, by transcoding and translating them into the code system and language of the country B. Finally, the pharmacist dispenses the medicine to the patient and updates the e-Prescription, i.e. it returns e-Dispensation documents to the NCPs.

By looking at the epSOS specifications, we can deduce a set of business requirements concerning the access control system managing the e-Prescription service. For instance, it is forbidden to pharmacists to write e-Prescriptions, which is instead obviously granted to a doctor having a specific set of permissions. In Table 2.1, we report in a *closed-world* form (i.e., everything not reported has to be forbidden) the self-explanatory requirements we focus on when we deal with this case study. The first three requirements concern access restrictions, while the others concern additional functionalities that sophisticated access control systems, like the one we will present, can provide.

The specification of the access control system enforcing the requirements of Table 2.1 is reported in Chapter 3, while its analysis is used in Chapter 4 to present the functionalities of the approach we propose.

2.3.2 A Robot-Swarm Disaster Scenario

Robot-swarm [Ben05] is a typical example of autonomic system. It consists of a large collection of robots that collaborate together to achieve desirable objectives, such as robustness to individual failures and enhanced performance. This kind of systems is studied within a large variety of application domains. We consider here a disaster recovery scenario [PBMD15] where a search-and-rescue operation must be performed by a swarm of robots in an hazardous environment.

Figure 2.5 graphically depicts a disaster scenario. The swarm of robots is placed in a deployment area and aims at spreading throughout the given area where some kind of disaster has happened. The goal of the robots is to locate and rescue possible victims. As a matter of fact, a single robot cannot rescue a victim alone: a task-oriented ensemble of robot is needed.

Our focus is not on the design of single robots, but on the behaviour of the robot swarm as a whole. Such behaviour emerges from the collaboration and interactions of single robots, which are all defined by the same specification. Thus, we organise the behaviours of single robots in terms of abstract roles, whose dynamic variation, according to the information sensed from the environment, permits rescuing the victims.

Table 2.2 reports in the form of requirements the expected behaviours a robot playing a certain role has to ensure. Each robot initially plays the *explorer* role in order to look within the arena for victims. When a robot finds a victim, it changes to the *rescuer* role starting the victim rescuing and indicating the victim's position to the other robots. As

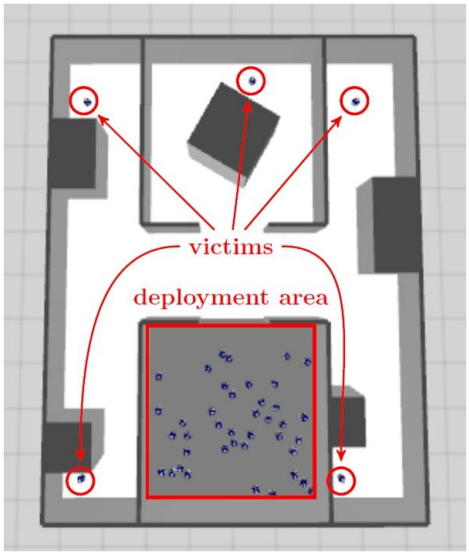


Figure 2.5: Robot-swarm case study: disaster scenario

soon as another robot receives the victim's position, it changes to the *helpRescuer* role going to help other rescuers. During the exploration, in case of critical battery level, a robot changes to the *lowBattery* role to activate the battery charging.

The specification of the case study is given in Chapter 5, together with a Java-based runnable simulation.

Rs- #	*Description*
Rs-1	*Explorer*s look for victims throughout the arena
Rs-2	*Explorer*s notify the position of a not-discovered victim as soon as they find one
Rs-3	*Rescuer*s cooperate in a 4-robot ensemble for rescuing a discovered victim
Rs-4	*HelpRescuer*s move towards the received position of a victim to help other rescuers
Rs-5	*LowBattery*s move towards the deployment area to recharge critical batteries

Table 2.2: Requirements for the disaster scenario (robot roles are in italics)

2.3.3 An Autonomic Cloud Platform

Cloud computing is a well-established technology for the provisioning of IT resources in a dynamic and on-demand fashion. It easily supports the request of new services by adapting its infrastructure to the ever changing load configurations.

The *autonomic cloud* [MVK+15] is a platform-as-a-service computing infrastructure, named Science Cloud, that is formed by a loose collection of voluntarily provided heterogeneous nodes. The individual nodes correspond to (virtual) machines running a *Science Cloud Platform instance* (SCPi), an expressly developed middleware offering peer-to-peer communications among geographically distributed nodes. The cooperative and collaborative interactions of nodes permit offering computational services and ensuring certain quality of service. In this context, the case study considers a *high load* scenario where, for each location-based ensemble, a special application, named APP, running on one or more nodes has the duty of offering computational services to the other nodes.

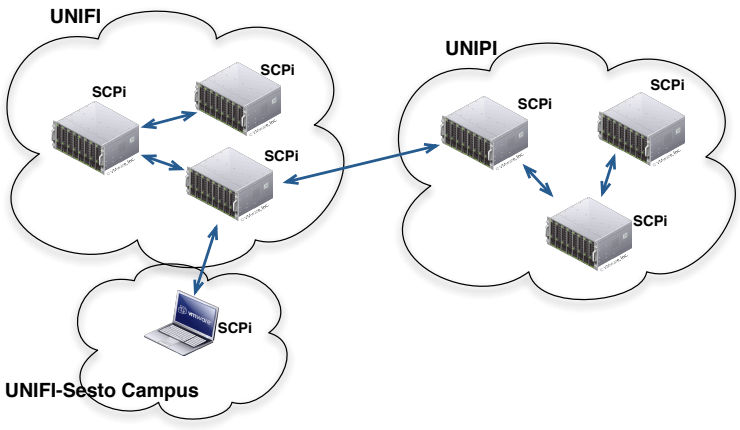


Figure 2.6: Autonomic cloud case study: high-load scenario

A small-size setup of the Science Cloud is depicted in Figure 2.6: a group of SCPis is located at UNIPI and another group at UNIFI, and another one (running on top of a mobile device) at the UNIFI Sesto Campus. For each group, one member has the duty of a gateway, i.e. it collects information about the whole platform and, if necessary, notifying the other members. We assume that, besides a node with role *gateway*, a node running the application APP has the role *server*, while the nodes requesting the computational service have the role *client*. Differently from the robot-swarm case study, here a node cannot change dynamically its role and to each role corresponds a different specification.

Table 2.3 reports in the form of requirements the expected behaviours each role has to ensure. Each location initially has a single (node with role) *server* offering the computational service and a collections of *client*s preparing tasks to be computed. Servers dynamically retrieve tasks from clients by enforcing a confidentiality security policy based on the MAC model. Specifically, each server and client has associated a security level, i.e. high and low, and a server with level low cannot retrieve tasks from clients with level high. To ensure the availability of the computational service, servers can dynamically spawn a co-located server when their load is over a certain threshold. Besides the associated security level, it is requested to clients to incrementally enumerate the tasks. To avoid overloading of clients, the space for storing tasks is assumed to be limited. Finally, *gateway*s collect information from clients about the tasks retrieved by servers.

The specification of the case study is given in Chapter 6 and its runtime behaviour is used as motivating example for a static analysis approach we will present.

Cl- #	*Description*
Cl-1	*Server*s offer a computational service
Cl-2	*Server*s enforce a MAC security policy on the tasks to compute
Cl-3	*Server*s ensure the availability of the service by possibly spawning a new server
Cl-4	*Client*s locally enumerate and store tasks to be computed
Cl-5	*Client*s can store tasks provided that their load is under a given threshold
Cl-6	*Gateway*s collect information on the computed tasks

Table 2.3: Requirements of the High Load scenario (node roles are in italics)

Chapter 3

The FACPL Language

ABAC systems are nowadays a state-of-the-art solution to design secure computing systems. A well-established specification approach for ABAC is that based on the XACML standard. However, as pointed out in the previous chapter, its verbose XML-based syntax and the lack of a formal semantics do not permit the specification of compact policies and the development of rigorous analysis techniques.

In this chapter, we introduce *Formal Access Control Policy Language (FACPL)*, a formal language that defines a core, yet expressive syntax for the specification of attribute-based access control policies. It is partially inspired by XACML (with which it shares the main traits of the policy structure), but it refines some aspects of XACML and introduces novel features from the access control literature. Indeed, the FACPL formal semantics, which is defined in a denotational style, permits a precise management of intricate aspects of access controls like, e.g., management of missing attributes (i.e., attributes requested by a policy but not provided by the request to authorise) and formalisation of combining algorithms (i.e., strategies to resolve conflictual decisions that policy evaluation can generate). To effectively deploy the language, FACPL is equipped with a fully-implemented Java-based toolchain. The key software tool is an Eclipse-based *Integrated Development Environment (IDE)* that offers a tailored development environment for FACPL policies.

The FACPL language, together with its denotational semantics, provides solid formal foundations that can be used as the basis of the security model of ABAC systems. In particular, the semantics has been exploited towards a twofold direction: to provide a precise formalisation of the evaluation process, thus to drive the implementation of the FACPL Java-based library, and to support the formalisation and verification of properties on policies. In particular, by relying on the semantics, the constraint-based representation of FACPL policies (which is presented in Chapter 4) has been proved sound.

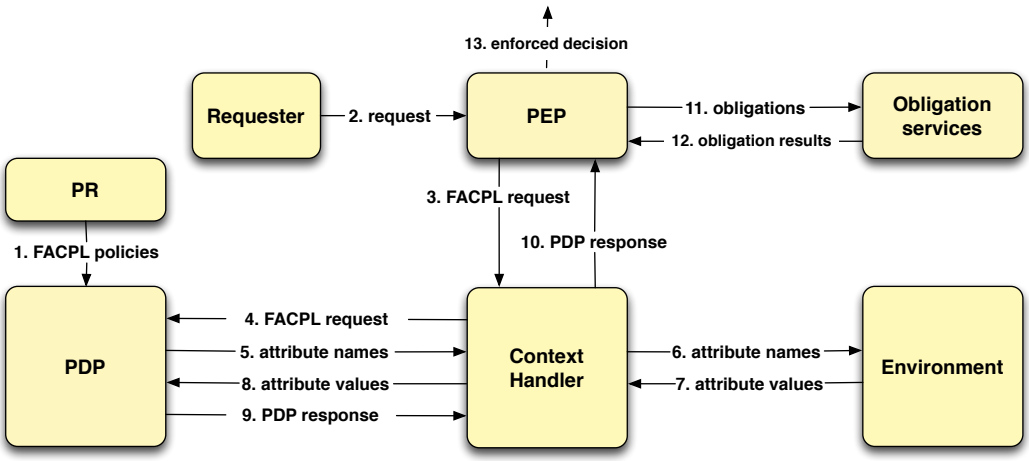


Figure 3.1: FACPL evaluation process

Structure of the chapter. The rest of this chapter is organised as follows. Section 3.1 overviews the FACPL evaluation process. Section 3.2 presents the syntax of FACPL. Section 3.3 informally explains the semantics of FACPL. Section 3.4 describes the FACPL policies defining the access control system of the e-Health case study presented in Section 2.3.1. Section 3.5 provides the formal account of the FACPL semantics. Section 3.6 outlines the main features of the FACPL toolchain. Section 3.7 concludes with some final remarks.

3.1 Evaluation Process

The evaluation process at the basis of FACPL-based access control systems is the policy-based evaluation process introduced in Section 2.1.3. Such a process has been tailored according to the specific features of FACPL and is shown in Figure 3.1.

The evaluation process assumes that system resources are paired with one or more FACPL policies, which define the credentials necessary to gain access to such resources. The PR stores the policies and makes them available to the PDP (step 1).

When a request is received by the PEP (step 2), the credentials contained in the request are encoded as a sequence of *attribute* elements (i.e., name-value pairs) forming a FACPL request (step 3).

The *context handler* sends the request to the PDP (step 4), by possibly adding environmental attributes, e.g. request receiving time, that may be used in the evaluation.

The PDP *authorisation process* computes the *PDP response* for the request by checking the attributes, that may belong either to the request or to the environment (steps 5-8), against the controls contained in the policies. Hence, an appropriate use of the context handler allows the authorisation process to depend on the evaluation environment and, e.g., to address location-based or data-dependent controls. The PDP response (steps 9-10) contains an authorisation *decision* and possibly some *obligations*.

A decision is the authorisation calculated for an access request and is one among permit, deny, not-app and indet. Their corresponding meanings follow

- permit: the access request is granted;

- deny: the access request is forbidden;
- not-app: there is no policy that applies to the access request;
- indet: some errors have occurred during the evaluation of the access request[1].

We will see that policies can automatically manage errors by using operators that combine, according to different strategies, indet decisions with the others.

Obligations, that is additional enforcement actions connected to authorisation decisions, are discharged by the PEP through appropriate *obligation services* (steps 11-12). In practice, obligations usually correspond to, e.g., updating a log file, sending a message or executing a command. The *enforcement process* performed by the PEP determines the *enforced decision* (step 13) on the basis of the obligation results. This decision could differ from the PDP one and is the overall outcome of the evaluation process.

Finally, it is worth noticing that obligations are intended to be actions executed during the enforcement process, rather than constraints on the future usage of the possibly granted accesses. The obligations used by FACPL are those used by XACML and proposed in [Slo94]. The management of obligations on future access usage is a topic worth to be studied in the context of, e.g., usage control [LMM10], and is indeed one of the research directions we want to further investigate (see Chapter 8 for more details).

3.2 Syntax

FACPL policies are hierarchically structured lists of elements containing controls on FACPL request attributes. Together with permit or deny decisions, policies define the combining algorithms to be used in their evaluation and the obligations for the enforcement process.

Formally, the syntax of FACPL is defined in Table 3.1. It is given through EBNF-like grammars, where as usual the symbol $?$ stands for optional items, $*$ for (possibly empty) sequences, and $+$ for non-empty sequences.

A top-level term is a *Policy Authorisation System (PAS)* encompassing the specifications of a PEP and a PDP. The PEP is defined in terms of the *enforcement algorithm* applied for establishing how decisions have to be enforced, e.g. if only decisions permit and deny are admissible, or also not-app and indet can be returned. The PDP is instead defined by a sequence of policies $Policy^+$ and an algorithm Alg for combining the results of the evaluation of these policies.

A *policy* can be a basic authorisation *rule* $(Effect\ \ \mathsf{target}: Expr\ \ \mathsf{obl}: Obligation^*)$ or a *policy set* $\{Alg\ \ \mathsf{target}: Expr\ \ \mathsf{policies}: Policy^+\ \ \mathsf{obl}: Obligation^*\}$ collecting rules and other policy sets, so that it defines policy hierarchies. A policy set specifies a *target*, that is an expression indicating the set of access requests to which the policy applies, a list of obligations, that defines mandatory or optional actions to be discharged by the enforcement process, a sequence of enclosed policies and an algorithm, that is used for combining the enclosed policies. A rule specifies an *effect*, that is the permit or deny decision returned when the rule is successfully evaluated, a target, that refines the one of the enclosing policy, and a list of obligations. Notably, obligations may be missing.

[1] Notably, the formal specification of FACPL, for the sake of presentation, only addresses a single indeterminate value, rather than the extended indeterminate values used by XACML. However, the FACPL supporting tools can also deal with these extended indeterminate values; see Section 3.6 for further details.

Policy Authorisation Systems	$PAS ::=$ (pep : $EnfAlg$ pdp : PDP)
Enforcement algorithms	$EnfAlg ::=$ base | deny-biased | permit-biased
Policy Decision Points	$PDP ::= \{Alg$ policies : $Policy^+\}$
Combining algorithms	$Alg ::=$ p-over$_\delta$ | d-over$_\delta$ | d-unless-p$_\delta$ | p-unless-d$_\delta$ | first-app$_\delta$ | one-app$_\delta$ | weak-con$_\delta$ | strong-con$_\delta$
Fulfilment strategies	$\delta ::=$ greedy | all
Policies	$Policy ::= (Effect$ target : $Expr$ obl : $Obligation^*$) | $\{Alg$ target : $Expr$ policies : $Policy^+$ obl : $Obligation^*\}$
Effects	$Effect ::=$ permit | deny
Obligations	$Obligation ::= [Effect \ \ ObType \ \ PepAction(Expr^*)]$
Obligation types	$ObType ::=$ M | O
Expressions	$Expr ::= Name \mid Value$ | and$(Expr, Expr)$ | or$(Expr, Expr)$ | not$(Expr)$ | equal$(Expr, Expr)$ | in$(Expr, Expr)$ | greater-than$(Expr, Expr)$ | add$(Expr, Expr)$ | subtract$(Expr, Expr)$ | divide$(Expr, Expr)$ | multiply$(Expr, Expr)$
Attribute names	$Name ::= Identifier/Identifier$
Literal values	$Value ::=$ true | false | $Double$ | $String$ | $Date$
Requests	$Request ::= (Name, Value)^+$

Table 3.1: Syntax of FACPL

An $attribute\ name$ is used to refer to the value of an attribute. This can either be contained in the request or retrieved from the environment by the context handler (steps 5-8 in Figure 3.1). To group attributes under categories, FACPL uses structured names of the form $Identifier/Identifier$, where the first identifier stands for a category name and the second for an attribute name. For example, the structured name subject/role represents the value of the attribute role within the category subject. Categories permit a fine-grained classification of attributes, varying from the classical categories of access control, i.e. *subject*, *resource* and *action*, to possibly application-dependent ones.

Expressions are built from attribute names and *literal* values, i.e. booleans, doubles, strings, and dates, by using standard operators. As usual, string values are written as sequences of characters delimited by double quotes. The expression syntax does not explicitly take into account the types of attribute names, because policies must be able to evaluate all possible access requests. However, at evaluation-time errors will be generated, and possibly managed, when expression operators are applied to arguments of unexpected types. Notably, FACPL supporting tools implement a type system to statically check how expression operators are combined, however it abstracts from the actual types assumed by attribute names, because they can represent any value. Moreover, the syntax of expressions accepted by the tools can be extended with additional operators (see Section 3.6 for further details).

A *combining algorithm* aims at resolving conflicts among the decisions resulting from policy evaluations, e.g. whenever both decisions permit and deny occur. The reported algorithms offer various strategies (e.g., the p-over$_\delta$ algorithm stating that 'decision permit

PDP responses	$PDPResponse$ ::=	$\langle\ Decision\quad FObligation^*\rangle$
Decisions	$Decision$ ::=	permit | deny | not-app | indet
Fulfilled obligations	$FObligation$ ::=	$[\ ObType\quad PepAction(Value^*)\ ]$

Table 3.2: Auxiliary syntax for FACPL responses

takes precedence over the others') and can be specialised by choosing different strategies for the fulfilment of obligations (e.g., the greedy strategy stating that 'only the obligations resulting from the evaluated policies are returned'). Note that algorithm names use 'p' and 'd' as shortcuts for permit and deny, respectively.

An *obligation* $[\ Effect\quad ObType\quad PepAction(Expr^*)\ ]$ specifies an applicability effect, a type, i.e. mandatory (M) or optional (O), and the identifier and the arguments of an action to be performed by the PEP. The set of action identifiers accepted by the PEP can be chosen, from time to time, according to the specific application (therefore, $PepAction$ is intentionally left unspecified). Action arguments are expressions.

A *request* consists of a non-empty sequence of *attributes*, i.e. name-value pairs, that enumerate request credentials in the form of literal values. Attributes are organised under categories by exploiting their structured names. *Multi-valued attributes*, i.e. names associated to a set of values, are rendered as multiple attributes sharing the same name.

The responses resulting from the evaluation of a FACPL request are written using the auxiliary syntax reported in Table 3.2.

The two-stage evaluation process described in Section 3.1 produces two different kinds of responses: *PDP responses* and *decisions* (i.e. responses by the PEP). The former ones, in case of decision permit and deny, pair the decision with a (possibly empty) sequence of fulfilled obligations. A *fulfilled obligation* is a pair made of a type (i.e., M or O) and an action whose arguments are values.

In the sequel, to simplify notations, we omit the keyword preceding a sub-term generated by the grammar in Table 3.1 whenever the sub-term is missing or is the expression true. Thus, e.g., the rule (deny target : true obl :) will be simply written as (deny). Moreover, when in the $PDPResponse$ the sequence of fulfilled obligations is empty, we sometimes write $Decision$ instead of $\langle Decision\rangle$.

3.3 Informal Semantics

We now informally explain how the FACPL linguistic constructs are dealt with in the evaluation process of access requests described in Section 3.1. We first present the PDP authorisation process and then the PEP enforcement process.

When the PDP receives an access request, first it evaluates the request on the basis of the available policies. Then, it determines the resulting decision by combining the decisions returned by these policies through the top-level combining algorithm.

The evaluation of a policy with respect to a request starts by checking its applicability to the request, which is done by evaluating the expression defining its target. Let us suppose that the applicability holds, i.e. the expression evaluates to true. In case of rules, the rule effect is returned. In case of policy sets, the result is obtained by evaluating the contained policies and combining their evaluation results through the specified algorithm.

In both cases, the evaluation ends with the fulfilment of the enclosed obligations. Let us suppose now that the applicability does not hold. If the expression evaluates to false, the policy evaluation returns not-app, while if the expression returns an error or a non-boolean value, the policy evaluation returns indet. Clearly, a policy with target expression true (resp., false) applies to all (resp., no) requests.

Evaluating expressions amounts to apply operators and to *resolve* the attribute names occurring within, that is to determine the value corresponding to each such name. If this is not possible, i.e. an attribute with that name is missing in the request and cannot be retrieved through the context handler, the special value $\bot$ is returned. This value can be exploited to enforce different strategies for managing the absence of attributes. In details, dealing with $\bot$ as an error would mean that all occurring attributes must be present in the request, otherwise the policy evaluation immediately returns indet. Instead, as chosen by the FACPL semantics, dealing with $\bot$ in a way similar to value false allows attributes to be missing without always generating errors. Indeed, using $\bot$ rather than an error value for dealing with missing attributes permits enforcing an appropriate management of requests only containing limited sets of attributes and reasoning on the role of missing attributes in the policy evaluation (see the formal analysis in Chapter 4 for further details).

The evaluation of expressions takes into account the types of the operators' arguments, and possibly returns the special values $\bot$ and error. In details, if the arguments are of the expected type, the operator is applied, else, i.e. at least one argument is error, error is returned; otherwise, i.e. at least one argument is $\bot$ and none is error, $\bot$ is returned. The operators and and or enforce a different treatment of these special values. Specifically, and returns true if both operands are true, false if at least one operand is false, $\bot$ if at least one operand is $\bot$ and none is false or error, and error otherwise (e.g. when an operand is not a boolean value). The operator or is the dual of and. Hence, and and or may mask $\bot$ and error. Instead, the unary operator not only swaps values true and false and leaves $\bot$ and error unchanged. In the following, we use operators and and or in infix notation, and assume that they are commutative and associative, and that operator and takes precedence over or.

The evaluation of a policy ends with the fulfilment of all obligations whose applicability effect coincides with the decision calculated for the policy. The fulfilment of an obligation consists in evaluating all the expression arguments of the enclosed action. If an error occurs, the policy decision is changed to indet. Otherwise, the fulfilled obligations are paired with the policy decision to form the PDP response.

Evaluating a policy set requires the application of the specified combining algorithm. Given a sequence of policies in input, the combining algorithms prescribe the sequential evaluation of the given policies and behave as follows:

- p-over$_\delta$ (d-over$_\delta$ is specular): if the evaluation of a policy returns permit, then the result is permit. In other words, permit takes precedence, regardless of the result of any other policy. Instead, if at least one policy returns deny and all others return not-app or deny, then the result is deny. If all policies return not-app, then the result is not-app. In the remaining cases, the result is indet.

- d-unless-p$_\delta$ (p-unless-d$_\delta$ is specular): similarly to p-over$_\delta$, this algorithm gives precedence to permit over deny, but it always returns deny in all the other cases.

- first-app$_\delta$: the algorithm returns the evaluation result of the first policy in the se-

quence that does not return not-app, otherwise the result is not-app.

- one-app$_\delta$: when exactly one policy is applicable, the result of the algorithm is that of the applicable policy. If no policy applies, the algorithm returns not-app, while if more than one policy is applicable, it returns indet.
- weak-con$_\delta$: the algorithm returns permit (resp., deny) if some policies return permit (resp., deny) and no other policy returns deny (resp., permit); if both decisions are returned, the algorithm returns indet. If policies only return not-app or indet, then indet, if present, takes precedence.
- strong-con$_\delta$: this algorithm is the stronger version of the previous one, in the sense that to obtain permit (resp., deny) all policies have to return permit (resp., deny), otherwise indet is returned. If all policies return not-app then the result is not-app.

The algorithms described in the first four items above are those popularised by XACML. They combine decisions according to a given precedence criterium or to policy applicability. The remaining two algorithms, instead, are borrowed from [LWQ+09] and compute the combined decision by achieving different forms of consensus.

If the resulting decision is permit or deny, each algorithm also returns the sequence of fulfilled obligations according to the chosen fulfilment strategy δ. There are two possible strategies. The all strategy requires evaluation of all policies in the input sequence and returns the fulfilled obligations pertaining to all decisions. Instead, the greedy strategy prescribes that, as soon as a decision is obtained that cannot change due to evaluation of subsequent policies in the input sequence, the execution halts. Hence, the result will not consider the possibly remaining policies and only contains the obligations already fulfilled. Therefore, the fulfilment strategies mainly affect the amount of fulfilled obligations possibly returned. Notice that the greedy strategy may significantly improve the policy evaluation performance and that it is inspired to the XACML management of obligations. Instead, the all strategy may require additional workload, but it ensures that all the policies and their obligations are always taken into account.

As last step, the calculated PDP response is sent to the PEP for the enforcement. To this aim, the PEP must discharge all obligations and decide, by means of the chosen enforcement algorithm, how to enforce decisions not-app and indet. In particular, the deny-biased (resp., permit-biased) algorithm enforces permit (resp., deny) only when all the corresponding obligations are correctly discharged, while enforces deny (resp., permit) in all other cases. Instead, the base algorithm leaves all decisions unchanged but, in case of decisions permit and deny, enforces indet if an error occurs while discharging obligations. This means that obligations not only affect the authorisation process due to their fulfilment, but also the enforcement one. It is worth noticing that errors caused by optional obligations, i.e. with type O, are safely ignored.

3.4 FACPL at work on the e-Health Case Study

We now use the FACPL linguistic abstractions to define an access control systems ensuring the requirements for the e-Health case study reported in Table 2.1. Indeed, such a system has to prevent unauthorised access to patient data and hence to ensure their confidentiality and integrity. The specification of this FACPL-based access control system is introduced

bottom-up, from single rules to whole policies, thus illustrating in a step-by-step fashion the combination strategies that could be pursued and their effects.

The system resources to protect via the access control system are *e-Prescriptions*. The access control rules need to deal with requester credentials, i.e. doctor and pharmacist roles, along with their assigned permissions, and with read or write actions.

Requirement (Eh-1), allowing doctors to write e-Prescriptions, can be formalised as a *positive* FACPL rule (i.e., with effect permit) as follows

```
( permit  target : equal(subject/role, "doctor") and equal(action/id, "write")
                   and in("e-Pre-Write", subject/permission)
                   and in("e-Pre-Read", subject/permission))
```

The rule target[2] checks if the requester role is doctor, if the action is write, and if the permissions include those for writing and reading an e-Prescription. Notably, that the resource type is equal to e-Prescription will be controlled by the target of the policy enclosing the rule. Due to the hierarchical processing of FACPL elements, this target is enough to ensure that the rule will only be applied to e-Prescriptions.

Requirement (Eh-2) can be expressed like the previous one: it differs for the action identifier and for the required permissions, i.e. only e-Pre-Read. Requirement (Eh-3) only differs from the second one for the role value.

These three rules, modelling Requirements (Eh-1), (Eh-2) and (Eh-3), can be combined together in a policy set whose target specifies the check on the resource type e-Prescription[3]. Since all granted requests are explicitly authorised, choosing the p-over$_{\text{all}}$ algorithm as combining strategy seems a natural choice. Let thus Policy (Eh-A) be defined as follows

```
{ p-over_all
   target : equal(resource/type, "e-Prescription")
   policies : ( permit  target : equal(subject/role, "doctor")and equal(action/id, "write")
                                 and in("e-Pre-Write", subject/permission)
                                 and in("e-Pre-Read", subject/permission))
              ( permit  target : equal(subject/role, "doctor")and equal(action/id, "read")
                                 and in("e-Pre-Read", subject/permission))
              ( permit  target : equal(subject/role, "pharmacist")and equal(action/id, "read")
                                 and in("e-Pre-Read", subject/permission))
   obl : [ permit  M  log(system/time, resource/type, subject/id, action/id) ] }
```

(Eh-A)

Policy (Eh-A) reports not only access controls but also an obligation formalising Requirement (Eh-4) about the logging of each authorised access. The arguments of the obligation action are separated by commas to increase their readability.

Let us now consider a FACPL request and evaluate it with respect to Policy (Eh-A). For the sake of presentation, hereafter we write $A \triangleq t$ to assign the symbolic name A to the term t. Let us suppose that doctor Dr. House wants to write an e-Prescription; the

[2] To improve code readability, we use the infix notation for operators, a textual notation for permissions and an additional check on the subject role. Of course, in a setting with semantically different roles, a standardised permission-based coding, as e.g. HL7 (http://www.hl7.org), should be used for defining role checks.

[3] Again, to improve code readability, the resource is encoded as text; in a real application, for interoperability reasons, the LOINC (http://loinc.org/) universal code system for clinical data should be used.

corresponding request is defined as follows

$$\begin{aligned}\mathsf{req1} \triangleq\ & (\mathsf{subject/id}, \text{“Dr. House”})\ (\mathsf{subject/role}, \text{“doctor”})\ (\mathsf{action/id}, \text{“write”})\\ & (\mathsf{resource/type}, \text{“e-Prescription”})\ (\mathsf{subject/permission}, \text{“e-Pre-Read”})\\ & (\mathsf{subject/permission}, \text{“e-Pre-Write”})\ \ldots\end{aligned}$$

where attributes are organised into the categories *subject*, *resource* and *action*. Additional attributes possibly included in the request are omitted because they are not relevant for this evaluation. Notice that subject/permission is a multi-valued attribute and it is properly handled in the previous rules by using the in operator, which verifies the membership of its first argument to the set that constitutes its second argument.

The authorisation process of req1 returns a permit decision. In fact, the request matches the policy target, as the resource type is e-Prescription, and exposes all the permissions required in the first rule for the write action and the doctor role. The response, that is a permit including a log obligation, is defined, e.g., as follows

$$\langle\ \mathsf{permit}\ [\,\mathsf{M}\ \ \mathsf{log}(\text{2016-01-22 10:15:12}, \text{“e-Prescription”}, \text{“Dr. House”}, \text{“write”})\,]\rangle$$

The fulfilled obligation indicates that the PDP succeeded in retrieving and evaluating all the attributes occurring within the arguments of the action; runtime information, such as the current time, is retrieved through the context handler.

The evaluation of req1 returns the expected result. We might be led to believe that due to the simplicity of Policy (Eh-A), this is true for all requests. However, this correctness property cannot be taken for granted as, in general, even though the meaning of a rule is straightforward, this may not be the case for a combination of rules. Depending on the chosen combination strategy, some unexpected results can arise. For example, a request from a pharmacist for a write action on an e-Prescription must be forbidden. In fact, this behaviour is not explicitly allowed (see Table 2.1), hence due to the *closed-world* assumption it has to be forbidden. However, the corresponding request

$$\begin{aligned}\mathsf{req2} \triangleq\ & (\mathsf{subject/id}, \text{“Dr. Wilson”})\ (\mathsf{subject/role}, \text{“pharmacist”})\ (\mathsf{action/id}, \text{“write”})\\ & (\mathsf{resource/type}, \text{“e-Prescription”})\ (\mathsf{subject/permission}, \text{“e-Pre-Read”})\ \ldots\end{aligned}$$

would evaluate to not-app. In fact, all enclosed rules do not apply (i.e., their targets do not match) and the resulting not-app decisions are combined by the p-over$_{\mathsf{all}}$ algorithm to not-app as well. Therefore, the enforcement algorithm of the PEP is entrusted with the task of taking the final decision for request req2. Even though this is correct in a setting where the PEP is well-defined, e.g. the epSOS system, it is not recommended when design assumptions on the PEP implementation are missing. In fact, a biased algorithm might transform not-app into permit, possibly causing unauthorised accesses.

To prevent not-app decisions to be returned by the policy, we can replace the combining algorithm of Policy (Eh-A) with the d-unless-p$_{\mathsf{all}}$ one. This implies that deny is taken as the default decision and is returned whenever no rule returns permit. Alternatively, we can obtain the same effect by using a policy set defined as the combination, through the p-over$_{\mathsf{all}}$ algorithm, of Policy (Eh-A) and a rule forbidding all accesses. This rule is simply defined as (deny): the absence of the target and the *negative* effect means that it always

returns deny. Now, let Policy (Eh-B) be defined as

```
{ p-over_all
    policies :
      { ... Policy (Eh-A) ... }
      (deny)
    obl : [ deny  M  mailTo(resource/patient-mail, "Data request by unauthorised subject") ]
          [ permit  O  compress( ) ] }
```
(Eh-B)

Policy (Eh-B) reports two obligations formalising, respectively, the last two requirements of Table 2.1: (i) a patient is informed about unauthorised attempts to access her data and (ii) if possible, data are exchanged in compressed form. Notably, the type 'optional' is exploited so that compressed exchanges are not strictly required but, e.g., only whenever the corresponding service is available.

Policy (Eh-B) can be used as a basis for the definition of the *patient informed consent* (see Section 2.3.1). For instance, Alice's policy for the management of her health data could be simply obtained by adding target : equal("Alice", resource/patient-id) to Policy (Eh-B), i.e. a check on the patient identifier to which the policy applies. In this way, Alice grants access to her e-Prescription data to the healthcare professionals that satisfy the requirements expressed in her consent policy. Another patient expressing a more restrictive consent, where e.g. writing of e-Prescriptions is disabled, will have a similar policy set where the rule modelling Requirement (Eh-1) is not included. In a more general perspective, the PDP could have a policy set for each patient, that encloses the policies expressing the consent explicitly signed by the patient. This is the approach followed, e.g., in the Austrian e-Health platform[4].

The FACPL-based access control system just presented highlights the mnemonic and compact notations provided by FACPL for the specification of access control policies. As a matter of fact, the XACML policy reported in Listing 2.1 could be defined in FACPL in few lines. However, as shown before, it can be challenging to identify unexpected authorisations and to determine whether policy fixes affect authorisations that should not be altered. The specification and combination of (a large number of) access control policies is indeed an error-prone task that has to be supported by effective analysis techniques.

Therefore, we equip FACPL with a formal semantics and then, as described in Chapter 4, we define a constraint-based analysis enabling the (automated) verification of multiple properties on policies.

3.5 Formal Semantics

In this section we present the formal semantics of FACPL by formalising the evaluation process introduced in Section 3.1 and detailed in Section 3.3. The semantics is defined by following a denotational approach which means that

- we introduce some semantic functions mapping each FACPL syntactic construct to an appropriate *denotation*, that is an element of a semantic domain representing the meaning of the construct;
- the semantic functions are defined in a *compositional* way, so that the semantic of each construct is formulated in terms of the semantics of its sub-constructs.

[4]For additional details see the Austria's ELGA system — http://www.elga.gv.at/

Syntactic category	Generic synt. elem.	Semantic function	Syntactic domain	Semantic domain
Attribute names	n		$Name$	
Literal values	v		$Value$	
Requests	req	$\mathcal{R}$	$Request$	$R \triangleq Name \to (Value \cup 2^{Value} \cup \{\bot\})$
Expressions	$expr$	$\mathcal{E}$	$Expr$	$R \to Value \cup 2^{Value} \cup \{\mathsf{error}, \bot\}$
Effects	e		$Effect$	
Obligation Types	t		$ObType$	
Pep Actions	$pepAct$		$PepAction$	
fulfilled obligations	fo		$FObligation$	
Obligations	o	$\mathcal{O}$	$Obligation$	$R \to FObligation \cup \{\mathsf{error}\}$
PDP Responses	res		$PDPReponse$	
Policies	p	$\mathcal{P}$	$Policy$	$R \to PDPReponse$
Policy Decision Points	pdp	$\mathcal{P}dp$	PDP	$R \to PDPReponse$
Combining algorithms	a	$\mathcal{A}$	$Alg \times Policy^+$	$R \to PDPReponse$
Decisions	dec		$Decision$	
Enforcement algorithms	ea	$\mathcal{E}A$	$EnfAlg$	$PDPReponse \to Decision$
Policy Auth. System	pas	$\mathcal{P}as$	PAS	$Request \to Decision$

Table 3.3: Correspondence between syntactic and semantic domains

To this purpose, we specify a family of semantic functions mapping each syntactic domain to a specific semantic domain. These functions are inductively defined on the FACPL syntax through appropriate semantic clauses following a 'point-wise' style. For instance, on the syntactic domain *Policy* representing all FACPL policies, we formalise the function $\mathcal{P}$ that defines a semantic domain mapping FACPL requests to PDP responses.

In the sequel, we convene that the application of the semantic functions is left-associative, omits parenthesis whenever possible, and surrounds syntactic objects with the emphatic brackets $[\![$ and $]\!]$ to increase readability. For instance, $\mathcal{E}[\![n]\!]r$ stands for $(\mathcal{E}(n))(r)$ and indicates the application of the semantic function $\mathcal{E}$ to (the syntactic object) n and (the semantic object) r. We also assume that each nonterminal symbol in Tables 3.1 and 3.2 (defining the FACPL syntax) denominates the set of constructs of the syntactic category defined by the corresponding EBNF rule, e.g. the nonterminal $Policy$ identifies the set of all FACPL policies. The used notations are summarised in Table 3.3 (the missing semantic domains coincide with the corresponding syntactic ones).

In the rest of this section we detail the semantics of requests (Section 3.5.1), PDP (Sections 3.5.2 and 3.5.3), PEP (Section 3.5.4), PAS (Section 3.5.5) and we conclude with some properties of the semantics (Section 3.5.6).

3.5.1 Requests

The meaning of a request[5] is a function of the set $R \triangleq Name \to (Value \cup 2^{Value} \cup \{\bot\})$, that is a total function that maps attribute names to either a literal value, or a set of values (in case of multi-valued attributes), or the special value $\bot$ (if the value for an attribute name is missing). The mapping from a request to its meaning is given by the semantic

[5]For simplicity's sake, here we assume that, when the evaluation of a request takes place, the original request has been already enriched with the information that would be retrieved at run-time through the Context Handler (steps 5-8 in Figure 3.1).

function $\mathcal{R} : \mathit{Request} \to R$, defined as follows:

$$\begin{aligned} \mathcal{R}[\![(n', v')]\!]n &= \begin{cases} v' & \texttt{if } n = n' \\ \bot & \texttt{otherwise} \end{cases} \\ \mathcal{R}[\![(n_i, v_i)^+(n', v')]\!]n &= \begin{cases} \mathcal{R}[\![(n_i, v_i)^+]\!]n \Cup v' & \texttt{if } n = n' \\ \mathcal{R}[\![(n_i, v_i)^+]\!]n & \texttt{otherwise} \end{cases} \end{aligned} \tag{S-1}$$

The semantics of a request, which is a function $r \in R$, is thus inductively defined on the length of the request. To deal with multi-valued attributes we introduce the operator $\Cup$, which is straightforwardly defined by case analysis on the first argument as follows

$$v \Cup v' = \{v, v'\} \qquad V \Cup v' = V \cup \{v'\} \qquad \bot \Cup v' = v'$$

where we let $V \in 2^{\mathit{Value}}$. Specifically, it composes an element of the set $\mathit{Value} \cup 2^{\mathit{Value}} \cup \{\bot\}$, which is returned by $\mathcal{R}[\![(n_i, v_i)^+]\!]n$, with a literal value in Value.

Remark 3.1. *In the definition of the semantic function $\mathcal{R}$, by making use of the abstraction notation from the λ-calculus to denote functions, the clause for a request formed by a single attribute could be expressed as follows*

$$\mathcal{R}[\![(n', v')]\!] \;=\; \lambda m.\; \mathit{if}\ [m = n']\ \mathit{then}\ v'\ \mathit{else}\ \bot\ .$$

From the principle of extensional equality *among functions, it follows that the returned function r is equivalent to that previously defined point-wise on the elements of its domain Name, as it can be easily seen by applying r to an argument n. For the sake of presentation, although in general we could define the semantic functions using the λ-notation, we prefer to use the more compact and intuitive 'point-wise definition style'.*

3.5.2 Policy Decision Process

We start defining the semantics of expressions and obligations that will be then exploited for defining the semantics of policies.

In Table 3.4 we report (an excerpt of) the clauses defining the function $\mathcal{E} : \mathit{Expr} \to (R \to \mathit{Value} \cup 2^{\mathit{Value}} \cup \{\mathsf{error}, \bot\})$ modelling the semantics of expressions. This means that the semantics of an expression is a function of the form $R \to \mathit{Value} \cup 2^{\mathit{Value}} \cup \{\mathsf{error}, \bot\}$ that, given a request, returns a literal value, or a set of values, or the special value $\bot$, or an error (e.g. when an argument of an operator has unexpected type). The evaluation order of sub-expressions is not relevant, as they do not generate side-effects.

The first raw of the table contains the clauses for basic expressions, i.e. attribute names and literal values. The semantics of the expression formed by a name n is a function that, given a (semantic) request r in input, returns the value that r associates to n. This is written as the clause $\mathcal{E}[\![n]\!]r = r(n)$. Similarly, the case of a value v is a function that always returns the value itself, that is the clause $\mathcal{E}[\![v]\!]r = v$.

The remaining clauses in Table 3.4 present (an excerpt of) the semantics of expression operators. In particular, each clause, one for each operator, uses straightforward semantic operators for composing denotations (e.g. $=$ corresponds to equal), and enforces the management strategy for the special values $\bot$ and error. They establish that error takes precedence over $\bot$ and is returned every time the operator arguments have unexpected types; whereas $\bot$ is returned when at least an argument is $\bot$ and there is no error. Notably,

$$\mathcal{E}[\![n]\!]r = r(n)$$

$$\mathcal{E}[\![v]\!]r = v$$

$$\mathcal{E}[\![\mathsf{or}(expr_1, expr_2)]\!]r = \begin{cases} \mathsf{true} & \texttt{if } \mathcal{E}[\![expr_1]\!]r = \mathsf{true} \vee \mathcal{E}[\![expr_2]\!]r = \mathsf{true} \\ \mathsf{false} & \texttt{if } \mathcal{E}[\![expr_1]\!]r = \mathcal{E}[\![expr_2]\!]r = \mathsf{false} \\ \bot & \texttt{if } \mathcal{E}[\![expr_i]\!]r = \bot \wedge \mathcal{E}[\![expr_j]\!]r \in \{\mathsf{false}, \bot\} \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathsf{not}(expr)]\!]r = \begin{cases} \mathsf{true} & \texttt{if } \mathcal{E}[\![expr]\!]r = \mathsf{false} \\ \mathsf{false} & \texttt{if } \mathcal{E}[\![expr]\!]r = \mathsf{true} \\ \bot & \texttt{if } \mathcal{E}[\![expr]\!]r = \bot \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathsf{and}(expr_1, expr_2)]\!]r = \begin{cases} \mathsf{true} & \texttt{if } \mathcal{E}[\![expr_1]\!]r = \mathcal{E}[\![expr_2]\!]r = \mathsf{true} \\ \mathsf{false} & \texttt{if } \mathcal{E}[\![expr_1]\!]r = \mathsf{false} \vee \mathcal{E}[\![expr_2]\!]r = \mathsf{false} \\ \bot & \texttt{if } \mathcal{E}[\![expr_i]\!]r = \bot \wedge \mathcal{E}[\![expr_j]\!]r \in \{\mathsf{true}, \bot\} \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathsf{equal}(expr_1, expr_2)]\!]r = \begin{cases} (\mathcal{E}[\![expr_1]\!]r = \mathcal{E}[\![expr_2]\!]r) & \texttt{if } \mathcal{E}[\![expr_1]\!]r \\ & \quad \wedge \mathcal{E}[\![expr_2]\!]r \in T \\ \bot & \texttt{if } \mathcal{E}[\![expr_i]\!]r = \bot \\ & \quad \wedge \mathcal{E}[\![expr_j]\!]r \neq \mathsf{error} \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathsf{in}(expr_1, expr_2)]\!]r = \begin{cases} (\mathcal{E}[\![expr_1]\!]r \in \mathcal{E}[\![expr_2]\!]r) & \texttt{if } \mathcal{E}[\![expr_1]\!]r \in T \ \wedge \ \mathcal{E}[\![expr_2]\!]r \in 2^T \\ \bot & \texttt{if } \mathcal{E}[\![expr_i]\!]r = \bot \ \wedge \mathcal{E}[\![expr_j]\!]r \neq \mathsf{error} \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathsf{add}(expr_1, expr_2)]\!]r = \begin{cases} (\mathcal{E}[\![expr_1]\!]r + \mathcal{E}[\![expr_2]\!]r) & \texttt{if } \mathcal{E}[\![expr_1]\!]r, \mathcal{E}[\![expr_2]\!]r \in \mathit{Double} \\ \bot & \texttt{if } \mathcal{E}[\![expr_i]\!]r = \bot \ \wedge \mathcal{E}[\![expr_j]\!]r \neq \mathsf{error} \\ \mathsf{error} & \texttt{otherwise} \end{cases}$$

Table 3.4: Semantics of (an excerpt of) FACPL expressions (T stands for one of the sets of literal values or for the powerset of the set of all literal values, and $i, j \in \{1, 2\}$ with $i \neq j$)

the clauses of operators and and or possibly mask these special values by implementing the behaviour informally described in Section 3.3. It is worth noticing that the explicit management of missing attributes and evaluation errors ensure a full account of crucial aspects of access control policy evaluations, usually neglected by other proposals from the literature (see, e.g., [JSS97, RRNN12, ACC14]). The only proposal taking into account the role of missing attributes is that in [CM12]. However, it considers a simplified language, e.g. obligations are missing, and assumes that expressions cannot generate errors.

Function $\mathcal{E}$ is straightforwardly extended to sequences of expressions by the clauses

$$\mathcal{E}[\![\epsilon]\!]r = \epsilon \qquad \mathcal{E}[\![expr' \ \ expr^*]\!]r = \mathcal{E}[\![expr']\!]r \bullet \mathcal{E}[\![expr^*]\!]r \tag{S-2}$$

The operator $\bullet$ denotes concatenation of sequences of semantic elements and ϵ denotes the empty sequence. We assume that $\bullet$ is strict on error and $\bot$, i.e. error is returned whenever an error or $\bot$ is in the sequence. Therefore, the evaluation of $\mathcal{E}[\![expr^*]\!]r$ fails if any of the expressions forming $expr^*$ evaluates to error or $\bot$.

The semantics of the fulfilment of obligations is formalised by the function $\mathcal{O} : \mathit{Obligation} \rightarrow (R \rightarrow \mathit{FObligation} \cup \{\mathsf{error}\})$ defined by the following clause

$$\mathcal{O}[\![\,[\, e \ \ t \ \ \mathit{pepAct}(expr^*)\,]\,]\!]r \quad = \quad \begin{cases} [\, t \ \ \mathit{pepAct}(w^*)\,] & \texttt{if } \mathcal{E}[\![expr^*]\!]r = w^* \\ \mathsf{error} & \texttt{otherwise} \end{cases} \tag{S-3a}$$

where w stands for a literal value or a set of literal values. Thus, the fulfilment of an obligation, given a request, returns a fulfilled obligation when the evaluation of every expression argument of the action returns a value. Otherwise, it returns an error.

Function $\mathcal{O}$ is straightforwardly extended to sequences of obligations as follows

$$\mathcal{O}[\![\epsilon]\!]r = \epsilon \qquad \mathcal{O}[\![o' \;\; o^*]\!]r = \mathcal{O}[\![o']\!]r \bullet \mathcal{O}[\![o^*]\!]r \tag{S-3b}$$

Notably, a sequence of fulfilled obligations is returned only if every obligation in the sequence successfully fulfils; otherwise, error is returned (indeed, $\bullet$ is strict on error).

We can now define the semantics of a policy as a function that, given a request, returns an authorisation decision paired with a (possibly empty) sequence of fulfilled obligations. Formally, it is given by the function $\mathcal{P} : \mathit{Policy} \to (R \to \mathit{PDPReponse})$ that has two defining clauses: one for rules and one for policy sets. The clause for rules is

$$\mathcal{P}[\![(e \;\; \mathsf{target} : \mathit{expr} \;\; \mathsf{obl} : o^* \,)]\!]r = \begin{cases} \langle e \;\; fo^* \rangle & \texttt{if } \mathcal{E}[\![\mathit{expr}]\!]r = \mathsf{true} \;\wedge\; \mathcal{O}[\![o^*|_e]\!]r = fo^* \\ \mathsf{not\text{-}app} & \texttt{if } \mathcal{E}[\![\mathit{expr}]\!]r = \mathsf{false} \;\vee\; \mathcal{E}[\![\mathit{expr}]\!]r = \bot \\ \mathsf{indet} & \texttt{otherwise} \end{cases} \tag{S-4a}$$

Thus, the rule effect is returned as a decision when the target evaluates to true, which means that the rule applies to the request, and all obligations with the same applicability effect as the rule successfully fulfil. In this case, the fulfilled obligations are also part of the response. Otherwise, it could be the case that (i) the rule does not apply to the request, i.e. the target evaluates to false or to $\bot$, or that (ii) an error has occurred while evaluating the target or fulfilling the obligations with the same effect as the rule. Notation $o^*|_e$ indicates the subsequence of o^* made of those obligations whose effect is e. Formally, its definition is as follows

$$\begin{aligned} \epsilon|_e &= \epsilon \\ ([\; e' \;\; t \;\; \mathit{pepAct}(\mathit{expr}^*)\;]\;\; o^*)|_e &= \begin{cases} [\; e' \;\; t \;\; \mathit{pepAct}(\mathit{expr}^*)\;]\;\; (o^*|_e) & \texttt{if } \; e' = e \\ o^*|_e & \texttt{otherwise} \end{cases} \end{aligned}$$

The semantics of policy sets relies on the semantics of combining algorithms. Indeed, as detailed in Section 3.5.3, we use a semantic function $\mathcal{A}$ to map each combining algorithm a to a function that, to a sequence of policies, associates a function from requests to PDP responses. The clause for policy sets is

$$\mathcal{P}[\![\{a \;\; \mathsf{target} : \mathit{expr} \;\; \mathsf{policies} : p^+ \; \mathsf{obl} : o^* \,\}]\!]r = \begin{cases} \langle e \;\; fo_1^* \bullet fo_2^* \rangle & \texttt{if } \mathcal{E}[\![\mathit{expr}]\!]r = \mathsf{true} \;\wedge\; \mathcal{A}[\![a, p^+]\!]r = \langle e \;\; fo_1^* \rangle \wedge\; \mathcal{O}[\![o^*|_e]\!]r = fo_2^* \\ \mathsf{not\text{-}app} & \texttt{if } \mathcal{E}[\![\mathit{expr}]\!]r = \mathsf{false} \;\vee\; \mathcal{E}[\![\mathit{expr}]\!]r = \bot \\ & \qquad \vee\; (\mathcal{E}[\![\mathit{expr}]\!]r = \mathsf{true} \;\wedge\; \mathcal{A}[\![a, p^+]\!]r = \mathsf{not\text{-}app}) \\ \mathsf{indet} & \texttt{otherwise} \end{cases} \tag{S-4b}$$

Thus, the policy set applies to the request when the target evaluates to true, the semantic of the combining algorithm a (which is applied to the enclosed sequence of policies and the request) returns the effect e and a sequence of fulfilled obligations fo_1^*, and all enclosed obligations with effect e successfully fulfil and return a sequence fo_2^*. In this case, the PDP response contains e and the concatenation of sequences fo_1^* and fo_2^*. Instead, if the target evaluates to false or to $\bot$, or the combining algorithm returns not-app, the policy set does not apply to the request. In the remaining cases, an error has occurred and the response is indet.

Finally, the semantic of a PDP is that function from requests to PDP responses obtained by applying the combining algorithm to the enclosed sequence of policies, that is

$$\mathcal{P}dp[\![\{a \;\; \mathsf{policies} : p^+\}]\!]r = \mathcal{A}[\![a, p^+]\!]r \tag{S-5}$$

3.5.3 Combining Algorithms

The semantics of combining algorithms is defined in terms of a family of binary operators. Let alg denote the name of a combining algorithm (i.e., p-over, d-over, etc.); the corresponding semantic operator is identified as $\otimes$alg and defined by means of a two-dimensional matrix that, given two PDP responses, calculates the resulting combined response. The combining matrices of the $\otimes$alg operators are reported in Table 3.5. Basically, the matrices specify the precedences among the permit, deny, not-app and indet decisions, and show how the resulting (sequence of) fulfilled obligations is obtained, i.e. by concatenating the fulfilled obligations of the responses whose decision matches the combined one. Notice that the operators are not commutative (in fact, the matrices are not symmetric because the order in which sequences of obligations are combined does matter).

