UNIVERSITÀ DEGLI STUDI FIRENZE

DISIA
DIPARTIMENTO DI STATISTICA, INFORMATICA, APPLICAZIONI "GIUSEPPE PARENTI"

A Rigorous Framework for Specification, Analysis and Enforcement of Access Control Policies

Andrea Margheri, Massimiliano Masi,
Rosario Pugliese, Francesco Tiezzi

DISIA WORKING PAPER
2016/05

© Copyright is held by the author(s).

A Rigorous Framework for Specification, Analysis and Enforcement of Access Control Policies

ANDREA MARGHERI, Università degli Studi di Firenze, Università di Pisa
MASSIMILIANO MASI, Tiani "Spirit" GmbH
ROSARIO PUGLIESE, Università degli Studi di Firenze
FRANCESCO TIEZZI, Università di Camerino

Access control systems are widely used means for the protection of computing systems. They are defined in terms of access control policies regulating the accesses to system resources. In this paper, we introduce a formally-defined