The semantics of the combining algorithms can be now formalised by the function $\mathcal{A} : Alg \times Policy^{+} \rightarrow (R \rightarrow \ PDPReponse)$. This function is defined in terms of the iterative application of the binary combining operators by means of two definition clauses according to the adopted fulfilment strategy: the all strategy always requires evaluation of all policies, while the greedy strategy halts the evaluation as soon as a final decision is determined (i.e. without necessarily taking into account all policies in the sequence). If the all strategy is adopted, the definition clause is as follows

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1 \ \ldots \ p_s]\!]r = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\ldots \otimes \mathsf{alg}(\mathcal{P}[\![p_1]\!]r, \mathcal{P}[\![p_2]\!]r), \ldots), \mathcal{P}[\![p_s]\!]r) \tag{S-6a}$$

meaning that the combining operator is sequentially applied to the denotations of all input policies[6]. Instead, if the greedy strategy is used, the definition clause is as follows

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1 \ \ldots \ p_s]\!]r = \begin{cases} res_1 & \texttt{if} \quad \mathcal{P}[\![p_1]\!]r = res_1 \ \wedge \ isFinal_{\mathsf{alg}}(res_1) \\ res_2 & \texttt{elseif} \quad \otimes\mathsf{alg}(res_1, \mathcal{P}[\![p_2]\!]r) = res_2 \ \wedge \ isFinal_{\mathsf{alg}}(res_2) \\ \vdots & \vdots \\ res_{s\text{-}1} & \texttt{elseif} \quad \otimes\mathsf{alg}(res_{s\text{-}2}, \mathcal{P}[\![p_{s\text{-}1}]\!]r) = res_{s\text{-}1} \ \wedge \ isFinal_{\mathsf{alg}}(res_{s\text{-}1}) \\ \otimes\mathsf{alg}(res_{s\text{-}1}, \mathcal{P}[\![p_s]\!]r) & \texttt{otherwise} \end{cases} \tag{S-6b}$$

where the `elseif` notation is a shortcut to represent mutually exclusive conditions. The auxiliary predicates $isFinal_{\mathsf{alg}}$ (one for each combining algorithm alg), given a response in input, check if the response decision is final with respect to the algorithm alg, i.e. if such decision cannot change due to further combinations. Their definition is in Table 3.6; as a matter of notation, we use $res.dec$ to indicate the decision of response res. These predicates are straightforwardly derived from the combination matrices of the binary operators, thus we only comment on salient points. In case of the p-over algorithm (and similarly for the others in the first two rows of the table), the permit decision is the only decision that can never be overwritten, hence, it is final. Instead, in case of the first-app algorithm, all decisions except not-app are final since they represent the fact that the first applicable policy has been already found. Both consensus algorithms have indet as final decision, because no form of consensus can be reached once an indet is obtained. Similarly, the one-app algorithm has indet as final decision.

[6]Notably, in case of a single policy, operators $\otimes$p-unless-d and $\otimes$d-unless-p turn the not-app and indet responses into, respectively, $\langle$permit $\epsilon\rangle$ and $\langle$deny $\epsilon\rangle$, while the remaining operators leave them unchanged.

$\otimes$p-over $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$
$\langle$deny $FO_1\rangle$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_1\rangle$	indet
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
indet	$\langle$permit $FO_2\rangle$	indet	indet	indet

$\otimes$d-over $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$permit $FO_1\rangle$	indet
$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
indet	indet	$\langle$deny $FO_2\rangle$	indet	indet

$\otimes$d-unless-p $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$
$\langle$deny $FO_1\rangle$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$deny $\epsilon\rangle$	$\langle$deny $\epsilon\rangle$
indet	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$deny $\epsilon\rangle$	$\langle$deny $\epsilon\rangle$

$\otimes$p-unless-d $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$
$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$permit $\epsilon\rangle$	$\langle$permit $\epsilon\rangle$
indet	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	$\langle$permit $\epsilon\rangle$	$\langle$permit $\epsilon\rangle$

$\otimes$first-app $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1\rangle$
$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$	$\langle$deny $FO_1\rangle$
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$one-app $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	indet	indet	$\langle$permit $FO_1\rangle$	indet
$\langle$deny $FO_1\rangle$	indet	indet	$\langle$deny $FO_1\rangle$	indet
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$weak-con $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	indet	$\langle$permit $FO_1\rangle$	indet
$\langle$deny $FO_1\rangle$	indet	$\langle$deny $FO_1 \bullet FO_2\rangle$	$\langle$deny $FO_1\rangle$	indet
not-app	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$strong-con $_{res_1}\backslash^{res_2}$	$\langle$permit $FO_2\rangle$	$\langle$deny $FO_2\rangle$	not-app	indet
$\langle$permit $FO_1\rangle$	$\langle$permit $FO_1 \bullet FO_2\rangle$	indet	indet	indet
$\langle$deny $FO_1\rangle$	indet	$\langle$deny $FO_1 \bullet FO_2\rangle$	indet	indet
not-app	indet	indet	not-app	indet
indet	indet	indet	indet	indet

Table 3.5: Combination matrices for $\otimes$alg operators (res_1 and res_2 indicate the first and the second argument, respectively)

$$isFinal_{\text{p-over}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{permit} \\ \text{false} & \texttt{otherwise} \end{cases} \qquad isFinal_{\text{d-over}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{deny} \\ \text{false} & \texttt{otherwise} \end{cases}$$

$$isFinal_{\text{d-unless-p}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{permit} \\ \text{false} & \texttt{otherwise} \end{cases} \qquad isFinal_{\text{p-unless-d}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{deny} \\ \text{false} & \texttt{otherwise} \end{cases}$$

$$isFinal_{\text{first-app}}(res) = \begin{cases} \text{false} & \texttt{if } res.dec = \text{not-app} \\ \text{true} & \texttt{otherwsise} \end{cases} \qquad isFinal_{\text{one-app}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{indet} \\ \text{false} & \texttt{otherwsise} \end{cases}$$

$$isFinal_{\text{weak-con}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{indet} \\ \text{false} & \texttt{otherwsise} \end{cases} \qquad isFinal_{\text{strong-con}}(res) = \begin{cases} \text{true} & \texttt{if } res.dec = \text{indet} \\ \text{false} & \texttt{otherwsise} \end{cases}$$

Table 3.6: Definition of the $isFinal_{\text{alg}}(res)$ predicate

It is worth noticing that the combination matrices we propose are a refined version of those in [LWQ+09], i.e. we explicit address the combination of obligations. Furthermore, differently from other formalisations of combining algorithms (see, e.g., [RLB+09, RRNN12, CM12, ACC14]), we formalise their semantics in terms of different fulfilment strategies, thus to further take into account the role of obligations in algorithm evaluation. In general, the introduction of δ paves the way to the definition of more sophisticated management strategies as, e.g., only obligations that are not in conflict each other (due to the type of actions they define) are combined and then returned.

3.5.4 Policy Enforcement Process

The semantics of the enforcement process defines how the PEP discharges obligations and enforces authorisation decisions. To define this process, we use the auxiliary function $(\!(\ )\!) : FObligation^* \to \{\text{true}, \text{false}\}$ that, given a sequence of fulfilled obligations, executes such obligations and returns a boolean value that indicates whether the evaluation is successfully completed. Notably, since failures caused by optional obligations can be safely ignored by the PEP, only failures of mandatory obligations (i.e. of type M) have to be taken into account. The function is thus defined as follows

$$\begin{aligned} (\!(\epsilon)\!) &= \text{true} \\ (\!([\,\text{O}\ \ pepAct(w^*)\,] \bullet fo^*)\!) &= (\!(fo^*)\!) \\ (\!([\,\text{M}\ \ pepAct(w^*)\,] \bullet fo^*)\!) &= \begin{cases} (\!(fo^*)\!) & \texttt{if } pepAct(w^*) \Downarrow \text{ok} \\ \text{false} & \texttt{otherwise} \end{cases} \end{aligned}$$

where $\Downarrow$ ok denotes that the discharge of the action $pepAct(w^*)$ succeeded. Since the set of action identifiers is intentionally left unspecified (see Section 3.2), the definition of predicate $\Downarrow$ ok is hence unspecified too (in other words, the syntactic domain $PepAction$ is a parameter of the syntax, while $\Downarrow$ ok is a parameter of the semantics); we just assume that it is total and deterministic.

The semantics of PEP is thus defined with respect to the enforcement algorithms. Formally, given an enforcement algorithm and a PDP response, the function $\mathcal{E}A : EnfAlg \to (PDPReponse \to Decision)$ returns the enforced decision. It is defined by three clauses, one for each algorithm. The clause for the deny-biased algorithm follows

$$\mathcal{E}A[\![\text{deny-biased}]\!]res = \begin{cases} \text{permit} & \texttt{if } res.dec = \text{permit} \ \wedge \ (\!(res.fo)\!) \\ \text{deny} & \texttt{otherwise} \end{cases} \tag{S-7a}$$

Likewise $res.dec$ that indicates the decision of the response res, notation $res.fo$ indicates the sequence of fulfilled obligations of res. The permit decision is enforced only if this

is the decision returned by the PDP and all accompanying obligations are successfully discharged. If an error occurs, as well as if the PDP decision is not permit, a deny is enforced. The clause for the permit-biased algorithm is the dual one, whereas the clause for the base algorithm is as follows

$$\mathcal{E}A[\![\mathsf{base}]\!]res = \begin{cases} \mathsf{permit} & \texttt{if } res.dec = \mathsf{permit} \ \wedge \ (\!(res.fo)\!) \\ \mathsf{deny} & \texttt{if } res.dec = \mathsf{deny} \ \wedge \ (\!(res.fo)\!) \\ \text{not-app} & \texttt{if } res.dec = \text{not-app} \\ \mathsf{indet} & \texttt{otherwise} \end{cases} \tag{S-7b}$$

Both decisions permit and deny are enforced only if all obligations in the PDP response are successfully discharged, otherwise indet is enforced. Instead, decisions not-app and indet are enforced without modifications.

3.5.5 Policy Authorisation System

The semantics of a PAS is defined in terms of the composition of the semantics of PEP and PDP. It is given by the function $\mathcal{P}as : PAS \to (Request \to Decision)$ defined by the following clause

$$\mathcal{P}as[\![\{\ \mathsf{pep} : ea \ \ \mathsf{pdp} : pdp\ \}, req]\!] = \mathcal{E}A[\![ea]\!](\mathcal{P}dp[\![pdp]\!](\mathcal{R}[\![req]\!])) \tag{S-8}$$

Basically, given a request req in the FACPL syntax, this is converted into its functional representation by the function $\mathcal{R}$ (see Section 3.5.1). This result is then passed to the semantics of the PDP, i.e. $\mathcal{P}dp[\![pdp]\!]$, which returns a response that on its turn is passed to the semantics of the PEP, i.e. $\mathcal{E}A[\![ea]\!]$. The latter function returns then the final decision of the PAS when given the request req in input.

3.5.6 Properties of the Semantics

We conclude this section with some properties and results of the FACPL semantics. In particular, we address the so-called 'reasonability' properties of [TK06] that precisely characterise the expressiveness of a policy language. These properties are written in italics between double quotes and accompanied by their informal description.

The main result is that FACPL semantics is "*deterministic*" and "*total*", i.e. the semantic is a total function. Informally, this means that, given a FACPL specification, i.e. a PAS, and a possible request, multiple evaluations of such request produce the same decision.

To formally prove this result, we reason by induction on the *depth* of policies, i.e. the number of nested policies, that is defined by induction on the syntax of policies as follows

$$depth((e \ \ \mathsf{target} : expr \ \ \mathsf{obl} : o^*\ )) = 0$$
$$depth(\{a \ \ \mathsf{target} : expr \ \ \mathsf{policies} : p^+ \ \mathsf{obl} : o^*\ \}) = 1 + max\{depth(p) \mid p \in p^+\}$$

Indeed, policies with depth 0 are rules, the other ones are policies containing other policies. Notationally, we will use p^i to mean that policy p has depth i and $(p^+)^i$ to mean that at least a policy in the sequence p^+ has depth i and the others have depth at most i.

Theorem 3.1 (Total Semantics). *For all $pas \in PAS$, $req \in Request$ and $dec, dec' \in Decision$, it holds that*

$$\mathcal{P}as[\![pas, req]\!] = dec \ \ \wedge \ \ \mathcal{P}as[\![pas, req]\!] = dec' \ \ \Rightarrow \ \ dec = dec'$$

that is, $\mathcal{P}as$ is a total function.

Proof. We prove that $\mathcal{P}as$ is a *total function*, i.e. it uniquely associates a decision to every input (pas, req). From the clause (S-8) we have

$$\mathcal{P}as[\![\{\ \mathsf{pep} : ea\ \ \mathsf{pdp} : pdp\ \}, req]\!] = \mathcal{E}A[\![ea]\!](\mathcal{P}dp[\![pdp]\!](\mathcal{R}[\![req]\!]))$$

thus, since the composition of total functions is a total function, it is enough to prove that $\mathcal{R}$, $\mathcal{P}dp$ and $\mathcal{E}A$ are total functions. The proofs proceed by inspecting their defining clauses with aim of checking that they satisfy the two requirements below

R1 there is one, and only one, clause that applies to each syntactic domain element (this usually follows since the definition is syntax-driven and considers all the syntactic forms that the input can assume);

R2 for each defining clause,

- the conditions of the right hand side are mutually exclusive (from the systematic use of the `otherwise` condition, it directly follows that they cover all the possible cases for the syntactic domain elements of the form occurring in the left hand side),
- the values assigned in each case of the right hand side are obtained by only using total functions and/or total and deterministic operators/predicates.

Case $\mathcal{R}$ From its defining clauses (S-1) we get that $\mathcal{R}$ is defined on all non-empty sequences of attributes, i.e. all requests. Moreover, the conditions of the right hand side of each clause are mutually exclusive and the operator $\Cup$ is total and deterministic by definition. Thus R1 and R2 hold, which means that $\mathcal{R}$ is a total function.

Case $\mathcal{P}dp$ To prove this case, we first prove that $\mathcal{E}$, $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{P}$ are total functions.

Case $\mathcal{E}$ By an easy inspection of the clauses defining $\mathcal{E}$, an excerpt of which are in Table 3.4, it is not hard to believe that they satisfy R1 (since the application of the clauses is driven by the syntactic form of the input expression) and R2 above, hence $\mathcal{E}$ is a total function. Moreover, since the operator $\bullet$ is total and deterministic, from the clauses (S-2) it follows that $\mathcal{E}$ remains a total function also when extended to sequences of expressions.

Case $\mathcal{O}$ Since $\mathcal{E}$ is a total function also on sequences of expressions, from the clauses (S-3a) and (S-3b) it follows that requirements R1 and R2 hold, thus $\mathcal{O}$ is a total function both on single obligations and on sequences of obligations.

Cases $\mathcal{A}$ **and** $\mathcal{P}$ The definitions of $\mathcal{P}$ and $\mathcal{A}$ are syntax-driven and consider all the syntactic forms that the input can assume, thus R1 is satisfied. Now, since $\mathcal{P}$ and $\mathcal{A}$ are mutually recursive, we prove by induction on the depth of their arguments that their defining clauses satisfy R2 for all input policies.

Base Case ($i = 0$**)** Let us start from $\mathcal{P}$. p^0 is of the form $(e\ \mathsf{target} : expr\ \mathsf{obl} : o^*)$. We have hence to prove that the clause (S-4a), which is the defining clause of $\mathcal{P}$ that applies to p^0, satisfies R2. This directly follows from the fact that $\mathcal{E}$ and $\mathcal{O}$ are total functions, as well as it is by definition the function corresponding to notation $o^*|_e$. Now, let us consider $\mathcal{A}$ and proceed by case analysis on a.

($a = \mathsf{alg}_{\mathsf{all}}$ **for any** alg) Since the clause (S-4a) satisfies (R1 and) R2, for each p_j^0 in $(p^+)^0$, $\mathcal{P}[\![p_j^0]\!]r$ is uniquely defined. Thus, since each operator $\otimes\mathsf{alg}$ is total and deterministic by construction, the clause (S-6a), to be used since the form of a, satisfies R2 (when all the input policies have depth 0).

($a = \mathsf{alg}_{\mathsf{greedy}}$ **for any** alg) This case is similar to the previous one, but involves the clause (S-6b) that satisfies R2 (when all the input policies have depth 0) since its conditions of the right hand side are mutually exclusive by construction (notably, each predicate $isFinal_{\mathsf{alg}}$ and each operator $\otimes\mathsf{alg}$ is total and deterministic).

Inductive Case ($i = n + 1$) Let us start from $\mathcal{P}$. p^{n+1} is of the form $\{a \ \ \mathsf{target} : expr \ \mathsf{policies} : (p^+)^n \ \mathsf{obl} : o^*\}$. By the induction hypothesis, for any r, a and p_j^k in $(p^+)^n$, with $k \leq n$, the clauses defining $\mathcal{P}$ and $\mathcal{A}$ satisfy (R1 and) R2, that is $\mathcal{P}[\![p_j^k]\!]r$ and $\mathcal{A}[\![a, (p^+)^n]\!]r$ are uniquely defined. Hence, the clause (S-4b), to be used since the form of p^{n+1}, satisfies R2 as well. For $\mathcal{A}$, we can reason like in the base case by exploiting the induction hypothesis. We can thus conclude that both the clauses (S-6a) and (S-6b) satisfy R2 (for any input policies).

Therefore, $\mathcal{P}$ and $\mathcal{A}$ are total functions.

Now, that $\mathcal{P}dp$ is a total function directly follows from its defining clause (S-5).

Case $\mathcal{E}A$ The requirement R1 is satisfied by definition. Moreover, since the predicate $\Downarrow \mathsf{ok}$ is total and deterministic, the same holds for the function $(\!(\ )\!)$. Therefore, also R2 is satisfied by each defining clause (the conditions on $res.dec$ are trivially mutually exclusive). Hence, $\mathcal{E}A$ is a total function.

□

Notably, the fact that FACPL has a total semantics is somehow expected due to its domain-specific nature, indeed FACPL is not a general-purpose Turing-complete language.

Furthermore, concerning compositionality of policies, FACPL ensures "*independent composition*", i.e. the results of the combining algorithms depend only on the decisions of the policies given in input. This clearly follows from the use of combination matrices.

On the contrary, FACPL ensures neither "*safety*", i.e. a request that is granted (resp., forbidden) may not be granted (resp., forbidden) anymore if new attributes are introduced in the request, nor "*monotonicity*", i.e. a request that is granted (resp., forbidden) may not be granted (resp., forbidden) anymore if a new policy is introduced in a combination. These properties are ensured neither by XACML nor by other policy languages featuring deny rules and combining algorithms like those we have shown.

In addition to these properties, we finally highlight the relationship between attribute names occurring in a policy and names defined by requests. By letting $Names(p)$ to indicate the set of attribute names occurring in (the expressions within) p, we can state the following result.

Lemma 3.2. *For all $p \in Policy$ and $r, r' \in R$ such that $r(n) = r'(n)$ for all $n \in Names(p)$, it holds that $\mathcal{P}[\![p]\!]r = \mathcal{P}[\![p]\!]r'$.*

Proof. The statement is based on an analogous result concerning expressions

$$\begin{array}{l}\text{for all } expr \in Expr \text{ and } r_1, r_1' \in R \text{ such that } r_1(n) = r_1'(n) \text{ for all} \\ n \in Names(expr), \text{ it holds that } \mathcal{E}[\![expr]\!]r_1 = \mathcal{E}[\![expr]\!]r_1'\end{array} \quad \text{(R)}$$

which can be easily proven by structural induction on the syntax of expressions. Functions r_1 and r_1' are only exploited in the base case when evaluating a name $n \in Names(expr)$ for which, by definition and hypothesis, we have $\mathcal{E}[\![n]\!]r_1 = r_1(n) = r_1'(n) = \mathcal{E}[\![n]\!]r_1'$.

Since for any $expr$ occurring in p, we have that $Names(expr) \subseteq Names(p)$, from (R), by taking $r_1 = r$ and $r'_1 = r'$, it follows that

$$\text{for all } expr \text{ occurring in } p,\ \mathcal{E}[\![expr]\!]r = \mathcal{E}[\![expr]\!]r' \tag{R-E}$$

From (R-E), it also immediately follows that

$$\text{for all } o \text{ occurring in } p,\ \mathcal{O}[\![o]\!]r = \mathcal{O}[\![o]\!]r' \tag{R-O}$$

Now we can prove the main statement by induction on the depth i of p.

Base Case ($i = 0$) p^0 has the form $(e \text{ target } : expr \text{ obl } : o^*)$, thus the clause (S-4a) is used to determine $\mathcal{P}[\![p]\!]r$. The thesis then trivially follows from (R-E) and (R-O).

Inductive Case ($i = n + 1$) p^{n+1} is of the form $\{a \text{ target} : expr \text{ policies} : (p^+)^n \text{ obl} : o^*\}$, thus the clause (S-4b) is used to determine $\mathcal{P}[\![p]\!]r$. By the induction hypothesis, for any p_j^k in $(p^+)^n$, with $k \leq n$, it holds that $\mathcal{P}[\![p_j^k]\!]r = \mathcal{P}[\![p_j^k]\!]r'$. This, due to the clauses (S-6a) and (S-6b), implies that $\mathcal{A}[\![a, (p^+)^n]\!]r = \mathcal{A}[\![a, (p^+)^n]\!]r'$, for any algorithm a. The thesis then follows from this fact and from (R-E) and (R-O).

□

3.6 Supporting Tools

To effectively support the specification, coding and enforcement of FACPL-based access control systems, we develop a fully integrated Java-based software toolchain[7], graphically depicted in Figure 3.2. The key element of the toolchain is an Eclipse-based IDE that provides features like, e.g., static code checks and automatic generation of runnable Java code. An expressly developed Java library is used to compile and execute the Java code. The figure also reports the SMT-based functionalities supporting the analysis we introduce in Chapter 4. Additional comments on these tools are reported in Section 4.5.

To provide interoperability with the standard XACML and the variety of available tools supporting it (e.g. XCREATE [BDLM12], Margrave [FKMT05] and Balana [WSO15]), the IDE automatically translates FACPL code into XACML one and vice-versa. Because of a slightly different expressivity (e.g. FACPL supports more combining algorithms), there are some limitations in FACPL and XACML interoperability. Section 7.1 reports a detailed comparison between them.

Furthermore, to allow newcomer users to directly experiment with FACPL, the web application "Try FACPL in your Browser", reachable from the FACPL website, offers an online editor for creating and evaluating FACPL policies; the e-Health case study is there reported as a running example. Additionally, it is also available from such web application a proof-of-concept interface showing how a FACPL-based access control system can be exploited for provisioning e-Health services.

In the rest of this section, we detail the FACPL Java library (Section 3.6.1) and IDE (Section 3.6.2). Instead, performance and functionality comparisons with other similar tools are reported in Section 7.5.

[7]The FACPL supporting tools are freely available and open-source; binary files, source files, unit tests and a user's guide can be found at the FACPL website [FAC16].

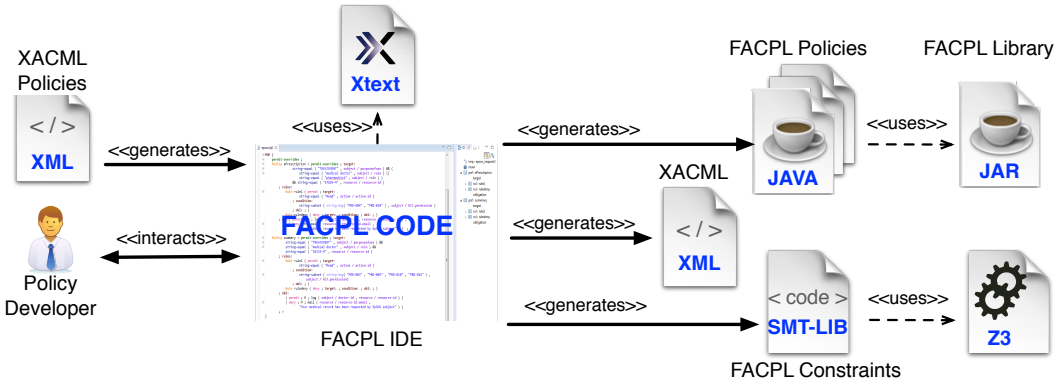


Figure 3.2: FACPL toolchain

3.6.1 The FACPL Library

The Java library we provide aims at representing and evaluating FACPL policies, hence at fully implementing the evaluation process formalised in Section 3.5. To this aim, driven by the formal semantics, we have defined a conformance test-suite that systematically verifies each library unit (e.g., expressions and combining algorithms) with respect to its formal specification.

For each element of the language the library contains an abstract class that provides its evaluation method. In practice, a FACPL policy is translated into a Java class that instantiates the corresponding abstract one and adds, by means of specific methods (e.g., `addObligation`), its forming elements. Similarly, a request corresponds to a Java class containing the request attributes and a reference to a context handler that can be used to dynamically retrieve additional attributes at evaluation-time.

Evaluating requests amounts to invoke the evaluation method of a policy, which coordinates the evaluation of its enclosed elements in compliance with its formal specification. In addition to the authorisation process, the library supports the enforcement process by defining the three enforcement algorithms and a minimal set of pre-defined PEP actions, i.e. `log`, `mailTo` and `compress`. Additional actions can be dynamically introduced by providing their implementation classes to the PEP initialisation method.

By way of example, we report in Listings 3.1 (an excerpt of) the Java code of Policy (Eh-A) introduced in Section 3.4. Besides the specific methods used for adding policy elements, the previous Java code highlights the use of class references for selecting expression operators and combining algorithms. This design choice, together with the use of Java reflection and best-practices of object-oriented programming, allows the library to be easily extended with, e.g., new expression operators, combining algorithms and enforcement actions. Note also that rules are defined as private inner classes, because they cannot be referred by policy sets different from the enclosing one.

Besides the four-valued decisions considered so far, the FACPL library also supports the extended indeterminate values used by XACML, i.e. indetP, indetD and indetDP. They specify the potential decision (permit, deny and both, respectively) that could have been taken if an error, leading to the decision indeterminate, would not have occurred during

```
public class PolicySet_e-Prescription extends PolicySet{

   public PolicySet_e-Prescription(){
      addCombiningAlg(PermitOverrides.class);
      addTarget(...new ExpressionFunction(Equal.class, "e-Prescription",
           new AttributeName("resource","type"))...);
      addPolicyElement(new rule1());
      addPolicyElement(new rule2());
      addPolicyElement(new rule3());
      addObligation(new Obligation("log",Effect.PERMIT,ObligationType.M,
         new AttributeName("system","time"),new AttributeName("resource","type"),
         new AttributeName("subject","id"),new AttributeName("action","id")));
   }

   private class Rule_rule1 extends Rule{
      Rule_rule1 (){
        addEffect(Effect.PERMIT);
        addTarget(...new ExpressionFunction(In.class,
            new AttributeName("subject","permission"),"e-Pre-Write"),...);
      }
   }

   private class Rule_rule2 extends Rule{
      Rule_rule2 (){...}
   }

   private class Rule_rule3 extends Rule{
      Rule_rule3 (){...}
   }
}
```

Listing 3.1: E-Health case study: (an excerpt of) Java code of the e-Prescription policy

the evaluation. Extended indeterminate values allow the PDP to obtain additional information about policy evaluation, which can be exploited, e.g., during policy debugging for improving the treatment of errors. However, their usage may require additional workload. For example, if the target of a policy set evaluates to error, rather than terminating the evaluation process, it continues the computation by processing the enclosed policies in order to calculate an extended indeterminate value. Therefore, the use of extended indeterminate values can be enabled or disabled, by setting a boolean parameter, each time the PDP decision process is invoked.

3.6.2 The FACPL IDE

The FACPL IDE is developed as an Eclipse plug-in and aims to bring together the available functionalities and tools. Indeed, it fully supports writing, evaluating and analysing of FACPL specifications. Its development rests on Xtext [Bet13, Xte16], a Java Eclipse framework for the design and implementation of domain-specific languages.

The plug-in accepts an enriched version of the FACPL language, which contains high-level features facilitating the coding tasks. In particular, each policy has an identifier that can be used as a reference to include the policy within other policies, while specific linguistic handles enable the definition of new expression operators and combining algorithms. Notably, to ease the organisation of large policy specifications, the plug-in supports modularisation of files and import commands extending file scopes.

The development environment provided by the plug-in is standard, as it is shown in

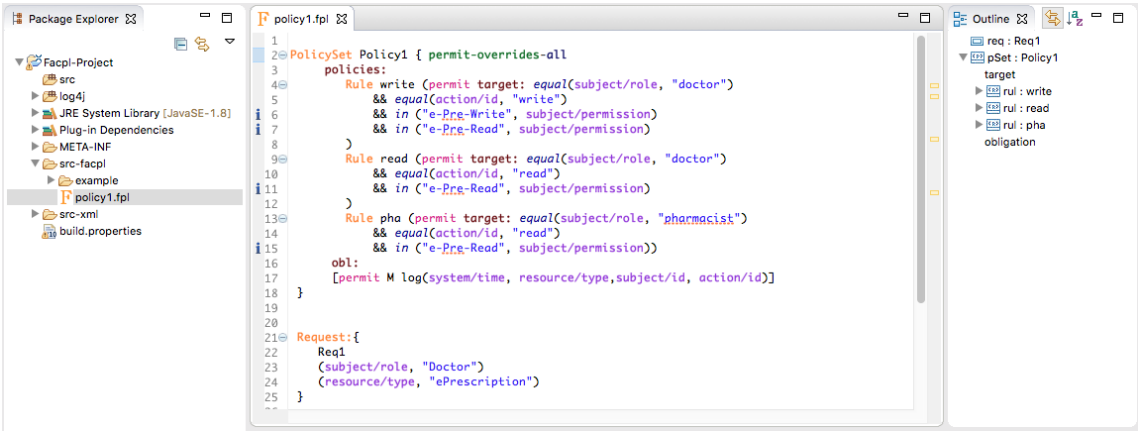


Figure 3.3: FACPL Eclipse plug-in

Figure 3.3. It offers graphical features (e.g., keywords highlighting, code suggestion and navigation within and among files), static controls on code (e.g., uniqueness of identifiers and type checking), and automatic generation of Java, XACML and SMT-LIB code. As the latter functionalities require additional libraries to work, a dedicated wizard creates an opportunely configured FACPL-type project.

As previously pointed out, the Java library is flexible enough to be easily extended. The plug-in facilitates this task by means of dedicated commands. For instance, to define a new expression operator, once a developer has defined the signature of the new function (which is used for type checking and inference), a template of its Java implementation is automatically generated. The actual implementation of the Java class is left to the developer.

Finally, we report a general insight on the usage of the FACPL IDE tool[8]. All the mentioned FACPL functionalities are offered via the customised FACPL project created by the dedicated wizard. Once the project has been created, the policy developer can start coding from the basic FACPL and XACML examples already provided or starting from scratch. A FACPL file is a generic text file with extension *.fpl*, which has dedicated text editor, outline view and contextual menus. The supporting code functionalities, e.g. code suggestion and auto-completion, are available via the classical Eclipse shortcuts and menus. In particular, from either the toolbar menu or the right-click editor menu, the developer can find a set of pre-defined commands to generate Java, XACML and SMT-LIB code, or to open a step-by-step wizard for the definition of authorisation and structural properties.

3.7 Concluding Remarks

In this chapter we have presented the FACPL language, its formal semantics and supporting tools. Here, we conclude by briefly commenting on the contributions of FACPL with respect to the research objectives of the thesis and to related and prior publications.

The syntax of FACPL, as shown in Section 3.4, accomplishes the objective **O1**, hence it provides compact, yet expressive means for the specification of ABAC systems. Indeed, it is expressive enough to represent real-world case studies with compact specifications,

[8]The FACPL user's guide can be found at http://facpl.sourceforge.net/guide/.

e.g. the FACPL epSOS policies are about 90% shorter than the corresponding XACML ones (see Section 7.1 for further details). The semantics of FACPL provides instead a full formal account of the evaluation of ABAC policies, hence accomplishing the objective **O2**. From an implementation point of view, the FACPL toolchain offers practical tools supporting the specification of policies and their practical deployment in real application domains. This accomplishes the objective **O4**; the details concerning the analysis tools are presented in Chapter 4.

In a more general perspective, FACPL brings together the benefits deriving from using a high-level, mnemonic rule-based language with the rigorous means provided by denotational semantics. In fact, the formal semantics provides a formalisation of complex access control features —including obligations and missing attributes, which are instead overlooked by many other proposals (see Section 7.2 for further details)— and lays the basis for developing analysis techniques and tools. Concerning the performance of the FACPL Java library, we ensure a mean request evaluation time comparable with that of state-of-the-art XACML tools. Indeed, by using a benchmark of reference, we obtain a mean time of 2,14ms for FACPL library and 1,85ms for the considered XACML tool. A full performance evaluation of the supporting tools is reported in Section 7.5.

The contents of this chapter are mainly based on the work in [MMPT16]; an initial modelling of the considered e-Health case study is also present in [MMPT13b]. The FACPL language and its Java library have also been exploited in [MMPT13a, CBT+15] within the context of Cloud computing to enforce resource usage strategies. This application, for the sake of presentation, is not reported here.

A preliminary version of FACPL was introduced in [MPT12] and was part of the Ph.D. thesis of Dr. Massimiliano Masi [Mas12]. The aim of this initial version was the formalisation of the semantics of XACML. The language presented here addresses a wider range of aspects concerning access control. Specifically, the syntax of the language is cleaned up and streamlined (e.g., rule conditions are integrated with rule targets and the policy structure is simplified) and, at the same time, it is extended with additional combining algorithms, the PEP specification, an explicit syntax for expressions, and obligations. This latter extension widens FACPL applicability range and expressiveness, as it provides the policy evaluation process with further, powerful means to affect the behaviour of controlled systems (see e.g. [MMPT13a] for a practical example of a policy-based manager for a Cloud platform). Additional significative differences concern the definition of the policy semantics: in [MPT12, Mas12] it is given in terms of partitions of the set of all possible requests, while here it is defined in a functional fashion with respect to a generic request. The new approach also features the formalisation of combining algorithms in terms of binary operators and fulfilment strategies, and the automatic management of missing attributes and evaluation errors throughout the evaluation process. Most of all, the aim of the version of FACPL here presented is significantly different: we do not only propose a different language, but we provide a complete methodology that encompasses all phases of the policy development life-cycle, i.e. specification, analysis and implementation. We indeed present the analysis in the next chapter. Finally, to effectively support all the functionalities, we provide a fully integrated software toolchain, that was not available before.

Chapter 4

Analysis of FACPL Policies

Attribute-based access control policies, like those expressible in FACPL, are sufficiently flexible and expressive to permit enforcing different types of security policies and representing different security models [JKS12]. Therefore, verifying if certain properties of interest are correctly enforced by an access control policy is essential. However, the hierarchical structure of policies, the presence of conflict resolution strategies and the intricacies deriving from the many involved controls do not permit a straightforward verification of properties. We thus need a formally-defined technique that can be supported by effective and practical software tools.

In this chapter, we address the specification and implementation of an analysis technique, based on constraints, for FACPL policies. Specifically, we devise a constraint formalism that permits defining a uniform, flat constraint-based representation of FACPL policies and performing (automatic) extensive checks. This constraint-based representation specifies satisfaction problems in terms of formulae based on multiple theories as, e.g., boolean and linear arithmetics. Such kind of formulae are usually called *satisfiability modulo theories* (SMT) formulae and are largely employed in diverse analysis applications [dB11].

The analysis of access control policies aims at verifying properties on the enforced authorisations. To precisely define properties on FACPL policies, we introduce and formalise a set of relevant *authorisation properties*, which explicitly address the peculiarities of attribute-based controls. Additionally, to reason on the whole set of authorisations enforced by one or more policies, we formalise a set of *structural properties* from the literature. The verification of all these properties can be automatically achieved by exploiting the constraint-based representation of FACPL policies and by using an SMT solver.

Before presenting the formalisation and implementation of this constraint-based analysis approach, we show the expressiveness of ABAC by providing a FACPL-based formalisation of traditional security policies concerning *confidentiality* and *integrity*. The verification that such FACPL policies enforce the expected authorisations exemplifies the intrinsic difficulties to tackle for defining and implementing an analysis technique.

Structure of the chapter. The rest of this chapter is organised as follows. Section 4.1 introduces the attribute-based formalisation of traditional security policies. Section 4.2 outlines the main issues to address in policy analysis. Section 4.3 introduces the constraint-based representation of FACPL policies. Section 4.4 formalises a set of properties of interests. Section 4.5 outlines the strategies for automatising the verification of properties by means of an SMT solver. Section 4.6 concludes with some final remarks.

4.1 Attribute-based Formalisation of Security Policies

The results in [JKS12] imply that the ABAC model subsumes the main other access control models, i.e. DAC, MAC and RBAC. To practically show this expressiveness of ABAC, we exploit FACPL to present an attribute-based formalisation of traditional security policies concerning *confidentiality* and *integrity*. In particular, we consider the formalisations of such policies given by established security models. Recall that, as described in Section 2.1, we only take into account access control systems being part of more complex systems, and that information flow and its related issues are here out of scope.

Classical security models are based on a state machine capturing the *secure* states of the controlled system. Each state corresponds to a secure configuration of the system, while each transition between states represents a permitted, hence secure, access to the system. These models thus rely on the so-called *closed-world* security policy, i.e. all accesses that are not explicitly granted must be forbidden.

The attribute-based formalisation we present addresses the well-known Bell-LaPadula [BL76] and Biba [Bib77] models concerning, respectively, confidentiality and integrity. Additionally, still in the context of integrity, we address the concept of *Separation of Duty (SoD)*, which was introduced in the Clark-Wilson model [CW87] and since then has been largely adopted, especially in RBAC systems.

Therefore, we first characterise in terms of FACPL rules the accesses these models allow (Section 4.1.1), then we formalise the semantic conditions corresponding to the enforcement of the closed-world policy (Section 4.1.2).

4.1.1 Attribute-based Characterisation

We start by providing a precise attribute-based definition of confidentiality and integrity in the context of access control. Given a controlled system, we let $res \in Res$, $Sub' \subseteq Sub$ and $Act' \subseteq Act$, where Res, Sub and Act are respectively the set of resources, subjects and actions involved in the access requests. The definitions thus follow

- *confidentiality*: the resource res has the property of *confidentiality* with respect to subjects Sub' and actions Act' if none of the subjects in Sub' can execute actions in Act' on res;

- *integrity*: the resource res has the property of *integrity* with respect to subjects Sub' and actions Act' if actions in Act' executed by subjects in Sub' cannot alter the trustworthiness of res.

On the base of these general definitions, we present the attribute-based characterisation of the considered models. Notably, since they are defined with respect to read and write actions, we assume that the set Act is only formed by those two actions. In the following, we use attribute names of the form subject/$*$, actions/$*$ and resource/$*$ to identify the characteristics of a *subject* willing to perform a given *action* on a *resource*. For instance, action/id returns the identifier of the requested action (in this case, one between "*read*" and "*write*").

Confidentiality

The security policies commonly referred to as *multi-level security* [BL76, San93] concern confidentiality and represent the type of access controls at the basis of MAC. Their goal is to prevent that a resource with a certain confidentiality level can be disclosed to a subject with a lower level. Hence, it is assumed that each subject and resource is assigned, through a function f_L, to a confidentiality level from a given partially ordered set $< L, \leq_L >$ of levels. Multiple approaches for the formalisation of these policies are present in the literature, our focus is on that of the Bell-LaPadula model [BL76].

This model permits the accesses that adhere to the following security properties

- *no read-up*: a subject s can read a resource res only if the security level of the subject dominates that of the resource, i.e. $f_L(res) \leq_L f_L(s)$;
- *no write-down*: a subject s can write a resource res only if the level of the subject s is dominated by that of the resource, i.e. $f_L(s) \leq_L f_L(res)$.

If we let the attributes subject/level and resource/level denote the confidentiality level assigned by function f_L to subjects and resources, respectively, then the previous properties can be characterised in terms of FACPL rules as follows

$$\begin{array}{l}(\text{permit}\ \ \texttt{target}: \text{equal}(\text{action/id}, \text{“}read\text{”})\ \text{and}\ \text{leq}(\text{resource/level}, \text{subject/level}))\\ (\text{permit}\ \ \texttt{target}: \text{equal}(\text{action/id}, \text{“}write\text{”})\ \text{and}\ \text{leq}(\text{subject/level}, \text{resource/level}))\end{array} \tag{4.1}$$

where the function leq is assumed to be a relational function corresponding to the partial order relation $\leq_L$.

The Bell-LaPadula model is usually extended to also consider a discretionary security policy, i.e. the resource owner has the discretion to establish for each action the subjects allowed to perform it. Thus, by assuming that the list of allowed subjects is grouped with respect to the resources (i.e., in terms of *access control list* [Sal74]), this security property on, e.g., the read action is characterised by the following FACPL rule

$$(\text{permit}\ \ \texttt{target}: \text{equal}(\text{action/id}, \text{“}read\text{”})\ \ \text{and}\ \ \text{in}(\text{subject/id}, \text{resource/read.ids})) \tag{4.2}$$

where we assume that the attribute resource/read.ids returns the set of all subjects allowed to execute the read action on the resource.

Integrity

The integrity property can concern multiple system aspects, but, since we only focus on access control, we address it in terms of the Biba model [Bib77] and the concept of SoD.

The Biba model formalises integrity with respect to read and write actions, and to integrity levels associated to subjects and resources. Assuming that the integrity levels are defined in the same way as the confidentiality ones, the Biba model is the 'dual' of the Bell-LaPadula one, i.e. it relies on the *no read-down* and *no write-up* security properties, which can thus be characterised as before.

The SoD ensures instead that when two or more actions are required to perform a critical transaction, these actions are performed by at least two different subjects. SoD is valuable in deterring fraudulent behaviours, since no single subject has the possibility to perform complex actions, but only well-defined, elementary actions.

A basic example of SoD is to prevent that a subject with assigned two conflicting roles can execute certain actions, i.e. there is no separation of duties among the actions that these roles permit. For instance, if we assume roles "$role1$" and "$role2$" to be in conflict, we can define a FACPL rule that permits a write action only when a subject exposes the first role but not the second one; the corresponding rule is as follows

$$\begin{aligned}(\mathsf{permit}\ \ \texttt{target} : \mathsf{equal}(\mathsf{action/id}, \text{“}write\text{”})\ \mathsf{and}\\ \mathsf{in}(\text{“}role1\text{”}, \mathsf{subject/role})\ \mathsf{and}\ \mathsf{not}(\mathsf{in}(\text{“}role2\text{”}, \mathsf{subject/role})))\end{aligned} \tag{4.3}$$

Indeed, the rule checks that the roles assigned to the subject, i.e. those obtained through the attribute subject/role, include "$role1$", which is required for executing the read action, and not "$role2$".

To sum up, exploiting attributes ensures a high level of abstraction to ABAC policies. In fact, by appropriately asserting on the information each attribute represents, we can easily characterise any type of security policy.

4.1.2 Semantic-based Formalisation

To formalise when a set of security properties of a considered model is correctly enforced by a FACPL policy, we need to explicitly reason on the authorisations the policy enforces. Thus, we use sets of access requests, expressed in the form of FACPL requests, to represent the (non)secure accesses with respect to a given property. The authorisations calculated for each request of these sets permit checking if a security property is correctly enforced.

Formally, given a security property pr of a security model, we let R_{pr} (resp., $\overline{R}_{pr}$) be the set of secure (resp., nonsecure) requests with respect to pr, and Sub_{pr} (resp., Res_{pr}) be the subset of subjects (resp., resources) for which the property pr is defined. Thus, a FACPL policy p containing the rules characterising pr correctly enforces it if the following conditions hold

$$\begin{aligned}&\forall\ r\ \in R_{pr}:\ r(resource/id) \in Res_{pr}\,,\ r\,(subject/id) \in Sub_{pr} \Rightarrow\ \mathcal{P}[\![p]\!]r = \langle \mathsf{permit}\ \ fo\rangle\\ &\forall\ r\ \in \overline{R}_{pr}:\ r(resource/id) \in Res_{pr}\,,\ r(subject/id) \in Sub_{pr} \Rightarrow\ \mathcal{P}[\![p]\!]r = \langle \mathsf{deny}\ \ fo\rangle\end{aligned}$$

for some sequence of fulfilled obligations fo. Notice that these sets are formed by functional FACPL requests of the form shown in Section 3.5.1 and that notation $r(\mathsf{attr\_name})$ indicates the value assigned to the attribute named attr_name by the request r. Hence,

we require that all the (secure) requests in R_{pr} evaluate to permit and all the (nonsecure) requests in $\overline{R}_{pr}$ evaluate to deny. Notably, we consider the requests that only refer to the subset of subjects and resources that the property pr takes into account. This means that the set $\overline{R}_{pr}$ is not the complementary set of R_{pr} with respect to the universe of all the possible requests of the system; rather it represents those requests that are considered nonsecure by the property pr due to the closed-world policy assumed by the security model.

In the sequel we report the definition of the (non)secure sets of requests for the properties we presented before.

Confidentiality

The secure accesses identified by the *no read-up* property corresponds to the set of requests R_{nru} whose elements r satisfy the following conditions

$$r(\mathsf{action/id}) = \text{“}read\text{”}\ ,\ r(\mathsf{resource/level}) = l_1\ ,\ r(\mathsf{subject/level}) = l_2 \quad : \quad l_1, l_2 \in L, l_1 \leq_L l_2$$

The set $\overline{R}_{nru}$ instead contains those requests satisfying the following conditions

$$r(\mathsf{action/id}) = \text{“}read\text{”}\ ,\ r(\mathsf{resource/level}) = l'_1\ ,\ r(\mathsf{subject/level}) = l'_2 \quad : \quad l'_1, l'_2 \in L, l'_1 \not\leq_L l'_2$$

The sets for the *no write-down* property are defined similarly. In case of the discretionary security properties as e.g. that defined by Rule (4.2), the requests of the set R_{dac} are characterised by the following conditions

$$r(\mathsf{action/id}) = \text{“}read\text{”}\ ,\ r(\mathsf{resource/read.ids}) = Sub_{res}\ ,\ r(\mathsf{subject/id}) = s \quad : \quad s \in Sub_{res}$$

where Sub_{res} is set of all subjects allowed to execute the read action on the resource res. Instead, the elements of the deny set $\overline{R}_{dac}$ must satisfy the following conditions

$$r(\mathsf{action/id}) = \text{“}read\text{”}\ ,\ r(\mathsf{resource/read.ids}) = Sub'_{res}\ ,\ r(\mathsf{subject/id}) = s \quad : \quad s \notin Sub'_{res}$$

Indeed, the set of discretionally granted subjects Sub'_{res} does not contain the subject s.

Integrity

The *no read-down* and the *no write-up* properties, representing the Biba model, are formalised similarly to the confidentiality ones.

Let us consider SoD for a write action expressed by Rule (4.3). If we let *Rol* be the collection of authorised non conflicting sets of roles, i.e. the collection of all the sets containing “$role1$” and not “$role2$”, the secure accesses R_{sod} are defined as follows

$$r(\mathsf{action/id}) = \text{“}write\text{”}\ ,\ r(\mathsf{subject/role}) = rol \quad : \quad rol \in Rol$$

The non secure accesses $\overline{R}_{sod}$ are instead defined as follows

$$r(\mathsf{action/id}) = \text{“}write\text{”}\ ,\ r(\mathsf{subject/role}) = rol' \quad : \quad rol' \in Rol_{all} \backslash Rol$$

where the set Rol_{all} represents the collection of all sets of roles that a subject can play in the system. Thus, a request is non secure when the set of subject roles does not contain “$role1$”, i.e. the subject has not the right to execute the write action, or contains “$role1$” and “$role2$” at the same time, i.e. the exposed roles are in conflict.

4.2 Verification of Security Policy Enforcement

The formalisation of security policies presented in the previous section determines the conditions stating when a policy correctly enforces a certain security property. We consider here two of the presented properties and propose different combinations of the corresponding FACPL rules. This allows us to demonstrate the difficulties to address for verifying the correct property enforcement. In particular, the hierarchical structure of policies and the various elements occurring in the policy evaluation make the analysis of policies cumbersome and error-prone.

By way of example, we consider a FACPL policy that has to enforce both the *no read-up* property and discretionary property for read actions requested by a set of subjects Sub' on the resource "res". We thus present different combinations of the Rules (4.1) and (4.2) by commenting on the properties that each combination actually enforces.

The first combination of the two rules is defined as follows

$$
\begin{array}{l}
\{\mathsf{p\text{-}over}_{\mathsf{all}} \\
\quad \texttt{target} : \mathsf{equal}(\mathsf{resource/id}, \text{“}res\text{”}) \text{ and } \mathsf{in}(\mathsf{subject/id}, Sub') \\
\quad \texttt{policies} : \\
\qquad (\mathsf{permit}\ \ \texttt{target} : \mathsf{equal}(\mathsf{action/id}, \text{“}read\text{”}) \text{ and } \mathsf{leq}(\mathsf{resource/level}, \mathsf{subject/level})) \\
\qquad (\mathsf{permit}\ \ \texttt{target} : \mathsf{equal}(\mathsf{action/id}, \text{“}read\text{”}) \text{ and } \mathsf{in}(\mathsf{subject/id}, \mathsf{resource/read.ids}))\}
\end{array}
$$

The chosen combining algorithm is p-over$_{\text{all}}$, which seems the natural choice since each allowed behaviour is explicitly authorised. However, likewise the specification of the e-Health case study commented in Section 3.4, the considered properties are not correctly enforced as we show below.

We consider the *no read-up* property. As formalised in Section 4.1.2, its secure accesses correspond to all the requests containing the resource and subject levels that respect the partial ordering relation. These ones clearly match the target of the first rule, hence this rule, as well as the p-over$_{\text{all}}$ algorithm, returns permit. The nonsecure accesses are instead represented by all the requests containing resource and subject levels not properly ordered. In this case, both internal rules do not apply and the p-over$_{\text{all}}$ algorithm returns not-app, because neither permit nor deny are returned by the rules. However, the nonsecure requests should be evaluated as deny, hence we can conclude that the policy does not properly enforce the *no read-up* property. The same also holds for the discretionary property.

To fix this drawback, we can replace the p-over$_{\text{all}}$ algorithm with the d-unless-p$_{\text{all}}$ one, which ensures that deny is taken as the default decision whenever no rule evaluates to permit. In this case all the nonsecure requests of both properties are properly forbidden. However, as we are addressing two properties, the secure accesses are all those ones that are secure, at the same time, for both properties. This means that permit must be returned only when the two rules apply at the same time as well, but this does not happen in the presented policies. In fact, the combining algorithm does not enforce any form of consensus between the two rules. For example, a subject only having the correct confidentiality level and not the discretionary access can circumvent the access control system when reading a resource.

This additional issue can be addressed by adding a new policy layer and requesting a

strong consensus between the rules. The modified policy is thus as follows

{d-unless-p_{all}
 policies :
  {strong-con_{all}
    target : equal(resource/id, "res") and in(subject/id, Sub')
    policies :
      (permit target : equal(action/id, "$read$") and leq(resource/level, subject/level))
      (permit target : equal(action/id, "$read$") and in(subject/id, resource/read.ids)) }}

The algorithm d-unless-p_{all} is thus used at top level to ensure that the resulting decisions will be only permit or deny. In the inner policy, the algorithm strong-con_{all} ensures that permit is returned only when both internal rules apply at the same time. In this case, all secure and nonsecure accesses of the two intended properties are properly enforced. Notably, we could achieve the same result by merging the two rules, thus avoiding the additional policy layer; however, taking apart the two rules improves readability and facilitate maintenance.

Verifying that a policy properly enforces a set of properties is indeed not straightforward. This example, which seems easy enough for being manually checked, shows us that also in case of simple policies we need an automated verification approach. Specifically, this approach must be capable to take into account all the aspects of a policy specification, e.g. policy stratification and combining algorithms, and to exhaustively check all the significant requests representing the possible accesses. A viable approach towards an automated verification of multiple properties of interest is outlined in the next subsection.

4.2.1 Towards an Automated Verification Approach

The analysis of FACPL policies is a challenging task that has to be supported by providing, on the one hand, a formalism capable of uniformly representing policies and, on the other hand, a set of properties expressively defined for FACPL policies, which hence addresses the peculiarities of attribute-based controls.

The formalism should be sufficiently flexible to deal with multiple domain values for attribute assignments and, at the same time, powerful enough to represent the different elements forming a policy. To this aim, we propose a constraint-based formalism and a formally-defined translation function.

The choice of constraints has been advocated by their flexibility and the numerous constraint solvers freely available. Specifically, constraints permit specifying satisfaction problems based both on boolean formulae and on formulae dealing with different theories as, e.g., linear arithmetics; the so-called *satisfiability modulo theories* (SMT) formulae. Recently, due to the relevant progress made in the development of automatic SMT solvers (e.g., Z3 [dB08], CVC4 [BCD+11], Yices [Dut14]), SMT has been extensively employed in diverse analysis applications [dB11], even in the context of access control policies (see, e.g., [ACC14, TdHRZ15]). In this context, the SMT-based approach has also been demonstrated more effective than many others from the literature, like, e.g., decision diagrams [FKMT05] and description logic [KHP07].

Regarding the properties of interest for access control policies, we can identify two main groups: *authorisation properties*, concerning the expected authorisations for (a set of) requests, and *structural properties*, concerning the structure of the sets of authorisations enforced by one or multiple policies, e.g. if a policy enforces the same set of authorisations

Constraints	*Constr*	::=	*Value* | *Name* | `isMiss`(*Constr*) | `isErr`(*Constr*) | `isBool`(*Constr*)
		|	$\neg$ *Constr* | $\dot{\neg}$ *Constr* | *Constr* cop *Constr*
	cop	::=	$\wedge$ | $\vee$ | $\dot{\wedge}$ | $\dot{\vee}$ | $=$ | $>$ | $\in$ | $+$ | $-$ | $*$ | $/$

Table 4.1: Constraint syntax

of another. In the literature, many structural properties have been proposed (see, e.g., in [FKMT05, KHP07]), whereas, to our knowledge, there is no well-established proposal for the specification of authorisation properties that takes into account the peculiarities of attribute-based policies, e.g. the fact that the "*safety*" property (see Section 3.5.6) is not usually guaranteed.

Therefore, we introduce in Section 4.3 a constraint formalism and a formal translation function for (automatically) representing FACPL policies in terms of constraints. Then, we formalise in Section 4.4 a set of properties of interest and we present in Section 4.5 the strategies for their automatic verification by means of an SMT solver.

4.3 A Constraint-based Representation for FACPL

This section introduces the constraint-based representation of FACPL policies. In particular, we first introduce the constraint formalism (Section 4.3.1), then we formalise how to generate constraints from FACPL policies (Section 4.3.2) and the properties enjoyed by this constraint-based representation (Section 4.3.3). We conclude with some examples of constraints obtained from the e-Health case study (Section 4.3.4).

4.3.1 A Constraint Formalism

The constraint formalism we present here extends boolean and inequality constraints with a few additional operators aiming at precisely representing FACPL constructs. Intuitively, a constraint is a relation defined by conditions on a set of attribute names[1]. An assignment of values to attribute names satisfies a constraint if its conditions are matched. Our formalism, besides classical operators and values, explicitly considers the role of missing attributes, by assigning $\bot$ to attribute names, and of run-time errors, i.e. type mismatches in constraint evaluations.

Syntax. Constraints are written according to the grammar shown in Table 4.1 (the nonterminals *Value* and *Name* are defined in Table 3.1). Thus, a constraint can be a literal value, an attribute name, or a more complex constraint obtained through predicates `isMiss`(), `isErr`() and `isBool`(), or through boolean, comparison and arithmetic operators. Notably, operators $\neg$, $\wedge$ and $\vee$ are the classical boolean ones, while $\dot{\neg}$, $\dot{\wedge}$ and $\dot{\vee}$ correspond to the 4-valued ones used by FACPL expressions.

In the sequel, in addition to the notations of Table 3.3, we use the letter c to denote a generic element of the set of all constraints identified by the nonterminal *Constr*.

Semantics. The semantics of constraints is modelled by the function $\mathcal{C} : \mathit{Constr} \to (R \to \mathit{Value} \cup 2^{\mathit{Value}} \cup \{\mathsf{error}, \bot\})$ inductively defined by the clauses in Table 4.2 (the clauses for

[1] In the literature, constraints are typically defined on a set of *variables*. In our framework, the role of variables is played by attribute names. Therefore, to maintain a coherent terminology throughout the thesis, we refer to constraint variables as attribute names.

$$\mathcal{C}[\![n]\!]r = r(n) \qquad \mathcal{C}[\![v]\!]r = v$$

$$\mathcal{C}[\![\texttt{isMiss}(c)]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \bot \\ \text{false} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![\texttt{isErr}(c)]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \text{error} \\ \text{false} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![\texttt{isBool}(c)]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c]\!]r \in \{\text{true}, \text{false}\} \\ \text{false} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![\neg\ c]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \text{false} \texttt{ or } \mathcal{C}[\![c]\!]r = \bot \\ \text{false} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![\dot\neg\ c]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \text{false} \\ \text{false} & \texttt{if } \mathcal{C}[\![c]\!]r = \text{true} \\ \bot & \texttt{if } \mathcal{C}[\![c]\!]r = \bot \\ \text{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![c_1\ \wedge\ c_2]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \text{true} \texttt{ and } \mathcal{C}[\![c_2]\!]r = \text{true} \\ \text{false} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![c_1\ \dot\wedge\ c_2]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \mathcal{C}[\![c_2]\!]r = \text{true} \\ \text{false} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \text{false} \texttt{ or } \mathcal{C}[\![c_2]\!]r = \text{false} \\ \bot & \texttt{if } \mathcal{C}[\![c_i]\!]r = \bot \texttt{ and } \mathcal{C}[\![c_j]\!]r \in \{\text{true}, \bot\} \\ \text{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![c_1\ \vee\ c_2]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \text{true} \texttt{ or } \mathcal{C}[\![c_2]\!]r = \text{true} \\ \text{false} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![c_1\ \dot\vee\ c_2]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \text{true} \texttt{ or } \mathcal{C}[\![c_2]\!]r = \text{true} \\ \text{false} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \mathcal{C}[\![c_2]\!]r = \text{false} \\ \bot & \texttt{if } \mathcal{C}[\![c_i]\!]r = \bot \texttt{ and } \mathcal{C}[\![c_j]\!]r \in \{\text{false}, \bot\} \\ \text{error} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![c_1\ =\ c_2]\!]r = \begin{cases} \text{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r, \mathcal{C}[\![c_2]\!]r \in T \texttt{ and } \mathcal{C}[\![c_1]\!]r = \mathcal{C}[\![c_2]\!]r \\ \text{false} & \texttt{if } \mathcal{C}[\![c_1]\!]r, \mathcal{C}[\![c_2]\!]r \in T \texttt{ and } \mathcal{C}[\![c_1]\!]r \neq \mathcal{C}[\![c_2]\!]r \\ \bot & \texttt{if } \mathcal{C}[\![c_i]\!]r = \bot \texttt{ and } \mathcal{C}[\![c_j]\!]r \neq \text{error} \\ \text{error} & \texttt{otherwise} \end{cases} \qquad \mathcal{C}[\![c_1\ +\ c_2]\!]r = \begin{cases} \mathcal{C}[\![c_1]\!]r + \mathcal{C}[\![c_2]\!]r & \texttt{if } \mathcal{C}[\![c_1]\!]r, \mathcal{C}[\![c_2]\!]r \in \mathit{Double} \\ \bot & \texttt{if } \mathcal{C}[\![c_i]\!]r = \bot \texttt{ and } \mathcal{C}[\![c_j]\!]r \neq \text{error} \\ \text{error} & \texttt{otherwise} \end{cases}$$

Table 4.2: Semantics of constraints (T stands for one of the sets of literal values or for the powerset of the set of all literal values, and $i, j \in \{1, 2\}$ with $i \neq j$)

$>$, $\in$, $-$, $*$ and $/$ are omitted as they are similar to those for $=$ or $+$). Hence, the semantics of a constraint is a function that, given the functional representation of a request (i.e., an assignment of values to attribute names), returns a literal value or a set of literal values or the special values $\bot$ and error.

The semantics of constraints, except for the cases of predicates and classical boolean operators, mimics the semantic definitions of the 'corresponding' FACPL expression operators defined in Table 3.4 (e.g., the constraint operator $\dot\vee$ corresponds to the expression operator or, as well as $+$ corresponds to add). The clause defining the semantics of predicate $\texttt{isMiss}(c)$ (resp. $\texttt{isErr}(c)$) returns true only if the constraint c evaluates to $\bot$ (resp. error), while that of predicate $\texttt{isBool}(c)$ returns true only if the constraint c evaluates to a boolean value. The clauses for classical boolean operators are instead defined ensuring that only boolean values can be returned. Specifically, they explicitly define conditions leading to result true, while in all the other cases the result is false. Notably, constraint $\neg\ c$ evaluates to true not only when the evaluation of c returns false, but also when it returns $\bot$. This is particularly convenient for translating FACPL policies because, in case of not-app decisions, $\bot$ is treated as false.

We conclude by proving that the constraint semantics is deterministic and total.

Theorem 4.1 (Total Constraint Semantics). *For all* $c \in Constr$, $r \in R$ *and* $el, el' \in (Value \cup 2^{Value} \cup \{\mathsf{error}, \bot\})$, *it holds that*

$$\mathcal{C}[\![c]\!]r = el \;\; \wedge \;\; \mathcal{C}[\![c]\!]r = el' \;\; \Rightarrow \;\; el = el'$$

Proof. The proof proceeds by structural induction on the syntax of c.

Base Case If $c = v$, the thesis immediately follows since $\mathcal{C}[\![v]\!]r = v$; otherwise, i.e. $c = n$, we have $\mathcal{C}[\![n]\!]r = r(n)$ and the thesis follows because r is a total function.

Inductive Case It is easy to check that all the defining clauses of $\mathcal{C}$ are such that the conditions of the right hand side are mutually exclusive and cover all the necessary cases. For each different form that c can assume, the thesis then directly follows by the induction hypothesis.

□

4.3.2 From FACPL Policies to Constraints

The constraint-based representation of FACPL policies is a logical combination of the constraints representing targets, obligations and combining algorithms occurring within policies. We present a *compositional* translation, defined by a family of translation functions $\mathcal{T}_{\cdot}$, that formally defines the constraints representing FACPL terms. We use the emphatic brackets {| and |} to represent the application of a translation function to a syntactic term. It is worth noticing that constraint-based representation of FACPL policies can only deal with statical aspects of policy evaluation, thus disregarding pure dynamic aspects like the greedy fulfilment strategy and the PEP evaluation.

We start by presenting the translation of FACPL expressions, whose operators are very close to (some of) those on constraints. The translation is formally given by the function $\mathcal{T}_E : Expr \to Constr$, whose defining clauses are given below

$$\begin{aligned} &\mathcal{T}_E\{\!|v|\!\} = v \qquad \mathcal{T}_E\{\!|n|\!\} = n \qquad \mathcal{T}_E\{\!|\mathsf{not}(expr)|\!\} = \dot{\neg}\mathcal{T}_E\{\!|expr|\!\} \\ &\mathcal{T}_E\{\!|\mathsf{op}(expr_1, expr_2)|\!\} = \mathcal{T}_E\{\!|expr_1|\!\} \;\; \mathtt{getCop}(\mathsf{op}) \;\; \mathcal{T}_E\{\!|expr_2|\!\} \end{aligned} \tag{T-1}$$

Thus, $\mathcal{T}_E$ acts as the identity function on attribute names and values, and as an homomorphism on operators. In fact, FACPL negation corresponds to the constraint operator $\dot{\neg}$, while the binary FACPL operators correspond to the constraint operators returned by the auxiliary function `getCop`(), which is defined as follows

$$\begin{array}{lll} \mathtt{getCop}(\mathsf{and}) = \dot{\wedge} & \mathtt{getCop}(\mathsf{or}) = \dot{\vee} & \mathtt{getCop}(\mathsf{equal}) = = \\ \mathtt{getCop}(\mathsf{in}) = \in & \mathtt{getCop}(\mathsf{greater\text{-}than}) = > & \mathtt{getCop}(\mathsf{add}) = + \\ \mathtt{getCop}(\mathsf{subtract}) = - & \mathtt{getCop}(\mathsf{multiply}) = * & \mathtt{getCop}(\mathsf{divide}) = / \end{array}$$

The translation of (sequences of) obligations returns a constraint whose satisfiability corresponds to the successful fulfilment of all the obligations. The translation function $\mathcal{T}_{Ob} : Obligation^* \to Constr$ is defined as follows

$$\begin{aligned} &\mathcal{T}_{Ob}\{\!|\epsilon|\!\} = \mathsf{true} \qquad \mathcal{T}_{Ob}\{\!|o \;\; o^*|\!\} = \mathcal{T}_{Ob}\{\!|o|\!\} \wedge \mathcal{T}_{Ob}\{\!|o^*|\!\} \\ &\mathcal{T}_{Ob}\{\!|[e \;\; t \;\; PepAction(expr^*)]|\!\} = \textstyle\bigwedge_{expr \in expr^*} \neg\mathtt{isMiss}(\mathcal{T}_E\{\!|expr|\!\}) \wedge \neg\mathtt{isErr}(\mathcal{T}_E\{\!|expr|\!\}) \end{aligned} \tag{T-2}$$

Hence, a sequence of obligations corresponds to the conjunction of the constraints representing each obligation. When translating a single obligation, predicates `isMiss`() and

$\texttt{isErr}()$ are used to check the fulfilment conditions, i.e. that the occurring expressions cannot evaluate to $\bot$ or error. Notably, the n-ary conjunction operator returns true if the considered obligation contains no expression (i.e., when $expr^* = \epsilon$).

The translation function for policies, $\mathcal{T}_P$, exploits the translation functions previously introduced, as well as a function $\mathcal{T}_A$ representing the effect of applying a combining algorithm to a sequence of policies. Functions $\mathcal{T}_P$ and $\mathcal{T}_A$ are indeed mutually recursive. Moreover, for representing all the decisions that a policy can return, both these two functions return 4-constraint tuples of the form

$$\langle \mathsf{permit} : c_p \quad \mathsf{deny} : c_d \quad \mathsf{not\text{-}app} : c_n \quad \mathsf{indet} : c_i \rangle$$

where each constraint represents the conditions under which the corresponding decision is returned. We call these tuples *policy constraint tuples* and denote their set by PCT. As a matter of notation, we will use the projection operator $\downarrow_l$ which, when applied to a constraint tuple, returns the value of the field labelled by l', where l is the first letter of l' (e.g., $\downarrow_p$ returns the permit constraint c_p).

The function $\mathcal{T}_P : Policy \to PCT$ is defined by two clauses for rules, i.e. one for each effect, and one clause for policy sets. The clause for rules with effect permit is as follows

$$\begin{aligned}
&\mathcal{T}_P\{\!|(\mathsf{permit}\ \ \mathsf{target} : expr\ \ \mathsf{obl} : o^*\ )|\!\} = \\
&\quad \langle\ \mathsf{permit} :\ \mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\} \\
&\quad\ \ \mathsf{deny} :\ \mathsf{false} \\
&\quad\ \ \mathsf{not\text{-}app} :\ \neg\ \mathcal{T}_E\{\!|expr|\!\} \\
&\quad\ \ \mathsf{indet} : \neg\,(\texttt{isBool}(\mathcal{T}_E\{\!|expr|\!\})\ \vee\ \texttt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})) \\
&\qquad\qquad \vee\,(\mathcal{T}_E\{\!|expr|\!\} \wedge \neg\,\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\})\ \rangle
\end{aligned} \tag{T-3a}$$

(the clause for effect deny is omitted, as it just swaps the permit and deny constraints). The clause takes into account the rule constituent parts and combines them according to the rule semantics (see clause (S-4a)). Notably, because of the semantics of the constraint operator $\neg$, the not-app constraint is satisfied when the constraint corresponding to the target expression evaluates to false or to $\bot$. Instead, the negation of a constraint corresponding to a sequence of obligations represents the failure of their fulfilment. Likewise FACPL semantics, the operator $|_e$ returns the subsequence of obligations defined on the effect e. In the indet constraint, together with condition $\neg\ \texttt{isBool}(\mathcal{T}_E\{\!|expr|\!\})$, we introduce $\neg\ \texttt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})$ because we want to exclude that $\mathcal{T}_E\{\!|expr|\!\} = \bot$ (otherwise, we would fall in the case of decision not-app).

The clause for policy sets is as follows

$$\begin{aligned}
&\mathcal{T}_P\{\!|\langle a\ \ \mathsf{target} : expr\ \ \mathsf{policies} : p^+\ \mathsf{obl} : o^*\ \rangle|\!\} = \\
&\quad \langle\ \mathsf{permit} :\ \mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_p \wedge\ \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\} \\
&\quad\ \ \mathsf{deny} :\ \mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_d \wedge\ \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{deny}}\ |\!\} \\
&\quad\ \ \mathsf{not\text{-}app} :\ \neg\ \mathcal{T}_E\{\!|expr|\!\} \vee\ (\mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_n) \\
&\quad\ \ \mathsf{indet} : \neg\,(\texttt{isBool}(\mathcal{T}_E\{\!|expr|\!\})\ \vee\ \texttt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})) \\
&\qquad\qquad \vee\,(\mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_i) \\
&\qquad\qquad \vee\,(\mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_p \wedge \neg\,\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}) \\
&\qquad\qquad \vee\,(\mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|a, p^+|\!\} \downarrow_d \wedge \neg\,\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{deny}}\ |\!\})\ \rangle
\end{aligned} \tag{T-3b}$$

With respect to the clauses for rules, it additionally takes into account the effect of the application of the combining algorithm according to the policy set semantics (see clause (S-4b)). It is worth noticing that the exclusive use of operators $\neg$, $\wedge$ and $\vee$ ensures that constraint tuples are only formed by boolean constraints.

$$
\begin{array}{rcl}
\textsf{p-over}(A,B) & = & \langle\, \textsf{permit} : A\downarrow_p \vee B\downarrow_p \\
& & \quad \textsf{deny} : (A\downarrow_d \wedge B\downarrow_d) \vee\ (A\downarrow_d \wedge B\downarrow_n) \vee\ (A\downarrow_n \wedge B\downarrow_d) \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : (A\downarrow_i \wedge \neg B\downarrow_p) \vee\ (\neg A\downarrow_p \wedge B\downarrow_i)\rangle \\
\textsf{d-over}(A,B) & = & \langle\, \textsf{permit} : (A\downarrow_p \wedge B\downarrow_p) \vee\ (A\downarrow_p \wedge B\downarrow_n) \vee (A\downarrow_n \wedge B\downarrow_p) \\
& & \quad \textsf{deny} : A\downarrow_d \vee B\downarrow_d \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : (A\downarrow_i \wedge \neg B\downarrow_d) \vee\ (\neg A\downarrow_d \wedge B\downarrow_i)\rangle \\
\textsf{d-unless-p}(A,B) & = & \langle\, \textsf{permit} : A\downarrow_p \vee B\downarrow_p \\
& & \quad \textsf{deny} : \neg A\downarrow_p \wedge \neg B\downarrow_p \wedge (A\downarrow_d \vee A\downarrow_n \vee A\downarrow_i) \wedge (B\downarrow_d \vee B\downarrow_n \vee B\downarrow_i) \\
& & \quad \textsf{not-app} : \textsf{false} \\
& & \quad \textsf{indet} : \textsf{false}\rangle \\
\textsf{p-unless-d}(A,B) & = & \langle\, \textsf{permit} : \neg A\downarrow_d \wedge \neg B\downarrow_d \wedge (A\downarrow_p \vee A\downarrow_n \vee A\downarrow_i) \wedge (B\downarrow_p \vee B\downarrow_n \vee B\downarrow_i) \\
& & \quad \textsf{deny} : A\downarrow_d \vee B\downarrow_d \\
& & \quad \textsf{not-app} : \textsf{false} \\
& & \quad \textsf{indet} : \textsf{false}\rangle \\
\textsf{first-app}(A,B) & = & \langle\, \textsf{permit} : A\downarrow_p \vee (B\downarrow_p \wedge A\downarrow_n) \\
& & \quad \textsf{deny} : A\downarrow_d \vee (B\downarrow_d \wedge A\downarrow_n) \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : A\downarrow_i \vee (A\downarrow_n \wedge B\downarrow_i)\rangle \\
\textsf{one-app}(A,B) & = & \langle\, \textsf{permit} : (A\downarrow_p \wedge B\downarrow_n) \vee\ (A\downarrow_n \wedge B\downarrow_p) \\
& & \quad \textsf{deny} : (A\downarrow_d \wedge B\downarrow_n) \vee\ (A\downarrow_n \wedge B\downarrow_d) \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : A\downarrow_i \vee B\downarrow_i \vee ((A\downarrow_p \vee A\downarrow_d) \wedge\ (B\downarrow_p \vee B\downarrow_d))\rangle \\
\textsf{weak-con}(A,B) & = & \langle\, \textsf{permit} : (A\downarrow_p \wedge B\downarrow_p) \vee\ (A\downarrow_p \wedge \neg B\downarrow_d) \vee\ (\neg A\downarrow_d \wedge B\downarrow_p) \\
& & \quad \textsf{deny} : (A\downarrow_d \wedge B\downarrow_d) \vee\ (A\downarrow_d \wedge \neg B\downarrow_p) \vee\ (\neg A\downarrow_p \wedge B\downarrow_d) \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : (A\downarrow_p \wedge B\downarrow_d) \vee\ (A\downarrow_d \wedge B\downarrow_p) \vee\ A\downarrow_i \vee B\downarrow_i \\
\textsf{strong-con}(A,B) & = & \langle\, \textsf{permit} : A\downarrow_p \wedge B\downarrow_p \\
& & \quad \textsf{deny} : A\downarrow_d \wedge B\downarrow_d \\
& & \quad \textsf{not-app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \textsf{indet} : A\downarrow_i \vee B\downarrow_i \vee (A\downarrow_n \wedge \neg B\downarrow_n) \vee\ (\neg A\downarrow_n \wedge B\downarrow_n) \\
& & \qquad\qquad \vee (A\downarrow_p \wedge B\downarrow_d) \vee\ (A\downarrow_d \wedge B\downarrow_p)\rangle
\end{array}
$$

Table 4.3: Constraint combination strategies for the combining algorithms

Combining algorithms are dealt with by the function $\mathcal{T}_A : Alg \times Policy^+ \to PCT$ that, given an algorithm (using the all fulfilment strategy) and a sequence of policies, returns a constraint tuple representing the effect of the algorithm application. Its definition is

$$\mathcal{T}_A\{\!|\textsf{alg}_{\textsf{all}}, p_1\ \ldots\ p_s|\!\} = \textsf{alg}(\ldots \textsf{alg}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\}), \ldots, \mathcal{T}_P\{\!|p_s|\!\}) \qquad \text{(T-4)}$$

By means of $\mathcal{T}_P$, the policies given in input are translated into constraint tuples which are then iteratively combined, two at a time, according to the algorithm combination strategy. Table 4.3 reports the combination of two constraint tuples, say A and B, according to the various combining algorithms. If $s = 1$, i.e. the algorithm must be applied to one tuple only, all the algorithms leave the input tuple unchanged, but for p-unless-d, which given

an input tuple A returns the tuple

$$\langle \mathsf{permit} : A\downarrow_p \vee A\downarrow_n \vee A\downarrow_i \quad \mathsf{deny} : A\downarrow_d \quad \mathsf{not\text{-}app} : \mathsf{false} \quad \mathsf{indet} : \mathsf{false}\rangle$$

and d-unless-p, which behaves similarly.

Finally, the translation of top-level PDP terms $\{Alg \;\; \mathsf{policies} : Policy^+\}$ is the same as that of the corresponding policy sets with target true and no obligations, i.e. $\{Alg \;\; \mathsf{target} : \mathsf{true} \;\; \mathsf{policies} : Policy^+\}$.

4.3.3 Properties of the Representation

We formalise the key results ensured by the constraint-based approach described so far: the correspondence between the semantics of the constraint-based representation of a policy and the semantics of the policy itself. The correspondence is clearly limited to only those policies using the fulfilment strategy all.

Before presenting the semantic correspondence result (Theorem 4.5), we show that the semantics correspondence is guaranteed by the constraint-based representation of expressions (Lemma 4.2), obligations (Lemma 4.3) and combining algorithms (Lemma 4.4). Therefore, the theorem means that the properties verified over constraints would return the same results if they were directly proven on FACPL policies. Hence, it ensures that the verification of properties we present in Section 4.4 is sound.

Lemma 4.2. *For all $expr \in Expr$ and $r \in R$, it holds that*

$$\mathcal{E}[\![expr]\!]r = \mathcal{C}[\![\mathcal{T}_E\{\!|expr|\!\}]\!]r$$

Proof. We proceed by structural induction on the syntax of $expr$ according to the translation rules of the clause (T-1).

($expr = n$) Since $\mathcal{T}_E\{\!|n|\!\} = n$, the thesis follows because $\mathcal{E}[\![n]\!]r = r(n) = \mathcal{C}[\![n]\!]r$.

($expr = v$) Since $\mathcal{T}_E\{\!|v|\!\} = v$, the thesis follows because $\mathcal{E}[\![v]\!]r = v = \mathcal{C}[\![v]\!]r$.

($expr = \mathsf{not}(expr_1)$) Since $\mathcal{T}_E\{\!|expr|\!\} = \dot{\neg}\, \mathcal{T}_E\{\!|expr_1|\!\}$ and, by the induction hypothesis, $\mathcal{E}[\![expr_1]\!]r = \mathcal{C}[\![\mathcal{T}_E\{\!|expr_1|\!\}]\!]r$, the thesis follows due to the correspondence of the semantic clause of the operator $\dot{\neg}$ in Table 4.2 and that of the operator not in Table 3.4.

($expr = \mathsf{op}(expr_1, expr_2)$) Since $\mathcal{T}_E\{\!|expr|\!\} = \mathcal{T}_E\{\!|expr_1|\!\}\ \mathsf{getOp(op)}\ \mathcal{T}_E\{\!|expr_2|\!\}$ and, by the induction hypothesis, $\mathcal{E}[\![expr_1]\!]r = \mathcal{C}[\![\mathcal{T}_E\{\!|expr_1|\!\}]\!]r$ and $\mathcal{E}[\![expr_2]\!]r = \mathcal{C}[\![\mathcal{T}_E\{\!|expr_2|\!\}]\!]r$, the thesis follows due to the correspondence of the semantic clause of the expression operator op in Table 4.2 and that of the constraint operator getOp(op) in Table 3.4.

□

Lemma 4.3. *For all $o \in Obligation$ and $r \in R$ it holds that*

$$\mathcal{O}[\![o]\!]r = fo \;\; \Leftrightarrow \;\; \mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o|\!\}]\!]r = \mathsf{true} \;\; and \;\; \mathcal{O}[\![o]\!]r = \mathsf{error} \;\; \Leftrightarrow \;\; \mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o|\!\}]\!]r = \mathsf{false}$$

Proof. We only prove the ($\Rightarrow$) implication as the proof for the other direction proceeds in a specular way. Let $o = [e\ t\ PepAction(expr^*)]$ with $expr^* = expr_1 \ldots expr_n$. By the clause (T-2), it is translated into the constraint

$$c = \bigwedge_{expr_j \in expr^*} \neg\mathtt{isMiss}(\mathcal{T}_E\{\!|expr_j|\!\}) \wedge \neg\mathtt{isErr}(\mathcal{T}_E\{\!|expr_j|\!\})$$

We now proceed by case analysis on $\mathcal{O}[\![o]\!]r$.

($\mathcal{O}[\![o]\!]r = fo$) We have to prove that $\mathcal{C}[\![c]\!]r = \mathsf{true}$. By the definition of $\mathcal{C}$, $\mathcal{C}[\![c]\!]r = \mathsf{true}$ corresponds to

$$\forall j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\neg\ \texttt{isMiss}(\mathcal{T}_E\{\!|expr_j|\!\})]\!]r = \mathsf{true} \ \wedge \ \mathcal{C}[\![\neg\ \texttt{isErr}(\mathcal{T}_E\{\!|expr_j|\!\})]\!]r = \mathsf{true}$$

According to the constraint semantics of $\neg$, `isMiss` and `isErr`, this corresponds to

$$\forall j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r \neq \bot \ \wedge \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r \neq \mathsf{error}$$

By the hypothesis $\mathcal{O}[\![o]\!]r = fo$ and the clauses (S-3a) and (S-2), we have

$$\mathcal{E}[\![expr^*]\!]r = \mathcal{E}[\![expr_1]\!]r \bullet \ldots \bullet \mathcal{E}[\![expr_n]\!]r = w_1 \ \ldots w_n$$

where w_j stands for a literal value or a set of values. Thus, by Lemma 4.2, we get that

$$\forall j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r = w_j \notin \{\bot, \mathsf{error}\}$$

which proves the thesis.

($\mathcal{O}[\![o]\!]r = \mathsf{error}$) We have to prove that $\mathcal{C}[\![c]\!]r = \mathsf{false}$. By the definition of $\mathcal{C}$, $\mathcal{C}[\![c]\!]r = \mathsf{false}$ corresponds to

$$\exists j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\neg\ \texttt{isMiss}(\mathcal{T}_E\{\!|expr_j|\!\})]\!]r = \mathsf{false} \ \vee \ \mathcal{C}[\![\neg\ \texttt{isErr}(\mathcal{T}_E\{\!|expr_j|\!\})]\!]r = \mathsf{false}$$

According to the constraint semantics of $\neg$, `isMiss` and `isErr`, this corresponds to

$$\exists j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r = \bot \ \vee \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r = \mathsf{error}$$

By the hypothesis $\mathcal{O}[\![o]\!]r = \mathsf{error}$ and the clauses (S-3a) and (S-2), we have

$$\mathcal{E}[\![expr^*]\!]r = \mathcal{E}[\![expr_1]\!]r \bullet \ldots \bullet \mathcal{E}[\![expr_n]\!]r \neq w^* \ \Rightarrow \ \exists j \in \{1,\ldots,n\} : \mathcal{E}[\![expr_j]\!]r \in \{\bot, \mathsf{error}\}$$

Thus, by Lemma 4.2, we obtain that

$$\exists j \in \{1,\ldots,n\} \ : \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr_j|\!\}]\!]r \in \{\bot, \mathsf{error}\}$$

which proves the thesis.

□

Lemma 4.4. *For all* $\mathsf{alg}_{\mathsf{all}} \in Alg$, $r \in R$ *and policies* $p_1,\ldots,p_s \in Policy$ *such that* $\forall\, i \in \{1,\ldots,s\} \ : \ \mathcal{P}[\![p_i]\!]r = \langle dec_i \ fo_i^* \rangle \ \Leftrightarrow \ \mathcal{C}[\![\mathcal{T}_P\{\!|p_i|\!\} \downarrow_{dec_i}]\!]r = \mathsf{true}$*, it holds that*

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1 \ \ldots \ p_s]\!]r = \langle dec \ fo^* \rangle \ \Leftrightarrow \ \mathcal{C}[\![\mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, p_1 \ \ldots \ p_s|\!\} \downarrow_{dec}]\!]r = \mathsf{true}$$

Proof. Since the considered algorithms use the all fulfilment strategy, by the hypothesis and the clauses (S-6a) and (T-4), the thesis is equivalent to prove that

$$\otimes\mathsf{alg}(\otimes\mathsf{alg}(\ldots \otimes \mathsf{alg}(\langle dec_1 \ fo_1^* \rangle, \langle dec_2 \ fo_2^* \rangle), \ldots), \langle dec_s \ fo_s^* \rangle) = \langle dec \ fo^* \rangle$$
$$\Longleftrightarrow$$
$$\mathcal{C}[\![\mathsf{alg}(\mathsf{alg}(\ldots \mathsf{alg}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\}), \ldots), \mathcal{T}_P\{\!|p_s|\!\}) \downarrow_{dec}]\!]r = \mathsf{true}$$

The proof proceeds by case analysis on alg. In what follows, we only report the case of the p-over algorithm, as the other ones are similar and derive directly from Tables 3.5 and 4.3.

Notably, when $s = 1$, we have $\otimes\mathsf{p\text{-}over}(\mathcal{P}[\![p_1]\!]r) = \mathcal{P}[\![p_1]\!]r$ and $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}) = \mathcal{T}_P\{\!|p_1|\!\}$ by definition, hence the thesis directly follows from the hypothesis that $\mathcal{P}[\![p_1]\!]r = \langle dec_1 \ fo_1^* \rangle \ \Leftrightarrow \ \mathcal{C}[\![\mathcal{T}_P\{\!|p_1|\!\} \downarrow_{dec_1}]\!]r = \mathsf{true}$. For the remaining cases, we proceed by induction on the number s of policies to combine.

Base Case $(s = 2)$ We must prove that

$$\otimes\mathsf{p\text{-}over}(\langle dec_1\ fo_1^*\rangle, \langle dec_2\ fo_2^*\rangle) = \langle dec\ fo^*\rangle \Leftrightarrow \mathcal{C}[\![\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\})\downarrow_{dec}]\!]r = \mathsf{true}\,.$$

For the sake of simplicity, in the following we omit the sequences of fulfilled obligations, as their combination does not affect the decision dec returned by $\otimes$p-over. We proceed by case analysis on the decision dec.

($dec = \mathsf{permit}$) It follows that $dec_1 = \mathsf{permit}$ or $dec_2 = \mathsf{permit}$. Moreover, by definition we have $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\})\downarrow_p = \mathcal{T}_P\{\!|p_1|\!\}\downarrow_p \vee \mathcal{T}_P\{\!|p_2|\!\}\downarrow_p$.

($dec = \mathsf{deny}$) It follows that $dec_1, dec_2 \in \{\mathsf{deny}, \mathsf{not\text{-}app}\}$. Moreover, by definition we have $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\})\downarrow_d = (\mathcal{T}_P\{\!|p_1|\!\}\downarrow_d \wedge \mathcal{T}_P\{\!|p_2|\!\}\downarrow_d) \vee (\mathcal{T}_P\{\!|p_1|\!\}\downarrow_d \wedge \mathcal{T}_P\{\!|p_2|\!\}\downarrow_n) \vee (\mathcal{T}_P\{\!|p_1|\!\}\downarrow_n \wedge \mathcal{T}_P\{\!|p_2|\!\}\downarrow_d)$.

($dec = \mathsf{not\text{-}app}$) It follows that $dec_1 = dec_2 = \mathsf{not\text{-}app}$. Moreover, by definition we have $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\})\downarrow_n = \mathcal{T}_P\{\!|p_1|\!\}\downarrow_n \wedge \mathcal{T}_P\{\!|p_2|\!\}\downarrow_n$.

($dec = \mathsf{indet}$) It follows that $dec_1 = \mathsf{indet}$ or $dec_2 = \mathsf{indet}$ and $dec_1, dec_2 \neq \mathsf{permit}$. Moreover, by definition we have $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\})\downarrow_i = (\mathcal{T}_P\{\!|p_1|\!\}\downarrow_i \wedge \neg\mathcal{T}_P\{\!|p_2|\!\}\downarrow_p) \vee (\neg\mathcal{T}_P\{\!|p_1|\!\}\downarrow_p \wedge \mathcal{T}_P\{\!|p_2|\!\}\downarrow_i)$.

In any case, thesis follows from the hypothesis on $\mathcal{T}_P\{\!|p_i|\!\}$ and the definition of $\mathcal{C}$.

Inductive Case $(s = k + 1)$ By the induction hypothesis the thesis holds for k policies, that is

$$\otimes\mathsf{alg}(\otimes\mathsf{alg}(\ldots\otimes\mathsf{alg}(\langle dec_1\ fo_1^*\rangle, \langle dec_2\ fo_2^*\rangle), \ldots), \langle dec_k\ fo_k^*\rangle) = \langle dec'\ fo'^*\rangle$$
$$\Longleftrightarrow$$
$$\mathcal{C}[\![\mathsf{alg}(\mathsf{alg}(\ldots\mathsf{alg}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\}), \ldots), \mathcal{T}_P\{\!|p_k|\!\})\downarrow_{dec'}]\!]r = \mathsf{true}$$

The thesis then follows by repeating the case analysis on decision dec of the 'Base Case' once we replace $\langle dec_1\ fo_1^*\rangle$, $\langle dec_2\ fo_2^*\rangle$, $\mathcal{T}_P\{\!|p_1|\!\}$ and $\mathcal{T}_P\{\!|p_2|\!\}$ by $\langle dec'\ fo'^*\rangle$, $\langle dec_s\ fo_s^*\rangle$, $\mathsf{p\text{-}over}(\mathsf{p\text{-}over}(\ldots\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}, \mathcal{T}_P\{\!|p_2|\!\}), \ldots), \mathcal{T}_P\{\!|p_k|\!\})$ and $\mathcal{T}_P\{\!|p_s|\!\}$, respectively.

□

Theorem 4.5 (Policy Semantic Correspondence)**.** *For all $p \in Policy$ enclosing combining algorithms only using* all *as fulfilment strategy, and $r \in R$, it holds that*

$$\mathcal{P}[\![p]\!]r = \langle dec\ fo^*\rangle \quad\Leftrightarrow\quad \mathcal{C}[\![\mathcal{T}_P\{\!|p|\!\}\downarrow_{dec}]\!]r = \mathsf{true}$$

Proof. The proof proceeds by induction on the depth i of p (see definition of depth in Section 3.5.6).

Base Case $(i = 0)$ This means that p is of the form $(e\ \ \mathsf{target} : expr\ \ \mathsf{obl} : o^*)$. We proceed by case analysis on dec.

($dec = \mathsf{permit}$) By the clause (S-4a), it follows that

$$\mathcal{E}[\![expr]\!]r = \mathsf{true} \ \wedge\ \mathcal{O}[\![o^*|_{\mathsf{permit}}]\!]r = fo^*$$

Thus, by Lemma 4.2, it follows that

$$\mathcal{C}[\![\mathcal{T}_E\{\!|expr|\!\}]\!]r = \mathsf{true}$$

and, by Lemma 4.3 and the clause (T-2), it follows that

$$\mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}]\!]r = \mathsf{true}$$

On the other hand, by the clause (T-3a), we have that

$$\mathcal{T}_P\{\!|(\mathsf{permit}\ \ \mathsf{target}:expr\ \ \mathsf{obl}:o^*)|\!\}\downarrow_p = \mathcal{T}_E\{\!|expr|\!\} \wedge \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}$$

Hence, by the definition of $\mathcal{C}$, we can conclude that

$$\begin{aligned}\mathcal{C}[\![\mathcal{T}_P\{\!|(\mathsf{permit}\ \ \mathsf{target}:expr\ \ \mathsf{obl}:o^*)|\!\}\downarrow_p]\!]r &= \\ \mathcal{C}[\![\mathcal{T}_E\{\!|expr|\!\}]\!]r \wedge \mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}]\!]r &= \\ \mathsf{true} \wedge \mathsf{true} &= \mathsf{true}\end{aligned}$$

which proves the thesis.

($dec = \mathsf{deny}$) We omit the proof since it proceeds like the previous case.

($dec = \mathsf{not\text{-}app}$) By the clause (S-4a), it follows that

$$\mathcal{E}[\![expr]\!]r = \mathsf{false}\ \vee\ \mathcal{E}[\![expr]\!]r = \bot$$

By the clause (T-3a), we have that

$$\mathcal{T}_P\{\!|(e\ \ \mathsf{target}:expr\ \ \mathsf{obl}:o^*)|\!\}\downarrow_n = \neg\,\mathcal{T}_E\{\!|expr|\!\}$$

Hence, the thesis directly follows by Lemma 4.2 and the definition of $\mathcal{C}$.

($dec = \mathsf{indet}$) By the clause (S-4a), the `otherwise` condition holds, that is

$$\neg(\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = fo^*)\ \wedge\ \neg(\mathcal{E}[\![expr]\!]r = \mathsf{false}\ \vee\ \mathcal{E}[\![expr]\!]r = \bot)$$

By applying standard boolean laws and reasoning on function codomains, this condition can be rewritten as follows

$$\begin{aligned}&\neg(\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = fo^*)\ \wedge\ \neg(\mathcal{E}[\![expr]\!]r = \mathsf{false}\ \vee\ \mathcal{E}[\![expr]\!]r = \bot)\\ &= (\mathcal{E}[\![expr]\!]r \neq \mathsf{true}\ \vee\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error})\ \wedge\ (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\})\\ &= \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error})\\ &= \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee\\ &\quad (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \vee\\ &\quad (\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error})\\ &= \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee (\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error})\end{aligned}$$

On the other hand, by the clause (T-3a), we have that

$$\begin{aligned}&\mathcal{T}_P\{\!|(e\ \ \mathsf{target}:expr\ \ \mathsf{obl}:o^*)|\!\}\downarrow_i =\\ &\quad \neg\,(\mathtt{isBool}(\mathcal{T}_E\{\!|expr|\!\})\ \vee\ \mathtt{isMiss}(\mathcal{T}_E\{\!|expr|\!\}))\ \vee\ (\mathcal{T}_E\{\!|expr|\!\} \wedge \neg\,\mathcal{T}_{Ob}\{\!|o^*|_e\ |\!\})\end{aligned}$$

The thesis then follows by Lemmas 4.2 and 4.3 and the definition of $\mathcal{C}$.

Inductive Case ($i = k + 1$) p is of the form $\{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target}:expr\ \ \mathsf{policies}:(p^+)^k\ \mathsf{obl}:o^*\}$. We proceed by case analysis on dec.

($dec = \mathsf{permit}$) By the clause (S-4b), it follows that

$$\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle\mathsf{permit}\ \ fo_1^*\rangle\ \wedge\ \mathcal{O}[\![o^*|_{\mathsf{permit}}]\!]r = fo_2^*$$

Thus, by Lemma 4.2, it follows that

$$\mathcal{E}[\![expr]\!]r = \ \mathcal{C}[\![\mathcal{T}_E\{\!|expr|\!\}]\!]r = \mathsf{true}$$

and, by Lemma 4.3 and the clause (T-2), it follows that

$$\mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}]\!]r = \mathsf{true}$$

Since by the induction hypothesis, for all p_i^h in $(p^+)^k$ with $h \leq k$, it holds that

$$\mathcal{P}[\![p_i^h]\!]r = \langle dec_i\ fo^* \rangle \quad \Leftrightarrow \quad \mathcal{C}[\![\mathcal{T}_P\{\!|p_i^h|\!\}\downarrow_{dec_i}]\!]r = \mathsf{true}$$

then, by Lemma 4.4, it follows that

$$\mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_p = \mathsf{true}$$

On the other hand, by the clause (T-3b), we have that

$$\begin{array}{c}\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target}: expr\ \ \mathsf{policies}: (p^+)^k\ \mathsf{obl}: o^*\ \}|\!\}\downarrow_p = \\ \mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_p \wedge\ \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}\end{array}$$

Hence, by the definition of $\mathcal{C}$, we can conclude that

$$\begin{array}{rcl}\mathcal{C}[\![\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target}: expr\ \ \mathsf{policies}: (p^+)^k\ \mathsf{obl}: o^*\ \}|\!\}\downarrow_p]\!]r & = & \\ \mathcal{C}[\![\mathcal{T}_E\{\!|expr|\!\}]\!]r \wedge \mathcal{C}[\![\mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_p]\!]r \wedge \mathcal{C}[\![\mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}\ |\!\}]\!]r & = & \\ \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true} & = & \mathsf{true}\end{array}$$

which proves the thesis.

($dec = \mathsf{deny}$) We omit the proof since it proceeds like the previous case.

($dec = \mathsf{not\text{-}app}$) By the clause (S-4b), it follows that

$$\mathcal{E}[\![expr]\!]r = \mathsf{false} \vee\ \mathcal{E}[\![expr]\!]r = \perp \vee (\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{not\text{-}app})$$

By the clause (T-3b), we have that

$$\begin{array}{l}\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target}: expr\ \ \mathsf{policies}: (p^+)^k\ \mathsf{obl}: o^*\ \}|\!\}\downarrow_n = \\ \neg\ \mathcal{T}_E\{\!|expr|\!\} \vee\ (\mathcal{T}_E\{\!|expr|\!\} \wedge\ \mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_n)\end{array}$$

The thesis then directly follows by Lemmas 4.2 and 4.4, due to the induction hypothesis and the definition of $\mathcal{C}$.

($dec = \mathsf{indet}$) By the clause (S-4b), the `otherwise` condition holds, that is

$$\begin{array}{l}\neg(\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e\ \ fo_1^*\rangle \wedge\ \mathcal{O}[\![o^*|_e]\!]r = fo_2^*) \wedge \\ \neg(\mathcal{E}[\![expr]\!]r = \mathsf{false} \vee\ \mathcal{E}[\![expr]\!]r = \perp \vee (\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{not\text{-}app}))\end{array}$$

By applying standard boolean laws and reasoning on function codomains, this condition can be rewritten as follows

$$\begin{array}{l}\neg(\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e\ \ fo_1^*\rangle \wedge\ \mathcal{O}[\![o^*|_e]\!]r = fo_2^*) \wedge \\ \neg(\mathcal{E}[\![expr]\!]r = \mathsf{false} \vee\ \mathcal{E}[\![expr]\!]r = \perp \vee (\mathcal{E}[\![expr]\!]r = \mathsf{true}\ \wedge\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{not\text{-}app})) \\ = \\ (\mathcal{E}[\![expr]\!]r \neq \mathsf{true} \vee\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \in \{\mathsf{not\text{-}app}, \mathsf{indet}\} \vee\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \wedge \\ (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \perp\} \wedge (\mathcal{E}[\![expr]\!]r \neq \mathsf{true} \vee\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app})) \\ = \\ (\mathcal{E}[\![expr]\!]r \neq \mathsf{true} \vee\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \in \{\mathsf{not\text{-}app}, \mathsf{indet}\} \vee\ \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \wedge \\ (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \perp\} \vee (\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \perp\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app}))\end{array}$$

$$
\begin{aligned}
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \in \{\mathsf{not\text{-}app}, \mathsf{indet}\}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \in \{\mathsf{not\text{-}app}, \mathsf{indet}\}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r \neq \mathsf{not\text{-}app} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet} \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e \;\; fo^* \rangle \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \wedge \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e \;\; fo^* \rangle \wedge \mathcal{O}[\![o^*|_e]\!]r = \mathsf{error}) \\
&= \\
&\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\wedge\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}\;) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\wedge\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle \mathsf{permit} \;\; fo^* \rangle \wedge \mathcal{O}[\![o^*|_{\mathsf{permit}}]\!]r = \mathsf{error}) \vee \\
&(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\wedge\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle \mathsf{deny} \;\; fo^* \rangle \wedge \mathcal{O}[\![o^*|_{\mathsf{deny}}]\!]r = \mathsf{error})
\end{aligned}
$$

where the last step exploits the fact that $e \in \{\mathsf{permit}, \mathsf{deny}\}$.
On the other hand, by the clause (T-3b), we have that

$$
\begin{aligned}
&\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}} \;\; \mathsf{target} : expr \;\; \mathsf{policies} : (p^+)^k \; \mathsf{obl} : o^* \;\}|\!\} \downarrow_{\mathsf{indet}} = \\
&\quad \neg\,(\mathtt{isBool}(\mathcal{T}_E\{\!|expr|\!\}) \;\vee\; \mathtt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})) \\
&\quad \vee\, (\mathcal{T}_E\{\!|expr|\!\} \wedge \;\mathcal{T}_A\{\!|a, (p^+)^k|\!\} \downarrow_i) \\
&\quad \vee\, (\mathcal{T}_E\{\!|expr|\!\} \wedge \;\mathcal{T}_A\{\!|a, (p^+)^k|\!\} \downarrow_p \wedge \neg\, \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{permit}}|\!\}\;) \\
&\quad \vee\, (\mathcal{T}_E\{\!|expr|\!\} \wedge \;\mathcal{T}_A\{\!|a, (p^+)^k|\!\} \downarrow_d \wedge \neg\, \mathcal{T}_{Ob}\{\!|o^*|_{\mathsf{deny}}|\!\}\;)
\end{aligned}
$$

The thesis then follows by Lemmas 4.2, 4.3 and 4.4, due to the induction hypothesis and the definition of $\mathcal{C}$.

□

It is worth noticing that these results can be easily tailored to extensions of FACPL. For instance, in case of additional expression operators, it only requires devising a constraint

operator (or a combination thereof) that faithfully represents the semantics of the new operator.

Additionally, from the previous theorems it follows that policy constraint tuples partition the set of input requests, that is each request satisfies only one of the constraints of a tuple. Basically, the following corollary extends Theorem 3.1 to constraint tuples.

Corollary 4.6 (Constraint-based partition)**.** *For all* $r \in R$ *and* $p \in Policy$*, such that* $\mathcal{T}_P\{\!|p|\!\} = \langle \mathsf{permit} : c_1 \ \mathsf{deny} : c_2 \ \mathsf{not\text{-}app} : c_3 \ \mathsf{indet} : c_4 \rangle$*, it holds that*

$$\exists! k \in \{1,\ldots,4\} \ : \ \mathcal{C}[\![c_k]\!]r = \mathsf{true} \ \wedge \ \textstyle\bigwedge_{j\in\{1,\ldots,4\}\setminus\{k\}} \mathcal{C}[\![c_j]\!]r = \mathsf{false}$$

Proof. The thesis immediately follows from Theorems 3.1 and 4.5. □

4.3.4 Constraint-based Representation of the e-Health Case Study

We now apply the translation functions just introduced to (a part of) the e-Health case study formalised in Section 3.4. For the sake of presentation, we shorten the attribute names used within the defined FACPL policies. For instance, the rule addressing Requirement (Eh-1) becomes as follows

$$\begin{array}{l}(\ \mathsf{permit}\ \ \mathsf{target}: \mathsf{equal}(\mathsf{sub/role}, \text{“doctor”})\ \mathsf{and}\ \mathsf{equal}(\mathsf{act/id}, \text{“write”}) \\ \qquad\qquad\qquad\qquad \mathsf{and}\ \mathsf{in}(\text{“e-Pre-Write”}, \mathsf{sub/perm})\ \mathsf{and}\ \mathsf{in}(\text{“e-Pre-Read”}, \mathsf{sub/perm}))\end{array}$$

Its translation starts by applying function $\mathcal{T}_E$ to the target expression. The resulting constraint is as follows

$$\begin{array}{l} c_{trg1} \triangleq \mathsf{sub/role} = \text{“doctor”} \ \dot\wedge \ \mathsf{act/id} = \text{“write”} \ \dot\wedge \ \text{“e-Pre-Write”} \in \mathsf{sub/perm} \\ \qquad \dot\wedge \ \text{“e-Pre-Read”} \in \mathsf{sub/perm} \end{array}$$

The translation proceeds by considering obligations; in this case they are missing (i.e., they correspond to the empty sequence ϵ), hence the constraint true is obtained. Function $\mathcal{T}_P$ finally defines the constraint tuple for the rule as follows

$$\begin{array}{lll} \langle\ \mathsf{permit}: & c_{trg1} \wedge \mathsf{true} & \mathsf{deny}: \ \mathsf{false} \\ \ \ \mathsf{not\text{-}app}: & \neg c_{trg1} & \mathsf{indet}: \ \neg(\mathtt{isBool}(c_{trg1}) \ \vee \ \mathtt{isMiss}(c_{trg1})) \vee (c_{trg1} \wedge \neg\mathsf{true}) \ \rangle \end{array}$$

The tuples for the rules addressing Requirements (Eh-2) and (Eh-3) are defined similarly, they only differ in the constraints representing their targets, which are denoted as c_{trg2} and c_{trg3}, respectively.

We can now define the constraint-based representation of Policy (Eh-A). Besides the target expression, which is straightforwardly translated to the constraint $c_{trgP} \triangleq \mathsf{res/typ} =$ “e-Pre”, the constraint tuple is built up from the result of function $\mathcal{T}_A$ representing the application of the algorithm p-over. Specifically, the constraint tuples of rules are iteratively combined according to Table 4.3. For example, the combination of the first two rules generates the following tuple

$$\begin{array}{l} \langle\ \mathsf{permit}: \ (c_{trg1} \wedge \mathsf{true}) \vee (c_{trg2} \wedge \mathsf{true}) \\ \ \ \mathsf{deny}: \ (\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \mathsf{false}) \\ \ \ \mathsf{not\text{-}app}: \ \neg c_{trg1} \wedge \neg c_{trg2} \\ \ \ \mathsf{indet}: ((\neg(\mathtt{isBool}(c_{trg1}) \vee \ \mathtt{isMiss}(c_{trg1})) \vee (c_{trg1} \wedge \neg\mathsf{true})) \wedge \neg(c_{trg2} \wedge \mathsf{true})) \\ \qquad \vee(\neg(c_{trg1} \wedge \mathsf{true}) \wedge (\neg(\mathtt{isBool}(c_{trg2}) \ \vee \ \mathtt{isMiss}(c_{trg2})) \vee (c_{trg2} \wedge \neg\mathsf{true}))) \ \rangle \end{array}$$

Notably, the deny constraint is never satisfied, because it is a disjunction of conjunctions having at least one false term as argument. This is somewhat expected, because the rules have the permit effect and the used combining algorithm is p-over. This tuple is then combined with that of the remaining rule in a similar way.

To generate the constraint tuple of the policy, we also need the constraint-based representation of its obligations. The policy contains only one obligation, which has effect permit. The corresponding constraint is as follows

$$c_{obl\_p} \triangleq \bigwedge_{n \in \{\text{sys/time,res/typ,sub/id,act/id}\}} \neg\texttt{isMiss}(n) \wedge \neg\texttt{isErr}(n)$$

The constraint for obligations with effect deny, which are missing, is instead true.

Finally, the constraint tuple of Policy (Eh-A) generated by function $\mathcal{T}_P$ is as follows

$$\begin{aligned}
&\langle \text{permit} : c_{trgP} \wedge ((c_{trg1} \wedge \text{true}) \vee (c_{trg2} \wedge \text{true}) \vee (c_{trg3} \wedge \text{true})) \wedge c_{obl_p} \\
&\quad \text{deny} : c_{trgP} \wedge ((((\text{false} \wedge \text{false}) \vee (\text{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \text{false})) \wedge \text{false}) \\
&\qquad \vee (((\text{false} \wedge \text{false}) \vee (\text{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \text{false})) \wedge \neg c_{trg3}) \\
&\qquad \vee ((\neg c_{trg1} \wedge \neg c_{trg2}) \wedge \text{false})) \wedge \text{true} \\
&\quad \text{not-app} : \neg c_{trgP} \vee (c_{trgP} \wedge (\neg c_{trg1} \wedge \neg c_{trg2} \wedge \neg c_{trg3})) \\
&\quad \text{indet} : \neg(\texttt{isBool}(c_{trgP}) \vee \texttt{isMiss}(c_{trgP})) \\
&\qquad \vee (c_{trgP} \wedge (((\neg(\texttt{isBool}(c_{trg1}) \vee \texttt{isMiss}(c_{trg1})) \vee (c_{trg1} \wedge \neg\text{true})) \wedge \neg(c_{trg2} \wedge \text{true})) \\
&\qquad\quad \vee \neg((c_{trg1} \wedge \text{true}) \wedge (\neg(\texttt{isBool}(c_{trg2}) \vee \texttt{isMiss}(c_{trg2})) \vee (c_{trg2} \wedge \neg\text{true}))) \wedge \neg(c_{trg3} \wedge \text{true})) \\
&\qquad\quad \vee (\neg((c_{trg1} \wedge \text{true}) \vee (c_{trg2} \wedge \text{true})) \wedge (\neg(\texttt{isBool}(c_{trg3}) \vee \texttt{isMiss}(c_{trg3})) \vee (c_{trg3} \wedge \neg\text{true}))) \\
&\qquad \vee (c_{trgP} \wedge ((c_{trg1} \wedge \text{true}) \vee (c_{trg2} \wedge \text{true}) \vee (c_{trg3} \wedge \text{true})) \wedge \neg c_{obl_p}) \\
&\qquad \vee (c_{trgP} \wedge ((((\text{false} \wedge \text{false}) \vee (\text{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \text{false})) \wedge \text{false}) \\
&\qquad\quad \vee (((\text{false} \wedge \text{false}) \vee (\text{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \text{false})) \wedge \neg c_{trg3}) \\
&\qquad\quad \vee ((\neg c_{trg1} \wedge \neg c_{trg2}) \wedge \text{false})) \wedge \neg\text{true}) \qquad \rangle
\end{aligned}$$

This example demonstrates that the constraints resulting from the translation are a single-layered representation of policies that fully details all the aspects of policy evaluation. It is also worth noticing that the translation functions are applied without considering possible optimisations, e.g., simplifications of unsatisfiable constraints like that of deny. It is also evident that the evaluation, as well as the generation, of such constraints cannot be done manually, but requires a tool support.

4.4 Formalisation of Properties

This section formalises a set of properties of interest for FACPL policies. In particular, we formalise properties that refer to the expected authorisation of single requests, i.e. *authorisation properties* (Section 4.4.1), and to the relationships among policies with respect to the authorisations they enforce, i.e. *structural properties* (Section 4.4.2). We conclude with some examples of properties from the e-Health case study (Section 4.4.3).

4.4.1 Authorisation Properties

An authorisation property aims at predicting the possible authorisations a policy calculates for a given (partial) request. Since FACPL policies and, in general, ABAC policies, do not enjoy the *safety* property (see Section 3.5.6), authorisation properties should also analyse how additional attributes possibly added to a request, thus extending it, might change its authorisation in a possibly unexpected way. Intuitively, it is important to consider not only

the authorisation decisions of specific requests, but also those of their extensions because, e.g., a malicious user could try to exploit them to circumvent the access control system. To this aim, we introduce and exploit the notion of *request extension set*.

The request extension set of a given request r is defined as follows

$$Ext(r) \triangleq \{r' \in R \mid r(n) \neq \bot \;\; \Rightarrow \;\; r'(n) = r(n)\}$$

The set is formed by all those requests that possibly extend request r with new attributes assignments not already defined by r.

The formalisation of the authorisation properties we propose follows.

Evaluate-To. This property, written r `eval` dec, requires the policy under examination to evaluate the request r to decision dec. The satisfiability, written `sat`, of the *Evaluate-To* property by a policy p is defined as follows

$$p \texttt{ sat } r \texttt{ eval } dec \qquad \textit{iff} \qquad \mathcal{P}[\![p]\!]r = \langle dec \;\; fo^* \rangle$$

In practice, the verification of the property boils down to apply the semantic function $\mathcal{P}$ to p and r, and check that the resulting decision is dec.

May-Evaluate-To. This property, written r $\texttt{eval}_{\texttt{may}}$ dec, requires that *at least one* request extending the request r evaluates to decision dec. The satisfiability of the *May-Evaluate-To* property by a policy p is defined as follows

$$p \texttt{ sat } r \texttt{ eval}_{\texttt{may}} \; dec \qquad \textit{iff} \qquad \exists\, r' \in Ext(r) \;:\; \mathcal{P}[\![p]\!]r' = \langle dec \;\; fo^* \rangle$$

This property, as well as the next one, addresses additional attributes extending the request r by considering the requests in its extension set $Ext(r)$.

Must-Evaluate-To. This property, written r $\texttt{eval}_{\texttt{must}}$ dec, differs from the previous one as it requires *all* the extended requests to evaluate to decision dec. The satisfiability of the *Must-Evaluate-To* property by a policy p is defined as follows

$$p \texttt{ sat } r \texttt{ eval}_{\texttt{must}} \; dec \qquad \textit{iff} \qquad \forall r' \in Ext(r) \;:\; \mathcal{P}[\![p]\!]r' = \langle dec \;\; fo^* \rangle$$

Notably, additional properties can be obtained by combining the previous ones, like a property requiring, e.g., that all requests in $Ext(r)$ may evaluate to dec and must not evaluate to dec'. Indeed, request extensions can be exploited to track down possibly unexpected authorisations.

It is worth noticing that the formalisation approach based on request extensions is practically feasible, although such sets might be infinite. Indeed, Lemma 3.2 ensures that the attribute names whose assignments generate significant extensions of a given request are only those belonging to the finite set of attribute names occurring within the considered policy. This fact paves the way for carrying out property verification by means of SMT solvers.

4.4.2 Structural Properties

A structural property refers to the structure of the sets of authorisations enforced by one or multiple policies. In case of multiple policies, the properties aim at characterising the relationships among the policies. Different structural properties have been proposed in

the literature (e.g. in [FKMT05] and [KHP07]) by pursuing different approaches for their definition and verification. Here, we consider a set of commonly addressed properties and provide a uniform characterisation thereof in terms of requests and policy semantics.

Completeness. A policy is `complete` if it applies to all requests. Thus, the satisfiability of the property by a policy p is defined as follows

$$p \texttt{ sat complete} \qquad \textit{iff} \qquad \forall\, r \in R \,:\, \mathcal{P}[\![p]\!]r = \langle dec \;\; fo^* \rangle, dec \neq \textsf{not-app}$$

Essentially, we require that the policy applies to any request, i.e. it always returns a decision different from not-app. Notably, in this formulation indet is considered as an acceptable decision; a more restrictive formulation could only accept permit and deny.

Disjointness. Disjointness among policies means that such policies apply to disjoint sets of requests. Thus, this property, written `disjoint` p', requires that there is no request for which both the policy under examination and the policy p' evaluate to permit or deny. The satisfiability of the property by a policy p is defined as follows

$$p \texttt{ sat disjoint } p' \qquad \textit{iff} \qquad \forall\, r \in R \,: \\ \mathcal{P}[\![p]\!]r = \langle dec \;\; fo^* \rangle, \mathcal{P}[\![p']\!]r = \langle dec' \;\; fo'^* \rangle, \{\, dec, dec' \,\} \not\subseteq \{\textsf{permit}, \textsf{deny}\}$$

It is worth noticing that disjoint polices can be combined with the assurance that the allowed or forbidden authorisations enforced by each of them are not in conflict, which simplifies the choice of the combining algorithm to be used.

Coverage. Coverage among policies means that one of such policies enforces the same decisions as the other ones. More specifically, the property `cover` p' requires that for each request r for which p' evaluates to an admissible decision, i.e. permit or deny, the policy under examination evaluates to the same decision. The satisfiability of the property by a policy p is defined as follows

$$p \texttt{ sat cover } p' \qquad \textit{iff} \qquad \forall\, r \in R \,: \\ \mathcal{P}[\![p']\!]r = \langle dec \;\; fo^* \rangle, dec \in \{\textsf{permit}, \textsf{deny}\} \;\; \Rightarrow \;\; \mathcal{P}[\![p]\!]r = \langle dec \;\; fo'^* \rangle$$

Thus, p calculates at least the same admissible decisions as p'. Consequently, if p' also covers p, the two policies enforce exactly the same admissible authorisations.

These structural properties statically predicate the relationships among policies and support system designers in developing and maintaining policies. One technique they enable is the *change-impact analysis* [FKMT05]. This analysis examines the effect of policy modifications for discovering unintended consequences of such changes.

4.4.3 Properties on the e-Health Case Study

By way of example, we address in terms of authorisation and structural properties the case of pharmacists willing to write an e-Prescription in the e-Health case study.

Given the patient consent policies in Section 3.4, i.e. Policies (Eh-A) and (Eh-B), we can verify whether they disallow the access to a pharmacist that wants to write an e-Prescription. To this aim, we define an *Evaluate-To* property[2] as follows

$$(\textsf{sub/role}, \text{“pharmacist”})(\textsf{act/id}, \text{“write”})(\textsf{res/typ}, \text{“e-Pre”}) \quad \texttt{eval} \quad \textsf{deny} \qquad \text{(Pr1)}$$

[2]For the sake of presentation, in this subsection we write requests using the FACPL syntax (i.e., they are specified as sequences of attributes) rather than using their semantics, i.e. functional representation.

which requires that such request evaluates to deny. Alternatively, by exploiting request extensions, we can check if there exists a request for which a pharmacist acting on e-Prescription can be evaluated to not-app. This corresponds to the *May-Evaluate-To* property defined as follows

$$(\text{sub/role}, \text{“pharmacist”})(\text{res/typ}, \text{“e-Pre”}) \quad \texttt{eval}_{\texttt{may}} \quad \text{not-app} \tag{Pr2}$$

The verification of these properties with respect to Policy (Eh-A) results in

$$Policy\ (Eh\text{-}A)\ \texttt{unsat}\ (Pr1) \qquad Policy\ (Eh\text{-}A)\ \texttt{sat}\ (Pr2)$$

where unsat indicates that the policy does not satisfy the property. Indeed, as already discussed in Section 3.4, each request assigning to act/id a value different from read evaluates to not-app, hence property (Pr1) is not satisfied while property (Pr2) holds. On the contrary, the verification with respect to Policy (Eh-B) results in

$$Policy\ (Eh\text{-}B)\ \texttt{sat}\ (Pr1) \qquad Policy\ (Eh\text{-}B)\ \texttt{unsat}\ (Pr2)$$

Both results are due to the internal policy (deny) which, together with the algorithm p-over, prevents not-app to be returned and enforces deny as default decision.

The analysis can also be conducted by relying on the structural properties. By verifying completeness, we can check if there exists a request that evaluates to not-app, and we get

$$Policy\ (Eh\text{-}A)\ \texttt{unsat complete} \qquad Policy\ (Eh\text{-}B)\ \texttt{sat complete}$$

As expected, Policy (Eh-A) does not satisfy completeness, i.e. there is at least one request that evaluates to not-app, whereas Policy (Eh-B) is complete. Instead, we can check if Policy (Eh-B) correctly refines Policy (Eh-A) by simply verifying coverage. We get

$$Policy\ (Eh\text{-}B)\ \texttt{sat cover}\ Policy\ (Eh\text{-}A)$$

This follows from the fact that Policy (Eh-B) evaluates to permit the same set of requests as Policy (Eh-A) and that Policy (Eh-A) never returns deny; the opposite coverage property does not clearly hold. It is also worth noticing that the two policies are not disjoint (in fact, they share the set of permitted requests).

4.5 Automated Property Verification

The verification of the properties we have just introduced requires extensive checks on large (possibly infinite) amounts of requests, hence, in order to be practically effective, tool support is essential. To this aim, we express the constraint formalism introduced in Section 4.3 by means of the SMT-LIB language [BST10], that is a standardised constraint language accepted by most of the SMT solvers. Intuitively, SMT-LIB is a strongly typed functional language expressly defined for the specification of constraints. On the basis of this SMT-LIB coding, we can define the strategies to follow for automatising the verification of properties by means of an SMT solver. Of course, the feasibility of the SMT-based reasoning crucially depends on decidability of the satisfiability checks to be done; in other words, the used SMT-LIB constructs must refer to decidable theories, as e.g. uninterpreted

$$\frac{v \in \mathit{Bool}}{\Gamma \vdash v : \mathit{Bool} \mid \mathsf{true}} \qquad \frac{v \in \mathit{Double}}{\Gamma \vdash v : \mathit{Double} \mid \mathsf{true}} \qquad \frac{v \in \mathit{String}}{\Gamma \vdash v : \mathit{String} \mid \mathsf{true}} \qquad \frac{v \in \mathit{Date}}{\Gamma \vdash v : \mathit{Date} \mid \mathsf{true}}$$

$$\frac{v \in 2^{\mathit{Value}}}{\Gamma \vdash v : 2^{\mathit{Value}} \mid \mathsf{true}} \qquad \frac{\Gamma(n) = X}{\Gamma \vdash n : X \mid \mathsf{true}} \qquad \frac{\Gamma \vdash \mathit{expr} : U \mid C}{\Gamma \vdash \mathsf{not}(\mathit{expr}) : \mathit{Bool} \mid C \wedge U = \mathit{Bool}}$$

$$\frac{\Gamma \vdash \mathit{expr}_1 : U_1 \mid C_1 \qquad \Gamma \vdash \mathit{expr}_2 : U_2 \mid C_2}{\Gamma \vdash \mathsf{eop}(\mathit{expr}_1, \mathit{expr}_2) : \mathit{Bool} \mid C_1 \wedge C_2 \wedge U_1 = \mathit{Bool} \wedge U_2 = \mathit{Bool}} \ \mathsf{eop} \in \{\mathsf{and}, \mathsf{or}\}$$

$$\frac{\Gamma \vdash \mathit{expr}_1 : U_1 \mid C_1 \qquad \Gamma \vdash \mathit{expr}_2 : U_2 \mid C_2}{\Gamma \vdash \mathsf{equal}(\mathit{expr}_1, \mathit{expr}_2) : \mathit{Bool} \mid C_1 \wedge C_2 \wedge U_1 = U_2} \qquad \frac{\Gamma \vdash \mathit{expr}_1 : U_1 \mid C_1 \qquad \Gamma \vdash \mathit{expr}_2 : 2^{U_2} \mid C_2}{\Gamma \vdash \mathsf{in}(\mathit{expr}_1, \mathit{expr}_2) : \mathit{Bool} \mid C_1 \wedge C_2 \wedge U_1 = U_2}$$

Table 4.4: Type inference rules for (an excerpt of) FACPL expressions (we use X as a type variable, U as a type name or a type variable, and we assume that Bool, Double, String, Date, 2^{Value} identify both the values' domains and their type names)

function and array theories. All these tasks are fully supported by automatic functionalities of the FACPL tools.

In the rest of this section, we first outline the SMT-LIB coding of our constraints (Section 4.5.1), then we present the strategies for the automated verification of properties (Section 4.5.2), and we conclude by commenting on the functionalities of the FACPL analysis tools (Section 4.5.3).

4.5.1 Expressing Constraints with SMT-LIB

We provide here a few insights on the SMT-LIB coding of our constraints. The key element of the coding strategy is the parametrised record type representing attributes. This type, named `TValue`, is defined as follows

```
(declare-datatypes (T) ((TValue (mk-val (val T)(miss Bool)(err Bool)))))
```

Hence, each attribute consists of a 3-valued record, whose first field `val` is the value with parametric type `T` assigned to the attribute, while the boolean fields `miss` and `err` indicate, respectively, if the attribute value is missing or has an unexpected type. Additional assertions, not shown here for the sake of presentation, ensure that the fields `miss` and `err` cannot be true at the same time, and that, when one of the last two fields is true, it takes precedence over `val`. Of course, a specification formed by multiple assertions is satisfied when all the assertions are satisfied.

The declaration of `TValue` outlines the syntax of SMT-LIB and its strongly typed nature. This means that each attribute occurring in a policy has to be typed, by properly instantiating the type parameter `T`. Since FACPL is an untyped language, to reconstruct the type of each attribute, we define the type inference system (whose excerpt is) reported in Table 4.4. The rules are straightforward and infer the judgment $\Gamma \vdash \mathit{expr} : U \mid C$ which, under the typing context Γ, assigns the type (or the type variable) U to the FACPL expression expr and generates the typing constraint C. Specifically, Γ is an injective function that associates a type variable to each attribute name, while C is basically made of conjunctions and disjunctions of equalities between variables and types. The generated typing constraint will be processed at the end of the inference process to establish well-typedness of an expression. Thus, a FACPL expression is *well-typed* if C is satisfiable, i.e. there exists

a type assignment for the typing variables occurring in C that satisfies C. Moreover, a FACPL policy is *well-typed* if the typing constraints generated by all the expressions occurring in the policy are satisfied by a same assignment. These type assignments are used to instantiate the type parameters of the SMT-LIB constraints representing well-typed policies.

The type inference system aims at statically discarding policies containing expressions that are not well-typed. For instance, given the expression $\mathsf{or}(\mathsf{cat/id}, \mathsf{equal}(\mathsf{cat/id}, 5))$ and the typing context $\Gamma(\mathsf{cat/id}) = X_{\mathsf{cat/id}}$, the inference rules assign the type $Bool$ to the expression and generate the constraint $X_{\mathsf{cat/id}} = Double \wedge X_{\mathsf{cat/id}} = Bool \wedge Bool = Bool$. This constraint is clearly unsatisfiable (as attribute cat/id cannot simultaneously be a double and a boolean), hence a policy containing such expression is not well-typed and would be statically discarded. Notably, the use of the field `err` allows the analysis to still address the role of errors in policy evaluations, i.e. reasoning on the authorisations of requests assigning unexpected values to attribute names.

On top of the `TValue` datatype we build the uninterpreted functions expressing the constraint operators of Table 4.1. By way of example, the 4-valued operator $\dot{\wedge}$ corresponds to the `FAnd` function defined as follows

```
(define-fun FAnd ((x (TValue Bool)) (y (TValue Bool))) (TValue Bool)
    (ite (and (isTrue x) (isTrue y))
        (mk-val true false false)
        (ite (or (isFalse x) (isFalse y))
            (mk-val false false false)
            (ite (or (err x) (err y))
                 (mk-val false false true)
                 (mk-val false true  false)))))
```

where `mk-val` is the constructor of `TValue` records. Hence, the function takes as input two `TValue Bool` records, i.e. type `Bool` is the instantiation of the type parameter `T`, and returns a `Bool` record as well. The conditional if-then-else assertions `ite` are nested to form a structure that mimics the semantic conditions of Table 4.2, so that different `TValue` records are returned according to the input. The function `isFalse` (resp. `isTrue`) is used to compactly check that all fields of the record are false (resp. only the field `val` is true). All the other constraint operators, except $\in$, are defined similarly.

To express the operator $\in$, we need to represent multi-valued attributes. Firstly, we define an array datatype, named `Set`, to model sets of elements as follows

```
(define-sort Set (T) (Array Int T))
```

where the type parameter `T` is the type of the elements of the array. By definition of array, each element has an associated integer index that is used to access the corresponding value. Thus, a multi-valued attribute is represented by a `TValue` record with type an instantiated `Set`, e.g. `(TValue (Set Int))` is an attribute whose value is a set of integers. Consequently, we can build the uninterpreted function modelling the constraint operator $\in$. In case of integer sets, the function is

```
(define-fun inInt ((x (TValue  Int)) (y (TValue (Set Int)))) (TValue Bool)
   (ite (or (err x)(err y))
       (mk-val false false true)
```

```
(ite (or (miss x) (miss y))
      (mk-val false true false)
      (ite (exists ((i Int)) (=  (val x) (select (val y) i)))
            (mk-val true  false false)
            (mk-val false false false)))))
```

where the command `(select (val y) i)` takes the value in position `i` of the set in the field `val` of the argument `y`. In addition to the conditional assertions, the function uses the existential quantifier `exists` for checking if the value of the argument `x` is contained in the set of the argument `y`.

The coding approach we pursue generates, in most of the cases, fully decidable constraints. In fact, since we support non-linear arithmetic, i.e. multiplication, it is possible to define constraints for which a constraint solver is not able to answer. Anyway, modern constraint solvers are actually able to resolve nontrivial nonlinear problems that, for what concerns access control policies, should prevent any undefined evaluation[3]. Similarly, the quantifier-based constraints are in general not decidable, but solvers still succeed in evaluating complicated quantification assertions due to, e.g., powerful pattern techniques (see, e.g., the documentation of Z3). Notice anyway that if we assume that each expression operator in (and, consequently, constraint operator $\in$) is applied to at most one attribute name, the quantifications are bounded by the number of literals defining the other operator argument.

Concerning the value types we support, SMT-LIB does not provide a primitive type for $Date$. Hence, we use integers to represent its elements. Furthermore, even though SMT-LIB supports the $String$ type, the Z3 solver we use does not. Thus, given a policy as an input, we define an additional datatype, say `Str`, with as many constants as the string values occurring in the policy. The string equality function is then defined over `TValue` records instantiated with type `Str`.

By way of example, the SMT-LIB code for the constraint c_{trg1} (see Section 4.3.4) is

```
(define-fun cns_target_Rule1 () (TValue Bool)
 (FAnd (equalStr n_sub/role cst_doc)
   (FAnd (equalStr n_act/id cst_write)
     (FAnd
       (inStr cst_permWrite n_sub/perm) (inStr cst_permRead n_sub/perm)))))
```

where identifiers starting with `n_` (resp. `cst_`) represent attribute names (resp. literals) of the represented expression. The whole SMT-LIB code for Policy (Eh-A) can be found at http://facpl.sourceforge.net/eHealth/.

4.5.2 SMT-LIB-based Property Verification

The SMT-LIB coding permits using SMT solvers to automatically verify the properties formalised in Section 4.4. In the following, given a FACPL policy p, we denote by $\langle \mathsf{permit} : smtlib\text{-}c_p \ \mathsf{deny} : smtlib\text{-}c_d \ \mathsf{not\text{-}app} : smtlib\text{-}c_n \ \mathsf{indet} : smtlib\text{-}c_i \rangle$ the tuple of SMT-LIB codes representing the formal constraints $\mathcal{T}_P\{\!|p|\!\} = \langle \mathsf{permit} : c_p \ \mathsf{deny} : c_d \ \mathsf{not\text{-}app} : c_n \ \mathsf{indet} : c_i \rangle$. Below, we present first the verification strategies to follow for the authorisation properties, then those for the structural properties.

[3]Notably, if at least one argument of each occurrence of the multiply operator is a numeric constant, the resulting non-linear arithmetic constraints are decidable.

Authorisation Properties

The automated verification of authorisation properties requires: (i) to introduce into the policy constraint of interest, which is chosen according to the property, the SMT-LIB coding of the request defined by the property; (ii) to check the satisfiability (or validity) of the resulting constraint.

Given a request r, the SMT-LIB coding of the request is defined as follows

$$r_{smtlib} \triangleq \left\{ \begin{array}{l} (\texttt{assert}\ (=\ (\texttt{val}\ \ n)\ \ v)) \\ (\texttt{assert}\ (\texttt{and}\ (\texttt{not}\ (\texttt{miss}\ \ n))\ (\texttt{not}\ (\texttt{err}\ \ n)))) \end{array} \middle|\ r(n) = v \right\}$$

Indeed, all attribute names n in r are asserted to be equal to their value v and to be neither missing nor erroneous. Furthermore, given a FACPL policy p, we also define the following SMT-LIB coding of the request

$$\overline{r_{smtlib}} \triangleq \left\{ \ (\texttt{assert}\ (\texttt{miss}\ \ n)) \ \middle|\ n \in Names(p)\ \wedge\ r(n) = \bot \ \right\}$$

where, as in Section 3.5.6, $Names(p)$ indicates the set of attribute names occurring in p. Indeed, all the names n that occur in p and are not assigned to a value in r are asserted as missing attributes.

By exploiting this SMT-LIB coding of requests, we define the automated verification (i.e., via an SMT solver) of the authorisation properties as follows

$$\begin{array}{rlll} p\ \texttt{sat}\ r\ \texttt{eval}\ dec & \texttt{iff} & smtlib\text{-}c_{dec} \circ r_{smtlib} \circ \overline{r_{smtlib}} & \texttt{is sat} \\ p\ \texttt{sat}\ r\ \texttt{eval}_{\texttt{may}}\ dec & \texttt{iff} & smtlib\text{-}c_{dec} \circ r_{smtlib} & \texttt{is sat} \\ p\ \texttt{sat}\ r\ \texttt{eval}_{\texttt{must}}\ dec & \texttt{iff} & smtlib\text{-}c_{dec} \circ r_{smtlib} & \texttt{is valid} \end{array}$$

where $\circ$ is used to indicate the concatenation of SMT-LIB code and `valid` means that the corresponding SMT-LIB code is a valid set of assertions. Some comments follow.

The *Evaluate-To* property does not exploit request extensions, hence all attribute names not assigned by the considered request can only assume the special value $\bot$. This means that the request r is coded in SMT-LIB with r_{smtlib} and $\overline{r_{smtlib}}$. The satisfiability of the property thus corresponds to that of the resulting SMT-LIB code.

To verify the *May-Evaluate-To* property, since it considers request extensions, the request has to be coded only with r_{smtlib}. As before, the satisfiability of the property corresponds to that of the resulting SMT-LIB code.

Finally, to verify the *Must-Evaluate-To* property, we code again the request with r_{smtlib}, but we check the validity of the resulting SMT-LIB code, i.e. that it is satisfied by all the assignments for the attribute names. This amounts to check if the negation of the resulting SMT-LIB code is not satisfiable, in which case the property holds.

Structural Properties

The automated verification of structural properties does not require to modify policy constraints, but rather to check the unsatisfiability of combinations of constraints. The auto-

mated verification of the structural properties is as follows

$$p \texttt{ sat complete} \quad \texttt{iff} \quad smtlib\text{-}c_n \quad \texttt{is unsat}$$

$$p \texttt{ sat disjoint } p' \quad \texttt{iff} \quad \begin{cases} smtlib\text{-}c_p \circ smtlib\text{-}c'_p & \texttt{is unsat} \\ smtlib\text{-}c_p \circ smtlib\text{-}c'_d & \texttt{is unsat} \\ smtlib\text{-}c_d \circ smtlib\text{-}c'_p & \texttt{is unsat} \\ smtlib\text{-}c_d \circ smtlib\text{-}c'_d & \texttt{is unsat} \end{cases}$$

$$p \texttt{ sat cover } p' \quad \texttt{iff} \quad \begin{cases} \neg\ smtlib\text{-}c_p \circ smtlib\text{-}c'_p & \texttt{is unsat} \\ \neg\ smtlib\text{-}c_d \circ smtlib\text{-}c'_d & \texttt{is unsat} \end{cases}$$

where $smtlib\text{-}c'_{dec}$ refers to the SMT-LIB code modelling decision dec of policy p'. Some comments follow.

The trivial case is that of the *completeness* property, which only amounts to check if the constraint modelling the decision not-app is not satisfiable, i.e. if its negation is valid; if it is, the property holds.

The *disjointness* of two policies is verified by checking, one at a time, if the conjunctions between the permit or deny constraint of the first policy and the permit or deny constraint of the second policy are not satisfiable[4]. If this holds for the four possible combinations of those constraints, the property holds.

The *coverage* of policy p on policy p' is verified by checking if the conjunction between the negation of the permit (resp., deny) constraint of p and the permit (resp., deny) constraint of p' is not satisfiable. Intuitively, if the policy p does not calculate a permit or deny decision (i.e., $\neg\ smtlib\text{-}c_p$ and $\neg\ smtlib\text{-}c_d$ hold), policy p' cannot do it as well, otherwise the property is not satisfied. Therefore, if this holds for the two conjunctions separately, the property holds.

Finally, it is worth noticing that we are not considering the set R of all possible requests because, due to Lemma 3.2, only the attribute names occurring in the policies of interest are relevant for the analysis; any other name cannot affect policy evaluation.

4.5.3 Supporting Tools

To effectively support the analysis of FACPL policies, the FACPL software toolchain introduced in Section 3.6 offers automatic generation of SMT-LIB code and aided specification of authorisation and structural properties. These functionalities, graphically depicted in Figure 3.2, are offered by the FACPL IDE.

The generation of SMT-LIB code is defined by using the code generator offered by Xtext and by pursuing the coding strategies outlined in Section 4.5.1. Indeed, the IDE implements the type inference system of Table 4.4 in order to properly instantiate the generated SMT-LIB constructs, that are generated according to the properly implemented translation function of Section 4.3.2. To support date/time and string values, an automatic procedure creates the appropriate numeric and datatype representations, respectively.

The FACPL IDE supports the specification of authorisation and structural properties by means of graphical wizards. These wizards permit choosing the type of the property and its forming elements (i.e., one or more policies and possibly a request), and then

[4]Notably, the satisfiability of two (sets of) SMT-LIB assertions amounts to check if they both hold at the same time, hence checking their conjunction.

they generate the corresponding SMT-LIB file ready to be evaluated. Such SMT-LIB file is generated by appropriately extending the SMT-LIB code representing the considered policy (or policies) according to the strategies reported in Section 4.5.2. To facilitate the evaluation of such file, it is also generated an execution script for the Z3 solver [dB08]. It is worth noticing that the generated SMT-LIB files can also be evaluated by any other solver accepting SMT-LIB and supporting the theories we use. Our choice has been Z3 due to its large support of SMT-LIB and its remarkable performance, see, e.g., the results of the last SMT competitions[5].

4.6 Concluding Remarks

In this chapter we have presented a constraint-based analysis approach for FACPL policies, from its formal definition to its implementation in terms of automatic functionalities of the FACPL IDE. Here, we conclude by briefly commenting on the contributions of the analysis with respect to the research objectives of the thesis and to related and prior publications.

The analysis approach, together with the expressly developed authorisation properties, accomplishes the objective **O3**, hence it provides a formally-defined analysis approach that is capable of taking into account the peculiarities of ABAC. On the basis of the FACPL semantics, we have also formally proved the soundness of this constraint-based analysis. From an implementation point of view, the analysis is supported by practical functionalities of the FACPL IDE. Together with the rest of the FACPL software tools, this contributes accomplishing the objective **O4**.

Concerning other analysis approaches from the literature, our proposal addresses all the elements forming access control policies and permits reasoning on missing and erroneous attributes. The latter functionalities are the main improvements with respect to other approaches, like, e.g., those in [FKMT05, ACC14, TdHRZ15]. Additionally, the authorisation properties formalised in Section 4.4 permit defining properties that, differently from other proposals, explicitly address the peculiarities of ABAC. The concept of request extension set, that we exploit for property specifications, is similarly defined in [CMZ15], but there it has the aim of defining completely different probabilistic properties on access control policies. It is also worth noticing that the FACPL analysis tools ensure remarkable performance. For example, the verification of the `complete` property on a benchmark of reference requires around 120ms. Further details are reported in Section 7.5.

Generally speaking, with the development of FACPL, its constraint-based analysis and its software tools, we aim at devising a comprehensive methodology supporting the whole development life-cycle of policy-based access control systems, from their specification and analysis to their implementation. Each ingredient has been first formally defined and then implemented as a software tool. Moreover, our methodology allows access control system developers to use formally-defined automated functionalities without requiring them to be familiar with formal methods.

From a practical perspective, the analysis of the e-Health FACPL policies we proposed sheds light on some inconsistencies that were found in a preliminary specification of epSOS. Such an issue could have been found immediately by means of the FACPL analysis tools. In general, the automated analysis of FACPL policies can be exploited to reason on dynamic modifications of the access control policies enforced in a system.

[5]SMT Competition - http://smtcomp.sourceforge.net/

The contents of this chapter are mainly based on the work in [MMPT16], with the exception of the formalisation of security policies of Sections 4.1 and 4.2 that are based on the work in [MPT15]. The preliminary version of FACPL presented in [MPT12, Mas12] does not address any of the contents reported in this chapter.

Chapter 5

The PSCEL Language

> It's not about keeping pace with Moore's Law, but rather dealing with the consequences of its decades-long reign.
>
> Paul Horn [Hor01]

Autonomic computing is a recently devised paradigm to build computing systems. The self-managing functionalities fostered by the autonomic approach aim at simplifying the management and interoperability of systems. However, to practically develop autonomic systems, we need a principled approach that permits structured design of the forming components and precise formalisation of their interactions [BCG+12]. A crucial aspect to address in the development is how to specify and enforce adaptation strategies, i.e. the means that allow systems to autonomously adapt to changing operating conditions.

In this chapter, we present *Policed-SCEL (PSCEL)*, a FACPL-based instantiation of the SCEL language. SCEL, as outlined in Section 2.2.1, provides a set of linguistic abstractions to specify autonomic systems, but it is parametric with respect to the formalism to be concretely used for the specification and enforcement of adaptation strategies. To this aim, we opportunely specialise FACPL, i.e. its targets and obligations, to regulate SCEL process interactions and to define rule-based adaptation strategies. Intuitively, target functions are specialised so that it is possible to define pattern-matching controls on process actions, while obligations are instantiated as SCEL process actions. Indeed, FACPL policies act as *Event-Condition-Action (ECA)* rules, where the event is an authorisation request representing a process action to authorise, the condition is a rule target and the action is an adaptation strategy defined in term of obligations. The interplay between policies and processes, generated by the evaluation of FACPL policies, is precisely formalised in terms

of a formal semantics. To effectively deploy the language, PSCEL is also equipped with a Java runtime environment providing a practical framework for developing autonomic systems and an IDE supporting the PSCEL coding and analysis tasks.

Structure of the chapter. The rest of this chapter is organised as follows. Section 5.1 outlines the main design principles of PSCEL. Section 5.2 presents the syntax of PSCEL. Section 5.3 describes the PSCEL formal semantics. Section 5.4 describes the PSCEL system modelling the robot-swarm case study presented in Section 2.3.2. Section 5.5 outlines the main functionalities of the PSCEL supporting tools. Section 5.6 concludes with some final remarks.

5.1 PSCEL: a FACPL-based Instantiation of SCEL

The PSCEL language is a full-fledged instance of SCEL, where policies are defined by a dialect of the FACPL language. We present here the main design principles underlying the conceptualisation of PSCEL and its functionalities.

The use of declarative policies for defining self-adaptation strategies is largely advocated in the literature (see, e.g., in [HM08, LMD13]). Since adaptation strategies usually consist of sets of actions to be executed by the controlled system, we can opportunely exploit FACPL obligations to define such strategies. Choosing FACPL obligations ensures a dynamical fulfilment of adaptation actions, i.e. action arguments can be dynamically retrieved at evaluation time. This approach clearly ensures high flexibility, but it also requires a technique for enforcing obligations.

Techniques for achieving self-adaptation in computing systems have received large attention and prompted the introduction of various approaches like, e.g., Aspect Oriented Programming (AOP) [KLM+97, KHH+01]. AOP crucially relies on the idea that definite parts of a program, called join-points, trigger the execution of *before* and *after* actions, i.e. actions that will be performed before or after the join-point. The AOP approach is widely used and has proven to be flexible and effective enough to easily deal with multiple adaptation and behavioural strategies. For example, it has been used in [DL06] to enforce adaptation in the component-based framework FRACTAL [BCL+06] and in [CM07] to dynamically compose web services. Therefore, in order to define FACPL-based adaptation strategies in SCEL, we instantiate FACPL obligations as SCEL process actions and, by taking inspiration from AOP, we specialise each obligation with a type representing if the corresponding actions are *before* or *after* actions. Additionally, we consider all process actions as join-points triggering the evaluation of policies. We represent each process action and its evaluation context as an attribute-based request that is evaluated by the FACPL policies in order for the corresponding action to be authorised for execution.

The principled combination of attribute-based requests, authorisation rules and obligations allows policies to regulate the authorisation of process actions, filter the interactions among components, and enforce adaptation strategies. Furthermore, as requests contain context information, policies can base their evaluation on the system context, like, e.g., location and availability of computing resources. This ensures that context awareness can drive system evolution.

To sum up, FACPL policies are specialised by instantiating obligation actions as the set of SCEL process actions and by defining adequate target functions that permits checking

SYSTEMS:	S	::=	$\mathcal{I}[\mathcal{K}, \Pi, P] \mid S_1 \parallel S_2 \mid (\nu n)S$
INTERFACES:	$\mathcal{I}$	::=	$(\mathsf{id} : n\ (, \mathsf{n} : e)^*)$
PROCESSES:	P	::=	$\mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid \underline{X} \mid A(\tilde{p})$
ACTIONS:	a	::=	$\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ $\mid \mathbf{upd}(\mathsf{n}, e) \mid \mathbf{read}(?\,\underline{x}, \mathsf{n})$
DESTINATIONS:	c	::=	$n \mid \underline{x} \mid \mathsf{self} \mid \mathcal{P}$
PREDICATES:	$\mathcal{P}$	::=	$\mathsf{true} \mid \mathsf{n} = e \mid \mathsf{n} < e \mid \neg\,\mathcal{P} \mid \mathcal{P}_1 \vee \mathcal{P}_2$
KNOWLEDGE:	$\mathcal{K}$	::=	$\emptyset \mid \langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$
ITEMS:	t	::=	$e \mid c \mid P \mid t_1, t_2$
TEMPLATES:	T	::=	$e \mid c \mid ?\,\underline{x} \mid ?\,\underline{X} \mid T_1, T_2$
EXPRESSIONS:	e	::=	$v \mid \underline{x} \mid \underline{X} \mid \neg\, e \mid e_1\ \mathsf{op}\ e_2 \qquad \mathsf{op} \in \{\wedge, =, <, +, -\}$
VALUES:	v	::=	$\mathsf{true} \mid \mathsf{false} \mid n \mid i$

Table 5.1: Programming constructs (POLICIES Π are defined in Table 5.2; n and n range over the set of names $\mathcal{N}$, while i ranges over integers)

conditions on the attribute-based requests representing process actions. Instead, the SCEL semantics is extended thus to interact with FACPL policies both by asking authorisation to execute process actions and by enforcing the obligations possibly returned by policies.

5.2 Syntax

The syntax of PSCEL includes the following two sets of constructs

- *programming constructs*: they define the components forming a system and their computational behaviours, i.e. the component processes;
- *policy constructs*: they define the policies of components specifying the authorisation and adaptation logic of a system.

The programming constructs are mainly borrowed from SCEL, but we introduce the management of contextual information by defining a syntax for component interfaces, predicates and actions to operate on interfaces. Instead, the policy constructs are a simplified version of FACPL and provide specification means for the definition of authorisation controls on process actions and of adaptation strategies in the form of AOP-inspired *before* and *after* obligations.

The syntax of programming and policy constructs is given by the BNF grammars in Tables 5.1 and 5.2; as before, the symbol * stand for (possibly empty) sequences, and $+$ for non-empty sequences. We convene that variables are underlined, e.g. $\underline{x}$. Moreover, we assume that a set of *names* $\mathcal{N}$ is given and we will use n to denote a generic name and n to denote the name of a specific syntactic element[1].

Programming Constructs. The key construct is the *component*, $\mathcal{I}[\mathcal{K}, \Pi, P]$, consisting of an interface $\mathcal{I}$, a knowledge repository $\mathcal{K}$, a (FACPL) policy Π and a process P.

[1]Different notations for names are used only for the sake of presentation. Their different meaning should be always clear from the context where they occur (e.g. names occurring in predicates are references to interface elements)

SYSTEMS aggregate components through the *composition* operator $S_1 \parallel S_2$. The *name restriction* operator $(\nu n)S$ restricts the scope of the name n to S.

INTERFACES are non empty lists of *features*, i.e. pairs $\mathsf{n} : e$. We assume that the names n in the interface features are pairwise distinct and act as references to *closed* (i.e. without variables) expressions. Notably, the features can be dynamically modified by the component process except for the feature id, which represents the component name and, by design assumption, is uniquely and persistently defined. The expressions reported at the outset in the interface specification correspond to the initialisation values.

PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), interleaved *parallel composition* ($P_1 \mid P_2$), *process variable* ($\underline{X}$), and *parametrised process invocation* ($A(\tilde{p})$). Process variables support *higher-order* communication, while parametrised process identifiers, that are ranged over by A, permit defining *recursive processes*. We assume that each process identifier A has a *single* definition of the form $A(\tilde{f}) \triangleq P$. Notably, $\tilde{p}$ and $\tilde{f}$ denote lists of values/processes and variables, respectively.

Processes can perform seven different types of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage knowledge repositories by withdrawing/retrieving/adding information items from/to the, possibly remote, repositories identified by c. Actions **get** and **qry** rely on a classical pattern-matching mechanism and block the enclosing process until a knowledge item t that matches the template T is available in the repositories identified by c; the two actions differ because **get** removes the item t from the repository, while **qry** leaves the repository unchanged. Action $\mathbf{fresh}(n)$ introduces a scope restriction for the name n, while action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$. The remaining actions are used to act on interface features. Action $\mathbf{upd}(\mathsf{n}, e)$ updates the value referred to by n with the (evaluation of) e, while action $\mathbf{read}(?\,\underline{x}, \mathsf{n})$ assigns the value referred to by n to the variable $\underline{x}$.

The DESTINATION c of an action can be either a component name n, or a variable $\underline{x}$, or the reserved variable self, which refers to the name of the component executing the action, or a predicate $\mathcal{P}$. PREDICATES are boolean expressions on feature names and are used to dynamically identify the component(s) destination of an action. Using a name or a predicate as a destination produces different forms of communication: *point-to-point* or (a sort of) *group-oriented* communication, respectively. Indeed, the group variant of **put** generates a broadcast, while the **get** and **qry** ones define interactions with a non deterministically chosen component among those satisfying the predicate.

Knowledge ITEMS and TEMPLATES are sequences of different elements. Both can contain expressions and destinations but differ for the fact that items can contain processes, while templates can contain binders for variables. We convene that variables $\underline{x}$ and $\underline{X}$ refer to values and processes, respectively, thus $?\underline{x}$ and $?\underline{X}$ are their respective binders. KNOWLEDGE repositories are, possibly empty, multisets of evaluated items (i.e. items with no occurring variables). EXPRESSIONS can be either VALUES (i.e. booleans, names or integers) or variables, or can be built by applying standard operators to simpler expressions[2].

Policy Constructs. These constructs represent a dialect of FACPL expressly designed to interact with the just presented programming constructs. To get a better intuition of the role

[2]We only take into account linear arithmetic operators to ensure the full decidability of our analysis. Nonlinear arithmetic operators, as e.g. $*$, are not decidable, although they can be managed in nontrivial expressions by the automatic solvers we exploit.

POLICIES:	Π	::=	$\langle alg$ rules : $\rho^+ \rangle$
ALGORITHMS :	alg	::=	d-unless-p | p-unless-d
RULES:	ρ	::=	n(d target : τ obl : o^*)
DECISIONS:	d	::=	permit | deny
TARGETS:	τ	::=	true | $f(sn, pv)$ | $\tau_1 \wedge \tau_2$ | $\tau_1 \vee \tau_2$ | $\neg\, \tau$
FUNCTIONS:	f	::=	equal | greater-than | less-than | pattern-match
STRUCTURED NAMES:	sn	::=	action/n | subject/n | object/n
POLICY VALUES:	pv	::=	pt | (pt) | **put** | **get** | **qry** | **new** | **fresh** | **upd** | **read** | this
POLICY TUPLES:	pt	::=	pe | _ | pt_1, pt_2
POLICY EXPRESSIONS:	pe	::=	v | n | $\neg\, pe$ | pe_1 op pe_2 op $\in \{\wedge, =, <, +, -\}$
OBLIGATIONS:	o	::=	[B $a(.\, a)^*$] | [A $a(.\, a)^*$]

Table 5.2: Policy constructs (OBLIGATIONS use the ACTIONS a of Table 5.1 where structured names can occur instead of names n; v stands for the VALUES of Table 5.1, while n range over the set of names $\mathcal{N}$)

of each syntactic construct, consider that policies will be evaluated against authorisation requests that are generated at runtime to enable the execution of process actions. Indeed, policies act as ECA rules, where the event is an authorisation request, the condition is a rule target and the action is an adaptation strategy.

POLICIES are formed by a combining algorithm and a sequence of rules. The combining ALGORITHM can be d-unless-p or p-unless-d. RULES are identified by a name n and consist of a positive or negative DECISION, i.e. permit or deny, a target and a sequence of obligations.

A TARGET is a boolean expression: it is either true, or an atomic target, or a combination of simpler targets through the boolean operators $\wedge$, $\vee$ and $\neg$. An *atomic target* $f(sn, pv)$ is a triple denoting the application of a relational FUNCTION to a structured name and a policy value.

STRUCTURED NAMES are a specialised form of attribute name. They are composed by a category name among action, subject and object, and by a name n. Indeed, all the authorisation requests adhere to a pre-defined structure, i.e. the category name of each attribute must be of the three ones just reported. By way of example, action/id refers to the name of the action generating the request, while subject/label refers to the value of the interface feature label of the component executing the action.

POLICY VALUES can be either policy tuples (round brackets are used to delimit them), or the *action identifiers* (i.e. **get**, **qry**, **put**, **fresh**, **new**, **upd** and **read**), or the reserved variable this, which refers to the name of the component where the policy is in force. POLICY TUPLES are sequences of elements containing policy expressions and the symbol '_', which is a wildcard representing any value. POLICY EXPRESSIONS are values, name references or can be built by applying standard operators to simpler expressions (differently from the EXPRESSIONS of Table 5.1, they cannot contain variables).

OBLIGATIONS are made of a type and a sequence of process actions. The type, i.e. B or A, indicates if the obligation has to be executed before or after the action whose authorisation request has triggered the evaluation. We assume that within obligation actions structured names can occur instead of names n. Thus, e.g., the action $\mathbf{put}(log, \text{action/id})@\text{self}$ can be used to locally add an item formed by log and the action identifier referred to by the structured name action/id.

Remark 5.1 (On the differences with FACPL)**.** *The policy constructs reported in Table 5.2 are a dialect of the FACPL language whose syntax is defined in Table 3.1. These constructs share their main traits with FACPL, but they have the following substantial differences: (i) there is only a basic set of combining algorithms; (ii) target expressions are specialised to express controls on process actions; (iii) obligation actions are instantiated as process actions; (iv) obligation types are "before" and "after" instead of "mandatory" and "optional". Notably, only* p-unless-d *and* d-unless-p *algorithms are supported, because we want to ensure that only* permit *and* deny *decisions are returned by policy evaluations. These observations explain the choice of a few different names and notations for the PSCEL policy constructs with respect to the FACPL ones.*

5.3 Formal Semantics

The semantics of PSCEL consists of two parts: a denotational semantics of policy constructs, and three Labeled Transition Systems (LTSs) defining the operational semantics of the programming constructs. The LTSs exploit the denotational semantics in order to check policy evaluation results.

As a general insight, each process action in order to be executed has to be first authorised by the policies of the involved components. Since processes can perform three different kinds of actions, we have the following cases: local actions (i.e. actions for which executing and destination components coincide) only involve one component, point-to-point actions involve two specific components, while group-oriented actions can involve more than two components dynamically determined through a destination predicate. The first authorisation phase amounts to evaluate the action arguments, check the policies of the involved components, possibly install adaptation actions and, when permission for execution is provided, mark the evaluated action as authorised. The consequent execution phase amounts to execute the authorised actions and apply the effects of their execution to the involved components. When a process wants to execute a group-oriented action, the authorisation phase determines the actual component(s) authorised to act as a destination: in case of **get** and **qry**, the destination is a component randomly chosen among those satisfying the destination predicate; in case of **put**, all the components satisfying the predicate act as a destination (thus, if one of them is not authorised by the policy of the executing component, the action itself is not authorised).

In the sequel, to define the PSCEL semantics, we rely on the auxiliary functions whose signatures are reported in Table 5.3. Most (co)domains are sets of terms generated by the BNF rules of Tables 5.1 and 5.2. The correspondence is obvious: e.g. $Actions$ identifies the set of all actions. Instead, $Requests$ (ranged over by req) is the set of all *(authorisation) requests*. A request req is a (syntactical) collection of *attributes*, i.e. pairs of the form (sn, pv), mapping structured names to policy values. Hence, differently from FACPL (see Section 3.5.1), PSCEL semantics is defined on the base of syntactical requests, i.e. they have not any name associated to the special value $\perp$. Anyway, when convenient, we interpret a syntactical request $req = \{(sn_i, pv_i) \mid i \in I \subseteq \mathbb{N}\}$ as a function mapping the structured name sn_i to the policy value pv_i, for each $i \in I$. Hence, the notation $req(sn)$ has the obvious meaning. Furthermore, both semantics are parametric with respect to the primitive function $[\![\cdot]\!]$ that, given a (policy) expression, returns a (boolean, name or integer) value. Notationally, we will use ϵ to denote an empty sequence of elements and

Aux. Function	Domain	Codomain
$[\![\cdot]\!]$	$Expressions \cup PolicyExpressions$	$Values$
Policy Constructs		
$\mathcal{F}$	$Actions \times Requests$	$Actions$
Programming Constructs		
$\mathcal{A}$	$Actions \times \mathcal{N}$	$Actions$
sat	$Predicates \times Interfaces$	{true, false}
req	$Interfaces \times Actions \times Interfaces$	$Requests$
acid	$Actions$	{**put**, **get**, **qry**, **new**, **fresh**, **upd**, **read**}
dst	$Actions$	$Destinations$
match	$Templates \times Items$	$2^{\{x,X\}} \times Values$

Table 5.3: Auxiliary functions

s to denote a sequence of actions ($Actions^*$ is the set of all sequences of actions); if s is empty, $s.s'$ (and $s'.s$) and $s.P$ stand for s' and P, respectively.

In the rest of this section, we present first the semantics of the policy constructs (Section 5.3.1), then that of the programming constructs (Section 5.3.2).

5.3.1 Policy Constructs

The semantics of policy constructs is expressed by a family of semantic functions mapping each syntactic domain to a specific semantic domain in a way similar to what presented for FACPL. The functions are reported in Table 5.4. As they correspond to a simplified form of those of FACPL (see Section 3.5), we only comment on the crucial points. Moreover, the semantics is only defined for policy constructs without occurrences of the reserved variable this.

Remark 5.2 (On the use of this in policy constructs)**.** *By using* this *to refer to the name of the component where a policy is in force, we can statically assign the same policy to different system components. During system evolution, the occurrences of the variable* this *in the policy of any component will be replaced by the name of the component itself. Hence, when a request must be authorised, the involved policy does not contain occurrences of* this *(see the semantics of the programming constructs in Section 5.3.2).*

The semantics of policies and rules is defined by the function $\mathcal{P} : Policies \cup Rules \rightarrow (Requests \rightarrow \{\mathsf{permit}, \mathsf{deny}, \mathsf{not\text{-}app}\} \times Actions^* \times Actions^*)$ that, given a request, returns a decision d and two (possibly empty) sequences s_{B} and s_{A} of before and after actions, respectively. It is worth noticing that the semantics of rules can return the not-app decision, while the semantics of policies, since the application of algorithms always provides a default permit or deny decision (see the matrices in Table 5.5), never returns not-app.

The case of function $\mathcal{P}$ defining the evaluation of policies corresponds to the application of the algorithm alg to the sequence of enclosed rules. The semantics of algorithm is defined by the function $\mathcal{A}lg : Algorithms \times Rules^* \rightarrow (Requests \rightarrow \{\mathsf{permit}, \mathsf{deny}\} \times Actions^* \times Actions^*)$. Its definition corresponds to the case of the FACPL fulfilment strategy all (see Section 3.5.3) according to the combination matrices reported in Table 5.5.

The case of function $\mathcal{P}$ defining the evaluation of rules is as expected. If the target matches the request, i.e. it evaluates to true, the rule decision is returned, together with

POLICIES AND RULES

$$\mathcal{P}[\![\langle alg \;\; \mathsf{rules} : \rho^+ \rangle]\!] req = d, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathtt{if} \quad \mathcal{A}lg[\![alg, \rho^+]\!] req = d, s_{\mathtt{B}}, s_{\mathtt{A}}$$

$$\mathcal{P}[\![\mathsf{n}(d \;\; \mathsf{target} : \tau \;\; \mathsf{obl} : o^* \,)]\!] req = \begin{cases} d, s_{\mathtt{B}}, s_{\mathtt{A}} & \mathtt{if} \quad \mathcal{E}[\![\tau]\!] req = \mathsf{true} \;\wedge\; \mathcal{O}[\![o^*]\!] req = s_{\mathtt{B}}, s_{\mathtt{A}} \\ \mathsf{not\text{-}app}, \epsilon, \epsilon & \mathtt{if} \quad \mathcal{E}[\![\tau]\!] req = \mathsf{false} \end{cases}$$

ALGORITHMS

$$\mathcal{A}lg[\![alg, \rho_1 \ldots \rho_s]\!] req = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\ldots \otimes \mathsf{alg}(\mathcal{P}[\![\rho_1]\!] req, \mathcal{P}[\![\rho_2]\!] req), \ldots), \mathcal{P}[\![\rho_s]\!] req)$$

TARGETS

$$\mathcal{E}[\![\mathsf{true}]\!] req = \mathsf{true} \qquad \mathcal{E}[\![\neg\tau]\!] req = \neg\, \mathcal{E}[\![\tau]\!] req$$

$$\mathcal{E}[\![\tau_1 \wedge \tau_2]\!] req = \mathcal{E}[\![\tau_1]\!] req \;\wedge\; \mathcal{E}[\![\tau_2]\!] req \qquad \mathcal{E}[\![\tau_1 \vee \tau_2]\!] req = \mathcal{E}[\![\tau_1]\!] req \;\vee\; \mathcal{E}[\![\tau_2]\!] req$$

$$\mathcal{E}[\![f(sn, pv)]\!] req = \begin{cases} \mathsf{true} & \mathtt{if} \quad f([\![\, req(sn) \,]\!], [\![\, pv \,]\!]) = \mathsf{true} \\ \mathsf{false} & \mathtt{if} \quad f([\![\, req(sn) \,]\!], [\![\, pv \,]\!]) = \mathsf{false} \end{cases}$$

OBLIGATIONS

$$\mathcal{O}[\![\epsilon]\!] req = \epsilon, \epsilon$$

$$\mathcal{O}[\![o \; o^*]\!] req = \quad s'_{\mathtt{B}}.s''_{\mathtt{B}}, s''_{\mathtt{A}}.s'_{\mathtt{A}} \quad \mathtt{if} \quad \mathcal{O}[\![o]\!] req = s'_{\mathtt{B}}, s'_{\mathtt{A}} \;\wedge\; \mathcal{O}[\![o^*]\!] req = s''_{\mathtt{B}}, s''_{\mathtt{A}}$$

$$\mathcal{O}[\![[\mathtt{B} \;\; a(.\, a)^*]]\!] req = \quad s_{\mathtt{B}}, \epsilon \quad \mathtt{if} \quad \mathcal{F}(a, req)(.\, \mathcal{F}(a, req))^* = s_{\mathtt{B}}$$

$$\mathcal{O}[\![[\mathtt{A} \;\; a(.\, a)^*]]\!] req = \quad \epsilon, s_{\mathtt{A}} \quad \mathtt{if} \quad \mathcal{F}(a, req)(.\, \mathcal{F}(a, req))^* = s_{\mathtt{A}}$$

Table 5.4: Semantics of policy constructs (s is a sequence of actions)

the two sequences of actions resulting from the evaluation of obligations. If the target does not match the request, i.e. it evaluates to false, the not-app decision is returned.

The semantics of targets is defined by the function $\mathcal{E} : \mathit{Targets} \rightarrow (\mathit{Requests} \rightarrow \{\mathsf{true}, \mathsf{false}\})$ that, given a request, returns a boolean value. Obviously, a target matches a request req if $\mathcal{E}$ returns true; otherwise, i.e. when it returns false, the target does not match the request. A target true matches all possible requests, while the boolean operators have the classical two-valued semantics. An atomic target $f(sn, pv)$ matches req according to the application of function f to the value identified by the structured name sn in the request, i.e. $[\![\, req(sn) \,]\!]$, and the value resulting from the evaluation of the policy value pv, i.e. $[\![\, pv \,]\!]$. Thus, when f returns true the target matches, otherwise it does not match. Notably, evaluation of policy values requires to evaluate the possible occurring expressions.

The semantics of obligations is defined by the function $\mathcal{O} : \mathit{Obligations}^* \rightarrow$

$\otimes$d-unless-p	permit, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	not-app, ϵ, ϵ
permit, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	permit, $s'_{\mathtt{B}}.s''_{\mathtt{B}}, s''_{\mathtt{A}}.s'_{\mathtt{A}}$	permit, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	permit, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$
deny, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	permit, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, $s'_{\mathtt{B}}.s''_{\mathtt{B}}, s''_{\mathtt{A}}.s'_{\mathtt{A}}$	deny, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$
not-app, ϵ, ϵ	permit, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, ϵ, ϵ

$\otimes$p-unless-d	permit, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	not-app, ϵ, ϵ
permit, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	permit, $s'_{\mathtt{B}}.s''_{\mathtt{B}}, s''_{\mathtt{A}}.s'_{\mathtt{A}}$	deny, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	permit, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$
deny, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	deny, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$	deny, $s'_{\mathtt{B}}.s''_{\mathtt{B}}, s''_{\mathtt{A}}.s'_{\mathtt{A}}$	deny, $s'_{\mathtt{B}}, s'_{\mathtt{A}}$
not-app, ϵ, ϵ	permit, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	deny, $s''_{\mathtt{B}}, s''_{\mathtt{A}}$	permit, ϵ, ϵ

Table 5.5: Combination matrices for $\otimes$alg operators

($Requests \rightarrow Actions^* \times Actions^*$) that, given a request, fulfils each obligation in the sequence in input and composes the results according to the obligation type and the order of occurrence. In case of an empty sequence of obligations, two empty sequences of actions are returned. Fulfilling an obligation amounts to evaluate each enclosed action through the function $\mathcal{F}$. This function takes in input an action a and a request req, and returns a modified by replacing the possibly occurring structured names sn with the value (resulting from the evaluation of the policy value) which they are mapped to by req, i.e. $[\![\, req(sn) \,]\!]$. $\mathcal{F}$ can be defined inductively on the syntax of actions and their subterms. Its definition is lengthy but straightforward, hence it is omitted.

Differently from FACPL semantics, it is worth noticing that this semantics is only defined with respect to: (i) authorisation requests mapping all the structured names occurring within the policy constructs; (ii) relational functions applied to values of the expected type. The semantics could be easily extended to overcome these limitations and return, e.g., the not-app decision when the two conditions above are not met (see Remark 5.3 below). However, since in PSCEL neither not-app decisions nor explicit managements of errors are crucial, we prefer to work under the previous two simplifying assumptions.

Remark 5.3 (Partial Requests and Type Mismatch)**.** *For the sake of completeness, we report below a straightforward management, based on the special value $\bot$, of evaluation errors due to, e.g., partial requests or type mismatches of function arguments. Specifically, the definition of function $\mathcal{E}$ is updated as follows*

$$\mathcal{E}[\![\mathsf{true}]\!]req = \mathsf{true}$$

$$\mathcal{E}[\![\neg\tau]\!]req = \begin{cases} \neg\, \mathcal{E}[\![\tau]\!]req & \texttt{if} \quad \mathcal{E}[\![\tau]\!]req \in \{\mathsf{true}, \mathsf{false}\} \\ \bot & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\tau_1 \wedge\ \tau_2]\!]req = \begin{cases} \mathcal{E}[\![\tau_1]\!]req\ \wedge\ \mathcal{E}[\![\tau_2]\!]req & \texttt{if} \quad \mathcal{E}[\![\tau_1]\!]req, \mathcal{E}[\![\tau_2]\!]req \in \{\mathsf{true}, \mathsf{false}\} \\ \bot & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![\tau_1 \vee\ \tau_2]\!]req = \begin{cases} \mathcal{E}[\![\tau_1]\!]req\ \vee\ \mathcal{E}[\![\tau_2]\!]req & \texttt{if} \quad \mathcal{E}[\![\tau_1]\!]req, \mathcal{E}[\![\tau_2]\!]req \in \{\mathsf{true}, \mathsf{false}\} \\ \bot & \texttt{otherwise} \end{cases}$$

$$\mathcal{E}[\![f(sn, pv)]\!]req = \begin{cases} \mathsf{true} & \texttt{if} \quad f([\![\, req(sn) \,]\!], [\![\, pv \,]\!]) = \mathsf{true} \\ \mathsf{false} & \texttt{if} \quad f([\![\, req(sn) \,]\!], [\![\, pv \,]\!]) = \mathsf{false} \\ \bot & \texttt{otherwise} \end{cases}$$

Indeed, the new rules return the special value $\bot$ when an evaluation error occurs.

As now targets can also evaluate to $\bot$, we also need to modify the case of function $\mathcal{P}$ for the rule evaluation as follows

$$\mathcal{P}[\![\mathsf{n}(d\ \ \mathsf{target}:\tau\ \ \mathsf{obl}:o^*)]\!]req \quad = \quad \begin{cases} d, s_{\mathtt{B}}, s_{\mathtt{A}} & \texttt{if} \quad \mathcal{E}[\![\tau]\!]req = \mathsf{true}\ \wedge\ \mathcal{O}[\![o^*]\!]req = s_{\mathtt{B}}, s_{\mathtt{A}} \\ \mathsf{not\text{-}app}, \epsilon, \epsilon & \texttt{if} \quad \mathcal{E}[\![\tau]\!]req \in \{\mathsf{false}, \bot\} \end{cases}$$

which ensures that a rule evaluates to not-app *when its target evaluates to $\bot$. In this way, the decision returned by the evaluation of the policy is entrusted to the other rules of the policy and to the combining algorithm. Finally, notice that unsuccessful applications of the function $\mathcal{F}$ could be managed similarly.*

5.3.2 Programming Constructs

The operational semantics of programming constructs provides a full description of system behaviour, addressing as two distinguished steps the authorisation and execution of process actions. The presentation is structured into two parts. First, the *process semantics* specifies the *commitments*, that is the actions that processes are willing to perform and the continuation process obtained after each such action. Then, by considering commitments and system configurations, the *system semantics* describes the effects of action authorisation and execution at system level.

To define the semantics we need to introduce the notions of *bound* and *free* variables/-names. Actions **qry**(T) and **get**(T) *bind* the variables occurring in T preceded by the symbol '?', as well as action **read**$(?\,\underline{x}, \mathsf{n})$ *binds* the variable $\underline{x}$, in the continuation process. Recall that variables $\underline{x}$ and $\underline{X}$ refer to values and processes, respectively. Action $\mathbf{fresh}(n)$ *binds* the name n in the continuation process and, similarly, the restriction operator (νn) *binds* n in the system to which it is applied. Variables and names which are not bound are called *free*. Function $\mathrm{n}(\_)$ returns the set of (free and bound) names occurring in the term _. We say that two terms are *alpha-equivalent*, written $\equiv$, if one can be obtained from the other by renaming bound variables/names.

The semantics is only defined for *closed extended* systems. These are systems defined by the grammars in Tables 5.1 and 5.2 where all variables are bound and pairwise distinct, bound names are pairwise distinct and different from the free ones, and the syntax of actions is extended to also include *authorised* actions. These actions are process actions that have already obtained permission to be executed and whose arguments have been evaluated, i.e. items and templates do not contain composed expressions and destinations do not contain the reserved variable self. As a matter of notation, we write $\mathbf{a}$ to denote an evaluated action and $\bar{\mathbf{a}}$ to denote an authorised action; α will indicate either an action a or an authorised action $\bar{\mathbf{a}}$. Moreover, we use $\mathcal{I}$ and $\mathcal{J}$ to range over interfaces.

Semantics of Processes

We use P and Q to range over processes and write $P \downarrow_\alpha Q$ to mean that "P can commit to perform action α and become Q after doing so". The relation $\downarrow$ defining the semantics of processes is the least relation induced by the inference rules in Table 5.6. The first rule says that a process of the form $\alpha.P$ can commit to perform action α and continue as process P. The following two pairs of rules deal with (non-deterministic) choice and (interleaving) parallel operators, respectively, and are quite explicative. The next rule states that a process invocation $A(\tilde{p})$ behaves like the invoked process P, where the formal parameters $\tilde{f}$ are replaced by the actual parameters $\tilde{p}$. The last rule states that alpha-equivalent processes have the same commitments.

$$\frac{-}{\alpha.P \downarrow_\alpha P} \qquad \frac{P \downarrow_\alpha P'}{P + Q \downarrow_\alpha P'} \qquad \frac{Q \downarrow_\alpha Q'}{P + Q \downarrow_\alpha Q'} \qquad \frac{P \downarrow_\alpha P'}{P \mid Q \downarrow_\alpha P' \mid Q}$$

$$\frac{Q \downarrow_\alpha Q'}{P \mid Q \downarrow_\alpha P \mid Q'} \qquad \frac{P\{\tilde{p}/\tilde{f}\} \downarrow_\alpha P'}{A(\tilde{p}) \downarrow_\alpha P'}\ A(\tilde{f}) \triangleq P \qquad \frac{P' \downarrow_\alpha P''}{P \downarrow_\alpha P''}\ P \equiv P'$$

Table 5.6: Semantics of processes

$$\frac{-}{\mathsf{sat}(\mathsf{true},\mathcal{I}) = \mathsf{true}} \qquad \frac{\mathsf{sat}(\mathcal{P},\mathcal{I}) = b}{\mathsf{sat}(\neg\,\mathcal{P},\mathcal{I}) = \neg\, b} \qquad \frac{\mathsf{sat}(\mathcal{P}_1,\mathcal{I}) = b_1 \quad \mathsf{sat}(\mathcal{P}_2,\mathcal{I}) = b_2}{\mathsf{sat}(\mathcal{P}_1 \vee \mathcal{P}_2,\mathcal{I}) = b_1 \vee b_2}$$

$$\frac{(\mathsf{n} : e') \in I \quad [\![\, e' = e \,]\!] = b}{\mathsf{sat}(\mathsf{n} = e,\mathcal{I}) = b} \qquad \frac{(\mathsf{n} : e') \in I \quad [\![\, e' < e \,]\!] = b}{\mathsf{sat}(\mathsf{n} < e,\mathcal{I}) = b}$$

Table 5.7: Evaluation of predicates (b stands for a boolean value)

Semantics of Systems

The operational semantics of systems is defined by three LTSs: (i) the first one models the authorisation of actions with respect to the policies of components; (ii) the second one models the execution of authorised actions; (iii) the third one models the semantics of generic systems by considering name restrictions, which are instead neglected by the previous two LTSs. The LTSs, together with the definitions of the auxiliary functions they exploit, are presented in the following paragraphs.

Semantics of Systems (1 of 3): authorisation rules. The rules defining the transition relation modelling the authorisation of actions use the following auxiliary functions: (i) dst, to get the destination of an action, i.e. a name n or a predicate $\mathcal{P}$; (ii) acid, to get the action identifier, e.g. **put**; (iii) sat, to establish if an interface satisfies a predicate; (iv) $\mathcal{A}$, to evaluate actions with respect to a name; (v) req, to generate the authorisation request corresponding to a process commitment. Actions that do not explicitly specify a destination (e.g. actions **new** or **upd**) have the name of the executing component as destination. We also use the following notations: $\mathcal{I}.\mathsf{id}$ (resp., $\mathcal{I}.\pi$) indicates the name (resp., the policy) of the component having interface $\mathcal{I}$; $(\mathsf{n} : e) \in \mathcal{I}$ means that the feature $\mathsf{n} : e$ is exposed by $\mathcal{I}$ and the name n is referring to the expression e in $\mathcal{I}$.

The function sat, inductively defined on the syntax of predicates by the rules in Table 5.7, returns true when the interface taken as a second argument satisfies the predicate taken as a first argument. All rules are straightforward and amount to evaluate the predicate, once the feature names possibly occurring within are replaced by the value (resulting from evaluating the expression) which they refer to in $\mathcal{I}$.

The function $\mathcal{A}$, inductively defined on the syntax of actions and their arguments (i.e. items, templates and destinations) by the rules in Table 5.8, evaluates the (argu-

$$\mathcal{A}(\mathbf{put}(t)@c, n) = \mathbf{put}(\mathcal{A}(t,n))@\mathcal{A}(c,n)$$

$$\mathcal{A}(\mathbf{qry}(T)@c, n) = \mathbf{qry}(\mathcal{A}(t,n))@\mathcal{A}(c,n) \quad \mathcal{A}(\mathbf{get}(T)@c, n) = \mathbf{get}(\mathcal{A}(t,n))@\mathcal{A}(c,n)$$

$$\mathcal{A}(\mathbf{fresh}(n'), n) = \mathbf{fresh}(n') \quad \mathcal{A}(\mathbf{new}(\mathcal{I},\mathcal{K},\Pi,P), n) = \mathbf{new}(\mathcal{I},\mathcal{K},\Pi,P)$$

$$\mathcal{A}(\mathbf{upd}(\mathsf{n},e), n) = \mathbf{upd}(\mathsf{n},\mathcal{A}(e,n)) \qquad \mathcal{A}(\mathbf{read}(?\,\underline{x},\mathsf{n}), n) = \mathbf{read}(?\,\underline{x},\mathsf{n})$$

$$\mathcal{A}((t_1,t_2), n) = \mathcal{A}(t_1,n), \mathcal{A}(t_2,n) \qquad \mathcal{A}((T_1,T_2), n) = \mathcal{A}(T_1,n), \mathcal{A}(T_2,n)$$

$$\mathcal{A}(?\,\underline{x}, n) = ?\,\underline{x} \quad \mathcal{A}(?\,\underline{X}, n) = ?\,\underline{X} \quad \mathcal{A}(v,n) = v \quad \mathcal{A}(\mathcal{P},n) = \mathcal{P} \quad \mathcal{A}(P,n) = P$$

$$\frac{[\![\, e \,]\!] = v' \quad [\![\, \neg\, v' \,]\!] = v}{\mathcal{A}(\neg\, e, n) = v} \qquad \frac{[\![\, e_1 \,]\!] = v_1 \quad [\![\, e_2 \,]\!] = v_2 \quad [\![\, v_1 \ \mathsf{op}\ v_2 \,]\!] = v}{\mathcal{A}(e_1 \ \mathsf{op}\ e_2, n) = v} \qquad \frac{-}{\mathcal{A}(\mathsf{self}, n) = n}$$

Table 5.8: Evaluation of actions

ments of the) action taken as first argument by using the name taken as second argument. Applying the function to an action amounts to apply it to its arguments. On action arguments, the function acts as the identity, except for (closed) expressions that are evaluated and for the occurrences of the reserved variable self that are replaced by the name taken in input. For instance, $\mathcal{A}(\mathbf{put}(1+2)@\mathsf{self}, \mathsf{nm1})$ returns the evaluated action $\mathbf{put}(3)@\mathsf{nm1}$.

The function req generates the authorisation request corresponding to a process commitment. The request collects attributes representing the action to authorise and the interface features of executing and destination components. The function req is defined by case analysis on the syntax of actions as follows

$$\begin{aligned}
\mathsf{req}(\mathcal{I}, \mathbf{fresh}(n), \mathcal{I}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{fresh})\} \\
&\quad \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P), \mathcal{I}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \\
&\quad \cup \{(\mathsf{action/id}, \mathbf{new})\} \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{get}(T)@c, \mathcal{J}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{get})\} \\
&\quad \cup \{(\mathsf{action/arg}, (\!|\mathcal{A}(T, \mathcal{I}.\mathsf{id})|\!))\} \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{J}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{qry}(T)@c, \mathcal{J}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{qry})\} \\
&\quad \cup \{(\mathsf{action/arg}, (\!|\mathcal{A}(T, \mathcal{I}.\mathsf{id})|\!))\} \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{J}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{put}(t)@c, \mathcal{J}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{put})\} \\
&\quad \cup \{(\mathsf{action/arg}, (\!|\mathcal{A}(T, \mathcal{I}.\mathsf{id})|\!))\} \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{J}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{upd}(\mathsf{n}, e), \mathcal{I}) &= \{(\mathsf{subject/n}, e') \mid (\mathsf{n} : e') \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{upd})\} \\
&\quad \cup \{(\mathsf{action/arg}, (\!|\mathsf{n}, \mathcal{A}(e, \mathcal{I}.\mathsf{id})|\!))\} \cup \{(\mathsf{object/n}, e') \mid (\mathsf{n} : e') \in \mathcal{I}\} \\
\mathsf{req}(\mathcal{I}, \mathbf{read}(?\underline{x}, \mathsf{n}), \mathcal{I}) &= \{(\mathsf{subject/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\} \cup \{(\mathsf{action/id}, \mathbf{read})\} \\
&\quad \cup \{(\mathsf{action/arg}, (\!|?\underline{x}, \mathsf{n}|\!))\} \cup \{(\mathsf{object/n}, e) \mid (\mathsf{n} : e) \in \mathcal{I}\}
\end{aligned}$$

The attributes with category action describe the action identifier and the policy tuple representing the evaluated action arguments, while those with category subject (resp., object) report the contextual information bound to the interface features of the component executing (resp., destination of) the action. Actions **fresh**, **new**, **upd** and **read** are local, i.e. they only involve the component $\mathcal{I}$. Besides actions **fresh** and **new** that operate neither on the knowledge repository nor on the interface, the remaining actions define the attribute named action/arg. This attribute is associated to the policy tuple obtained by possibly applying first the function $\mathcal{A}$ for evaluating the action arguments and then the function $(\!| \cdot |\!)$ for abstracting from all those elements of items/templates that cannot occur in policy tuples, i.e. variable binders, predicates and processes (compare ITEMS and TEMPLATES of Table 5.1 with POLICY TUPLES of Table 5.2); all these elements are replaced by the wildcard '_'. Formally, the function $(\!| \cdot |\!)$ is defined as follows

$$\begin{aligned}
&(\!|t_1, t_2|\!) = (\!|t_1|\!), (\!|t_2|\!) \qquad (\!|T_1, T_2|\!) = (\!|T_1|\!), (\!|T_2|\!) \qquad (\!|v|\!) = v \\
&(\!|?x|\!) = _ \qquad (\!|?X|\!) = _ \qquad (\!|P|\!) = _ \qquad (\!|\mathcal{P}|\!) = _
\end{aligned}$$

Notably, $(\!| \cdot |\!)$ is only defined on evaluated items/templates of committed actions.

The transition relation modelling the authorisation of actions is defined by the rules in Tables 5.9 and 5.10. Table 5.9 reports first the rules for local actions, then those for point-to-point actions, and finally those for the group-oriented variants of **get** and **qry**. The rules for the group-oriented variants of **put** are instead in Table 5.10. The generated transition labels are as follows:

- τ, representing a single policy authorisation;
- $\mathcal{I} : \mathbf{a}, p, s_{\mathtt{B}}, s_{\mathtt{A}}$, representing (part of) a multiparty communication taking place to authorise the group-oriented variant of **put**.

Additionally, we use the following abbreviations: $r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}}$ indicates the request $\mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J})$; $\Pi^{\mathcal{I}}$ indicates the policy obtained by replacing all the occurrences of the reserved variable this in Π with $\mathcal{I}$.id.

We now comment the rules in Table 5.9. Rules *(l-p)* and *(l-d)* regulate, respectively, the permit and deny authorisation of local actions. In case of permit, the resulting process contains the evaluated action marked as authorised (i.e. by using notation $\bar{\mathbf{a}}$), and the sequences of actions $s_{\mathtt{B}}$ and $s_{\mathtt{A}}$ returned by the policy evaluation. Notably, the before actions $s_{\mathtt{B}}$ prefix $\bar{\mathbf{a}}$, while the after actions $s_{\mathtt{A}}$ follow $\bar{\mathbf{a}}$ and prefix the continuation process P'. This ensures that all adaptation actions will be executed in the correct order with respect to action $\bar{\mathbf{a}}$ and, in any case, before the rest of the process. When the authorisation is deny, although the action is forbidden, the resulting sequences of actions are installed; they in fact might adapt the system to permit a subsequent successful authorisation of the action or to enable an alternative execution path. The sequences $s_{\mathtt{A}}$ and $s_{\mathtt{B}}$ both prefix P, hence the forbidden action a, thus to not cause unexpected deadlocks. For instance, if an action in $s_{\mathtt{B}}$ has the duty of taking a lock and one in $s_{\mathtt{A}}$ of releasing it, such lock will be never released if a would never be authorised. Notably, since we assumed that bound variables are all distinct, the variable bindings possibly occurring within obligation actions do not change the scope of those variables that are already bound in the process.

The authorisation of remote actions is established by considering the possibly different authorisations resulting from the involved components. First, we comment the rules for point-to-point actions, then those for group-oriented actions.

A point-to-point action involves two different components and for its authorisation we may have three different cases. When both components authorise the action (rule *(ptp-pp)*), it is marked as authorised and the appropriate sequences of actions are installed in both components. When the executing component authorises the action while the destination component does not (rule *(ptp-pd)*), the sequences of actions are installed in the destination component, while the executing component remains unchanged. When the executing component does not authorise the action (rule *(ptp-d*)*), only such component installs the returned sequences of actions, while the other component remains unchanged.

Authorisation of the group-oriented variants of **get**/**qry** operates similarly. In fact, yet again the interaction only involves two components, although the destination one is nondeterministically chosen at runtime among those components satisfying a destination predicate. In case of positive authorisations from both components, the evaluated action is marked as authorised and modified by letting the destination to be the name $\mathcal{I}_2$.id of the interacting component. Rules *(gr-get-pp)* and *(gr-qry-pp)* detail this case for **get** and **qry**, respectively. The other two cases, dealt with by rules *(gr-get-qry-pd)* and *(gr-get-qry-d)*, are similar to the point-to-point variants previously described.

The authorisation rules for the group-oriented variants of action **put** are in Table 5.10. Such an action is authorised when all those components dynamically identified by the destination predicate are authorised by the executing component and at least one of them authorises the action too. Rule *(gr-put-p)* models indeed the positive authorisation by both the executing component and an arbitrarily chosen destination component. In addition to the interface of the executing component and the evaluated action, the transition label

LOCAL ACTIONS

$$\frac{P \downarrow_a P' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}.\mathsf{id} \quad \mathcal{P}[\![\Pi^{\mathcal{I}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{I}} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}}}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}, \Pi, s_{\mathtt{B}}.\bar{\mathbf{a}}.s_{\mathtt{A}}.P']} \textit{ (l-p)}$$

$$\frac{P \downarrow_a P' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}.\mathsf{id} \quad \mathcal{P}[\![\Pi^{\mathcal{I}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{I}} = \mathsf{deny}, s_{\mathtt{B}}, s_{\mathtt{A}}}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}, \Pi, s_{\mathtt{B}}.s_{\mathtt{A}}.P]} \textit{ (l-d)}$$

POINT-TO-POINT ACTIONS

$$\frac{\begin{array}{c} P_1 \downarrow_a P_1' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}_1.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}_2.\mathsf{id} \\ \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}' \end{array}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.\bar{\mathbf{a}}.s_{\mathtt{A}}.P_1'] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}'.s_{\mathtt{A}}'.P_2]} \textit{ (ptp-pp)}$$

$$\frac{\begin{array}{c} P_1 \downarrow_a P_1' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}_1.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}_2.\mathsf{id} \\ \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{deny}, s_{\mathtt{B}}', s_{\mathtt{A}}' \end{array}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}'.s_{\mathtt{A}}'.P_2]} \textit{ (ptp-pd)}$$

$$\frac{P_1 \downarrow_a P_1' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}_1.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}_2.\mathsf{id} \quad \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{deny}, s_{\mathtt{B}}, s_{\mathtt{A}}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.s_{\mathtt{A}}.P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2]} \textit{ (ptp-d*)}$$

GROUP GET/QRY

$$\frac{\begin{array}{c} P_1 \downarrow_{\mathbf{get}(T)@\mathcal{P}} P_1' \quad \mathbf{a} = \mathbf{get}(\mathcal{A}(T, \mathcal{I}_1.\mathsf{id}))@\mathcal{P} \quad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \\ \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}' \end{array}}{\begin{array}{l} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \\ \qquad \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.\overline{\mathbf{get}(\mathcal{A}(T, \mathcal{I}_1.\mathsf{id}))@\mathcal{I}_2.\mathsf{id}}.s_{\mathtt{A}}.P_1'] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}'.s_{\mathtt{A}}'.P_2] \end{array}} \textit{ (gr-get-pp)}$$

$$\frac{\begin{array}{c} P_1 \downarrow_{\mathbf{qry}(T)@\mathcal{P}} P_1' \quad \mathbf{a} = \mathbf{qry}(\mathcal{A}(T, \mathcal{I}_1.\mathsf{id}))@\mathcal{P} \quad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \\ \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}' \end{array}}{\begin{array}{l} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \\ \qquad \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.\overline{\mathbf{qry}(\mathcal{A}(T, \mathcal{I}_1.\mathsf{id}))@\mathcal{I}_2.\mathsf{id}}.s_{\mathtt{A}}.P_1'] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}'.s_{\mathtt{A}}'.P_2] \end{array}} \textit{ (gr-qry-pp)}$$

$$\frac{\begin{array}{c} P_1 \downarrow_a P_1' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}_1.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{P} \quad \mathsf{acid}(\mathbf{a}) \neq \mathbf{put} \quad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \\ \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{deny}, s_{\mathtt{B}}', s_{\mathtt{A}}' \end{array}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}'.s_{\mathtt{A}}'.P_2]} \textit{ (gr-get-qry-pd)}$$

$$\frac{\begin{array}{c} P_1 \downarrow_a P_1' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}_1.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{P} \\ \mathsf{acid}(\mathbf{a}) \neq \mathbf{put} \quad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \quad \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{deny}, s_{\mathtt{B}}, s_{\mathtt{A}} \end{array}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.s_{\mathtt{A}}.P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2]} \textit{ (gr-get-qry-d)}$$

Table 5.9: Semantics of programming constructs (1 of 3): authorisation rules (1 of 2) ($r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}}$ stands for $\mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J})$, $\Pi^{\mathcal{I}}$ stands for the policy obtained by replacing all the occurrences of the variable this in Π with $\mathcal{I}.\mathsf{id}$)

also reports a predicate indicating the name of those components already authorised to act as a destination component, and the sequences of before and after actions representing the results of local authorisation. These latter sequences are updated while the inference proceeds, i.e. other components join this multiparty interaction, and finally used to replace the placeholder $\bullet$ prefixing the continuation process P_1'. Indeed, when a system produces a label of the form $\mathcal{I} : \mathbf{a}, p, s_{\mathtt{B}}, s_{\mathtt{A}}$, any additional component satisfying the destination predi-

GROUP PUT

$$\frac{P_1 \downarrow_{\mathbf{put}(t)@\mathcal{P}} P_1' \qquad \mathbf{a} = \mathbf{put}(\mathcal{A}(t, \mathcal{I}_1.\mathsf{id}))@\mathcal{P} \qquad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \qquad \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}} \qquad \mathcal{P}[\![\Pi_2^{\mathcal{I}_2}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}'}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\mathcal{I}_1:\mathbf{a},\mathsf{id}=\mathcal{I}_2.\mathsf{id},s_{\mathtt{B}},s_{\mathtt{A}}} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, \bullet.P_1'] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, s_{\mathtt{B}}', s_{\mathtt{A}}'.P_2]} \textit{ (gr-put-p)}$$

$$\frac{P_1 \downarrow_{\mathbf{put}(t)@\mathcal{P}} P_1' \quad \mathbf{a} = \mathbf{put}(\mathcal{A}(t, \mathcal{I}_1.\mathsf{id}))@\mathcal{P} \quad \mathsf{sat}(\mathcal{P}, \mathcal{I}_2) = \mathsf{true} \quad \mathcal{P}[\![\Pi_1^{\mathcal{I}_1}]\!] r_{\mathbf{a}}^{\mathcal{I}_1,\mathcal{I}_2} = \mathsf{deny}, s_{\mathtt{B}}, s_{\mathtt{A}}}{\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1, s_{\mathtt{B}}.s_{\mathtt{A}}.P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2]} \textit{ (gr-put-d)}$$

$$\frac{S \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathtt{B}},s_{\mathtt{A}}} S' \qquad \mathbf{a} = \mathbf{put}(t)@\mathcal{P} \qquad \mathsf{sat}(\mathcal{P}, \mathcal{J}) = \mathsf{true} \qquad \mathcal{P}[\![\mathcal{I}.\pi^{\mathcal{I}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}' \qquad \mathcal{P}[\![\Pi^{\mathcal{J}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{permit}, s_{\mathtt{B}}'', s_{\mathtt{A}}''}{S \parallel \mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{a},p\vee\mathsf{id}=\mathcal{J}.\mathsf{id},s_{\mathtt{B}}.s_{\mathtt{B}}',s_{\mathtt{A}}'.s_{\mathtt{A}}} S' \parallel \mathcal{J}[\mathcal{K}, \Pi, s_{\mathtt{B}}''.s_{\mathtt{A}}''.P]} \textit{ (gr-put-pp)}$$

$$\frac{S \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathtt{B}},s_{\mathtt{A}}} S' \qquad \mathbf{a} = \mathbf{put}(t)@\mathcal{P} \qquad \mathsf{sat}(\mathcal{P}, \mathcal{J}) = \mathsf{true} \qquad \mathcal{P}[\![\mathcal{I}.\pi^{\mathcal{I}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{permit}, s_{\mathtt{B}}', s_{\mathtt{A}}' \qquad \mathcal{P}[\![\Pi^{\mathcal{J}}]\!] r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{deny}, s_{\mathtt{B}}'', s_{\mathtt{A}}''}{S \parallel \mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathtt{B}},s_{\mathtt{A}}} S' \parallel \mathcal{J}[\mathcal{K}, \Pi, s_{\mathtt{B}}''.s_{\mathtt{A}}''.P]} \textit{ (gr-put-pd)}$$

$$\frac{S \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathtt{B}},s_{\mathtt{A}}} S' \qquad \mathbf{a} = \mathbf{put}(t)@\mathcal{P} \qquad \mathsf{sat}(\mathcal{P}, \mathcal{J}) = \mathsf{false}}{S \parallel \mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathtt{B}},s_{\mathtt{A}}} S' \parallel \mathcal{J}[\mathcal{K}, \Pi, P]} \textit{ (engr-put)}$$

$$\frac{S_1 \xrightarrow{\tau} S_1'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1' \parallel S_2} \textit{ (as-aut)}$$

Table 5.10: Semantics of programming constructs (1 of 3): authorisation rules (2 of 2) ($r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}}$ stands for $\mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J})$, $\Pi^{\mathcal{I}}$ stands for the policy obtained by replacing all the occurrences of the variable this in Π with $\mathcal{I}.\mathsf{id}$, $\mathcal{I}.\pi$ is the policy of the component with interface $\mathcal{I}$)

cate and authorising the action can join the interaction (rule *(gr-put-pp)*): in the transition label, the predicate p is enriched by a disjunctive assertion of the form $\mathsf{id} = \mathcal{J}.\mathsf{id}$ and the sequences $s_{\mathtt{B}}$ and $s_{\mathtt{A}}$ are extended as well. Instead, rule *(gr-put-pd)* deals with any component satisfying the predicate but not authorising the action: in this case, the transition label is left unchanged. Notably, those components not satisfying the predicate do not affect the authorisation of a group-oriented **put** (rule *(engr-put)*). Instead, when a group-oriented **put** is not locally authorised, rule *(gr-put-d)* locally installs the sequences of before and after actions produced by policy evaluation and generates a τ-labelled transition. Finally, rule *(as-aut)* states that all those authorisations different from the positive authorisation of a group-oriented **put** can be performed by only involving some of the components. This exclusion ensures that when a component is going to authorise a group-oriented **put** all potential destination components are considered.

Remark 5.4 (On the approach followed by the rules in Table 5.10)**.** *The authorisation of a group-oriented **put** must be denied if a component satisfing the destination predicate is not authorised by the policy of the executing component. The authorisation rules in Table 5.10 enforce this behaviour by relying on the fact that at most one of the inferences generated from the rules (gr-put-p) and (gr-put-d) can actually take place. Indeed, given an action* $\mathbf{put}(t)@\mathcal{P}$*, let* F *be the set of (the interfaces of) the components satisfying* $\mathcal{P}$*, i.e. the destination components. If the policy of the executing component authorises the action for each component in*

$$\mathsf{match}(v, v) = \emptyset \qquad \mathsf{match}(\mathcal{P}, \mathcal{P}) = \emptyset$$

$$\mathsf{match}(?\,\underline{x}, v) = [v/\underline{x}] \qquad \mathsf{match}(?\,\underline{x}, \mathcal{P}) = [\mathcal{P}/\underline{x}] \qquad \mathsf{match}(?\,\underline{X}, P) = [P/\underline{X}]$$

$$\frac{\mathsf{match}(T_1, t_2) = \sigma_1 \quad \mathsf{match}(T_3, t_4) = \sigma_2}{\mathsf{match}((T_1, T_3)\,,\,(t_2, t_4)) = \sigma_1 \uplus \sigma_2}$$

Table 5.11: Matching rules

F and at least one of them authorises the action too, the only applicable rules are (gr-put-p), (gr-put-pp) (gr-put-pd) and (engr-put), and the action is actually authorised. Similarly, when all local authorisations are denied, only rules (gr-put-d) and (as-aut) apply, hence the action is denied. Instead, if the policy of the executing component authorises the action for some components in F and denies it for the remaining ones, then only the inference starting from rule (gr-put-d) successfully terminates by returning the expected system behaviour, i.e. the action is denied, whereas the inference starting from rule (gr-put-p) gets stuck because there is no rule able to manage the transition label it generates in presence of local negative authorisations.

Semantics of Systems (2 of 3): execution rules. Tables 5.12 and 5.13 report the rules defining the transition relation modelling the execution of authorised local and remote actions, respectively. In case of action **put**, rules and transition labels for point-to-point actions differ from those for group-oriented variants. The set of transition labels, ranged over by λ, is generated by the following productions

$$\begin{aligned}\lambda ::= \; & n : \mathbf{get}(T)@n' \mid n : \mathbf{qry}(T)@n' \mid n : \mathbf{fresh}(n') \mid n : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \\ & \mid n : \mathbf{put}(t)@n' \mid n : \mathbf{put}(t)@p \mid n : \mathbf{upd}(\mathsf{n}, e) \mid n : \mathbf{read}(\mathsf{n})\end{aligned}$$

where T/t are templates/items of evaluated actions, while p are predicates defined by the authorisation rules for group-oriented variants of **put**.

The execution rules for **get** and **qry** rely on the partial function match that, given a template T and an item t in input, implements a classical pattern-matching mechanism. Its definition is reported in Table 5.11. The rules state that a template T matches an item t, if they have the same number of elements and the corresponding elements have matching values or variable binders; variable binders match any value of the same type and two values match only if they are identical. When the arguments match, the function returns a substitution σ associating to the variables bound by T the corresponding values in t; otherwise, it is undefined. We use $\emptyset$ to denote the empty substitution and $\sigma_1 \uplus \sigma_2$ to denote the composition of substitutions σ_1 and σ_2 when they have disjoint domains.

We now comment the rules for local actions in Table 5.12. Rules *(l-put)*, *(l-get)* and *(l-qry)* model the management of local knowledge, i.e. when the destination of actions **put**/**get**/**qry** is the executing component itself. As effect of execution of action **put**, the item t (which has been evaluated during the authorisation phase) argument of the action is added to the local knowledge. Instead, as effect of execution of action **get**, the item t matching the template T argument of the action is removed from the knowledge repository. Execution of action **qry** only differs because the item is read but not removed from the knowledge. If more items match the template, one of them is arbitrarily chosen. In any case, the substitution σ generated by match is applied to the process continuation P' for replacing the variables bound by T with the corresponding values contained in the matched item.

LOCAL ACTIONS

$$\frac{P \downarrow_{\overline{\mathbf{put}(t)@n}} P' \qquad n = \mathcal{I}.\mathsf{id}}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@n} \mathcal{I}[\mathcal{K} \parallel t, \Pi, P']} \textit{(l-put)} \qquad \frac{P \downarrow_{\overline{\mathbf{get}(T)@n}} P' \quad n = \mathcal{I}.\mathsf{id} \quad \mathsf{match}(T,t) = \sigma}{\mathcal{I}[\mathcal{K} \parallel t, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{get}(t)@n} \mathcal{I}[\mathcal{K}, \Pi, P'\sigma]} \textit{(l-get)}$$

$$\frac{P \downarrow_{\overline{\mathbf{qry}(T)@n}} P' \quad n = \mathcal{I}.\mathsf{id} \quad \mathsf{match}(T,t) = \sigma}{\mathcal{I}[\mathcal{K} \parallel t, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{qry}(t)@n} \mathcal{I}[\mathcal{K} \parallel t, \Pi, P'\sigma]} \textit{(l-qry)} \qquad \frac{P \downarrow_{\overline{\mathbf{fresh}(n)}} P' \quad n \notin \mathrm{n}(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}])}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{fresh}(n)} (\nu n)\,\mathcal{I}[\mathcal{K}, \Pi, P']} \textit{(freshn)}$$

$$\frac{P \downarrow_{\overline{\mathbf{new}(\mathcal{J},\mathcal{K}_j,\Pi_j,P_j)}} P'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{new}(\mathcal{J},\mathcal{K}_j,\Pi_j,P_j)} \mathcal{I}[\mathcal{K}, \Pi, P'] \parallel \mathcal{J}[\mathcal{K}_j, \Pi_j, P_j]} \textit{(newc)}$$

$$\frac{P \downarrow_{\overline{\mathbf{upd}(\mathsf{n},v)}} P' \qquad \mathsf{n} \neq \mathsf{id}}{((\mathsf{n}' : e',)^*\ \mathsf{n} : e\ (,\mathsf{n}'' : e'')^*)[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{upd}(\mathsf{n},v)} ((\mathsf{n}' : e',)^*\ \mathsf{n} : v\ (,\mathsf{n}'' : e'')^*)[\mathcal{K}, \Pi, P']} \textit{(upd)}$$

$$\frac{P \downarrow_{\overline{\mathbf{read}(?\,\underline{x},\mathsf{n})}} P' \qquad \sigma = [\mathcal{A}(e, \mathcal{I}.\mathsf{id})/\underline{x}]}{((\mathsf{n}' : e',)^*\ \mathsf{n} : e\ (,\mathsf{n}'' : e'')^*)[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{read}(\mathsf{n})} ((\mathsf{n}' : e',)^*\ \mathsf{n} : e\ (,\mathsf{n}'' : e'')^*)[\mathcal{K}, \Pi, P'\sigma]} \textit{(read)}$$

Table 5.12: Semantics of programming constructs (2 of 3): execution rules (1 of 2)

Execution of action **fresh** (rule *(freshn)*) introduces a scope restriction for the name n. To this aim, the name n must not be used in the executing component except for its process (that indeed will likely use n). This condition, expressed by the hypothesis $n \notin \mathrm{n}(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}])$, can be always satisfied by exploiting alpha-equivalence among terms. Execution of action **new** (rule *(newc)*) causes the creation of the new component argument of the action.

Execution of action **upd** (rule *(upd)*) updates the value referred to by the interface feature n to the value v which has been obtained by evaluating the expression e during the authorisation phase; the condition $\mathsf{n} \neq \mathsf{id}$ ensures that the feature id cannot be changed. Execution of action **read** (rule *(read)*) retrieves the value (obtained by evaluating the expression e) referred to by the name n argument of the action and generates a substitution σ for the variable $\underline{x}$. The substitution is then applied to the process continuation P'. Both the rules *(upd)* and *(read)* exploit the notation $((\mathsf{n}' : e',)^*\ \mathsf{n} : e\ (,\mathsf{n}'' : e'')^*)$ to indicate that the feature n can be at any position of the list of pairs defining an interface.

The comments on the rules for remote actions in Table 5.13 follow. Rules *(rem-get)* and *(rem-qry)* deal with execution of remote variants of actions **get** and **qry**, respectively. The rules differ from the corresponding local ones only for the condition on the action destination, i.e. the destination name n has to be equal to the name of the component (with interface) $\mathcal{J}$ appearing in (the left-hand side of the conclusion of) the rule. Checking this condition is possible both for point-to-point and group-oriented variants because, due to the authorisation phase, an authorised **get** or **qry** action always contains the name of the destination component. Rule *(ptp-put)*, modelling execution of point-to-point variants of action **put**, differs from its local counterpart for similar aspects.

Execution of group-oriented variants of action **put** requires to add the action argument t to all the components authorised to receive it, i.e. all those components whose name satisfies the predicate p determined during the authorisation phase. Therefore, rule *(gr-put)* generates a label of the form $\mathcal{I}.\mathsf{id} : \mathbf{put}(t)@p$ reporting the needed information so that, thanks to rule *(gr-put-y)*, the intended destination components can receive t. Rule *(gr-put-n)*

REMOTE ACTIONS

$$\frac{P \downarrow_{\overline{\mathbf{get}(T)@n}} P'' \qquad n = \mathcal{J}.\mathsf{id} \qquad \mathsf{match}(T,t) = \sigma}{\mathcal{I}[\mathcal{K},\Pi,P] \parallel \mathcal{J}[\mathcal{K}' \parallel t,\Pi',P'] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{get}(t)@n} \mathcal{I}[\mathcal{K},\Pi,P''\sigma] \parallel \mathcal{J}[\mathcal{K}',\Pi',P']} \textit{(rem-get)}$$

$$\frac{P \downarrow_{\overline{\mathbf{qry}(T)@n}} P'' \qquad n = \mathcal{J}.\mathsf{id} \qquad \mathsf{match}(T,t) = \sigma}{\mathcal{I}[\mathcal{K},\Pi,P] \parallel \mathcal{J}[\mathcal{K}' \parallel t,\Pi',P'] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{qry}(t)@n} \mathcal{I}[\mathcal{K},\Pi,P''\sigma] \parallel \mathcal{J}[\mathcal{K}' \parallel t,\Pi',P']} \textit{(rem-qry)}$$

$$\frac{P \downarrow_{\overline{\mathbf{put}(t)@n}} P'' \qquad n = \mathcal{J}.\mathsf{id}}{\mathcal{I}[\mathcal{K},\Pi,P] \parallel \mathcal{J}[\mathcal{K}',\Pi',P'] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@n} \mathcal{I}[\mathcal{K},\Pi,P''] \parallel \mathcal{J}[\mathcal{K}' \parallel t,\Pi',P']} \textit{(ptp-put)}$$

$$\frac{P \downarrow_{\overline{\mathbf{put}(t)@p}} P'}{\mathcal{I}[\mathcal{K},\Pi,P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@p} \mathcal{I}[\mathcal{K},\Pi,P']} \textit{(gr-put)}$$

$$\frac{S \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@p} S' \qquad \mathsf{sat}(p,\mathcal{J}) = \mathsf{true}}{S \parallel \mathcal{J}[\mathcal{K},\Pi,P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@p} S' \parallel \mathcal{J}[\mathcal{K} \parallel t,\Pi,P]} \textit{(gr-put-y)}$$

$$\frac{S \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@p} S' \qquad \mathsf{sat}(p,\mathcal{J}) = \mathsf{false}}{S \parallel \mathcal{J}[\mathcal{K},\Pi,P] \xrightarrow{\mathcal{I}.\mathsf{id}:\mathbf{put}(t)@p} S' \parallel \mathcal{J}[\mathcal{K},\Pi,P]} \textit{(gr-put-n)}$$

$$\frac{S_1 \xrightarrow{\lambda} S_1' \qquad \lambda \neq \mathcal{I}.\mathsf{id} : \mathbf{put}(t)@p}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2} \textit{(as-ex)}$$

Table 5.13: Semantics of programming constructs (2 of 3): execution rules (2 of 2) (p stands for the predicate determined by the rules in Table 5.10)

ensures that the components not satisfying the predicate cannot affect its execution.

Finally, similarly to rule *(as-aut)*, rule *(as-ex)* states that execution of all actions different from group-oriented variants of **put** can be performed by only involving some of the system components. This exclusion ensures that when a component is going to execute a group-oriented **put** all potential destination components are taken into account.

Semantics of Systems (3 of 3): system transition rules. Table 5.14 reports the rules defining the transition relation modelling the semantics of a generic system (with names restrictions). The rules generate τ-labelled transitions, corresponding to action authorisations, or λ-labelled transitions, corresponding to action executions. As a matter of notation, γ ranges over both kinds of transition labels. Moreover, $\tilde{n}$ denotes a (possibly empty) sequence of names and $\tilde{n}, n'$ is the sequence obtained by composing $\tilde{n}$ and n'. $(\nu\tilde{n})S$ abbreviates $(\nu n_1)((\nu n_2)(\cdots(\nu n_m)S\cdots))$, if $\tilde{n} = n_1, n_2, \cdots, n_m$ with $m > 0$, and S, otherwise. $S\{n'/n\}$ denotes the system obtained by replacing any free occurrence of n in S with n'.

Rule *(acc-put)* deals with the authorisation of a group-oriented action **put**. When it is applied, all system components possibly matching the predicate have been taken into account to determine the predicate p, thus the placeholder $\bullet$ can be safely replaced by the sequence of actions $s_{\mathtt{B}}.\overline{\mathbf{put}(t)@p}.s_{\mathtt{A}}$ formed by the elements carried by the label. Notably, the action **put** has the predicate p as destination and it is prefixed (resp., followed) by the collected sequence of before (resp., after) actions. Hence, finally, the authorisation of

$$\frac{S \xrightarrow{\mathcal{I}:\mathbf{a},p,s_{\mathrm{B}},s_{\mathrm{A}}} S' \quad \mathbf{a} = \mathbf{put}(t)@\mathcal{P}}{(\nu\tilde{n})S \overset{\tau}{\rightarrowtail} (\nu\tilde{n})S'[s_{\mathrm{B}}.\overline{\mathbf{put}(t)@p}.s_{\mathrm{A}}/\bullet]}\ (acc\text{-}put) \qquad \frac{S \xrightarrow{\gamma} S'}{(\nu\tilde{n})S \overset{\gamma}{\rightarrowtail} (\nu\tilde{n})S'}\ (aut\text{-}exec)$$

$$\frac{(\nu\tilde{n}, n')(S_1 \parallel S_2) \overset{\gamma}{\rightarrowtail} S'}{(\nu\tilde{n})(S_1 \parallel (\nu n')S_2) \overset{\gamma}{\rightarrowtail} S'}\ (top) \qquad \frac{(\nu\tilde{n})(S_2 \parallel S_1) \overset{\gamma}{\rightarrowtail} S'}{(\nu\tilde{n})(S_1 \parallel S_2) \overset{\gamma}{\rightarrowtail} S'}\ (comm) \qquad \frac{(\nu\tilde{n})((S_1 \parallel S_2) \parallel S_3) \overset{\gamma}{\rightarrowtail} S'}{(\nu\tilde{n})(S_1 \parallel (S_2 \parallel S_3)) \overset{\gamma}{\rightarrowtail} S'}\ (assoc)$$

Table 5.14: Semantics of programming constructs (3 of 3): system transition rules

a group-oriented **put** boils down to a τ-labelled transition as well. Rule *(aut-exec)* states that all the other transitions corresponding to action authorisations and all the transitions corresponding to action executions generate a system transition.

Rule *(top)* manipulates the syntax of systems for moving all name restrictions at top level, thus ensuring that one of the first two rules apply. These rules indeed apply only when all (possible) name restrictions are at top level (otherwise the transitions in the premises could not be inferred). Finally, rules *(comm)* and *(assoc)* make system composition a commutative and associative operator.

5.4 PSCEL at work on the Robot-Swarm Case Study

In this section we employ PSCEL, enriched with support for adaptive policies, to model the robot-swarm case study presented in Section 2.3.2. Therefore, we first introduce the PSCEL extension supporting adaptive policies (Section 5.4.1), then we present the PSCEL specification of the case study (Section 5.4.2).

5.4.1 A PSCEL Extension: Adaptive Policies

The PSCEL language, in the context of the ASCENS project, has been extended in order to support additional functionalities [DLL+15]. Here, we only illustrate the support of adaptive policies, i.e. policies that can dynamically change while components evolve.

To explicitly express the fact that the policy in force at any given component can dynamically change over time, we use an automaton, somehow reminiscent of *security automata* [Sch00], whose states identify FACPL policies and transitions between states model possible modifications of the policy actually in force at the component. From time to time, the policy of the current state is the policy actually in force. We outline below the needed modifications to PSCEL syntax and semantics for supporting this extension.

Syntactically, the policy Π of components is replaced by a POLICY AUTOMATON formed by a pair $\langle A, p \rangle$. Its definition follows

- A is an automaton of the form $\langle Pol, Trg, \mathcal{T}r \rangle$ where: (i) $Pol \subseteq Policies$ is the set of states identifying all the policies that can be in force at different times; (ii) $Trg \subseteq Targets$ is the set of conditions that can trigger policy modifications; (iii) $\mathcal{T}r \subseteq (Pol \times Trg \times Pol)$ is the set of all the labelled transitions between states;
- $p \in Pol$ is the current state of A.

The sets $Policies$ and $Targets$ represent, respectively, the sets of all POLICIES and TARGETS from Table 5.2.

Semantically, the automaton intervenes when the authorisation requests corresponding to process actions are evaluated. Namely, the premise $\mathcal{P}[\![\Pi^{\mathcal{I}}]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = d, s_{\mathtt{B}}, s_{\mathtt{A}}$ occurring in Tables 5.9 and 5.10 has to address the evaluation of the automaton and its possible modifications. Therefore, if we let $\Pi \triangleq \langle A, p\rangle$ the previous premise is replaced by $\mathcal{PA}[\![\Pi^{\mathcal{I}}]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = d, s_{\mathtt{B}}, s_{\mathtt{A}}, \Pi'$. Its definition follows

$$\mathcal{PA}[\![\Pi^{\mathcal{I}}]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \begin{cases} d, s_{\mathtt{B}}, s_{\mathtt{A}}, \langle A, p'\rangle & \texttt{if } \mathcal{P}[\![p[\mathcal{I}.\mathsf{id}/\mathsf{this}]]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = d, s_{\mathtt{B}}, s_{\mathtt{A}} \\ & \quad \wedge \langle p, \tau, p'\rangle \texttt{ in } A \wedge\ \mathcal{E}[\![\tau]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{true} \\ d, s_{\mathtt{B}}, s_{\mathtt{A}}, \Pi & \texttt{if } \mathcal{P}[\![p[\mathcal{I}.\mathsf{id}/\mathsf{this}]]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = d, s_{\mathtt{B}}, s_{\mathtt{A}} \\ & \quad \wedge ((\langle p, \tau, p'\rangle \texttt{ not in } A) \\ & \qquad \vee (\langle p, \tau, p'\rangle \texttt{ in } A \wedge\ \mathcal{E}[\![\tau]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} = \mathsf{false})) \end{cases}$$

where notation $\Pi^{\mathcal{I}}$ is used to identify the component interface $\mathcal{I}$ whose feature id is used to replace the occurrences of the variable this. The semantic function makes use of the semantics of targets and policies of Table 5.4. Intuitively, an action $\mathbf{a}$ is authorised to decision d, together with the sequences of additional actions $s_{\mathtt{B}}$ and $s_{\mathtt{A}}$, by the policy p of current automaton state. Moreover, if for some target $\tau \in Targets$, such that $\mathcal{E}[\![\tau]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{J}} =$ true, the automaton A has a transition $\langle p, \tau, p'\rangle$, then the state of A after the request evaluation becomes Π'. Notably, the current policy in Π does not change unless there is a target τ matching the request and producing a transition in the policy automaton. Of course, if the automaton has a single state or an empty set of transitions, the policy in force at a component never changes. Instead, to address the automaton modifications, we need to enforce the possibly modified automaton Π' in the resulting components of the rules in Tables 5.9 and 5.10. By way of example, the rule modelling the authorisation of local actions is updated as follows

$$\frac{P \downarrow_a P' \quad \mathbf{a} = \mathcal{A}(a, \mathcal{I}.\mathsf{id}) \quad \mathsf{dst}(\mathbf{a}) = \mathcal{I}.\mathsf{id} \quad \mathcal{PA}[\![\Pi^{\mathcal{I}}]\!]r_{\mathbf{a}}^{\mathcal{I},\mathcal{I}} = \mathsf{permit}, s_{\mathtt{B}}, s_{\mathtt{A}}, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\ \tau\ } \mathcal{I}[\mathcal{K}, \Pi', s_{\mathtt{B}}.\bar{\mathbf{a}}.s_{\mathtt{A}}.P']} \textit{ (l-p)}$$

Indeed, the automaton Π' returned by the evaluation of Π is placed as the policy automaton of the resulting component.

Dynamically changing policies is a powerful mechanism that permits controlling, in a natural and clear way, the evolution of adaptive systems. We exploit it in the PSCEL modelling of the robot-swarm case study.

5.4.2 PSCEL Specification

According to the separation of concerns principle fostered by PSCEL, to model the robot-swarm case study we define (i) a process, defining the functional behaviour of robots; and (ii) a collection of policies organised in term of a policy automaton, regulating the interactions among robots and with the environment, and generating the adaptation actions necessary to react to the changing operating conditions. The interplay between processes and policies permits a convenient design and enacts a collaborative swarm behaviour aiming at implementing the robot behaviours described by the Requirements of Table 2.2, hence at accomplishing the goal of rescuing victims.

The specification relies on the fact that processes can read from the knowledge items produced by robot sensors (e.g. the item $\langle collision, \mathsf{true}\rangle$ indicating that an imminent collision with an arena wall has been detected), and can add items triggering robot actuators

that force the activation of specific functionalities (e.g. the item $\langle stop \rangle$ forcing the robot to halt its movement). As a matter of fact, we do not explicitly model sensors and actuators, because our focus is on the collaborative behaviours of robots. We only assume that the actuator triggered by items of the form $\langle goTo, nDest, x, y \rangle$ forces the robot to reach the position (x, y) and signals the arrival at such position by means of the item $\langle arrived, nDest \rangle$.

The robot-swarm case study is modelled as the following PSCEL system

$$\mathcal{I}_{R_1}[\mathcal{K}_{R_1}, \Pi_R, P_R] \parallel \ldots \parallel \mathcal{I}_{R_n}[\mathcal{K}_{R_n}, \Pi_R, P_R]$$

Each robot interface $\mathcal{I}_R$ contains, besides id, the features role, battery, xRobot, yRobot, xStart and yStart. In details, role represents the abstract role currently played by the robot, i.e. one among $explorer$, $rescuer$, $helpRescuer$, $lowBattery$; battery represents the level of charge of the battery, i.e. an integer between 0 and 100; xRobot and yRobot represent the current position of the robot; xStart and yStart represent the starting position of the robot and where the recharging station is placed. Thus, a robot interface can be of the form $\mathcal{I}_{R_1} \triangleq (\mathsf{id}\colon r1, \mathsf{role}\colon explorer, \mathsf{battery}\colon 56, \mathsf{xRobot}\colon 4.35, \mathsf{yRobot}\colon 8.17, \mathsf{xStart}\colon 1.2, \mathsf{yStart}\colon 5.3)$. It follows that updating, e.g., the role amounts to execute the action **upd**(role, $rescuer$), while reading the value of, e.g., the feature xStart amounts to execute the action **read**($?\,\underline{x}$, xStart). Notice that initially the robot role is $explorer$.

Intuitively, the robot behaviours are structured as follows: process P_R specifies all the basic behaviours a robot can do, while the policy automaton Π_R controls and adapts such behaviours according to the current role of the robot, i.e. depending on the policy enforced by the current automaton state.

The process P_R of each robot is defined as follows

$$\begin{aligned} P_R \triangleq (\ & \mathbf{qry}(victimPerceived, \mathsf{true})@\mathsf{self}.\mathbf{read}(?\,\underline{x}, \mathsf{xRobot}).\mathbf{read}(?\,\underline{y}, \mathsf{yRobot}). \\ & \quad \mathbf{put}(victim, \underline{x}, \underline{y}, 3)@\mathsf{self}.\ \mathbf{put}(rescue)@\mathsf{self} \\ & + \mathbf{get}(victim, ?\underline{xVictim}, ?\underline{yVictim}, ?\underline{count})@(\mathsf{role}{=}rescuer \vee \mathsf{role}{=}helpRescuer). \\ & \quad HelpingRescuer\ ) \\ & \mid RandomWalk \mid IsMoving \end{aligned}$$

According to Requirement (Rs-1) the robots follow a random walk to explore the arena. To this aim, the process *RandomWalk* randomly selects a direction that is followed until either a wall is hit or a $stop$ signal is sent to the wheel actuator. The robot recognises the presence of a victim by means of the **qry** action, while the action **get** permits it to help other robots to rescue a victim, that is robots that have role $rescuer$ or $helpRescuer$ (i.e. it is used the predicate $\mathsf{role}{=}rescuer \vee \mathsf{role}{=}helpRescuer$). When a victim is found, information about its position, which is retrieved by means of actions **read** reading features xRobot and yRobot, and the number of additional robots, i.e. 3, needed for rescuing the victim is locally published. This behaviour corresponds to Requirement (Rs-2). To effectively start the rescuing procedure, the item $\langle rescue \rangle$ is then locally added.

The *RandomWalk* process calculates the random direction followed by the robot to explore the arena. The robot starts moving as soon as the first direction is calculated. When the proximity sensor signals a possible collision, by means of the item $\langle collision, \mathsf{true} \rangle$, a new random direction is calculated. This behaviour corresponds to the following process

$$\mathit{RandomWalk} \triangleq \mathbf{put}(direction, 2\pi\mathsf{rand}())@\mathsf{self}.\mathbf{qry}(collision, \mathsf{true})@\mathsf{self}.RandomWalk$$

where $2\pi\mathsf{rand}()$ is assumed to be a random angular direction. Notice that the process defines only the direction of the motion and not the will of moving.

The *HelpingRescuer* process is defined as follows

$$\begin{aligned} \mathit{HelpingRescuer} \triangleq\ & \mathbf{if}\ (\underline{\mathit{count}} > 1)\ \mathbf{then}\ \{\ \mathbf{put}(\mathit{victim}, \underline{\mathit{xVictim}}, \underline{\mathit{yVictim}}, \underline{\mathit{count}}\text{-}1)@\mathsf{self}\ \}. \\ & \mathbf{put}(\mathit{goTo}, \mathit{victim}, \underline{\mathit{xVictim}}, \underline{\mathit{yVictim}})@\mathsf{self}. \\ & \mathbf{get}(\mathit{arrived}, \mathit{victim}, \underline{\mathit{xVictim}}, \underline{\mathit{yVictim}})@\mathsf{self}.\ \mathbf{put}(\mathit{rescue})@\mathsf{self} \end{aligned}$$

where the high-level construct **if-then** is used, with its obvious meaning, to simplify the process specification[3]. This process is triggered by a victim (knowledge) item retrieved from the ensemble of robots satisfying the predicate $\mathsf{role}{=}\mathit{rescuer} \vee \mathsf{role}{=}\mathit{helpRescuer}$ (see process P_R). The item indicates that additional robots (whose number is stored in count) are needed at position $(\underline{\mathit{xVictim}}, \underline{\mathit{yVictim}})$ to rescue a victim. If more than one robot is needed, a new victim item is published (with decremented counter). Then, the robot goes towards the victim position and once it reaches it (i.e. the **get** action completes), the local addition of the item $\langle \mathit{rescue} \rangle$ triggers the rescuing procedure; this behaviour, together with the **qry** action of P_R, corresponds to Requirements (Rs-3) and (Rs-4). It is worth noting that, if more victims are in the arena, different groups of rescuers will be spontaneously organised to rescue them. To avoid that more than one group is formed to assist the same victim, we assume that the sensor used to perceive the victims is configured so that a victim that is already receiving assistance by some rescuers is not detected as a victim by further robots. This assumption is also feasible in a real scenario, where a light-based message communication among robots can be used [OGCD10]: when a robot reaches a victim, it uses a specific light colour to signal that the victim is already receiving assistance.

Notably, the effectiveness of this discover-and-rescue procedure relies on the assumption that robots cannot fail. In fact, when a robot that knows the victim position fails, it cannot be ensured that such position is correctly communicated. Anyway, specific handling can be used in such a case, e.g. by enabling the perception of a victim if the stationary robots close to it are not active.

Finally, in order for the policy to control the battery level during the exploration, we need to capture the movement status. This information is represented by the item $\langle \mathit{isMoving} \rangle$, which is produced by the wheel sensor, and read by the following process

$$\mathit{IsMoving} \triangleq \mathbf{qry}(\mathit{isMoving})@\mathsf{self}.\mathit{IsMoving}$$

The authorisation of this action allows policies to check the battery level and, when it is critical, to opportunely adapt the robot behaviour in order to recharge the battery according to Requirement (Rs-5).

The processes just presented need to be controlled and adapted in order to check context information, e.g. the battery level, and to execute additional actions, e.g. halting the movement and executing the battery recharging procedure. To this aim, each robot features the policy automaton Π_R reported in Figure 5.1. The automaton states correspond to the policy in force when the robot plays the same role as the state name. By design choice, we manage the role changing by means of obligation actions dynamically enforced (in fact, processes do not contain any **upd** action updating the feature role). Thus, the consequent authorisation of these obligation actions triggers automaton state changes. The

[3] Since $\underline{\mathit{count}}$ can assume a finite range of values, i.e. from 0 to 3, the **if-then** construct is just an abbreviation for a more lengthy specification based on pattern-matching conditions.

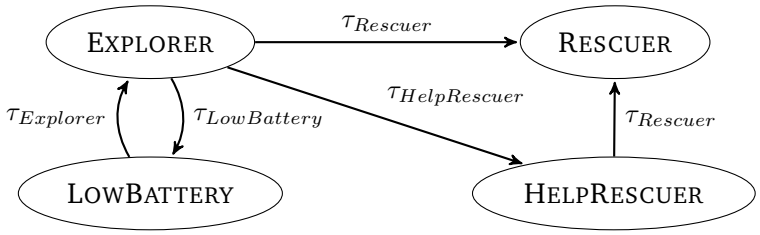


Figure 5.1: Robot-swarm case study: policy automaton

transition conditions are reported in Table 5.15. In the following, we present some of the policies of the automaton states and we informally comment on their runtime interplay with the process P_R.

The EXPLORER state identifies the Policy (Rs-E), which is defined as follows

$$
\begin{array}{l}
\langle \text{p-unless-d} \\
\quad \text{rules :} \\
\qquad \text{E1 (permit target : equal(action/id,\textbf{qry})} \\
\qquad\qquad \wedge \text{ pattern-match(action/arg,}(victimPerceived, \text{true})) \\
\qquad\quad \text{obl : [A } \mathbf{put}(stop)@\text{self.}\mathbf{upd}(\text{role}, rescuer)] \text{)} \\
\qquad \text{E2 (permit target : equal(action/id,\textbf{get})} \\
\qquad\qquad \wedge \text{ pattern-match(action/arg,}(victim, \_, \_)) \\
\qquad\quad \text{obl : [A } \mathbf{upd}(\text{role}, helpRescuer)] \text{)} \\
\qquad \text{E3 (deny target : equal(\textbf{qry},action/id) } \wedge \text{ pattern-match(action/arg,}isMoving) \\
\qquad\qquad \wedge \text{ less-than(subject/battery,}20) \\
\qquad\quad \text{obl : [B } \mathbf{put}(goTo, start, \text{subject/xStart, subject/yStart})@\text{self.} \\
\qquad\qquad \mathbf{upd}(\text{role}, lowBattery).\mathbf{get}(arrived, start)@\text{self}] \text{)} \rangle
\end{array}
\qquad \text{(Rs-E)}
$$

The rule E1 has the purpose of positively authorising the action **qry** sensing the perception of a victim and of returning the obligation actions $\mathbf{put}(stop)$@self and $\mathbf{upd}(\text{role}, rescuer)$. The action **put** instructs the wheel actuator to halt the robot movement. Thus, as soon as a victim is sensed (i.e. the corresponding action **qry** completes), such action, which is authorised by this policy (i.e. due to the p-unless-d algorithm), is executed and the robot stops. The subsequent **upd** action is authorised as well by also triggering an automaton state change; its execution changes the robot role to $rescuer$. Similarly, the rule E2 authorises the action **get** of the process P_R by returning a **upd** action that, when such **get** has completed, changes the robot role to *helpRescuer*. Finally, the rule E3 is used to check the battery and when the level is critical (i.e. lower than 20), it forbids the action **qry** of the $IsMoving$ process and returns the obligation $\mathbf{put}(goTo, start, \text{subject/xStart, subject/yStart})$@self for instructing the robot to reach the recharging station. The dynamical fulfilment of the attributes subject/xStart and subject/yStart permits retrieving (through the attribute-based request under evaluation)

Name	*Condition*
$\tau_{Rescuer}$	equal(action/id, **upd**) $\wedge$ pattern-match(action/arg, (role, *rescuer*))
$\tau_{HelpRescuer}$	equal(action/id, **upd**) $\wedge$ pattern-match(action/arg, (role, *helpRescuer*))
$\tau_{Explorer}$	equal(action/id, **upd**) $\wedge$ pattern-match(action/arg, (role, *explorer*))
$\tau_{LowBattery}$	equal(action/id, **upd**) $\wedge$ pattern-match(action/arg, (role, *lowBattery*))

Table 5.15: Robot-swarm case study: policy automaton transition conditions

the needed position from the robot interface. The additional obligations actions **upd** and **get** are used, respectively, to change the robot role to $lowBattery$ and check when the robot arrives at destination.

The policy of the HELPRESCUER state has only the duty of managing the change from the $helpRescuer$ role to the $rescuer$ one. Indeed, when the action **get** of the process $HelpingRescuer$ is authorised, the corresponding **upd** action is returned.

The LOWBATTERY state is instead associated to the following policy

$$\begin{array}{l}\langle \text{d-unless-p} \\ \quad \text{rules :} \\ \qquad \text{L1 (permit target : equal(action/id,}\mathbf{get}\text{)} \\ \qquad\qquad \wedge \text{ pattern-match(action/arg,}(arrived, start)\text{)} \\ \qquad\quad \text{obl : [A } \mathbf{put}(charge)@\text{self}.\mathbf{get}(charged)@\text{self}.\mathbf{upd}(\text{role}, explorer)]\rangle\end{array} \quad \text{(Rs-L)}$$

Policy (Rs-L) forbids all actions by means of the algorithm d-unless-p, but the **get** action checking the arrival of the robot at the recharging station. Such action is that enforced, while the robot is an $explorer$, by the rule E3 of Policy (Rs-E). This action needs to be authorised by the Policy (Rs-L), because the **upd** action preceding such **get**, i.e. that enforced by rule E3, changes the robot role to $lowBattery$ and the automaton state accordingly. As result of the authorisation, the rule L1 returns three actions. The first two ones are used to trigger and observe, respectively, the start and the end of the charging procedure, while the last one changes the robot state to $explorer$.

Finally, Policy (Rs-R) associated to the RESCUER state is defined as follows

$$\begin{array}{l}\langle \text{p-unless-d} \\ \quad \text{rules :} \\ \qquad \text{R1(permit target : equal(}\mathbf{put}\text{,action/id) } \wedge \text{ pattern-match(action/arg,}rescue\text{)} \\ \qquad\qquad \wedge \text{ greater-than(subject/battery,}40\text{)} \\ \qquad\quad \text{obl : [A } \mathbf{put}(camera, on)@\text{self]) }\rangle\end{array} \quad \text{(Rs-R)}$$

This policy does not forbid any action, it is only used for turning on the robot camera if there is enough battery; other functionalities could be activated as well.

5.5 Supporting Tools

In this section we present the supporting tools of PSCEL[4]: (i) jRESP, a Java runtime environment for developing autonomic systems according to the PSCEL approach; (ii) a PSCEL Eclipse-based IDE supporting coding and analysis of PSCEL specifications.

In the rest of this section, we outline the main functionalities of jRESP (Section 5.5.1) and of the PSCEL IDE (Section 5.5.2).

5.5.1 The PSCEL Java Runtime Environment

To effectively deploy PSCEL systems we provide the jRESP Java runtime environment. jRESP was originally proposed to support SCEL [DLPT14] by offering a set of APIs for the development of distributed applications based on (a part of) the programming constructs of Table 5.1. Thus, to fully support PSCEL, we enhanced the jRESP functionalities with

[4]The PSCEL supporting tools are freely available and open-source; binary files, source files, unit tests and a user's guide can be found at the PSCEL website [PSC16].

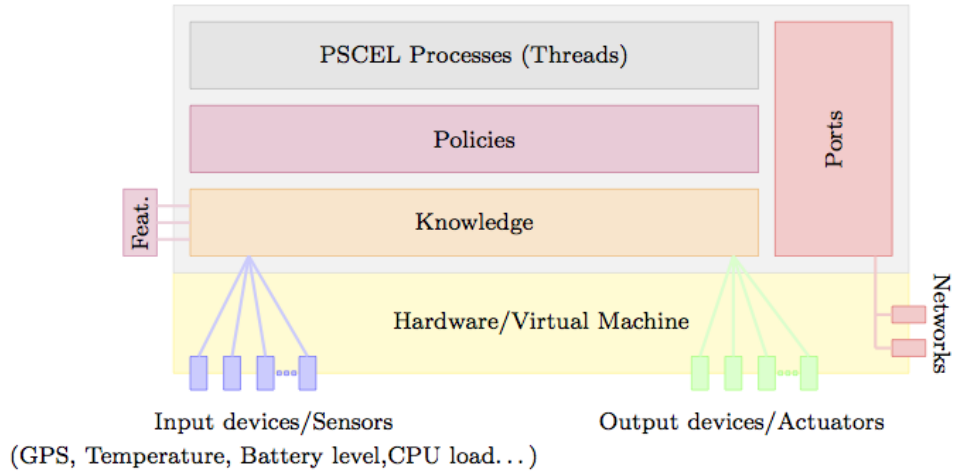


Figure 5.2: Architecture of Java PSCEL components

the support for the specification and evaluation of FACPL-based policy automata (components featuring single policies are supported as well, i.e. such policies are represented as automata with one state and no transitions).

In the following, we first present the design principles underlying the design of jRESP, then we outline the integration strategies of the FACPL-based policy automata, and we conclude by commenting on the jRESP simulation of the robot-swarm case study.

Design Principles

The key part of the jRESP framework is the implementation of PSCEL components. They are modelled via the class Node and their architecture is shown in Figure 5.2. A node aggregates an interface, a knowledge repository, a set of running processes, and a set of policies. A node interacts with other nodes via ports, which provide support to both point-to-point and group-oriented communications.

The node knowledge is modelled by the interface Knowledge, which indicates the high-level primitives to manage knowledge elements. By default, jRESP provides an item-based implementation of the knowledge (see Table 5.1) and supports the corresponding pattern-matching mechanism. The items can be produced by processes or by sensors. Each sensor can be associated to a logical functionality that produces data. Similarly, actuators can be used to send data, e.g., to external services.

The support for communication is achieved via the abstract class AbstractPort. According to the underlying communication infrastructure, this class is appropriately refined, e.g. the classes InetPort and P2PPort implement the TCP/UPD and peer-to-peer communications, respectively. Additionally, to enable local simulations of distributed communications, the class VitualPort models node interactions via a buffered memory.

The node processes are implemented as threads via the abstract class Agent. This class provides the methods implementing all the process actions.

The node policies are defined by the interface IPolicy, which defines the methods to be used to authorise local and remote actions. In details, when a method of the class Agent corresponding to a process action is invoked, its execution is delegated to the policy in force at the node where the agent is running. The policy can thus authorise or not the

execution of the action and possibly adapt the process behaviour by enforcing adaptation strategies. By default, IPolicy is instantiated by the class DefaultPermitPolicy, which allows any action to be executed.

FACPL Integration

To support the specification of FACPL-based policy automata, we instantiate the interface IPolicy accordingly. Specifically, we define a generic policy automaton and instantiate its state definition in order to refer to FACPL policies.

The class PolicyAutomaton implements a generic automaton whose current state contains the reference to the policy actually in force at the jRESP node. An automaton consists of (i) a set of classes IPolicyAutomatonState, each of which identifies the possible policies in force at the node; (ii) a set of transitions; (iii) a reference to the current automaton state identifying the actual policy.

The evaluation of the automaton is defined by instantiating the interface IPolicy. In details, the provided methods invoke the evaluation of the policy of the current state by taking as argument an instance of the class AutorisationRequest, which represents the action to authorise and its context (i.e. the request produced by the function req defined in Section 5.3.2). Each invocation can trigger in its own turn an update of the automaton state according to the satisfied transitions (see Section 5.4.1).

The full PSCEL support can be now achieved by defining the class FacplPolicyState, which extends IPolicyAutomatonState and wraps the Java-translated FACPL policies. This class overwrites the automaton evaluation methods and renders FACPL obligation actions as appropriate Agent instances. Clearly, the integration of FACPL policies relies on the FACPL Java library presented in Section 3.6.1.

Simulation of the Robot-Swarm Case Study

We comment here on the jRESP implementation of the PSCEL specification modelling the robot-swarm case study[5].

Each robot corresponds to a Node implementing the presented interface $\mathcal{I}_R$ and equipped with the appropriate sensors and actuators. For example, sensors include those used to detect victims, access the battery level and detect possible collisions. Instead, actuators include those used to set the robot direction, stop the movement and start the battery recharging procedure.

The node process is defined by four Agent instantances: Explorer, HelpingRescuer, RandomWalk and isMoving. Agents Explorer and HelpingRescuer represent, respectively, the two branches of the non-deterministic choice of the process P_R[6].

By way of example, we report in Listing 5.1 the jRESP code of the Explorer agent. Looking at the code, it is easy to see an almost one-to-one correspondence between the code itself and the corresponding branch of the PSCEL process P_R.

The policies in force at each node are managed by an instance of the class PolicyAutomaton implementing the automaton reported in Figure 5.1. The Java code of each FACPL

[5]The complete code for the scenario, together with a simulation environment, can be downloaded from the jRESP code repository [jRE16] or from the PSCEL website [PSC16].

[6]Non-deterministic choice is rendered as a concurrent execution of agents, i.e. Java threads, that are regulated by checks on the current robot role.

```
public class Explorer extends Agent {

  public Explorer() {
    super("Explorer");
  }

  protected void doRun() throws Exception {
    query(
        new Template(new ActualTemplateField("victimPerceived"),
            new ActualTemplateField(true)),
        Self.SELF
    );
    Object x = readAttribute("xRobot");
    Object y = readAttribute("yRobot");
    put(new Tuple("victim", x, y, 3, Self.SELF);
    put(new Tuple("rescue"), Self.SELF);
  }
}
```

Listing 5.1: Robot-swarm case study: jRESP code of the Explorer agent

policy has the form of that reported in Section 3.6.1, while the transition conditions of Table 5.15 are straightforwardly translated to Java code and integrated as part of the automaton.

5.5.2 The PSCEL IDE

The PSCEL IDE is an Eclipse plug-in offering a tailored development environment for PSCEL. Indeed, it supports with graphical features the coding of PSCEL code and permits automated generation of runnable jRESP and SMT-LIB code. The SMT-LIB code is used in the analysis of PSCEL specifications presented in Chapter 6; additional comments on the support of this analysis are reported in Section 6.4.2.

As in the case of FACPL, the PSCEL IDE relies on the Xtext framework. The plug-in accepts an enriched version of the PSCEL language, which contains a few high-level

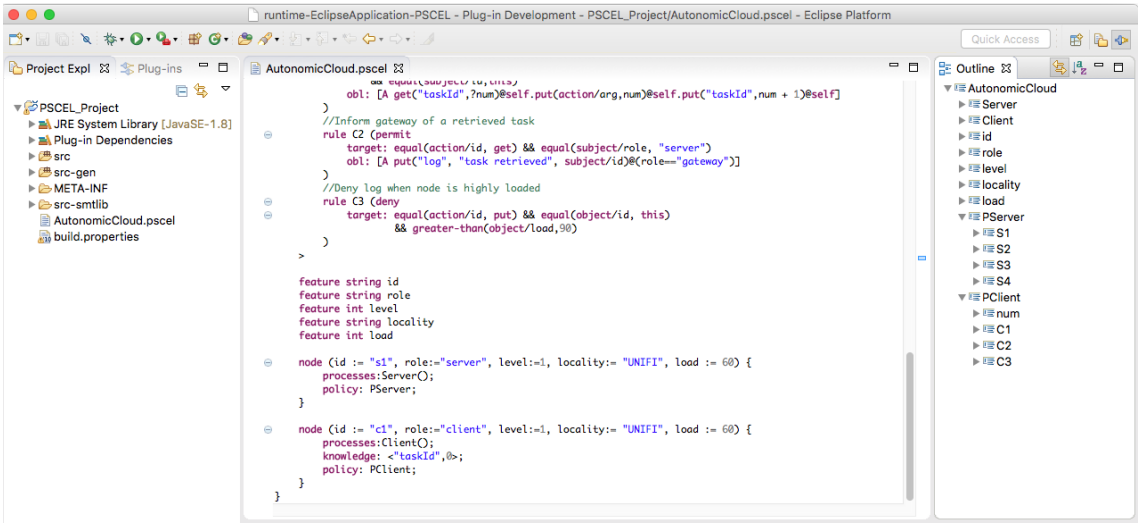


Figure 5.3: PSCEL Eclipse plug-in

features facilitating the coding tasks. Specifically, interface features and variables have a type, processes can contain conditional and loop commands, and specific linguistic handles are provided to define sensors and actuators of components.

The PSCEL development environment is standard and is shown in Figure 5.3. Besides the code generation and the graphical features like, e.g., code highlighting and suggestion, the plug-in defines static type controls. Similar to the FACPL IDE, this IDE offers a customised Java project where PSCEL files, i.e. those with extension *.pscel*, are automatically translated into Java and SMT-LIB files as soon as they are saved. The PSCEL features supporting the coding are available via the classical contextual menus and views of Eclipse.

5.6 Concluding Remarks

In this chapter we have presented the PSCEL language, starting from the design principles to the formal semantics and the supporting tools. Here, we conclude by briefly commenting on the contributions of PSCEL with respect to the research objectives of the thesis and to related and prior publications.

The specification of PSCEL accomplishes the objective **O5**, hence it instantiates SCEL with a tailored version of the FACPL language that, as formalised in Section 5.3 and shown in Section 5.4.2, is capable of regulating interactions among components and enforcing adaptation strategies. From an implementation point of view, the jRESP environment and the PSCEL IDE provide a practical framework to be used for the development of PSCEL-based autonomic systems. This accomplishes the objective **O7**; the details concerning the analysis tools are presented in Chapter 6.

The PSCEL specification approach features multiple powerful and flexible ingredients that, with respect to other approaches from the literature, offer many advantages. For example, differently from the rigid communication functionalities typical of component-based approaches like, e.g., FRACTAL [BCL+06], PSCEL permits defining flexible self-adaptive predicate-based communications. Further details are reported in Section 7.4. In a more general perspective, the main advantage of PSCEL is the fostered separation of concerns: the normal computational behaviour is defined through the processes, while the authorisation and adaptation logic is defined through the policies. This approach permits indeed a clear identification of the context-dependent adaptation strategies.

The interplay among processes and policies is a powerful feature of the language. Specifically, the dynamic fulfilment of obligation actions allows adaptation strategies to retrieve their arguments at runtime. These obligation actions, when injected in a process, are however normal process actions, that is they have to be authorised for execution. This design choice ensures that all actions a system executes adhere to the component policies, but it may generate recursive policy evaluations that pave the way to unforeseen interplays among processes and policies. On the base of the PSCEL semantics, we introduce in Chapter 6 an expressly devised analysis approach that aims at statically pointing out the potential evaluations of policies at runtime and their effects on the progress of systems.

The contents of this chapter are mainly based on the work in [MRNNP16a, MRNNP16b]. A preliminary version of PSCEL is also presented in [MPT13, DLL+15], while its application to the robot-swarm case study is outlined in [LMPT14]. The jRESP framework, besides its exploitation for the robot-swarm case study in [LMPT14], is de-

scribed in [ACH+15].

The development of PSCEL has taken advantage of the formal semantics framework of SCEL [DLPT14]. Besides the integration of the FACPL dialect, in the development of PSCEL we have also refined and fully instantiated different aspects abstracted by SCEL. In details, we give a syntax and a precise semantics to expressions, predicates and interfaces (indeed, the actions **upd** and **read** are not present in SCEL). Most of all, to enforce AOP-inspired adaptation strategies, we modify the semantics of programming constructs by decoupling, for each process action, the authorisation to perform it from its actual execution.

To conclude, we briefly comment on other specification approaches that PSCEL could support to define adaptation strategies. Currently, PSCEL defines adaptation strategies in terms of sequences of obligation actions and such design solution has been proven expressive enough to enforce the adaptation strategies requested by the considered case studies. More in general, we could take into account the enforcement of obligations formed by general processes rather than only sequences of actions. This extension can be easily implemented by acting on the syntax of obligations. The main benefit of this extension is a higher flexibility of process modifications, however appropriate analysis techniques should be provided to statically identify, e.g., possible deadlocks caused by unexpected interactions of the enforced processes.

Chapter 6

Analysis of PSCEL Specifications

Autonomic computing systems, like those expressible in PSCEL, experiment highly dynamic behaviours that are crucially affected by the authorisations and adaptation strategies enforced by system policies. The runtime evaluation of policies, possibly depending on the context, can indeed generate unforeseen behaviours leading to unexpected consequences, e.g. the preclusion of the system progress. Supporting the analysis of system policies is thus crucial to ensure that system behaviours are correctly regulated and adapted.

In this chapter, we address the analysis of the effects of policy evaluations in PSCEL systems. Specifically, since obligation actions are dynamically enforced and their execution requires additional policy evaluations, we aim at statically pointing out the potential policy evaluations and the possible flows of evaluation between rules, called *policy-flow*s. To this aim, we need to reason on policy rules and their applicability to process and obligation actions. By relying on the PSCEL formal semantics, we devise a PSCEL-oriented SMT-based constraint formalism that permits representing policy rules and approximating their applicability to actions. This constraint-based representation of policies permits defining a flow graph, called *Policy-Flow graph*, that represents all the policy-flows and the context dependencies that can take place at runtime. Additionally, we show that the graph can be used for inspecting the effects of policy evaluations on the progress of systems. The graph construction is also supported by practical functionalities of the PSCEL IDE.

Before presenting this analysis approach, we exemplify, by means of the PSCEL system modelling the autonomic cloud case study, a few of the unexpected behaviours generated by the interplay of PSCEL processes and policies.

Structure of the chapter. The rest of this chapter is organised as follows. Section 6.1 outlines the motivations underlying the policy analysis of PSCEL specifications. Section 6.2 describes the PSCEL system modelling the autonomic cloud case study presented in Section 2.3.3. The notion of policy-flow and the constraint-based representation of policies are introduced in Section 6.3. The Policy-Flow graph is defined in Section 6.4, while Section 6.5 exploits the graph to address the verification of progress properties on processes. Section 6.6 concludes with some final remarks.

6.1 Towards the Analysis of PSCEL Specifications

In this section, we first review the main ingredients of the design of PSCEL specifications, then we highlight the type of unexpected behaviours caused by inadequate design choices. Such undesired behaviours motivate our analysis. Notably, in this chapter we take into account the core version of the PSCEL language, i.e. we ignore the extension supporting adaptive policies (see Section 5.4.1).

The design of a PSCEL system involves the specification of three main parts: *processes*, *policy rules*, and *obligations*. Their principled interaction, which is formalised by the PSCEL semantics, provides flexible and powerful specification means. For example, the fact that an action can be defined as process or obligation action permits achieving different design solutions. Namely, choosing a process action means that it is always part of the system behaviour (it can be only denied but not removed), while choosing an obligation action means that it can be part of the system behaviour only when it is needed, e.g. according to context conditions. Generally speaking, the principles at the basis of the PSCEL design approach can be informally described as follows

- *processes*: they contain the actions that must be executed provided that authorisation rules grant them for execution;
- *policy rules*: they decide, possibly on the base of the context, whether to authorise an action and to enforce adaptation strategies;
- *obligations*: they define dynamically fulfilled actions that, if enforced, can be executed.

Therefore, the use of obligation actions is advocated when such actions do not need to be always executed or their execution is subordinated to certain context configurations.

As in the main development approaches for self-adaptive systems (see, e.g., the MAPE-K control loop [KC03] or the EDLC system life cycle [HKP+15]), PSCEL process actions (including obligations) are controlled and authorised before being executed. The additional authorisations of obligations can lead to unforeseen behaviours if the policies are not adequately designed. Indeed, the design of policy rules must consider that rules can apply not only to process actions, but also to the obligation actions possibly injected due to adaptation strategies. This interplay between policies and processes makes the prediction of the overall behaviour of PSCEL systems challenging. It is then worthwhile to devise a static analysis approach supporting the development of PSCEL systems.

Our analysis approach is based on constraints and on a flow graph, called Policy-Flow graph. The aim of the graph is to statically point out the relationships among the different policy rules whose dynamic evaluation can affect the system progress. Specifically, we first

investigate whether a policy may generate an infinite sequence of evaluations (because the obligations injected due to an evaluation recursively trigger further evaluations). Then, we analyse whether the progress of an authorised process action can be precluded (because the authorisation of some injected obligations is denied). Satisfiability of both properties is proved by defining specific conditions on the structure of the Policy-Flow graph. Before presenting this analysis approach, in the next section we further comment on the factors affecting the progress of PSCEL systems.

6.2 PSCEL at work on the Autonomic Cloud Case Study

In this section we employ PSCEL to model the autonomic cloud case study presented in Section 2.3.3. For simplicity's sake, we consider only a group of nodes placed at the locality *UNIFI*. After presenting the PSCEL system, we outline unexpected behaviours that unforeseen interplays between processes and policies can generate (Section 6.2.1).

The considered group of nodes can be rendered as the following PSCEL system

$$\mathcal{I}_1[\mathcal{K}, \Pi_S, P_S] \;\|\; \mathcal{J}_G[\mathcal{K}', \Pi_G, P_G] \;\|\; \mathcal{I}_{C_1}[\mathcal{K}_C, \Pi_C, P_C] \,\|\, \ldots \,\|\, \mathcal{I}_{C_k}[\mathcal{K}_C, \Pi_C, P_C]$$

where each component represents a (virtual) node of the platform. The component with interface $\mathcal{I}_1$ represents the *server* node, the one with interface $\mathcal{J}_G$ represents the *gateway* node. The other components represent *client* nodes. We report the specification of client and server components implementing the behaviours described by the Requirements of Table 2.3. The gateway component is not explicitly described because, in our simplified setting with only one locality, it does not have any duty besides collecting logs from clients.

Each component interface $\mathcal{I}$ contains, besides id, the features role, locality, level and load. In details, role is the type of the represented node, i.e. *server*, *client* or *gateway*; locality is the name of the physical locality where the node is placed, i.e. *UNIFI*; level is the confidentiality level of the node, i.e. *1* and *2* corresponding to low and high confidentiality levels respectively; load is the percentage of load of the hosting machine, i.e. an integer between 0 and 100 that, for simplicity's sake, aggregates both cpu and memory loads. E.g., the server interface is of the form $\mathcal{I}_1 \triangleq (\mathsf{id} : s1, \mathsf{role} : server, \mathsf{level} : 1, \mathsf{locality} : UNIFI, \mathsf{load} : 70)$. Consequently, the execution of, e.g., **upd**$(\mathsf{load}, 75)$ has the effect of updating the feature load to the value 75.

Let us first focus on the (single-threaded) server component $\mathcal{I}_1[\mathcal{K}, \Pi_S, P_S]$, that is the node offering the computational service to the other nodes of the locality. Its process P_S is defined as follows

$$\begin{aligned} P_S \triangleq\; & \mathbf{get}(task, ?\underline{owner}, ?\underline{X}, ?\underline{taskId})@(\mathsf{locality}{=}UNIFI). \\ & (\; X \;\mid\; \mathbf{get}(result, ?\underline{res})@\mathsf{self}.\mathbf{put}(result, \underline{taskId}, \underline{res})@\underline{owner}.\, P_S \;) \end{aligned}$$

The *group-oriented* **get** (non-deterministically) retrieves a task from a component among those that dynamically match the predicate $\mathsf{locality}{=}UNIFI$. A task is any process Q stored in an item of the form $\langle task, n, Q, i\rangle$ (n and i are a name and an integer), which is expected to terminate its execution by locally producing an item of the form $\langle result, v\rangle$. The retrieved task (bound to the process variable $\underline{X}$) is sent for execution by process P_S which then waits for the result via a local **get**. The retrieved result is then sent to the owner of the task through a *point-to-point* **put** and the process proceeds by retrieving a new task. This behaviour corresponds to Requirement (Cl-1).

The server policy Π_S controls P_S ensuring that Requirements (Cl-2) and (Cl-3) are guaranteed. Indeed, tasks are retrieved only from components with the same or lower confidentiality level than the local one, and availability of the computational service is ensured, even though the node is highly loaded. Therefore, when the value of load is greater than 90, the policy forbids the retrieval of new tasks and dynamically creates an additional (virtual) server node. Hence, the contextual information on the load triggers a self-adaptive behaviour of the system. The policy Π_S is defined as follows

$$
\begin{array}{l}
\langle\, \mathsf{p\text{-}unless\text{-}d} \\
\quad \mathsf{rules}: \\
\quad\quad \mathsf{S1}\,(\mathsf{deny}\ \ \mathsf{target}: \mathsf{equal}(\mathsf{action/id}, \mathbf{get}) \wedge\ \mathsf{equal}(\mathsf{subject/id}, \mathsf{this}) \\
\qquad\qquad\qquad \wedge\ \mathsf{pattern\text{-}match}(\mathsf{action/arg}, (task, \_, \_, \_)) \\
\qquad\qquad\qquad \wedge\ \mathsf{equal}(\mathsf{subject/level}, \mathit{1}) \wedge \mathsf{equal}(\mathsf{object/level}, \mathit{2})) \\
\quad\quad \mathsf{S2}\,(\mathsf{deny}\ \ \mathsf{target}: \mathsf{equal}(\mathsf{action/id}, \mathbf{get}) \wedge\ \mathsf{equal}(\mathsf{subject/id}, \mathsf{this}) \\
\qquad\qquad\qquad \wedge\ \mathsf{pattern\text{-}match}(\mathsf{action/arg}, (task, \_, \_, \_)) \qquad\qquad (\Pi_S) \\
\qquad\qquad\qquad \wedge\ \mathsf{greater\text{-}than}(\mathsf{subject/load}, 90) \\
\qquad\qquad \mathsf{obl}: \ [\mathsf{B}\ \ \mathbf{fresh}(n').\, \mathbf{new}(\mathcal{I}_1[\mathsf{id} := n'], \mathcal{K}, \Pi_S, P_S).\mathbf{read}(?\, \underline{load}, \mathsf{load})]) \\
\quad\quad \mathsf{S3}\,(\mathsf{deny}\ \ \mathtt{target}: \mathtt{equal}(\mathsf{action/id}, \mathbf{read})\ \wedge\ \mathsf{pattern\text{-}match}(\mathsf{action/arg}, (\_, \mathsf{load}))) \\
\qquad\qquad\qquad \wedge\ \mathsf{greater\text{-}than}(\mathsf{subject/load}, 60) \\
\quad\quad \mathsf{S4}\,(\mathsf{permit}\ \ \mathtt{target}: \mathtt{equal}(\mathsf{action/id}, \mathbf{put})\ \wedge\ \mathsf{equal}(\mathsf{subject/id}, \mathsf{this}) \\
\qquad\qquad \mathsf{obl}: \ [\mathsf{A}\ \ \mathbf{put}(log, \mathsf{action/arg})@\mathsf{self}])\, \rangle
\end{array}
$$

where, for simplicity's sake, we omit the keyword obl : whenever the sequence of obligations is empty. The policy is made of four rules and uses the p-unless-d algorithm. Rules S1 and S2 manage the action **get** for retrieving a new task, that is the action executed by the local process (as checked by equal(subject/id, this)) having as argument a specific template (as checked by pattern-match(action/arg,$(task, \_, \_, \_)$)). In details, rule S1 forbids the action **get** (i.e. the rule decision is deny) if the confidentiality level of the local component (i.e. the subject of the action) is low and that of the component from where the task should be retrieved (i.e. the object of the action) is high. Rule S2 enforces the adaptation strategy, thus it forbids the action **get** when the local load is greater than 90 and spawns a new server component, which only differs from the creating one for its name, via the action $\mathbf{new}(\mathcal{I}_1[\mathsf{id} := n'], \mathcal{K}, \Pi_S, P_S)$ ($\mathcal{I}_1[\mathsf{id} := n']$ denotes the interface obtained from $\mathcal{I}_1$ by initialising id with n'). Afterwards, the local process is blocked until the load is higher than 60, this is due to the combination of the last returned action **read** and rule S3. Finally, rule S4 records in a log, via an additional **put**, the action **put** that sends the result of the task to the owner. Notably, S4 is only needed to introduce the obligation, as actions **put** are authorised by the combining algorithm. Indeed, p-unless-d authorises any request when no specific rule applies. Additional controls on, e.g., the spawned component or task executions, are out of scope here.

Remark 6.1 (Variation of Load)**.** *According to the autonomic computing paradigm, the feature* load *is modified by a sensor attached to the component when significant variations of the load occur. The behaviour of such a sensor can be modelled as a sequence of* ***upd***(load, v) *actions updating the value of the feature. For simplicity's sake, process P_S does not include this behaviour.*

Let us now focus on the client components $\mathcal{I}_{C_k}[\mathcal{K}_C, \Pi_C, P_C]$. We expect that process P_C, besides some other actions, performs actions of the form $\mathbf{put}(task, loc\_res, Q)@\mathsf{self}$

that locally add an item containing a new task Q to execute, together with a component name loc_res where the result of the task execution should be sent. According to Requirement (Cl-4), before being executed, these tasks are incrementally numbered. This behaviour, together with those corresponding to Requirements (Cl-5) and (Cl-6), is enforced by the policy Π_C defined as follows

$$
\begin{array}{l}
\langle\, \mathsf{p\text{-}unless\text{-}d} \\
\quad \mathsf{rules:} \\
\quad\quad \mathsf{C1\,(permit\ \ target:\ equal(action/id,}\,\mathbf{put}) \wedge\ \mathsf{pattern\text{-}match(action/arg,}(task, \_, \_)) \\
\quad\quad\quad\quad\quad \wedge\ \mathsf{equal(subject/id,this)} \\
\quad\quad\quad\quad \mathsf{obl:}\ [\mathsf{A}\ \ \mathbf{get}(taskId, ?\underline{num})@\mathsf{self}.\mathbf{put}(\mathsf{action/arg}, \underline{num})@\mathsf{self}. \\
\quad\quad\quad\quad\quad\quad \mathbf{put}(taskId, \underline{num} + 1)@\mathsf{self}]) \\
\quad\quad \mathsf{C2\,(permit\ \ target:\ equal(action/id,}\mathbf{get}) \wedge\ \mathsf{equal(subject/role,}server) \\
\quad\quad\quad\quad \mathsf{obl:}\ [\mathsf{A}\ \ \mathbf{put}(log, task\ retrieved, \mathsf{subject/id})@(\mathsf{role} = gateway)]) \\
\quad\quad \mathsf{C3\,(deny\ \ target:\ equal(action/id,}\,\mathbf{put}) \wedge \mathsf{equal(object/id,this)} \\
\quad\quad\quad\quad\quad \wedge\ \mathsf{greater\text{-}than(object/load,}90))\,\rangle
\end{array}
\qquad (\Pi_C)
$$

Rule C1 applies to **put** actions of process P_C locally adding a new task, i.e. with subject the local component and as argument an item formed by $task$ and two elements. The rule thus accomplishes the incremental enumeration of tasks by means of three additional actions: action **get** retrieves the current task number, the first **put** locally adds the argument of the authorised **put** appropriately extended with the retrieved number, and the second **put** increments the task number[1]. Rule C2 injects a **put** action informing the gateway about the retrieval of a task by a **get** action originated by a server, while rule C3 forbids any **put** action with object the local component when the local load is higher that 90. These last two rules correspond, respectively, to Requirements (Cl-6) and (Cl-5).

6.2.1 Interplay between Policies and Processes

We now focus on unforeseen interplays between policies and processes. Indeed, the obligation actions injected in a process, like any other process action, need to be authorised before being executed. On the other hand, policy rules are initially designed to apply to certain process actions. Thus, if afterwards rules also apply to injected actions, as effect of policy evaluation they can possibly prevent processes from actually proceed.

For example, let us consider process P_S and its controlling policy Π_S. Actions **get** retrieving new tasks can be either authorised or not; moreover, application of rule S2 causes the injection of additional actions. Actions **fresh** and **new** are authorised due to the default decision of the algorithm, while actions **read** are authorised only when rule S3 does not apply, otherwise they are forbidden without injecting any obligation. In case of actions **put**, rule S4 also injects a new action **put** for logging purposes. As result of such an authorisation, process P_S dynamically evolves as follows

$$\overline{\mathbf{put}(result, 12, 5)@n'}.\mathbf{put}(log, result, 12, 5)@\mathsf{self}.\, P_S$$

where, bound variables occurring in the process syntax are replaced by realistic values (i.e., $\underline{owner}$, $\underline{taskId}$ and $\underline{res}$ have been replaced by n', 12 and 5, respectively). Notably, the obligation action has been fulfilled by replacing the structured name action/arg with

[1] We assume that at the outset the repository $\mathcal{K}_C$ contains the item $\langle taskId, 0\rangle$.

the argument of the authorised **put** (i.e., the overlined action). Once the authorised **put** has been executed, the authorisation of the injected **put** modifies the process as follows

$$\overline{\mathbf{put}(log, result, 12, 5)@s1}.\mathbf{put}(log, log, result, 12, 5)@\mathsf{self}.\, P_S$$

Basically, rule S4 has been applied again and injected an additional logging **put**! Clearly, this leads to an infinite introduction of actions, and hence of policy evaluations, which prevents process P_S from proceeding further.

A similar interplay might occur between P_C and Π_C. Indeed, local **put** actions adding new tasks trigger the evaluation of rule C1 that, consequently, injects a **get**. This action, on its own turn, can trigger rule C2 that injects a new **put** action. This might potentially generate a cyclic dependency. However, by reasoning in details on targets, we can easily see that C2 cannot trigger C1. In fact, the **put** injected by C2 has as argument an item with *log* as first element, while the pattern-match control of C1 requires *task* as first element. Similar situations can occur due to conflicting controls on features defined by the involved rules.

A different interplay can concern the type of authorisations enforced by rules. In particular, when an action is positively authorised, the action itself or its continuation can be precluded from progressing due to an injected action that is forbidden. This interplay can occur between rules C1 and C3: when rule C1 positively authorises an action **put**, the injected **put** action could be denied by rule C3, i.e. whenever the load of the destination component is higher than 90.

6.3 Policy-Flow: Definition and Constraint-based Analysis

The interplays presented in the previous section are due to unforeseen policy evaluations caused by dynamically injected obligation actions. Indeed, the injection of actions can trigger additional evaluations of rules and, thus, generate at runtime a sort of flow between policy rules, that we call *policy-flow*. To statically over-approximate all the potential flows, we now start introducing an analysis approach based on a constraint-based representation of policies that enables extensive (automated) checks on the applicability of obligation actions to policy rules.

Due to the static nature of our approach, the injected actions a to consider are those produced by the syntax. Hence, they may contain *open terms*, i.e. terms where variables and structured names can occur. To make these actions evaluable through $\mathcal{A}$, we must apply to them a 'closing' *substitution*, denoted by ξ, i.e. a function mapping their variables and structured names to values.

As a matter of notation, we write $\mathsf{Interf}(S)$ to denote the set of component interfaces in a system S, $\mathcal{I}.\mathsf{id}$ to refer to the name of the component having interface $\mathcal{I}$, and $\Pi(S, m)$ to make it explicit that the policy Π is in force at the component named m in S. Hence, we set the following definition of policy-flow.

Definition 6.1 (Policy-Flow)**.** *Given a system S with $\mathcal{I}, \mathcal{J} \in \mathsf{Interf}(S)$, there is a* flow *from rule ρ_i to rule ρ_j in the policy $\Pi(S, \mathcal{I}.\mathsf{id})$ if, for any request req, it holds that*

$$\mathcal{P}[\![\rho_i]\!]req = d, s_{\mathtt{B}}, s_{\mathtt{A}} \quad \mathsf{and} \quad \exists\, a \sqsubseteq s_{\mathtt{B}}.s_{\mathtt{A}}\ ,\ \xi\ :\ \mathcal{P}[\![\rho_j]\!]\mathsf{req}(\mathcal{I}, \mathcal{A}(a\xi, \mathcal{I}.\mathsf{id}), \mathcal{J}) = d', s'_{\mathtt{B}}, s'_{\mathtt{A}}$$

where $a \sqsubseteq s_{\mathtt{B}}.s_{\mathtt{A}}$ means that action a occurs in the sequence of actions $s_{\mathtt{B}}.s_{\mathtt{A}}$.

Constraints $cstr ::= \mathsf{true} \mid \neg\, cstr \mid cstr_1 \wedge cstr_2 \mid cstr_1 \vee cstr_2 \mid var = pv \mid var > pv$	
$\mid var < pv \mid var\ \mathsf{match}\ pv$	$var \in A_\Pi \cup F_S$

Table 6.1: Constraint Syntax (pv stands for the POLICY VALUES of Table 5.2)

The flows in a policy can be statically determined by checking whether the authorisation requests corresponding to obligation actions match rule targets. To this aim, we represent targets in terms of constraints and authorisation requests in terms of assignments for constraints. Existence of a flow is thus equivalent to satisfiability of a constraint with applied an assignment.

In the following, first we introduce a constraint formalism and the representation of obligation actions it enables (Section 6.3.1), then we formalise a translation procedure from rule targets to constraints (Section 6.3.2). Both the representation and the translation are proved to be in agreement with the PSCEL semantics.

6.3.1 A PSCEL-Oriented Constraint Formalism: Syntax and Exploitation

Constraints are written according to the grammar shown in Table 6.1. A constraint can be either the value true, or a comparison between a variable var and a policy value pv through a relational operator, or a boolean combination of simpler constraints. Policy values are the values referred to by the attribute names of authorisation requests like, e.g., the action identifier **get** or an item argument of an action. Variables model the structured names sn occurring within rule targets and can either belong to the set A_Π or to the set F_S.

The set A_Π, given a policy $\Pi \triangleq \langle alg\ \mathsf{rules}:\ \rho_1 \ldots \rho_k \rangle$, is defined as follows

$$A_\Pi \triangleq \{id\text{-}h, arg\text{-}h, sub\text{-}h, obj\text{-}h \mid h \in \{\mathsf{id}(\rho_1) \ldots \mathsf{id}(\rho_k)\}\}$$

where $\mathsf{id}(\rho_j)$ stands for the name of the rule ρ_j. Variables in A_Π model: (i) action identifiers referred to by $\mathsf{action/id}$; (ii) action arguments referred to by $\mathsf{action/arg}$; (iii) the name of the subject (resp., object) component referred to by $\mathsf{subject/id}$ (resp., $\mathsf{object/id}$). Due to the definition of function req (see Section 5.3.2) and the syntax of obligation actions, the domains of values that the variables in A_Π can assume are

$$dom(id\text{-}h) = \{\textbf{put}, \textbf{get}, \textbf{qry}, \textbf{new}, \textbf{fresh}, \textbf{upd}, \textbf{read}\}$$
$$dom(arg\text{-}h) = ExtendedPolicyTuples \qquad dom(sub\text{-}h) = dom(obj\text{-}h) = Destinations$$

where $ExtendedPolicyTuples$ is the set of all POLICY TUPLES where open terms can occur. Notationally, we use cv (which stands for *constraint value*) to indicate an element of any of the domains of values that the variables in A_Π can assume.

The set F_S includes variables modelling the features of the components in S. Given a system S, the set F_S is defined as follows

$$F_S \triangleq \{\mathsf{z}\text{-}n \mid \exists\, \mathcal{I} \in \mathsf{Interf}(S) : \mathcal{I}.\mathsf{id} = n \wedge (\mathsf{z} : e) \in \mathcal{I} \wedge \mathsf{z} \neq \mathsf{id}\}$$

thus, a variable $\mathsf{z}\text{-}n$ models the feature with name z of the component (whose feature id has value) n. By definition, features are associated to closed expressions, thus the domain of the variables in F_S is the set of values, i.e. $Values$.

Our constraint formalism can represent targets, however, to enable (automated) reasoning on obligation actions, we need to represent authorisation requests (corresponding

to obligation actions) as assignments for constraint variables. To this aim, we use the function $\langle\!\langle \cdot \rangle\!\rangle_m^h$ which, given in input an obligation action, the name m of a component (i.e. the intended subject) and the name h of a policy rule, returns the assignments (induced by the obligation action) for the variables of A_Π corresponding to rule h. The function is defined by case analysis on the syntax of obligation actions as follows

$$
\begin{array}{l}
\langle\!\langle \mathbf{get}(T)@c \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{get}, arg\text{-}h := (\!|T[m/\mathsf{self}]|\!), sub\text{-}h := m, obj\text{-}h := \langle\!\langle c \rangle\!\rangle_m^h] \\
\langle\!\langle \mathbf{put}(t)@c \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{put}, arg\text{-}h := (\!|t[m/\mathsf{self}]|\!), sub\text{-}h := m, obj\text{-}h := \langle\!\langle c \rangle\!\rangle_m^h] \\
\langle\!\langle \mathbf{qry}(T)@c \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{qry}, arg\text{-}h := (\!|T[m/\mathsf{self}]|\!), sub\text{-}h := m, obj\text{-}h := \langle\!\langle c \rangle\!\rangle_m^h] \\
\langle\!\langle \mathbf{fresh}(n) \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{fresh}, sub\text{-}h := m, obj\text{-}h := m] \\
\langle\!\langle \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{new}, sub\text{-}h := m, obj\text{-}h := m] \\
\langle\!\langle \mathbf{upd}(\mathsf{n}, e) \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{upd}, arg\text{-}h := (\!|\mathsf{n}, e[m/\mathsf{self}]|\!), sub\text{-}h := m, obj\text{-}h := m] \\
\langle\!\langle \mathbf{read}(?\,\underline{x}, \mathsf{n}) \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{read}, arg\text{-}h := (\!|?\,\underline{x}, \mathsf{n}|\!), sub\text{-}h := m, obj\text{-}h := m] \\
\\
\langle\!\langle \mathsf{self} \rangle\!\rangle_m^h = m \qquad \langle\!\langle n \rangle\!\rangle_m^h = n \qquad \langle\!\langle \underline{x} \rangle\!\rangle_m^h = \underline{x} \qquad \langle\!\langle sn \rangle\!\rangle_m^h = sn \qquad \langle\!\langle \mathcal{P} \rangle\!\rangle_m^h = \underline{x}
\end{array}
\tag{A-1}
$$

The action identifier is assigned to the variable $id\text{-}h$, while the name m of the subject component is assigned to $sub\text{-}h$ and that of the action destination $\langle\!\langle c \rangle\!\rangle_m^h$ is assigned to $obj\text{-}h$. Notably, if c is a predicate $\mathcal{P}$, it is mapped to some variable $\underline{x}$ (to which any name can be assigned), because the components satisfying the predicate cannot be statically determined. Instead, the policy tuple corresponding to the action arguments, obtained by applying the function $(\!| \cdot |\!)$ introduced in Section 5.3.2, is assigned to $arg\text{-}h$. For dealing with open terms, $(\!| \cdot |\!)$ is extended as follows

$$(\!|e|\!) = e \qquad (\!|\underline{x}|\!) = \underline{x} \qquad (\!|sn|\!) = sn \qquad (\!|\underline{X}|\!) = \_$$

Thus, $(\!| \cdot |\!)$ acts as an identity function for generic expressions e, value variables $\underline{x}$ and structured names sn, while it abstracts from process variables $\underline{X}$ that are replaced by the wildcard '_'. For example, given the action $a \triangleq \mathbf{put}(taskId, \underline{num} + 1)@\mathsf{self}$, we have $\langle\!\langle a \rangle\!\rangle_m^h = [id\text{-}h := \mathbf{put}, arg\text{-}h := (taskId, \underline{num} + 1), sub\text{-}h := m, obj\text{-}h := m]$.

Since constraint variables are possibly mapped to elements containing open terms, checking the satisfiability of a constraint means deciding if there exists a substitution ξ mapping variables and structured names to values such that the constraint evaluates to true. For instance, checking the satisfiability of a constraint with applied the previous assignment means identifying a substitution ξ for the variable $\underline{num}$ such that the constraint is satisfied. We write $\xi \models cstr\langle\!\langle a \rangle\!\rangle_m^h$ to mean that the constraint $cstr$, under the assignment $\langle\!\langle a \rangle\!\rangle_m^h$ induced by the obligation action a, is satisfiable through the substitution ξ.

Finally, we conclude by showing that any obligation action possibly executed at runtime can be statically approximated starting from its syntactical definition.

Lemma 6.1. *For any obligation action a such that $\mathcal{A}(a\xi', m) = \mathbf{a}$ for some substitution ξ', it holds that*

$$\exists\, \xi \;:\; \langle\!\langle \mathbf{a} \rangle\!\rangle_m^h = [\![\, \langle\!\langle a \rangle\!\rangle_m^h \xi \,]\!]$$

where $[\![\, \langle\!\langle a \rangle\!\rangle_m^h \xi \,]\!]$ denotes the assignment obtained from $\langle\!\langle a \rangle\!\rangle_m^h \xi$ by evaluating all the expressions occurring within.

Proof. From the definition of $\langle\!\langle \cdot \rangle\!\rangle_m^h$, it follows that

$$
\begin{array}{rcl}
\langle\!\langle \mathbf{a} \rangle\!\rangle_m^h & = & [id\text{-}h := cv_1, arg\text{-}h := cv_2, sub\text{-}h := cv_3, obj\text{-}h := cv_4] \\
\langle\!\langle a \rangle\!\rangle_m^h & = & [id\text{-}h := cv_1', arg\text{-}h := cv_2', sub\text{-}h := cv_3', obj\text{-}h := cv_4']
\end{array}
$$

where cv_i identifies an element of the corresponding variable domain. Thus, the statement amounts to prove that

$$\exists\, \xi \;:\; \forall i \in \{1,2,3,4\} \;\; cv_i = [\![\, cv'_i\, \xi \,]\!]$$

From (A-1), it follows that $cv_1 = cv'_1$ and $cv_3 = cv'_3$; hence, we only need to consider the remaining cases.

$(cv_2 = cv'_2)$ From (A-1) we have that cv_2 and cv'_2 are sequences of elements of same length that differ for the possible occurrence of open terms $\underline{x}$ and sn in cv'_2 (variables $\underline{X}$ cannot occur). Hence, we proceed by induction on the length of the sequences and by only analysing the different pairs.

Base Case $v = (e_1\ \mathtt{op})^*\ e\ (\mathtt{op}\ e_2)^*$ with $e \in \{\underline{x}, sn\}$. As op indicates boolean and linear arithmetic operators, it is always possible to compute a value v' such that $[\![\, (e_1\ \mathtt{op})^*\ v'\ (\mathtt{op}\ e_2)^* \,]\!] = v$. Hence, with $\xi \triangleq [v'/e]$ the thesis follows.

Inductive Case $cv_2, v = cv'_2, (e_1\ \mathtt{op})^*\ e\ (\mathtt{op}\ e_2)^*$ with $e \in \{\underline{x}, sn\}$. From inductive hypothesis we have that $\exists \xi' \;:\; cv_2 = cv'_2\xi'$. As we can compute a value v' such that $[\![\, (e_1\ \mathtt{op})^*\ v'\ (\mathtt{op}\ e_2)^* \,]\!] = v$, with $\xi \triangleq [v'/e] \uplus \xi'$ the thesis follows.

$(cv_4 = cv'_4)$ We proceed by case analysis on the possible different pairs.

$(n = \underline{x})$ The thesis immediately follows with $\xi \triangleq [n/\underline{x}]$ for some $n \in \mathcal{N}$.

$(n = sn)$ The thesis immediately follows with $\xi \triangleq [n/sn]$ for some $n \in \mathcal{N}$.

□

6.3.2 From PSCEL Policies To Constraints

To represent targets in terms of constraints, we define a formal translation procedure which, intuitively, works in two steps. First, we approximate the potential subject and object components of those actions matching the target. Then, we exploit them to translate targets into their corresponding constraints.

Step (1 of 2): Potential Subject and Object Components. The components possibly involved in the actions matching the target of a rule can be over-approximated by inspecting the controls occurring in the rule target. In fact, controls concerning component names, i.e. subject/id and object/id, or features, e.g. subject/level, statically limit the components that can be represented by those authorisation requests that match the target. The sets of the potential subject and object components of the actions matching the target, $\mathcal{S}$ and $\mathcal{O}$ resp., are determined by functions Sbj and Obj, resp. The function Sbj is defined inductively on the syntax of targets by the clauses in Table 6.2. The definition of Obj is similar, hence it is omitted. It only differs from that of Sbj because it swaps the categories subject and object.

The function Sbj, given in input a target τ and a system S, returns a set of component names. Without loss of generality, we only consider targets belonging to policies like $\Pi(S, m)$, that is where the reserved variable this does not occur, and where the operator $\neg$ is only applied to relational functions or the value true[2]. Let us now briefly comment the clauses. A target true matches all requests, thus the set $\mathcal{S}$ includes the names of all the

[2]This follows since, when a target is evaluated, each occurrence of this has been replaced by a component name and since, by means of standard boolean laws, the syntax of the target expression can be manipulated so that it is of the required form.

$$Sbj_S(\mathsf{true}) = \{\mathcal{I}.\mathsf{id} \mid \mathcal{I} \in \mathsf{Interf}(S)\} \qquad Sbj_S(\neg\,\mathsf{true}) = \{\}$$

$$Sbj_S(\tau_1 \wedge \tau_2) = Sbj_S(\tau_1) \cap Sbj_S(\tau_2) \qquad Sbj_S(\tau_1 \vee \tau_2) = Sbj_S(\tau_1) \cup Sbj_S(\tau_2)$$

$$Sbj_S(f(\mathsf{subject/id}, pv)) = \begin{cases} \{n\} & \text{if } [\![\, pv \,]\!] = n \wedge f \in \{\mathsf{equal}, \mathsf{pattern\text{-}match}\} \\ \{\} & \text{otherwise} \end{cases}$$

$$Sbj_S(\neg\, f(\mathsf{subject/id}, pv)) = \begin{cases} \{\mathcal{I}.\mathsf{id} \mid \mathcal{I} \in \mathsf{Interf}(S)\} \setminus \{n\} & \text{if } [\![\, pv \,]\!] = n \wedge f \in \{\mathsf{equal}, \mathsf{pattern\text{-}match}\} \\ \{\} & \text{otherwise} \end{cases}$$

$$Sbj_S(f(\mathsf{subject/z}, pv)) = Sbj_S(\neg\, f(\mathsf{subject/z}, pv)) = \{\mathcal{I}.\mathsf{id} \mid \mathcal{I} \in \mathsf{Interf}(S) \wedge (\mathsf{z} : e) \in \mathcal{I}\} \quad \text{with } \mathsf{z} \neq \mathsf{id}$$

$$Sbj_S(f(\mathsf{object/z}, pv)) = Sbj_S(\neg\, f(\mathsf{object/z}, pv)) = \{\mathcal{I}.\mathsf{id} \mid \mathcal{I} \in \mathsf{Interf}(S)\}$$

$$Sbj_S(f(\mathsf{action/z}, pv)) = Sbj_S(\neg\, f(\mathsf{action/z}, pv)) = \{\mathcal{I}.\mathsf{id} \mid \mathcal{I} \in \mathsf{Interf}(S)\}$$

Table 6.2: The function Sbj determining the set of the potential executing components

components of S. Conversely, a target $\neg$ true does not match any request, hence the set $\mathcal{S}$ is empty. The set resulting from a conjunction (resp., disjunction) of targets corresponds to the intersection (resp., union) of the sets calculated for the sub-targets. In case of relational functions, the resulting set $\mathcal{S}$ mainly depends on the occurring structured name sn. Let sn be subject/id. If f is an equality function and the policy value pv evaluates to a name n, $\mathcal{S}$ only contains such a name. Therefore, when the operator $\neg$ is applied to f, $\mathcal{S}$ contains all names except n. If f is not an equality function or the policy value pv does not evaluate to a name n, the target cannot match any request, hence $\mathcal{S}$ is empty. Let sn be subject/z with z $\neq$ id. The set $\mathcal{S}$ includes the names of those components exposing such a feature. Finally, when sn has category action or object, all component names are included in $\mathcal{S}$, because no additional information to restrict the potential subject components can be deduced.

Step (2 of 2): Generating the Constraints. The formal translation of targets into constraints is given by the function $\mathcal{T}$ inductively defined on the syntax of targets by the clauses in Table 6.3. The emphatic brackets $\{\!|$ and $|\!\}$ enclose the target τ to translate; in addition to it, the function takes in input the sets $\mathcal{S}$ and $\mathcal{O}$ calculated from τ and the name h of a policy rule. Thus, the translation of the target true corresponds to the value true itself, while that of a composed target is given compositionally using the corresponding constraint operators. The translation of function $f(sn, pv)$ exploits the sets $\mathcal{S}$ and $\mathcal{O}$. Indeed, if the structured name sn is a subject/object feature different from id, the generated constraint has as many variables z-n representing the feature z as the component names n in the set $\mathcal{S}/\mathcal{O}$. The disjunction ensures that the constraint addresses each possibly involved component. Instead, if the structured name represents the feature id or an attribute with category action, the generated constraint uses the variables referring to the rule named h. Notably, f is mapped to the corresponding constraint operator by the

$$\mathcal{T}\{\!|\mathsf{true}|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathsf{true} \qquad \mathcal{T}\{\!|\neg\tau|\!\}^h_{\mathcal{S},\mathcal{O}} = \neg\,\mathcal{T}\{\!|\tau|\!\}^h_{\mathcal{S},\mathcal{O}}$$

$$\mathcal{T}\{\!|\tau_1 \wedge \tau_2|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathcal{T}\{\!|\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}} \wedge \mathcal{T}\{\!|\tau_2|\!\}^h_{\mathcal{S},\mathcal{O}} \qquad \mathcal{T}\{\!|\tau_1 \vee \tau_2|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathcal{T}\{\!|\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}} \vee \mathcal{T}\{\!|\tau_2|\!\}^h_{\mathcal{S},\mathcal{O}}$$

$$\mathcal{T}\{\!|f(sn,pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = \begin{cases} \bigvee_{n\in\mathcal{S}} \mathsf{z}\text{-}n\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{subject/z \text{ and } z} \neq \mathsf{id} \\ \bigvee_{n\in\mathcal{O}} \mathsf{z}\text{-}n\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{object/z \text{ and } z} \neq \mathsf{id} \\ sub\text{-}h\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{subject/id} \\ obj\text{-}h\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{object/id} \\ id\text{-}h\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{action/id} \\ arg\text{-}h\ \mathsf{getOp}(f)\ pv & \text{if } sn = \mathsf{action/arg} \end{cases}$$

Table 6.3: Translation function for rule targets

function getOp defined as follows

$$\mathsf{getOp}(\mathsf{equal}) = = \qquad \mathsf{getOp}(\mathsf{greater\text{-}than}) = >$$
$$\mathsf{getOp}(\mathsf{less\text{-}than}) = < \qquad \mathsf{getOp}(\mathsf{pattern\text{-}match}) = \mathsf{match}$$

This means that the semantics of relational functions on targets is assumed to coincide with that of the corresponding constraint operator.

To conclude, we prove that the satisfiability of the constraints representing targets under the assignments induced by (authorisation requests corresponding to) obligation actions correctly over-approximates the set of policy flows (Corollary 6.4). This follows from the main result proving that the satisfiability of the constraint-based representation of targets over-approximate target applicability (Theorem 6.3). Before, we show that $\mathcal{S}$ and $\mathcal{O}$ correctly approximate the sets of potential subject and object components of those actions matching the rule target (Lemma 6.2).

Lemma 6.2. *Given a system S with $\mathcal{I},\mathcal{J} \in \mathsf{Interf}(S)$, for any rule of the policy $\Pi(S,\mathcal{I}.\mathsf{id})$ with target τ, and for any obligation action a such that $\mathcal{A}(a\xi',\mathcal{I}.\mathsf{id}) = \mathbf{a}$ for some substitution ξ', it holds that*

$$\mathcal{E}[\![\tau]\!]\mathsf{req}(\mathcal{I},\mathbf{a},\mathcal{J}) = \mathsf{true} \quad \Rightarrow \quad \mathcal{I}.\mathsf{id} \in \mathit{Sbj}_S(\tau) \ \textit{and}\ \ \mathcal{J}.\mathsf{id} \in \mathit{Obj}_S(\tau)\,.$$

Proof. We only prove that $\mathcal{I}.\mathsf{id} \in \mathit{Sbj}_S(\tau)$, as the proof that $\mathcal{J}.\mathsf{id} \in \mathit{Obj}_S(\tau)$ proceeds similarly. We let $req_{\mathbf{a}} \triangleq \mathsf{req}(\mathcal{I},\mathbf{a},\mathcal{J})$ and proceed by structural induction on the syntax of targets where, without loss of generality, we assume that the operator $\neg$ is only applied to relational functions or the value true.

($\tau = \mathsf{true}$) We have $\mathcal{S} \triangleq \mathit{Sbj}_S(\mathsf{true}) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\}$ hence we have $\mathcal{I}.\mathsf{id} \in \mathcal{S}$;

($\tau = \neg\,\mathsf{true}$) As the target cannot match any request, i.e. $\nexists\, \mathcal{I} : \mathcal{E}[\![\tau]\!]\mathsf{req}(\mathcal{I},\mathbf{a},\mathcal{J}) = \mathsf{true}$, the thesis immediately holds.

($\tau = f(sn,pv)$) We proceed by case analysis on sn.

($sn = \mathsf{subject/id}$) Let us suppose $f \in \{\mathsf{equal},\mathsf{pattern\text{-}match}\}$. As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $\mathcal{I}.\mathsf{id} = n = [\![pv]\!]$ for some name $n \in \mathcal{N}$. Hence, we have $\mathcal{I}.\mathsf{id} \in \mathcal{S} = \mathit{Sbj}_S(f(\mathsf{subject/id},pv)) = \{[\![\, pv \,]\!]\}$. Let us suppose $[\![\, pv \,]\!] \notin \mathcal{N}$ or $f \notin \{\mathsf{equal},\mathsf{pattern\text{-}match}\}$. As the target cannot match any request, the thesis immediately holds.

($sn = \mathsf{subject/z}$ **with** $\mathsf{z} \neq \mathsf{id}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $(\mathsf{z} : e) \in \mathcal{I} \wedge [\![e]\!] = v$ for some value v. Hence, as $\mathsf{z} \neq \mathsf{id}$, we have $\mathcal{I}.\mathsf{id} \in \mathcal{S} = Sbj_S(f(\mathsf{subject/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S) \wedge (\mathsf{z} : e') \in \mathcal{H}\}$.

($sn = \mathsf{action/z}$) We have $\mathcal{S} \triangleq Sbj_S(f(\mathsf{action/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\}$ hence we certainly have $\mathcal{I}.\mathsf{id} \in \mathcal{S}$;

($sn = \mathsf{object/z}$) We have $\mathcal{S} \triangleq Sbj_S(f(\mathsf{object/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\}$ hence we certainly have $\mathcal{I}.\mathsf{id} \in \mathcal{S}$;

($\tau = \neg\, f(sn, pv)$) We proceed by case analysis on sn.

($sn = \mathsf{subject/id}$) Let us suppose $f \in \{\mathsf{equal}, \mathsf{pattern\text{-}match}\}$. As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $\mathcal{I}.\mathsf{id} \neq [\![pv]\!]$ and $[\![pv]\!] \in \mathcal{N}$. Hence, we have $\mathcal{I}.\mathsf{id} \in \mathcal{S} = Sbj_S(\neg\, \mathsf{equal}(\mathsf{subject/id}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\} \setminus \{[\![pv]\!]\}$. Let us suppose $[\![pv]\!] \notin \mathcal{N}$ or $f \notin \{\mathsf{equal}, \mathsf{pattern\text{-}match}\}$. As the target cannot match any request, the thesis immediately holds.

($sn = \mathsf{subject/z}$ **with** $\mathsf{z} \neq \mathsf{id}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $(\mathsf{z} : e) \in \mathcal{I} \wedge [\![e]\!] = v$ for some value v. Hence, as $\mathsf{z} \neq \mathsf{id}$, we have $\mathcal{I}.\mathsf{id} \in \mathcal{S} = Sbj_S(\neg\, f(\mathsf{subject/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S) \wedge (\mathsf{z} : e') \in \mathcal{H}\}$.

($sn = \mathsf{action/z}$) We have $\mathcal{S} \triangleq Sbj_S(\neg\, f(\mathsf{action/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\}$ hence we certainly have $\mathcal{I}.\mathsf{id} \in \mathcal{S}$;

($sn = \mathsf{object/z}$) We have $\mathcal{S} \triangleq Sbj_S(\neg\, f(\mathsf{object/z}, pv)) = \{\mathcal{H}.\mathsf{id} \mid \mathcal{H} \in \mathsf{Interf}(S)\}$ hence we certainly have $\mathcal{I}.\mathsf{id} \in \mathcal{S}$;

($\tau = \tau_1 \wedge \tau_2$) We have $\mathcal{S} = Sbj_S(\tau_1 \wedge \tau_2) = Sbj_S(\tau_1) \cap Sbj_S(\tau_2)$; the thesis immediately follows by structural induction.

($\tau = \tau_1 \vee \tau_2$) We have $\mathcal{S} = Sbj_S(\tau_1 \wedge \tau_2) = Sbj_S(\tau_1) \cup Sbj_S(\tau_2)$; the thesis immediately follows by structural induction.

□

Theorem 6.3. *Given a system S with $\mathcal{I}, \mathcal{J} \in \mathsf{Interf}(S)$, for any rule ρ of the policy $\Pi(S, \mathcal{I}.\mathsf{id})$ with $\mathsf{id}(\rho) = h$ and target τ, and for any obligation action a such that $\mathcal{A}(a\xi', \mathcal{I}.\mathsf{id}) = \mathbf{a}$ for some substitution ξ', it holds that:*

$$\mathcal{E}[\![\tau]\!]\mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J}) = \mathsf{true} \quad \Rightarrow \quad \exists\ \xi\ :\ \xi \models \mathcal{T}\{\!|\tau|\!\}^h_{\mathcal{S},\mathcal{O}}\langle\!\langle a \rangle\!\rangle^h_m\,.$$

Proof. We let $req_{\mathbf{a}} \triangleq \mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J})$ and proceed by structural induction on the syntax of targets; the rules referred to throughout the proof are those of Table 5.4.

($\tau = \mathsf{true}$) From the definition of $\mathcal{E}$, we have $\mathcal{E}[\![\mathsf{true}]\!]req_{\mathbf{a}} = \mathsf{true}$ for any $req_{\mathbf{a}}$. Since $\mathcal{T}\{\!|\mathsf{true}|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathsf{true}$ and $\xi \models \mathsf{true}\langle\!\langle a \rangle\!\rangle^h_m$ for any ξ and a, the thesis follows due to Lemma 6.1.

($\tau = f(sn, pv)$) From the definition of $\mathcal{E}$, we have $\mathcal{E}[\![f(sn, pv)]\!]req_{\mathbf{a}} = \mathsf{true} \Rightarrow f([\![req_{\mathbf{a}}(sn)]\!], [\![pv]\!]) = \mathsf{true}$. As constraint operators resulting from $\mathsf{getOp}(f)$ have, by definition, the same semantics of f, i.e. $\mathsf{getOp}(f) \equiv f$, we proceed by case analysis on sn.

($sn = \mathsf{action/id}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $req_{\mathbf{a}}(\mathsf{action/id}) = \mathsf{acid}(\mathbf{a})$ such that $f(\mathsf{acid}(\mathbf{a}), [\![\ pv\ ]\!]) = \mathsf{true}$. Since $\mathcal{T}\{\!|f(\mathsf{action/id}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = id\text{-}h\ \mathsf{getOp}(f)\ pv$ and $\langle\!\langle a \rangle\!\rangle^h_m = [id\text{-}h := \mathsf{acid}(a), \ldots]$ where $\mathsf{acid}(a) = \mathsf{acid}(\mathbf{a})$ by definition, due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, there exists ξ so that the thesis follows.

($sn = \mathsf{action/arg}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $req_{\mathbf{a}}(\mathsf{action/arg}) = pt$ such that $f(pt, [\![\ pv\ ]\!]) = \mathsf{true}$ for a policy tuple $pt = (\!|t|\!)/(\!|T|\!)$ with t/T argument of $\mathbf{a}$. Since $\mathcal{T}\{\!|f(\mathsf{action/arg}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = arg\text{-}h\ \mathsf{getOp}(f)\ pv$ and $\langle\!\langle a \rangle\!\rangle^h_m = [arg\text{-}h := cv\ \ldots]$, we have, due to Lemma 6.1, that there exists ξ such that $pt = cv\xi$. Hence, due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, the thesis follows.

($sn = \mathsf{subject/id}$) As the target matches, by definition of $req_{\mathbf{a}}$, we have that $req_{\mathbf{a}}(\mathsf{subject/id}) = \mathcal{I}.\mathsf{id} = m$ such that $f(m, [\![\ pv\ ]\!]) = \mathsf{true}$. Since $\mathcal{T}\{\!|f(\mathsf{subject/id}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = sub\text{-}h\ \mathsf{getOp}(f)\ pv$ and $\langle\!\langle a \rangle\!\rangle^h_m = [sub\text{-}h := m, \ldots]$, due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, there exists ξ so that the thesis follows.

($sn = \mathsf{object/id}$) As the target matches, by definition of $req_{\mathbf{a}}$, we have that $req_{\mathbf{a}}(\mathsf{object/id}) = \mathcal{J}.\mathsf{id} = n$ such that $f(n, [\![\ pv\ ]\!]) = \mathsf{true}$. Since $\mathcal{T}\{\!|f(\mathsf{object/id}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = obj\text{-}h\ \mathsf{getOp}(f)\ pv$ and $\langle\!\langle a \rangle\!\rangle^h_m = [obj\text{-}h := cv, \ldots]$, we have, due to Lemma 6.1, that there exists ξ such that $n = cv\xi$. Hence, due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, the thesis follows.

($sn = \mathsf{subject/z}$ **with** $\mathsf{z} \neq \mathsf{id}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $(\mathsf{z} : e) \in \mathcal{I}$ and $[\![\ e\ ]\!] = v$ for some value v so that $f(v, [\![\ pv\ ]\!]) = \mathsf{true}$. Since $\mathcal{T}\{\!|f(\mathsf{subject/z}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = \bigvee_{l \in \mathcal{S}} \mathsf{z}\text{-}l\ \mathsf{getOp}(f)\ pv$, we have, due to Lemma 6.2, that $\mathcal{I}.\mathsf{id} = n \in \mathcal{S} = Sbj_S(\tau)$, hence $\mathsf{z}\text{-}n$ occurs. Thus, with $\xi \triangleq [v/\mathsf{z}\text{-}n, \ldots]$ and due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, the thesis follows.

($sn = \mathsf{object/z}$ **with** $\mathsf{z} \neq \mathsf{id}$) As the target matches the request, by definition of $req_{\mathbf{a}}$, we have that $(\mathsf{z} : e) \in \mathcal{J}$ and $[\![\ e\ ]\!] = v$ for some value v so that $f(v, [\![\ pv\ ]\!]) = \mathsf{true}$. Since $\mathcal{T}\{\!|f(\mathsf{object/z}, pv)|\!\}^h_{\mathcal{S},\mathcal{O}} = \bigvee_{l \in \mathcal{O}} \mathsf{z}\text{-}l\ \mathsf{getOp}(f)\ pv$, we have, due to Lemma 6.2, that $\mathcal{J}.\mathsf{id} = n \in \mathcal{O} = Obj_S(\tau)$, hence $\mathsf{z}\text{-}n$ occurs. Thus, with $\xi \triangleq [v/\mathsf{z}\text{-}n, \ldots]$ and due to $\mathsf{getOp}(f) \equiv f$ and Lemma 6.1, the thesis follows.

($\tau = \neg\tau_1$) We have $\mathcal{T}\{\!|\neg\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}} = \neg\mathcal{T}\{\!|\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}}$; due to the definition of $\mathcal{E}$, the thesis immediately follows by structural induction.

($\tau = \tau_1 \wedge \tau_2$) We have $\mathcal{T}\{\!|\tau_1 \wedge \tau_2|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathcal{T}\{\!|\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}} \wedge \mathcal{T}\{\!|\tau_2|\!\}^h_{\mathcal{S},\mathcal{O}}$; due to the definition of $\mathcal{E}$, the thesis immediately follows by structural induction.

($\tau = \tau_1 \vee \tau_2$) We have $\mathcal{T}\{\!|\tau_1 \vee \tau_2|\!\}^h_{\mathcal{S},\mathcal{O}} = \mathcal{T}\{\!|\tau_1|\!\}^h_{\mathcal{S},\mathcal{O}} \vee \mathcal{T}\{\!|\tau_2|\!\}^h_{\mathcal{S},\mathcal{O}}$; due to the definition of $\mathcal{E}$, the thesis immediately follows by structural induction.

□

Corollary 6.4. *Given a system S with $\mathcal{I}, \mathcal{J} \in \mathsf{Interf}(S)$, for any rule ρ of the policy $\Pi(S, \mathcal{I}.\mathsf{id})$ with $\mathsf{id}(\rho) = h$, and for any obligation action a such that $\mathcal{A}(a\xi', \mathcal{I}.\mathsf{id}) = \mathbf{a}$ for some substitution ξ', it holds that:*

$$\mathcal{P}[\![\rho]\!]\mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J}) = d', s'_{\mathtt{B}}, s'_{\mathtt{A}} \quad \Rightarrow \quad \exists\, \xi\ :\ \xi \models \mathcal{T}\{\!|\tau|\!\}^h_{\mathcal{S},\mathcal{O}} \langle\!\langle a \rangle\!\rangle^h_m .$$

Proof. From the definition of $\mathcal{P}$ we have that a rule ρ controls an action $\mathbf{a}$ when its corresponding request $req_{\mathbf{a}} = \mathsf{req}(\mathcal{I}, \mathbf{a}, \mathcal{J})$ matches the rule target, i.e. it evaluates to true. Hence, the thesis immediately follows from Theorem 6.3. □

6.4 Policy-Flow Graph

We now define the construction of a graph, called *Policy-Flow graph*, that, by relying on the previous constraint-based representation of rule targets and authorisation requests, graphically and compactly represents all the potential flows in a policy. Afterwards, we apply

the graph to the case study (Section 6.4.1) and outline the SMT-LIB coding of constraints that allows the PSCEL IDE to automatise the construction of flow graphs (Section 6.4.2).

As policies can check conditions on the *context* (which is made of the component features), the edges are annotated with the contextual conditions holding when the corresponding flow takes place. For convenience, we re-organise the constraints representing targets so that the constraints involving variables from the set A_Π are separated from those involving variables from the set F_S[3]. In the following, we thus assume they are written in the form $act \wedge ctx$, where act is the constraint on the *action*, while ctx is that on the *context*; if a constraint is empty, it corresponds to true. Furthermore, as a matter of notation, we use $\rho \rhd^{act \wedge ctx} s$ to indicate that $act \wedge ctx$ is the constraint representing the target of rule ρ and that s is the sequence of before and after obligations generated when the rule applies.

Intuitively, the nodes of the graph represent policy rules, or its combining algorithm, while the directed edges represent the flows. Hence, the graph *paths* estimate the sequences of policy evaluations that might occur at runtime. Formally, the Policy-Flow graph is defined as follows.

Definition 6.2 (Policy-Flow Graph)**.** *The* Policy-Flow graph G_Π *of a policy* $\Pi(S, m) \triangleq \langle alg \text{ rules} : \rho_1 \ \ldots \ \rho_k \rangle$ *is a doubly labelled directed graph* (N, F, T, L) *where*

- N*, i.e. the set of* nodes*, is* $\{\mathtt{id}(\rho_1), \ldots, \mathtt{id}(\rho_k), alg\}$ *(recall that* $\mathsf{id}(\rho_j)$ *is the name of rule* ρ_j*);*
- F*, i.e. the set of* edge labels*, is* $\{ctx_j \mid \rho_j \rhd^{act_j \wedge ctx_j} s_j \text{ with } j = 1, \ldots, k\}$*;*
- $T \subseteq N \times F \times N$*, i.e. the set of labelled* directed edges*, contains the elements*
  - $(\mathsf{id}(\rho_j), ctx_j, \mathsf{id}(\rho_l))$*: for each rule pair* ρ_j *and* ρ_l*, with* $\rho_j \rhd^{act_j \wedge\ ctx_j} s_j$ *and* $\rho_l \rhd^{act_l \wedge\ ctx_l} s_l$*, such that* $\exists\, a \in s_j\ , \exists\, \xi :\ \xi \models (act_l \wedge ctx_l)\langle\!\langle a \rangle\!\rangle^l_m$*;*
  - $(\mathsf{id}(\rho_j), ctx_j, alg)$*: for each rule* ρ_j*, with* $\rho_j \rhd^{act_j \wedge\ ctx_j} s_j$*, such that* $\exists\, a \in s_j\ ,\ \nexists\, \rho_l$*, with* $\rho_l \rhd^{act_l \wedge\ ctx_l} s_l$*, such that* $\forall \xi\ :\ \xi \models (act_l \wedge ctx_l)\langle\!\langle a \rangle\!\rangle^l_m$*;*
- $L : N \to \{\mathsf{p}, \mathsf{d}\}$*, i.e. the node* labelling function*, is defined as follows*

$$L(\mathsf{id}(\rho_j)) = \mathsf{p} \text{ if } \rho_j \text{ has decision } \mathsf{permit} \qquad L(alg) = \mathsf{p} \text{ if } alg = \text{p-unless-d}$$
$$L(\mathsf{id}(\rho_j)) = \mathsf{d} \text{ if } \rho_j \text{ has decision } \mathsf{deny} \qquad L(alg) = \mathsf{d} \text{ if } alg = \text{d-unless-p}$$

The graph has two types of edges: one representing a flow between two rules, the other representing a flow from a rule to the combining algorithm. In the first edge type, $\mathsf{id}(\rho_j)$ is connected to $\mathsf{id}(\rho_l)$ when there exists in the sequence s_j an action a whose induced assignment $\langle\!\langle a \rangle\!\rangle^l_m$ makes the constraint corresponding to the target of the rule ρ_l satisfiable, i.e. there exists ξ such that $\xi \models (act_l \wedge ctx_l)\langle\!\langle a \rangle\!\rangle^l_m$ holds. The edge is annotated with the contextual conditions ctx_j asserted by the target of ρ_j. In the second edge type, $\mathsf{id}(\rho_j)$ is connected to alg when there exists in the sequence s_j an action a whose induced assignment $\langle\!\langle a \rangle\!\rangle^l_m$ cannot make the constraint corresponding to the target of any rule ρ_l always satisfiable, i.e. there does not exist ρ_l such that for all substitutions ξ the condition $\xi \models (act_l \wedge ctx_l)\langle\!\langle a \rangle\!\rangle^l_m$ holds; the edge is annotated with ctx_j as well. If multiple edges

[3]This splitting can always be done by appropriately applying standard boolean laws because each relational operator takes at most one variable as argument.

with the same label connect a node $\mathsf{id}(\rho_j)$ to node alg, only one of them is retained. Notably, the same action a can cause, due to different substitutions, the creation of edges of both types. Since combining algorithms neither define controls nor obligations, alg has no outgoing edges. Notice that determining the set of edges T has a worst case complexity of $O(k^2\theta)$, where k is the number of policy rules and θ is the maximum number of actions forming the obligations of the policy rules (e.g., in the case of policy Π_S, θ has value 3 because of the obligation actions of rule S2). Indeed, both edge types require examining all the $k \times k$ pairs of rules. Each pair of both types requires examining at most θ obligation actions.

To sum up, the edges of the graphs approximate the flows between the policy rules, while the *paths* in the graph, i.e. sequences of connected edges, estimate the sequences of policy evaluations that might occur at runtime.

6.4.1 Policy-Flow Graph at work on the Autonomic Cloud Case Study

The policies Π_S and Π_C presented in Section 6.2 generate the flows graphically depicted by the graphs in Figure 6.1. Before commenting the construction of the graphs, we outline the constraint-based representation of some of the rule targets. For simplicity's sake, we consider a system S formed by three components: one server and two clients named $sr1$, $cl1$ and $cl2$, respectively. The set of system interfaces $\mathsf{Interf}(S)$ is defined as $\{\mathcal{I}_1, \mathcal{I}_{C_1}, \mathcal{I}_{C_2}\}$.

Let us first determine the sets of variables A_Π and F_S. The set F_S contains as many variables as the interface features of each of the three components, that is role, level, locality and load. Instead, the set A_Π depends on the policy. For example, in the case of Π_S, it is $\{id\text{-}h, arg\text{-}h, sub\text{-}h, obj\text{-}h \mid h \in \{S1, S2, S3, S4\}\}$. For ease of reference, as a subscript of constraint names we use the name of the rule enclosing the represented target; e.g., the constraint named act_{S1} represents the action conditions of rule S1. Furthermore, to increase readability of constraints, we underline constraint variables, e.g. $\underline{id\text{-}S1}$, and structured names occurring in constraint assignments, e.g. $\underline{\mathsf{action/arg}}$.

We report now some examples of constraint-based representation of targets. Let us consider rule S1 of the policy Π_S in force at the component $sr1$. Firstly, we determine the sets $\mathcal{S}$ and $\mathcal{O}$, i.e. via functions Sbj and Obj. We have that $\mathcal{S} = \{sr1\}$ due to the control $\mathsf{equal(subject/id}, sr1)$[4], while $\mathcal{O} = \{sr1, cl1, cl2\}$, i.e. it contains the names of all system components, because the only control restricting the set of names is that on the feature level that is anyway exposed by all components. Secondly, the translation function $\mathcal{T}$ is applied to the target of rule S1 and returns

$$\begin{array}{l}(\underline{id\text{-}S1} = \textbf{get}) \wedge (\underline{sub\text{-}S1} = sr1) \wedge (\underline{arg\text{-}S1}\ \mathsf{match}\ (task, \_, \_, \_)) \\ \wedge\ (\underline{\mathsf{level}\text{-}sr1} = 1) \wedge ((\underline{\mathsf{level}\text{-}sr1} = 2) \vee (\underline{\mathsf{level}\text{-}cl1} = 2) \vee (\underline{\mathsf{level}\text{-}cl2} = 2))\end{array} \tag{6.1}$$

The sub-constraints in the first row represent the target controls referring to the action; their conjunction forms the constraint act_{S1}. The sub-constraints in the second row represent the two target controls on the feature level: $(\underline{level\text{-}sr1} = 1)$ is obtained from the control $\mathsf{equal(subject/level}, 1)$ by exploiting the set $\mathcal{S}$, while the disjunction following the operator $\wedge$ is obtained from the control $\mathsf{equal(object/level}, 2)$ by exploiting the set $\mathcal{O}$. The conjunction of the sub-constraints in the second row forms the constraint ctx_{S1}.

[4]The name $sr1$ of the component where the policy Π_S is assumed to be in force is that referred to by the variable this occurring in the definition of rule S1.

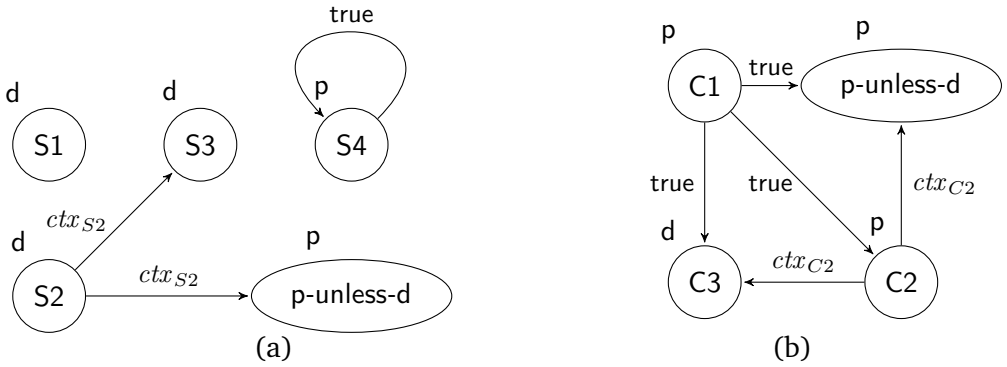


Figure 6.1: Autonomic cloud case study: Policy-Flow graphs (some conditions ctx_h are detailed in the text): (a) policy Π_S of servers; (b) policy Π_C of clients

In case of rule S4 of the policy Π_S, its target is represented by the constraint

$$(\underline{id\text{-}S4} = \textbf{put}) \wedge (\underline{sub\text{-}S4} = sr1) \tag{6.2}$$

It forms the constraint act_{S4}, while the constraint ctx_{S4} is true since there are no contextual controls in the rule target.

Let us consider rule C2 of the policy Π_C; its target is represented as follows

$$(\underline{id\text{-}C2} = \textbf{get}) \wedge (\underline{\text{role-}sr1} = server \vee \underline{\text{role-}cl1} = server \vee \underline{\text{role-}cl2} = server) \tag{6.3}$$

where $\mathcal{S} = \{sr1, cl1, cl2\}$ is used to define the constraints on the feature role.

The constraints just introduced can now be exploited to construct the Policy-Flow graphs of the policies Π_S and Π_C according to Definition 6.2. For simplicity's sake, we use the name of the rules and of the algorithm to identify the corresponding node in the graph. Since the policy rules that do not define obligations cannot trigger other rules the corresponding nodes have no outgoing edges. As depicted in Figure 6.1, this is the case, e.g., of rules S1 and S3 of policy Π_S. In the remaining cases, to determine the outgoing edges of a node, we check if any obligation action of its corresponding rule induces an assignment that makes the constraint representing a rule target satisfiable.

Let us consider the construction of the graph of the policy Π_S. Rule S4 returns the obligation action $\textbf{put}(log, \text{action/arg})@\text{self}$ that, relatively to node S4, induces the assignment $[\underline{id\text{-}S4} := \textbf{put}, \underline{arg\text{-}S4} := (log, \underline{\text{action/arg}}), \underline{sub\text{-}S4} := sr1, \underline{obj\text{-}S4} := sr1]$. This assignment makes the applicability constraint of S4, reported in (6.2), satisfiable. In fact, by applying the assignment, we get the constraint $\textbf{put} = \textbf{put} \wedge (sr1 = sr1)$ that clearly evaluates to true. Hence, there is a self loop on node S4 labelled by the contextual constraint ctx_{S4}, i.e. true. The other flows in policy Π_S are generated by the obligation actions within rule S2. By reasoning as before we can easily establish that its **fresh** and **new** actions do not match the target of any rule, therefore there is a flow from node S2 to node p-unless-d, while its **read** action can match the target of rule S3 when the subject load is higher than 60, hence there is a flow from node S2 to node S3. Notice that, when the load is less than 60, the action **read** does not match the target of rule S3, hence, as there does not exist a rule always applicable to such action, there can be also a flow to node p-unless-d. On the contrary, node S4 is not connected to node p-unless-d, because the rule itself is always applicable to its obligation (it follows directly from the fact that constraint (6.2), with applied the corresponding assignment, is satisfied and does not contain open terms).

The graph of the policy Π_C is constructed similarly. We only comment on the flow from node C1 to node C2 meaning that the obligation action **get** returned by rule C1 can be controlled by rule C2. Indeed, C2 applicability constraint, reported in (6.3), is satisfiable by applying the assignment $[\underline{id\text{-}C2} := \textbf{get}; \underline{arg\text{-}C2} := (taskId, \underline{num}); \underline{sub\text{-}C2} := cl1, \underline{obj\text{-}C2} := cl1]$ induced by the obligation action of C1 relatively to node C2. However, this flow cannot actually occur, because rule C2 checks if the action subject has role $server$, while the injected action **get** is locally executed by a $client$ component. This over-approximation derives from the fact that the static analysis we pursue abstracts from the actual values (and their modifications) of the context features.

Remark 6.2 (Assignment induced by the structured name action/arg)**.** *The obligation actions can use the structured name* action/arg *which dynamically will be replaced by the item/template argument of the action that has matched the target of the rule generating the obligation action. To statically determine the assignment induced by the action, it is then safe to map* action/arg *to a variable that can assume any policy value. For the sake of presentation, we do not introduce additional refinements, but it is worth noticing that we could use the controls occurring in the target of a rule to better approximate the set of potential values which* action/arg *can dynamically refer to. We briefly illustrate the point through the example below.*

Let us consider rule C1 *and its obligation action* $\mathbf{put}$(action/arg, $\underline{num}$)@self. *When the action is dynamically fulfilled, the name* action/arg *is replaced by the item/template argument of the action that has matched the target of* C1*. Therefore, from the control* pattern-match(action/arg,$(task, \_, \_)$) *we get that* action/arg *will necessarily represent an item/template of three elements with the value* $task$ *as its first one. Thus, since the item of the obligation action* ***put*** *extends the three-element item referred to by* action/arg *with* $\underline{num}$*, variables* $\underline{arg\text{-}h}$ *can only assume four-element item/template values that have* $task$ *as their first element. By the way, the addition of an element to the original item prevents the occurrence of a self loop on node* C1 *(indeed, its target only applies to three-element items/templates).*

6.4.2 Automated Policy-Flow Graph Construction

The construction of Policy-Flow graphs requires extensive checks on rules and obligations, hence, in order to be practically effective, tool support is essential. To this aim, we express the PSCEL-oriented constraint formalism introduced in Section 6.3.1 by means of the SMT-LIB language; this coding is done in a way similar to that of Section 4.5.1. On the basis of this SMT-LIB code, the PSCEL IDE can opportunely exploit the Z3 SMT solver to automatically construct Policy-Flow graphs.

The SMT-LIB coding of the PSCEL constraints corresponds to a simplified version of that of the FACPL ones. In fact, as policy-flows refer only to applicable (authorisation requests representing) actions and PSCEL rules are not applicable if an attribute is missing or erroneous (see Remark 5.3), the SMT-LIB coding models attributes as single (typed) variables, and not as 3-valued records like in FACPL.

First of all, given a PSCEL specification, we declare the variables forming the sets A_Π and F_S. The case of F_S is straightforward: each feature of each component corresponds to a variable. For instance, a server *sr1* of the autonomic cloud case study generates the following variable declarations

```
(declare-const n_sr1_id Str)
(declare-const n_sr1_role Str)
(declare-const n_sr1_level Int)
(declare-const n_sr1_locality Str)
(declare-const n_sr1_load Int)
```

where the datatype `Str` is used to represent in Z3 string values.

The case of A_{Π} requires instead a slightly more complicated coding, due to the variable $arg\text{-}h$ and the pattern-matching function defined on it. In fact, as SMT-LIB is strongly-typed, we cannot use a record type to code sequences of elements of different types or to compare sequences of different lengths. Therefore, given a rule h defining a pattern match control in its target, the variable $arg\text{-}h$ corresponds to: (i) a set of variables representing the elements of the template argument of the pattern match control; (ii) a variable representing the length of such template. The set of variable declarations for a rule containing, e.g., the control pattern-match(action/arg,$(task, \_, \_)$) is as follows

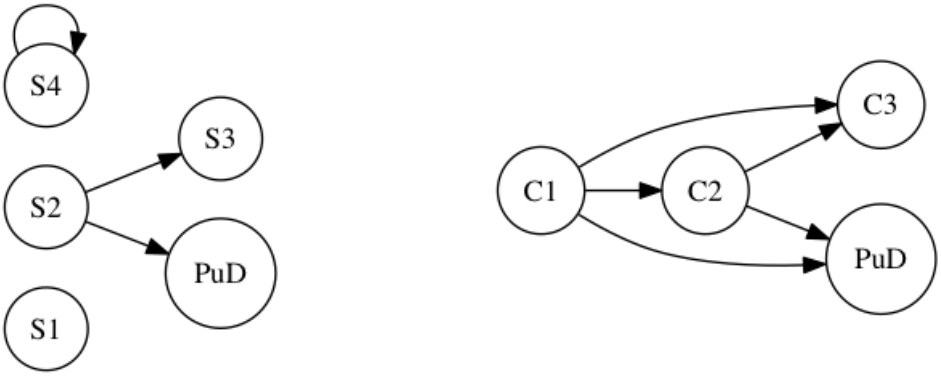
```
(declare-const n_sub/id Str)
(declare-const n_obj/id Str)
(declare-const n_act/id ACTIONID)
(declare-const n_act/arg_1 Str)
(declare-const n_act/arg_2 Str)
(declare-const n_act/arg_3 Str)
(declare-const n_act/argN Int)
```

where the datatype `ACTIONID` represents the PSCEL action identifiers, while the SMT-LIB variable named *n_act/argN* refers to the template length. Notice that the second and third element of the template are represented by SMT-LIB variables with type `Str`, because the PSCEL IDE requires a type for each template element.

Once variables are declared, the constraint functions of Table 6.1 and, consequently, the translation procedure of Table 6.3 can be straightforwardly defined. By way of example, we report in the following the SMT-LIB constraint `cns_S1` of rule S1 corresponding to the formal constraint reported in (6.1).

```
(define-fun cns_S1 () Bool
  (and
    (= n_act/id GET)
    (= n_sub/id s_sr1)
    (= n_act/arg_1 s_task)
    (= n_act/argN 4)
    (= n_sr1_level 1)
    (or
       (= n_sr1_level 2)
       (= n_cl1_level 2)
       (= n_cl2_level 2))))
```

To reason on applicability of obligations, we thus code each obligation action into a set of assertions on constraint variables. This coding is made according to the definitions in (A-1) and, e.g., in the case of obligation $\mathbf{put}(log, \text{action/arg})@\text{self}$ is as follows



(a) (b)

Figure 6.2: Autonomic cloud case study: Policy-Flow graphs generated by the PSCEL IDE (PuD stands for p-unless-d): (a) policy Π_S of servers; (b) policy Π_C of clients

```
(define-fun cns_Obl () Bool
  (and
    (= n_sub/id s_sr1)
    (= n_obj/id s_sr1)
    (= n_act/id PUT)
    (= n_act/arg_1 s_log)
    (= n_act/argN 2)))
```

Finally, once all rules and obligations are coded into SMT-LIB, the flow graph can be constructed by checking the conditions defining the graph edges. The first edge type requires a satisfiability check, while the second requires a set of validity checks.

The generation of the SMT-LIB code and the checks for the edge definition are automatically carried out by the PSCEL IDE. As a result, it is generated a text file representing the flow graph, that can be automatically drawn via the Graphviz tool[5]. Figure 6.2 reports the flow graphs generated by the PSCEL IDE for the autonomic cloud case study and drawn by Graphviz.

6.5 Progress Analysis of PSCEL Specifications

Policy evaluations may affect the progress of controlled processes. The effects on progress can be expressed in terms of the following properties

- *finite evaluation*: each action can only trigger a finite number of policy evaluations;
- *undeniable executability*: once an action is positively authorised, the execution of the controlled process cannot be blocked due to the injection of an action that is denied.

For example, rule S4 in the previous section shows a violation of the former property, while rules C1 and C3 show a violation of the latter one.

The properties above refer to the flows that a policy may generate and can be verified in terms of conditions on the structure of the Policy-Flow graph. Since the graph

[5]Graphviz - http://www.graphviz.org/

paths statically address conditions on the context, we need to assume that the context is somehow *stable*, that is it does not change along each path. To effectively reason on contextual conditions, we introduce the *characteristic formula* of a path, which is made of the contextual conditions occurring along the path, and the concept of *feasible* path.

As a matter of notation, we assume that the set N of nodes of a flow graph G_{Π} is ranged over by ν and that the function cstr, given in input a node ν, returns the constraint corresponding to the policy rule represented by ν; the function returns true if ν represents a combining algorithm.

In the following, we first consider a simplified situation which abstracts from the controlled processes (Section 6.5.1), then we illustrate how to extend our results when all intricacies come into the picture (Section 6.5.2). We conclude by defining a syntactic check to approximate the context stability of a policy (Section 6.5.3).

6.5.1 Context-stable Policies

Before presenting the structural conditions on the flow graph over-approximating the verification of the considered properties, we formalise the concepts of *context-stable* policy and that of *feasible* path.

Definition 6.3 (Context-stable Policy)**.** *A policy is* context-stable *if, along each path of its Policy-Flow graph, the features it checks do not change value.*

Intuitively, if a policy is context-stable then, given a feature n checked by (the target of a rule of) the policy, an action of the form **upd**(n, e) cannot interleave with the policy evaluations forming a path. This check could be done manually, e.g. in the case of our case study[6], or syntactically over-approximated by checking the policy specification. We refer to Section 6.5.3 for further details.

The paths of Policy-Flow graphs are annotated with constraints ctx, which represent the context conditions holding when the corresponding policy evaluations occur. To consider them in the analysis, we introduce the following notion.

Definition 6.4 (Characteristic Formula of a Path)**.** *Given a path formed by nodes $\nu_1 \dots \nu_k$, its* characteristic formula *is* $\mu \triangleq \bigwedge_{j=1}^{k} \mathsf{cstr}(\nu_j)$.

Notably, due to the context-stability assumption, it is enough to consider the context conditions occurring in a loop only once. A path is deemed *feasible* if, under the context-stability assumption, its characteristic formula is satisfiable. Unsatisfiable paths in the policy-flow graph represent sequences of flows that, due to conflictual contextual conditions, e.g. like $\underline{\mathsf{role}\text{-}sr1} = server$ and $\underline{\mathsf{role}\text{-}sr1} = client$, cannot actually occur in the system.

Finite Evaluation

A (context-stable) policy enjoys the *finite evaluation* property when each action matching the target of any rule can only trigger finite sequences of policy evaluations. It follows that the property holds when the Policy-Flow graph has no feasible infinite paths, i.e. loops, as stated by the following theorem.

[6] E.g., rule $S2$ of policy Π_S checks the feature load. Since $S2$ only generates paths of length one, possible updates of load cannot interleave with policy evaluations.

Theorem 6.5. *A context-stable policy enjoys the finite evaluation property only if its Policy-Flow graph contains no feasible loops.*

Proof. Let Π be a context-stable policy enjoying the finite evaluation property. Suppose that its graph G_Π has a feasible loop, that is a path of the form

$$\mathsf{id}(\rho_{j_0}) \xrightarrow{c_{j_0}} \ldots \xrightarrow{c_{j_{k-1}}} \mathsf{id}(\rho_{j_k}) \xrightarrow{c_{j_k}} \ldots \xrightarrow{c_{j_{n-1}}} \mathsf{id}(\rho_{j_0})$$

whose characteristic formula μ is satisfiable. Then, from Definition 6.2 and Corollary 6.4, it follows that the sequence of rules

$$\rho_{j_0} \cdots \rho_{j_{k-1}}\ \rho_{j_k} \cdots \rho_{j_0}$$

is such that, for each pair of rules $\rho_{j_{k-1}}\ \rho_{j_k}$, at least an obligation a injected by rule $\rho_{j_{k-1}}$ triggers the evaluation of rule ρ_{j_k}. This is a contradiction since rule ρ_{j_0} triggers infinite policy evaluations. □

As informally pointed out in Section 6.4.1, this structural condition is not met by policy Π_S, while it holds for policy Π_C.

Undeniable Executability

A policy enjoying the finite evaluation property can anyway undermine the progress of a controlled process due to the authorisations it enforces. The undeniable executability property addresses the case of injected obligation actions that are denied. Specifically, once an action has been permitted, the denying of some of the obligations whose injection was caused by the action authorisation may prevent the execution of the controlled process. It follows that the property holds when, in the Policy-Flow graph, each path containing nodes labelled by p (i.e., enforcing permit) does not contain nodes labelled by d (i.e., enforcing deny).

Theorem 6.6. *A context-stable policy enjoys the undeniable executability property only if for each feasible path in its Policy-Flow graph, if the path contains a node labelled by* p*, than after this node there is no node labelled by* d*.*

Proof. Let Π be a context-stable policy enjoying the undeniable executability property. Suppose that its graph G_Π has a feasible path containing at least a node labelled by p and one labelled by d, that is a path of the form

$$\mathsf{id}(\rho_{j_0}) \xrightarrow{c_{j_0}} \ldots \xrightarrow{c_{j_{k-1}}} \mathsf{id}(\rho_{j_k}) \xrightarrow{c_{j_k}} \ldots \xrightarrow{c_{j_{n-1}}} \mathsf{id}(\rho_{j_n})$$

where $L(\mathsf{id}(\rho_{j_0})) = \mathsf{p}$ and $L(\mathsf{id}(\rho_{j_n})) = \mathsf{d}$, whose characteristic formula μ is satisfiable. Thus, from Definition 6.2 and Corollary 6.4, it follows that the sequence of rules

$$\rho_{j_0} \cdots \rho_{j_{k-1}}\ \rho_{j_k} \cdots \rho_{j_n}$$

is such that, for each pair of rules $\rho_{j_{k-1}}\ \rho_{j_k}$, at least an obligation a injected by rule $\rho_{j_{k-1}}$ triggers the evaluation of rule ρ_{j_k}. This is a contradiction since rule ρ_{j_0} has decision permit, while ρ_{j_n} has decision deny. □

Concerning the graphs in Figure 6.1, it is easy to check that policy Π_C meets the condition of Theorem 6.5 while policy Π_S does not; instead, because of the path between rules C1 and C3, policy Π_C does not met the condition of Theorem 6.6.

The converse of Theorems 6.5 and 6.6 does not hold. This is a consequence of Theorem 6.4 and of the fact that the Policy-Flow graph does not take into account the evaluation of the combining algorithm, but it only considers rules separately.

6.5.2 Context-stable Policies with respect to a Process

The progress analysis presented for context-stable policies can be refined by taking into account the controlled process. In particular, the conditions of Theorems 6.5 and 6.6, as well as Definition 6.3, can be relaxed according to process definitions. Intuitively, when a process is taken into account, the theorem conditions can only address those graph paths that the process can trigger. Before presenting the refined results, we introduce some auxiliary notations.

Given a policy $\Pi(S, m)$ and its Policy-Flow graph $G_\Pi = (N, F, T, L)$, we define the set $\mathcal{P}_{G_\Pi}$ of all the possible paths of the graph as follows

$$\mathcal{P}_{G_\Pi} = \{\nu_{j_0} \dots \nu_{j_n} \mid \quad \nu_{j_k}, \nu_{j_{k+1}} \in N \ , \ (\nu_{j_k}, ctx_{j_k}, \nu_{j_{k+1}}) \in T \quad \text{with } k \in \{0, \dots, n-1\}, n \geq 1\}$$

While, given a process P and a policy $\Pi(S, m)$, the set $\mathcal{P}^P_{G_\Pi} \subseteq \mathcal{P}_{G_\Pi}$ of all paths of G_Π that the process P can trigger is defined as follows

$$\mathcal{P}^P_{G_\Pi} = \{\nu_{j_0} \dots \nu_{j_n} \mid \quad \nu_{j_0} \dots \nu_{j_n} \in \mathcal{P}_{G_\Pi} \ , \ \exists\, a \text{ in } P \ , \ \xi \ : \ \xi \models \mathsf{cstr}(\nu_{j_0})\langle\langle a \rangle\rangle^{j_0}_m\}$$

where $\mathsf{cstr}(\nu_{j_0})$ identifies the constraint of the rule represented by the node ν_{j_0}[7]. Indeed, the set $\mathcal{P}^P_{G_\Pi}$ contains all those paths that start from a node representing a rule whose target can match an action a occurring in P. This set contains, due to Corollary 6.4 and Definition 6.2, each possible path triggered by the actions of process P. Notice that we can apply the result of Corollary 6.4 because process actions are a limited version of obligation actions, i.e. structured names cannot occur in place of names.

Definition 6.5 (Context-stable Policy with respect to a Process)**.** *A policy $\Pi(S, m)$ is* context-stable with respect to a process *P, if along each path of $\mathcal{P}^P_{G_\Pi}$, the value of the features checked by Π do not change.*

Theorems 6.5 and 6.6 can be now relaxed by only considering the set of paths $\mathcal{P}^P_{G_\Pi}$. The following corollaries report, respectively, the conditions over-approximating the verification of the *finite evaluation* and *undeniable executability* properties when a process is taken into account.

Corollary 6.7. *A policy $\Pi(S, m)$ context-stable with a process P enjoys the finite evaluation property only if the set of paths $\mathcal{P}^P_{G_\Pi}$ contains no feasible loops.*

Proof. The proof proceeds like that of Theorem 6.5, but by taking a path from $\mathcal{P}^P_{G_\Pi}$. □

Corollary 6.8. *A policy $\Pi(S, m)$ context-stable with a process P enjoys the undeniable executability property only if all the feasible paths in $\mathcal{P}^P_{G_\Pi}$ containing nodes labelled by* p *do not contain nodes labelled by* d.

Proof. The proof proceeds like that of Theorem 6.6, but by taking a path from $\mathcal{P}^P_{G_\Pi}$. □

To sum up, if a process cannot trigger paths violating the considered property, the policy controlling such process enjoys the property, even though it does not by itself.

[7] The node ν_{j_0} cannot represent a combining algorithm, because such a node would have no outgoing edge by definition.

6.5.3 Context-Stability: Syntactic Check

Defining if a policy is context-stable (with respect to a process) can be statically approximated by examining the obligation actions defined in a policy and, if a process is taken into account, the process actions. Indeed, the set of feature names occurring in a policy is compared against the features possibly modified by obligation actions (and process actions). Recall that features can only be updated by actions $\mathbf{upd}(\mathsf{n}, e)$ and that the feature name n cannot be provided through variables nor structured names.

We define the function $\mathsf{c\_names}$ that, given a target τ, returns the set of the occurring feature names. Its definition is as follows

$$\mathsf{c\_names}(\tau) = \{\mathsf{n} \mid \exists\ sn = \mathsf{subject/n} \text{ or } sn = \mathsf{object/n} \text{ in } \tau\}$$

The resulting set contains all those feature names n occurring in the structured name sn. Both subject and object categories are considered, because a component can be either the subject or object component of an action.

The function $\mathsf{c\_names}$ is then extended to rules and policies as follows

$$\mathsf{c\_names}(\mathsf{n}(d\ \ \mathsf{target} : \tau\ \ \mathsf{obl} : o^*)) = \mathsf{c\_names}(\tau)$$
$$\mathsf{c\_names}(\langle alg\ \ \mathsf{rules} : \rho_1 \ldots \rho_k \rangle) = \bigcup_{j=1}^{k} \mathsf{c\_names}(\rho_j)$$

Namely, the function returns the set of names of the rule target, in the case of a rule, and the set of names resulting from the union of the sets representing the enclosed rules, in the case of a policy.

A policy Π is thus *context-stable* if the following condition holds

$$\forall\ \mathbf{upd}(\mathsf{n}, e) \text{ in } \Pi\ \Rightarrow\ \mathsf{n} \notin \mathsf{c\_names}(\Pi)$$

which means that no obligation action acting on features can modify the features checked in the policy, i.e. those in $\mathsf{c\_names}(\Pi)$.

Instead, a policy Π is *context-stable with respect to a process* P if the previous condition holds together with the following one

$$\forall\ \mathbf{upd}(\mathsf{n}, e) \text{ in } P\ \Rightarrow\ \mathsf{n} \notin \mathsf{c\_names}(\Pi)$$

which refers to the process actions acting on features. Conditions on both obligation and process actions are needed, because it is not possible to syntactically understand whether an action comes from an obligation injected in the process.

Notice anyway that this check over-approximates the context-stability of a policy. In fact, features checked by a policy along a path, but modified along another path, violate the check causing false-positive.

6.6 Concluding Remarks

In this chapter we have presented a static analysis approach for PSCEL that, based on the notion of Policy-Flow graph, permits reasoning on the potential policy evaluations and their effects on the progress of systems. Here, we conclude by briefly commenting on the contributions of this analysis with respect to the research objectives of the thesis and to related and prior publications.

The specification of this analysis approach accomplishes the objective **O6**, hence it supports the analysis of PSCEL specifications by pointing out the effects on system behaviours of policy evaluations. Together with jRESP, the functionalities of the PSCEL IDE supporting the automated construction of the Policy-Flow graph contribute accomplishing the objective **O7**. Notice that the progress analysis based on the Policy-Flow graph can be supported by leveraging on the SMT-LIB code generated by the PSCEL IDE.

The analysis of SCEL-like systems was also addressed in [DLL+14, DLL+15] by means of the Spin model checker [Hol97]. Specifically, it is provided a translation of a lightweight version of SCEL into Promela, i.e. the input language of Spin, and shown how to exploit it to verify some properties of interest. However, the policies and some programming constructs are not taken into account. Our approach crucially addresses the role of policies and permits verifying progress properties related to the potential evaluations of policies.

In the literature, analysis approaches for languages featuring AOP characteristics, like, e.g., AspectK [HNNY08], have also been proposed. For instance, [TNN12] concerns the analysis of AspectK and proposes an approach based on communicating pushdown systems [BET03] to discover undesired infinite executions. However, AspectK has significantly less powerful functionalities than PSCEL, e.g. predicate-based communication. Furthermore, our approach is completely statical and only relies on the abstractions of the flow graph.

The contents of this chapter are mainly based on the work in [MRNNP16a, MRNNP16b]. All the works concerning SCEL do not address any of the contents reported in this chapter.

To conclude, we briefly comment on the scientific contribution of the proposed analysis. In particular, to our knowledge, the Policy-Flow graph addresses for the first time a flow analysis of policies that can be used to reason on the dynamic behaviours of autonomic systems. Generally speaking, this sort of analysis does not only refer to PSCEL, but it can be easily adapted to any other language featuring some distinguishing traits of PSCEL, i.e. interfaces, attribute-based requests and obligation actions.

Chapter 7

Related Works

In this chapter we review more closely related works to the FACPL and PSCEL languages and their analysis approaches. Before commenting in details these works, we outline below some other relevant application domains of policy-based systems.

The use of policies is usually advocated to decouple the managing aspects of a system from its functional behaviour, thus to achieve separation of concerns in system design. One of the first examples was in the context of computer security: in [WL93] policies were used to specify authorisation rules for programs. Besides access control and autonomic computing, other application domains that exploit policies are emergency handling and network management.

Emergency handling concerns all those applications that have to fulfil strict requirements beyond merely data secrecy. For example, in healthcare systems the accomplishment of patient treatment has to be always guaranteed, according to the principle that 'nothing can interfere with delivery of care'. This is an instance of a more general principle known as 'break the glass' [Joi04], which means that access controls can be bypassed in case of emergency. Many research efforts are being devoted to investigate these issues. For example, [ADCdVF+10] proposes an approach based on different policy spaces that are combined to model the additional behaviours needed in case of emergency. The approaches in [BP09, MDS14] rely on a lattice ordering of policies, that are applied according to the lattice value corresponding to each specific emergency. Notably, all these proposals exploit the high abstraction level of attributes to capture emergency information, and only require suitable ways of combining policies. Therefore, we can use FACPL for implementing any of them. For instance, to manage emergencies within the e-Health case study, we can simply introduce an additional policy that bypasses, in case of emergency, the other ones.

Policy	Number of lines		Saved	Number of characters		Saved
	XACML	FACPL	lines	XACML	FACPL	characters
e-Prescription	239	24	89,95%	10.656	894	91,61%
e-Dispensation	239	24	89,95%	10.674	914	91,43%
Consent Policy	423	38	91,01%	19.195	1.558	91,88%

Table 7.1: FACPL vs. XACML on the e-Health case study

Networks management is another significative application domain for policy languages. In fact, high-level management policies are convenient for dealing with the variety of heterogeneous devices typically present in computer networks; such an approach is exploited, e.g., in [Ver02]. An appropriate use of attributes allows also FACPL to define similar policies.

Structure of the chapter. The rest of this chapter is organised as follows. Section 7.1 comments on differences and interoperability of FACPL with XACML. Section 7.2 compares the most relevant languages for the specification of access control policies. Section 7.3 reviews the different approaches to the analysis of access control policies. Section 7.4 outlines different design and analysis approaches for autonomic computing systems. Section 7.5 concludes by comparing functionalities and performance of supporting tools.

7.1 FACPL vs XACML

XACML [OAS13] is a well-established standard for the specification of attribute-based access control policies and requests. Its XML-based syntax and informal evaluation process have been already introduced in Section 2.1.4. We comment here on the main syntactical and semantic differences and similarities with FACPL.

From a merely lexical point of view, FACPL allows developers to define each policy element via a lightweight mnemonic syntax and leads to compact policy specifications. Instead, the XML-based syntax used by XACML ensures cross-platform interoperability, but generates verbose specifications that developers hardly immediately understand and, most of all, are not suitable for formally defining semantics and analysis techniques. Table 7.1 exemplifies a lexical comparison between the FACPL policies for the e-Health case study and the corresponding XACML ones (both groups of policies can be downloaded from `http://facpl.sourceforge.net/eHealth/`).

Although FACPL and XACML policies have a similar structure, there are quite a number of (semantic) differences. In the following, we outline the main ones.

In FACPL, request attributes are referred by structured names. In XACML, they are referred by either AttributeDesignator or AttributeSelector elements. The former one corresponds to a typed version of a structured name, while the latter one is defined in terms of XPath expressions, which are not supported by FACPL. Anyway, FACPL can represent some of them by appropriately using structured names; e.g. an AttributeSelector with category $subject$ and an XPath expression like $type/id/text()$ correspond to subject/type.id.

The XACML Target structure AnyOf-AllOf-Match can be rendered in FACPL by means of, respectively, the expression operators and-or-and. However, slightly different results can be obtained from target evaluations due to the management of errors and missing attributes.

Features	XACML	Ponder	ASL	PTaCL	[RLB+09]	[ACC14]	KAoS	FACPL
Rule-based	✓	✓						✓
Logic-based			✓	✓	✓	✓		
Ontology-based							✓	
Mnemonic spec.		✓						✓
Comb. algorithms	✓		✓*	✓*	✓	✓*	✓*	✓
Obligations	✓	✓					✓	✓
Missing attributes	✓			✓				✓
Error handling	✓							✓

Table 7.2: Comparison of a relevant set of policy languages (where ✓* means that user encoding are required)

Indeed, when a value is missing, XACML semantics returns false, and this occurs since the level of Match elements, whereas the FACPL semantics of the target elements returns $\perp$ until the level of policies is reached, where $\perp$ is converted to false. Thus, a missing attribute could be masked in XACML but not in the corresponding FACPL expression; the same occurs for evaluation errors. Additionally, the evaluation of Match functions in XACML is iteratively defined on all the retrieved attribute values. To ensure a similar behaviour in FACPL, an XACML expression such as, e.g., an equality comparison must be translated into an operator defined on sets, like e.g. in. Clearly, this limits the amount of XACML functions that can be faithfully represented in FACPL. Furthermore, XACML poses specific restrictions on PolicySet targets: they can only contain comparison functions and each comparison can only contain one attribute name.

XACML requires that Rules can be combined with other Rules but not with PolicySets. It supports fewer combining algorithms than FACPL, as well as fulfilment strategies (indeed, XACML can only render the greedy one). Furthermore, XACML specialises the decision indet into three extended indeterminate values: indetD, indetP and indetDP. As reported in Section 3.1, we have not considered the extended indeterminate values in the formal account of FACPL, but they are supported by the FACPL tools.

Additionally, XACML provides some constructs that do not crucially affect policy expressiveness and evaluation. For instance, Variable elements permit defining pointers to expression declarations. These constructs are not directly supported by FACPL.

It is finally worth noticing that the aim of FACPL is to propose and deploy a compact, yet expressive, language whose formal foundations enable tool-supported analysis techniques, rather than to supersede XACML or only face its semantic issues.

7.2 Languages for Access Control Policies

Languages for the specification of access control policies have recently been the subject of extensive research, both by industry and academia. In the following, we compare some of the main languages with FACPL; Table 7.2 summarises the comparison.

Among the many proposed languages, we can identify three main specification approaches: (i) *rule-based*, as e.g. the XACML standard and Ponder [DDLS01, TDLS09]; (ii) *logic-based*, as e.g. ASL [JSS97], PTaCL [CM12] and the logical frameworks in [ACC14]; (iii) *ontology-based*, as e.g. in KAoS [UBJ+04] and Rei [KFJ03]. Many other works, as e.g. [LWQ+09, RLB+09, RRN14], study (part of) XACML by formally addressing peculiar features of design and evaluation of access control policies.

In the rule-based approach, policies are structured into sets of declarative rules. The seminal work [Slo94] introduces two types of policies: authorisations and obligations. Policies of the former type have the aim of establishing if an access can be performed, while those of the latter type are basically ECA rules triggering the enforcement of adaptation actions. This setting is at the basis of Ponder and XACML.

Ponder [DDLS01, TDLS09] is a strongly-typed policy language that, differently from FACPL, takes authorisation and obligations policies apart. Ponder does not provide explicit strategies to resolve conflictual decisions possibly arising in policy evaluation, rather it relies on abductive reasoning to statically prevent the occurrence of conflicts, although no implementation or experimental results are presented. On the contrary, FACPL provides combining algorithms, as we think they offer higher degrees of freedom to policy developers for managing conflicts. Similarly to Ponder, FACPL uses a mnemonic textual specification language and addresses value types, although they are not explicitly reported. Finally, the FACPL evaluation process is triggered by requests and not by events as in Ponder. Anyway, the FACPL approach is as general as the Ponder one since, by exploiting attributes, requests can represent any event of a system.

The logic-based approach mainly exploits predicate or multi-valued logics. Most of these proposals are based on Datalog [CGT89] (see, e.g., [JSS97, DeT02, HKTT09]), which implies that the access rules are defined as first order logic predicates. In general, these approaches offer valuable means for a low-level design of rules, but the lack of high-level features, e.g. combining algorithms or obligations, prevent them from representing policies like those of FACPL.

ASL [JSS97] is one of the firstly defined logic-based languages. It expresses authorisation policies based on user identity credentials and authorisation privileges, and supports hierarchisation and propagation of access rights among roles and groups of users. Additional predicates enable the definition of (a posteriori) integrity checks on authorisation decisions, e.g. conflict resolution strategies. Differently from ASL, FACPL provides high-level constructs and offers by-construction many not straightforward features like, e.g., conflict resolution strategies. A suitable use of policy hierarchisation enables propagation of access rights also in FACPL specifications.

PTaCL [CM12] follows the logic-based approach as well, but it does not rely on Datalog. It defines two sets of algebraic operators based on a multi-valued logic: one modelling target expressions, the other one defining policy combinations. These operators emphasise the role of missing attributes in policy evaluation, in a way similar to FACPL, but address errors only partially. In fact, combination operators are not defined on error values: it is rather assumed that all target functions are string equalities that never produce errors. Similarly to FACPL, PTaCL permits reasoning on non-monotonicity and safety properties of attribute-based policies [TK06].

A similar study, but more focussed on the distinguishing features of XACML, is reported in [RRN14]. It introduces a formalisation of XACML in terms of multi-valued logics, by first considering 4-valued decisions and then 6-valued ones (i.e., including the extended-indeterminate decisions). Most of the XACML combining algorithms are formalised as operators on a partially ordered set of decisions, while the algorithms first-app and one-app are defined by case analysis. Differently from FACPL, this formalisation does not deal with missing attributes and obligations, which are instead crucial in XACML policy evaluation.

Another logic-based language is presented in [ACC14]. In this case, a policy is a list of constraint assertions that are evaluated by means of an SMT solver. The framework

supports reasoning about different properties, but any high-level feature, as e.g. combining algorithms, has to be encoded 'by hand' into low-level assertions. In addition, missing attributes, erroneous values and obligations are not addressed.

Multi-valued logics and the relative operators have also been exploited to model the behaviour of combining algorithms. For example, the *Fine-Integration Algebra* introduced in [RLB+09] models the strategies of XACML combining algorithms by means of a set of 3-valued (i.e., permit, deny and not-app) binary operators. The behaviour of each algorithm is then defined in terms of the iterative application of the operators to the policies of the input sequence. This approach significantly differs from the FACPL one since it does not consider the indet decision. Instead, [LWQ+09] explicitly introduces an error handling function that, given two decisions, determines whether their combination produces an error, i.e. an indet decision. Each (binary) operator is then defined using such error function. The formalisation of FACPL combining algorithms follows a similar approach, but it also deals with obligations and fulfilment strategies, which require different iterative applications of the operators. Furthermore, [LWQ+09] exploits also nonlinear constraints for the specification of combining algorithms, e.g. to return a decision dec if the majority of the input policies return dec. Such algorithms are not usually dealt with in the literature and cannot be expressed in terms of iterative applications of some binary operators.

The last major approach for specifying access control policies relies on ontologies. The use of ontologies permits representing the contextual information of an access control system (i.e., the knowledge of the system). An example of this approach is the KAoS framework [UBJ+04] that uses the OWL ontology language[1]. With respect to FACPL, the OWL representation directly enables various reasoning techniques, but it is more complex to read and specify (e.g., long declarative descriptions, cross-references to external resources, absence of explicit combining algorithms). Moreover, the interoperability gained by using ontologies can be somehow achieved in FACPL by means of an appropriate use of attributes [NIS14]. Another example is the Rei language [KFJ03], where a deontic ontology is used for specifying positive and negative authorisation controls and obligations. Rei offers various reasoning techniques and a less verbose specification language than KAoS. However, Rei comes without any enforcement mechanisms, as e.g. the Java FACPL library, that can be exploited for implementing concrete access control systems.

7.3 Analysis of Access Control Policies

The increasing diffusion of access control systems has prompted the development of many analysis approaches for access control policies. These approaches pursue different techniques, ranging from SMT formulae to multi-terminal binary decision diagrams (MTBDD) and different kinds of logics. Below, we review the most relevant approaches by commenting their main differences with that we propose for FACPL.

The works that are closer to our approach are of course those exploiting SMT formulae. In [TdHRZ15], a strategy for representing XACML policies in terms of SMT formulae is introduced. The representation, which is based on an informal semantics of XACML, supports integers, booleans and doubles, while the representation of sets of values and strings is only sketched. The combining algorithms are modelled as conjunctions and disjunctions of formulae representing the policies to be combined, i.e. in a form similar to the

[1] Web Ontology Language (OWL) - http://www.w3.org/TR/owl-features/

approach shown in Table 4.3. As a design choice, formulae corresponding to the not-app decision are not generated, because they can be inferred as the complementary of the other ones. Thus, in case of algorithms like d-unless-p, additional workload is required. Moreover, differently from the FACPL SMT-based approach, the representation assumes that each attribute name is assigned only to those values that match the implicit type of the attribute, hence the analysis cannot deal with missing attributes or erroneous values. Finally, it does not take into account obligations, which have instead an important role in the evaluation. Again, differently from the FACPL approach, the SMT-based framework of [ACC14], introduced in Section 7.2, suffers from similar drawbacks.

The only analysis approach that takes missing attributes into account is presented in [CMZ15]. The analysis is based on a notion of request extension, as we have done in Section 4.4. Differently from the FACPL approach, this analysis aims at quantifying the impact of possibly missing attributes on policy evaluations, thus verifying probabilistic properties.

The change-impact analysis of XACML policies presented in [FKMT05] aims at studying the consequences of policy modifications. In particular, to verify structural properties among policies by means of automatic tools, this approach relies on an MTBDD-based representation of policies. However, it cannot deal with many of the XACML combining algorithms and, as outlined in [ACC14], SMT-based approaches like that of FACPL scale significantly better than the MTBDD one.

Datalog-based languages, like e.g. ASL, only provide limited analysis functionalities, that are anyway significantly less performant than the FACPL SMT-based approach. In general, these languages are useful to reason on access control issues at an high abstraction level, but they neglect many of the advanced features of modern access control systems.

Description Logic (DL) is used in [KHP07] as a target formalism for representing a part of XACML. The approach does not take into account many combining algorithms and the not-app and indet decisions. Thus, it only permits reasoning on a set of properties significantly reduced with respect to that supported by the FACPL SMT-based approach. Furthermore, DL reasoners support the verification of structural properties of policies but suffer from the same scalability issues as the MTBDD-based reasoners.

Answer Set Programming (ASP) is used in [AHLM10, RRNN12] for encoding XACML and enabling verification of structural properties that are similar to the `complete` one defined in Section 4.4.2. This approach however suffers from some drawbacks due to the nature of ASP. In fact, differently from SMT, ASP does not support quantifiers and various theories like datatype and arithmetic. Some seminal extensions of ASP to "Modulo Theories" have been proposed, but, to the best of our knowledge, no effective solver like the Z3 one used by FACPL is available. Similarly, the work in [HB08] exploits the SAT-based tool Alloy [Jac02] to detect inconsistencies in XACML policies. However, as outlined in [ACC14] and [FKMT05], Alloy is not able to manage even quite small policies and, more importantly, it cannot reason on arithmetic or any additional theory.

Many other works deal with the analysis of access control policies. For instance, [Bry05] and [ZRG05] exploit, respectively, the process algebra CSP [Hoa85] and the description language RW [ZRG04] to verify XACML policies by means of model checking. Alternatively, in [BF07] it is exploited the model-oriented specification language VDM++ [FLM+05] to define abstract models of XACML policies. The various tools supporting VDM++ permit verifying if a policy enforces the expected requirements. However, besides the use of different off-the-shelf tools, these approaches only focus a limited part

of XACML, e.g. neither all combing algorithms nor obligations, and suffer from scalability issues with respect to the FACPL SMT-based approach.

It is also worth noticing that various analysis approaches using SAT-based tools have been developed for Ponder, see e.g. [BLR03]. These approaches, however, cannot actually be compared with ours due to the numerous differences among Ponder and FACPL.

In summary, all the approaches to the analysis of access control policies mentioned above are deficient in several respects with respect to the FACPL SMT-based approach. Those based on SMT formulae do not address relevant aspects like, e.g. missing attributes, while the other ones do not enjoy the benefits of using SMT, i.e. support of multiple theories and scalable performance.

7.4 Design and Analysis of Autonomic Computing Systems

Autonomic computing has recently been object of numerous research efforts by different communities. In the following, we review the most closely related approaches to the specification and analysis of PSCEL systems.

The proposed specification approaches are multiple and variegated. Some prominent examples are multi-agent systems [vdHlW08], component-based design [MSKC04] and context-oriented programming (COP) [HCN08].

Multi-agent approaches are largely used in the development of complex, self-adaptive software systems (see, e.g., [Win05, BCG07, Das08]). Similarly to PSCEL, they pursue the importance of the knowledge representation and how it is handled for choosing adaptation actions. However, PSCEL components can directly access knowledge repositories of other components (provided that it is allowed by the related policies), while agents need additional message exchanges to access it. In general, the PSCEL predicate-based communications, pattern-matching-based management of knowledge and policy-based adaptation strategies permit more flexible system specifications.

Component-based approaches, e.g. FRACTAL [BCL+06, DL06] and *Helena* [HK14], design systems in terms of basic components that re-organise themselves according to changes of operating conditions. FRACTAL fosters a hierarchical component model that, in addition to standard component-based systems, permits defining systems with a less rigid structure, where indeed components can be without completely fixed boundaries. However, communication among components is still defined via connectors and system adaptation is obtained by adding, removing or modifying components and/or connectors. Communication and adaptation in PSCEL are instead more flexible and, hence, more adequate to deal with highly dynamic systems. *Helena* suffers from similar drawbacks, but, in a way similar to the PSCEL policy automata, it permits dynamic changes of the rules regulating the behaviours of components.

COP-based approaches have been advocated to program autonomic systems [SGP11]. They exploit ad-hoc linguistic constructs to express context-dependent behavioural variations and their run-time activation. The most of the literature on COP is devoted to the design and implementation of concrete programming languages (a comparison can be found in [AHH+09]). All these approaches are however quite different from ours, that instead focusses on distribution and predicate-based communications, and supports a highly dynamic notion of policy-based adaptation strategies.

Concerning the use of policy languages, we can find in the literature some practical ex-

ploitations of policies in the autonomic computing context, like, e.g., in [Chi06, KJT+12]. Specifically, [Chi06] proposes an XML language based on ECA rules to regulate system behaviours. However, it comes without a precise syntax and semantics and aims only at being integrated within IBM frameworks. Instead, [KJT+12] introduces PobSAM, a policy-based formalism that combines an actor-based model, which specifies the computation of system elements, and a configuration algebra, which defines autonomic managers regulating system behaviours. These managers are specified by (a sort of) ECA rules that, in response to operating condition changes, adapt the system configuration. Instead, since PSCEL policy constructs are strictly integrated with programming ones, they permit enforcing more flexible and expressive adaptation strategies, mostly due to the dynamic fulfilment of actions.

Ponder, presented in Section 7.2, has also been applied to various autonomous and pervasive systems. As previously mentioned, the design choice of two separate types of policies, one for authorisation and another for obligations, do not permit flexible specification like those of PSCEL. Furthermore, obligation actions are not fulfilled at runtime.

Among other enforcement approaches for self-adaptation, many proposals address the principles of AOP. Some examples are the AOP extension of FRACTAL [DL06] (which suffers from the same drawbacks of FRACTAL) and AspectK [HNNY08], which enriches the distributed coordination language Klaim [DFP98] with AOP concepts. With respect to AspectK, the AOP support offered by PSCEL is more flexible and, most of all, it supports dynamically fulfilled actions. Additionally, the following work of AspectK in [HNN09] introduces Belnap policies to enforce authorisation controls on process actions. However, differently from PSCEL, it does not support both positive and negative authorisations, and explicit conflict resolution strategies.

Concerning the analysis of AOP-inspired systems, many works address general-purpose AOP programming languages. For example, AspectJ [KHH+01] (i.e., an AOP extension of Java) has been target of various analysis proposals (see, e.g., [RRST05, ZGLZ08]). A more closely work to the analysis of PSCEL is that in [TNN12]. In details, it concerns the analysis of AspectK specifications, hence of processes recursively modified by adaptation actions. The authors exploit communicating pushdown systems [BET03] and, by means of soft-constraint approximations, define a reachability analysis for discovering undesired infinite executions. The approach we propose is instead completely statical and relies only on the abstractions of the flow graph.

7.5 Supporting Tools

The effectiveness of supporting tools is a crucial point for the usability and exploitation of policy languages. In the context of autonomic computing, however, there are neither tools of reference (apart from commercial frameworks by IBM) nor tools implementing many of the languages presented in Section 7.4. Thus, we only focus on access control by comparing the performance of the FACPL tools with respect to that of the most representative tools from the literature. The tests we conducted are based on the CONTINUE case study [Kri03], which has been adopted as a standard benchmark in this context[2].

[2]The tests have been conducted on a MacBook Pro, 2.5 GHz Intel i5 - 8 Gb RAM running OS X El Capitan. The test suite of policies and requests, as well as the test results, is available at http://facpl.sourceforge.net/continue/.

XACML is by now the point-of-reference for industrial access control systems. To the best of our knowledge, the most up-to-date, freely available XACML tool is Balana [WSO15]. Balana manages XACML policies directly in XML and evaluates XACML requests in terms of a visit of the XML files, differently from FACPL that models policies as Java classes. We have compared the evaluation of more than 1.000 requests and obtained that the mean request execution time is 2,14ms for FACPL and 1,85ms for Balana. It must also be considered that Balana requires a set-up time of about 500ms to initially validate XACML policies, while FACPL initialises Java classes in about 200ms.

Concerning the analysis tools, as previously pointed out, the tool closer to ours is that of [ACC14], which relies on the SMT solver Yices [Dut14]. Differently from Z3, Yices does not support datatype theory, which is instead crucial for dealing with a wide range of policy aspects, as e.g. missing and erroneous attributes. To analyse the completeness of the CONTINUE policies, the Yices-based tool requires around 570ms[3], while our Z3-based tool requires around 120ms. Notably, other not SMT-based tools, like, e.g., Margrave [FKMT05], have significantly lower performance when policies scale. In fact, as reported in [ACC14], the increment of the number of possible values for the attributes occurring in the CONTINUE policies prevents Margrave to accomplish the analysis. On the contrary, SMT solvers can also deal with infinite attribute values, as e.g. integers.

Finally, we conclude commenting on the IDEs closer to the FACPL one. To the best of our knowledge, the only similar (freely available) IDEs are the ALFA Eclipse plug-in by Axiomatics (http://www.axiomatics.com/alfa-plugin-for-eclipse.html) and the graphical editor of the Balana-based framework (http://xacmlinfo.org/category/xacml-editor/). However, differently from our IDE, they only provide a high-level language for writing XACML policies. Additionally, ALFA does not provide any request evaluation engine, since the Axiomatics one is a proprietary software.

[3]This value is taken for granted from [ACC14], because the provided CONTINUE implementation only runs on Windows machines. Anyway, their hardware configuration is similar to ours.

Chapter 8

Concluding Remarks

This thesis attempts to provide a formal methodology supporting a principled exploitation of policies in the context of access control and autonomic computing systems. To sum up, the thesis contributions can be detailed as follows

- FACPL: a formal language for the specification and implementation of attribute-based access control systems;
- An automated analysis approach for FACPL: a constraint-based analysis approach for the automatic verification of authorisation and structural properties on FACPL policies;
- PSCEL: a formal language for the specification of autonomic computing systems that relies on FACPL in order to specify and enforce authorisation controls and adaptation strategies;
- A static analysis approach for PSCEL: a constraint-based analysis approach that aims at pointing out the effects of PSCEL policies on system behaviours.

Each ingredient of these contributions has been first formally introduced and then implemented in terms of practical software tools. To testify the effectiveness of the proposed solutions, we have exploited diverse case studies from real application domains.

As future works, we plan, on the one hand, to enhance the functionalities of FACPL in order to support continuative and history-dependent access controls and, on the other hand, to devise additional verification services for PSCEL systems. These contributions

will be first rigorously formalised and then practically implemented as additional functionalities of the software tools we presented. In the following, we briefly argument on these intended research directions.

Continuative and history-dependent access controls in FACPL. Continuative access control consists in the guarantee that when a subject access is in progress, the needed subject rights continue to hold. This type of access control is called Usage Control [LMM10] and has been formalised by various models in the literature. An established one is the UCON model [PS04], which provides a formalisation based on attributes representing information about the access in progress, and on specific obligations used to enforce limitations on the usage of the allowed accesses.

History-dependent access control concerns instead the evaluation of access requests by taking into account the access history. This type of access control permits enforcing dynamic security policies, i.e. an access is secure according to the accesses already authorised. Typical examples are *Chinese Wall* policies [BN89], where confidentiality controls depend on integrity controls addressing the access history. For example, a subject can access the resource A (resp., B) only if she has not accessed the resource B (resp., A) yet. A similar example is the dynamic variant on SoD, i.e. the integrity control of SoD depend on, e.g., the roles previously assumed by a subject.

We thus intend to enhance FACPL with functionalities supporting the definition of such continuative and history-dependent access controls. Specifically, we will appropriately exploit (i) attributes to both refer to the information on current and previous accesses, and (ii) obligations to enforce different types of usage limitations. The SMT-based FACPL analysis will be also tailored to adequately verify properties about these new types of access controls.

Model checking techniques for PSCEL. A model checking approach for PSCEL systems permits verifying that some expected behaviours are guaranteed. Due to the different entities involved in a PSCEL system, defining a formal machinery laying the basis for the exploitation of a model checker is a challenging task. A crucial point is addressing the dynamic evaluation of policies and, hence, their effects on processes. The approximation of the Policy-Flow graph is a starting point to opportunely represent PSCEL systems into a formalism accepted by model checkers.

To carry out this research direction, we will benefit from the work in [DLL+14], where SCEL systems are analysed by means of the Spin model checker [Hol97]. Specifically, it proposes a translation of a lightweight version of SCEL, i.e. it neglects policies and part of the programming constructs, into Promela, the input language of Spin. Our approach will take into account policies and, consequently, the authorisations and additional actions they enforce. As the authorisations of process actions possibly branch the evolution of PSCEL systems, we plan to define properties on the behaviours of PSCEL systems by means of a branching-time logic. Therefore, as Spin is a model checker for linear-time properties, we will exploit the NuSMV model checker [CCGR00] that, besides the support for branching-time properties, effectively combines state-of-the-art symbolic model checking techniques with SAT-based ones.

Bibliography

[ACC14] K. Arkoudas, R. Chadha, and C.-Y. J. Chiang. Sophisticated access control via SMT and logical frameworks. *ACM Trans. Inf. Syst. Secur.*, 16(4):17, 2014.

[ACH+15] D. B. Abeywickrama, J. Combaz, V. Horký, J. Keznikl, J. Kofron, A. Lluch-Lafuente, M. Loreti, A. Margheri, P. Mayer, G. V. Monreale, U. Montanari, C. Pinciroli, P. Tuma, A. Vandin, and E. Vassev. Tools for ensemble design and runtime. In Wirsing et al. [WHKM15], pages 429–448.

[ADCdVF+10] C. A. Ardagna, S. De Capitani di Vimercati, S. Foresti, T. Grandison, S. Jajodia, and P. Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 29(8):848–858, 2010.

[AHH+09] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *Proc. of COP*, pages 6:1–6:6. ACM, 2009.

[AHLM10] G. J. Ahn, H. Hu, J. Lee, and Y. Meng. Representing and reasoning about web access control policies. In *Proc. of COMPSAC*, pages 137–146. IEEE Computer Society, 2010.

[BCD+11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. of CAV*, LNCS 6806, pages 171–177. Springer, 2011.

[BCG07] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

[BCG+12] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. A conceptual framework for adaptation. In *Proc. of FASE*, LNCS 2012, pages 240–254. Springer, 2012.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. B. Stefani. The FRACTAL component model and its support in Java. *Softw., Pract. Exper.*, 36:1257–1284, 2006.

[BDLM12] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies. In *Proc. of WEBIST*, pages 155–160. SciTePress, 2012.

[Ben05] G. Beni. From swarm intelligence to swarm robotics. In *Proc. of SAB*, LNCS 3342, pages 1–9. Springer, 2005.

[BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551, 2003.

[Bet13] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[BF07] J. Bryans and J. S. Fitzgerald. Formal Engineering of XACML Access Control Policies in VDM++. In *Proc. of ICFEM*, LNCS 4789, pages 37–56. Springer, 2007.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical report, The MITRE Corporation, 1977.

[Bis02] M. A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley, 2002.

[BL76] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical report, The MITRE Corporation, 1976.

[BLR03] A. K. Bandara, E. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Proc. of POLICY*, page 26. IEEE Computer Society, 2003.

[BN89] D. F. C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proc. of Security and Privacy*, pages 206–214. IEEE Computer Society, 1989.

[BP09] A. D. Brucker and H. Petritsch. Extending access control models with break-glass. In *Proc. of SACMAT*, pages 197–206. ACM, 2009.

[Bry05] J. Bryans. Reasoning about XACML policies using CSP. In *Proc. of SWS*, pages 28–35. ACM, 2005.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[CBT+15] J. Combaz, S. Bensalem, F. Tiezzi, A. Margheri, R. Pugliese, and J. Kofron. Correctness of service components and service component ensembles. In Wirsing et al. [WHKM15], pages 107–159.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *Int. Jour. on Software Tools for Technology Transfer*, 2:2000, 2000.

[CD15] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0, 2015. http://www.w3.org/TR/xpath.

[CGT89] S. Ceri, G. Gottlob, and T. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[Chi06] S. Chilukuri. Autonomic Computing Policy Language - ACPL, 2006. http://www.ibm.com/developerworks/tivoli/tutorials/ac-spl/.

[CM07] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web*, (3):309–344, 2007.

[CM12] J. Crampton and C. Morisset. Ptacl: A language for attribute-based access control in open systems. In *Proc. of POST*, LNCS 7215, pages 390–409. Springer, 2012.

[CMZ15] J. Crampton, C. Morisset, and N. Zannone. On missing attributes in access control: Non-deterministic and probabilistic attribute retrieval. In *Proc. of SACMAT*, pages 99–109. ACM, 2015.

[CW87] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proc. of Security and Privacy*, pages 184–195. IEEE Computer Society, 1987.

[Das08] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[dB08] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, LNCS 4963, pages 337–340. Springer, 2008.

[dB11] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. of POLICY*, LNCS 1995, pages 18–38. Springer, 2001.

[DeT02] J. DeTreville. Binder, a logic-based security language. In *Proc. Security and Privacy*, pages 105–113. IEEE Computer Society, 2002.

[DFP98] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[DL06] P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *Proc. of Software Composition*, pages 82–97. Springer, 2006.

[DLL+14] R. De Nicola, A. Lluch-Lafuente, M. Loreti, A. Morichetta, R. Pugliese, V. Senni, and F. Tiezzi. Programming and verifying component ensembles. In *Proc. of FPS*, LNCS 8415, pages 69–83. Springer, 2014.

[DLL+15] R. De Nicola, D. Latella, A. Lluch-Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, and A. Vandin. The SCEL Language: Design, Implementation, Verification. In Wirsing et al. [WHKM15], pages 3–71.

[DLPT14] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7:1–7:29, 2014.

[Dut14] B. Dutertre. Yices 2.2. In *Proc. of CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[Eur95] European Parliament and Council. Directive 95/46/EC, 1995. Official Journal L 281 , 23/11/1995 P. 0031 - 0050. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML.

[FAC16] FACPL Website, 2016. http://facpl.sourceforge.net.

[FK92] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proc. of NCSC*, pages 554–563, 1992.

[FKMT05] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. of ICSE*, pages 196–205. ACM, 2005.

[FLM+05] J. S. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005.

[FP715] FP7 EU project. Autonomic Service-Component ENSembles (ASCENS), 2010-2015. Grant no. 257414 - http://www.ascens-ist.eu/.

[GD72] G. S. Graham and P. J. Denning. Protection: Principles and practice. In *Proc. of AFIPS*, pages 417–429. ACM, 1972.

[Gol11] D. Gollmann. *Computer Security (3. ed.)*. Wiley, 2011.

[Har13] D. Hardt. The OAuth 2.0 Authorization Framework, 2013.

[HB08] G. Hughes and T. Bultan. Automated verification of access control policies using a sat solver. *STTT*, 10(6):503–520, 2008.

[HCN08] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[HK14] R. Hennicker and A. Klarl. Foundations for ensemble modeling - The Helena approach. In *Proc. of Specification, Algebra, and Software*, LNCS 8373, pages 359–381. Springer, 2014.

[HKF15] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015.

[HKP+15] M. M. Hölzl, N. Koch, M. Puviani, M. Wirsing, and F. Zambonelli. The ensemble development life cycle and best practices for collective autonomic systems. In Wirsing et al. [WHKM15], pages 325–354.

[HKTT09] M. Hashimoto, M. Kim, H. Tsuji, and H. Tanaka. Policy description language for dynamic access control models. In *Proc. of DASC*, pages 37–42. IEEE, 2009.

[HL12] W. Han and C. Lei. A survey on policy languages in network and security management. *Computer Networks*, 56(1):477–489, 2012.

[HM08] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[HNN09] C. Hankin, F. Nielson, and H. Riis Nielson. Advice from Belnap policies. In *Proc. of CSF*, pages 234–247. IEEE Computer Society, 2009.

[HNNY08] C. Hankin, F. Nielson, H. Riis Nielson, and F. Yang. Advice for coordination. In *Proc. of COORDINATION*, LNCS 5052, pages 153–168. Springer, 2008.

[Hoa85] C. A. R. Hoare. *Commmunicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[Hor01] P. Horn. Autonomic computing: IBM's Perspective on the State of Information Technology, 2001.

[IBM06] IBM. An Architectural Blueprint for Autonomic Computing. June 2006.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[JKS12] X. Jin, R. Krishnan, and R. S. Sandhu. A unified attribute-based access control model covering dac, MAC and RBAC. In *Proc. of DBSec*, LNCS 7371, pages 41–55. Springer, 2012.

[Joi04] Joint SPC. Break-glass: An approach to granting emergency access to healthcare systems, 2004. White paper, Joint NEMA/COCIR/JIRA Security and Privacy Committee (SPC).

[jRE16] jRESP Code Repository, 2016. https://sourceforge.net/p/jresp/code/ci/master/tree/.

[JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. of Security And Privacy*, pages 31–42. IEEE Computer Society, 1997.

[JSSS01] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KFJ03] L. Kagal, T. W. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proc. of POLICY*, page 63. IEEE Computer Society, 2003.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of ECOOP*, LNCS 2072, pages 327–353. Springer, 2001.

[KHP07] V. Kolovski, J. A. Hendler, and B. Parsia. Analyzing web access control policies. In *Proc. of WWW*, pages 677–686. ACM, 2007.

[KJT+12] N. Khakpour, S. Jalili, C. L. Talcott, M. Sirjani, and M. R. Mousavi. Formal modeling of evolving self-adaptive systems. *Sci. Comput. Program.*, 78(1):3–26, 2012.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*, pages 220–242, 1997.

[Kov14] M. Kovac. E-health demystified: An e-government showcase. *IEEE Computer*, 47(10):34–42, 2014.

[Kri03] S. Krishnamurthi. The CONTINUE server (or, how I administered PADL 2002 and 2003). In *Proc. of PADL*, LNCS 2562, pages 2–16. Springer, 2003.

[Lam74] B. W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[LMD13] P. Lalanda, J. A. McCann, and A. Diaconescu. *Autonomic Computing - Principles, Design and Implementation*. Undergraduate Topics in Computer Science. Springer, 2013.

[LMM10] A. Lazouski, F. Martinelli, and P. Mori. Usage control in computer security: A survey. *Computer Science Review*, 4(2):81–99, 2010.

[LMPT14] M. Loreti, A. Margheri, R. Pugliese, and F. Tiezzi. On programming and policing autonomic computing systems. In *Proc. of ISoLA*, LNCS 8802, pages 164–183. Springer, 2014.

[LWQ+09] N. Li, Q. Wang, W. H. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: theory meets practice. In *Proc. of SACMAT*, pages 135–144. ACM, 2009.

[Mas12] M. Masi. *On Authentication and Authorisation Issues in e-Health Systems*. PhD thesis, Università degli Studi di Firenze, 2012.

[MDS14] S. Marinovic, N. Dulay, and M. Sloman. Rumpole: An introspective break-glass access control language. *ACM Trans. Inf. Syst. Secur.*, 17(1):2:1–2:32, 2014.

[MESW01] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. RFC 3060 (Proposed Standard), February 2001.

[MMPT13a] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation. A Practical Approach. In *Proc. of WSFM*, LNCS 8379, pages 85–105. Springer, 2013.

[MMPT13b] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. On a formal and user-friendly linguistic approach to access control of electronic health data. In *Proc. of HEALTHINF*. SciTePress, 2013.

[MMPT16] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. A rigorous framework for specification, analysis and enforcement of access control policies. *Submitted for publication*, 2016. Technical Report available at http://local.disia.unifi.it/wp_disia/2016/wp_disia_2016_05.pdf.

[MPT12] M. Masi, R. Pugliese, and F. Tiezzi. Formalisation and Implementation of the XACML Access Control Mechanism. In *Proc. of ESSoS*, LNCS 7159, pages 60–74. Springer, 2012.

[MPT13] A. Margheri, R. Pugliese, and F. Tiezzi. Linguistic abstractions for programming and policing autonomic computing systems. In *Proc. of UIC/ATC*, pages 404–409. IEEE Computer Society, 2013.

[MPT15] A. Margheri, R. Pugliese, and F. Tiezzi. On Properties of Policy-Based Specifications. In *Proc. of WWV*, EPTCS 188, pages 33–50, 2015.

[MRNNP16a] A. Margheri, H. Riis Nielson, F. Nielson, and R. Pugliese. Design, analysis and implementation of policy-based self-adaptive computing systems. *Submitted for publication*, 2016. Technical Report available at http://facpl.sourceforge.net/research/PSCEL-TR.pdf.

[MRNNP16b] A. Margheri, H. Riis Nielson, F. Nielson, and R. Pugliese. Towards static analysis of policy-based self-adaptive computing systems. In *Proc. of ISoLA*, *To Appear*. Springer, 2016.

[MSKC04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[MVK+15] P. Mayer, J. Velasco, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bures. The Autonomic Cloud. In Wirsing et al. [WHKM15], pages 495–512.

[NIS09] NIST. A survey of access control models, 2009. http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.

[NIS14] NIST. Guide to attribute based access control (ABAC) definitions and considerations, 2014. http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf.

[OAS05] OASIS XACML TC. Security Assertion Markup Language (SAML) version 2.0, 2005. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[OAS13] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 3.0 , January 2013. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[OGCD10] R. O'Grady, R. Groß, A. L. Christensen, and M. Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Auton. Robots*, 28(4):439–455, 2010.

[PBMD15] C. Pinciroli, M. Bonani, F. Mondada, and M. Dorigo. Adaptation and Awareness in Robot Ensembles: Scenarios and Algorithms. In Wirsing et al. [WHKM15], pages 471–494.

[PRI16] PRIN Italian project. Compositionality, Interaction, Negotiation, Autonomicity (CINA), 2013-2016. Grant no. 2010LHT4KM - http://sysma.imtlucca.it/cina/doku.php.

[Pro07] Project InterLink. http://interlink.ics.forth.gr, 2007.

[PS04] J. Park and R. S. Sandhu. The UCON_{ABC} usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[PSC16] PSCEL Website, 2016. http://facpl.sourceforge.net/pscel/.

[RLB+09] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of XACML policies. In *Proc. of SACMAT*, pages 63–72. ACM, 2009.

[RRN14] C. D. P. K. Ramli, Nielson H. Riis, and F. Nielson. The logic of XACML. *Sci. Comput. Program.*, 83:80–105, 2014.

[RRNN12] C. D. P. K. Ramli, H. Riis Nielson, and F. Nielson. XACML 3.0 in Answer Set Programming. In *Proc. of LOPSTR*, LNCS 7844, pages 89–105. Springer, 2012.

[RRST05] X. Ren, B. G. Ryder, M. Störzer, and F. Tip. Chianti: a change impact analysis tool for Java programs. In *Proc. of ICSE*, pages 664–665. ACM, 2005.

[Sal74] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, 1974.

[San93] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SdV00] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Proc. of FOSAD*, LNCS 2171, pages 137–196. Springer, 2000.

[SGP11] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-Oriented Programming: A Programming Paradigm for Autonomic Systems. *CoRR*, abs/1105.0069, 2011.

[SGP12] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[Slo94] M. Sloman. Policy driven management for distributed systems. *J. Network Syst. Manage.*, 2(4):333–360, 1994.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[TdHRZ15] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. Analysis of XACML policies with SMT. In *Proc. of POST*, LNCS 9036, pages 115–134. Springer, 2015.

[TDLS09] K. P. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A policy system for autonomous pervasive environments. In *Proc. of ICAS*, pages 330–335. IEEE Computer Society, 2009.

[The13] The Article 29 Data Protection WP, 2013. http://ec.europa.eu/justice/data-protection/article-29/.

[TK06] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proc. of SACMAT*, pages 160–169. ACM, 2006.

[TNN12] M. Terepeta, H. Riis Nielson, and F. Nielson. Recursive advice for coordination. In *Proc. of COORDINATION*, LNCS 7274, pages 137–151. Springer, 2012.

[UBJ+04] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and J. S. Aitken. Kaos policy management for semantic web services. *IEEE Intelligent Systems*, 19(4):32–41, 2004.

[vdHlW08] W. van der Hoek and M. l. Wooldridge. Multi-agent systems. In *Handbook of Knowledge Representation*, pages 887–928. Elsevier, 2008.

[Ver02] D. C. Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2):20–26, 2002.

[WHKM15] M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, editors. *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*. LNCS 8998. Springer, 2015.

[Win05] M. Winikoff. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer, 2005.

[WL93] T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Computer Security*, 2(2-3):107–136, 1993.

[WSO15] WSO2. Balana: Open source XACML implementation, 2015. https://github.com/wso2/balana.

[Xte16] Xtext - Language Engineering Made Easy, 2016. https://eclipse.org/Xtext/.

[YPG00] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. RFC 3060 (Proposed Standard), 2000.

[ZGLZ08] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. of ICSM*, pages 87–96. IEEE Computer Society, 2008.

[ZRG04] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proc. of FMSE*, pages 56–65. ACM, 2004.

[ZRG05] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. of ISC*, LNCS 3650, pages 446–460. Springer, 2005.

List of Tables

2.1 Requirements for the e-Health case study 19
2.2 Requirements for the robot-swarm case study 20
2.3 Requirements for the autonomic cloud case study 21

3.1 Syntax of FACPL 26
3.2 Auxiliary syntax for FACPL responses 27
3.3 Correspondence between syntactic and semantic domains 33
3.4 Semantics of FACPL expressions 35
3.5 Combination matrices for $\otimes_{\mathsf{alg}}$ operators 38
3.6 Definition of the $isFinal_{\mathsf{alg}}(res)$ predicate 39

4.1 Constraint syntax 56
4.2 Semantics of constraints 57
4.3 Constraint combination strategies for the combining algorithms 60
4.4 Type inference rules for FACPL expressions 72

5.1 Programming constructs 81
5.2 Policy constructs 83
5.3 Auxiliary functions 85
5.4 Semantics of policy constructs 86
5.5 Combination matrices for $\otimes_{\mathsf{alg}}$ operators 86
5.6 Semantics of processes 88
5.7 Evaluation of predicates 89
5.8 Evaluation of actions 89
5.9 Semantics of programming constructs (1 of 3): authorisation rules (1 of 2) . 92
5.10 Semantics of programming constructs (1 of 3): authorisation rules (2 of 2) . 93
5.11 Matching rules 94
5.12 Semantics of programming constructs (2 of 3): execution rules (1 of 2) . . . 95
5.13 Semantics of programming constructs (2 of 3): execution rules (2 of 2) . . . 96
5.14 Semantics of programming constructs (3 of 3): system transition rules . . . 97
5.15 Robot-swarm case study: policy automaton transition conditions 101

6.1 Constraint Syntax 115
6.2 Function for determining potential executing components 118
6.3 Translation function for rule targets 119

7.1 FACPL vs. XACML on the e-Health case study 134
7.2 Comparison of a relevant set of policy languages 135

List of Figures

2.1 Policy-based evaluation process 11
2.2 SCEL Autonomic Component 16
2.3 Abstract representation of SCEL Autonomic Component Ensembles 17
2.4 E-Health case study: e-Prescription service protocol 18
2.5 Robot-swarm case study: disaster scenario 20
2.6 Autonomic cloud case study: high-load scenario 21

3.1 FACPL evaluation process 24
3.2 FACPL toolchain 44
3.3 FACPL Eclipse plug-in 46

5.1 Robot-swarm case study: policy automaton 101
5.2 Architecture of Java PSCEL components 103
5.3 PSCEL Eclipse plug-in 105

6.1 Autonomic cloud case study: Policy-Flow graphs 124
6.2 Autonomic cloud case study: tool generated Policy-Flow graphs 127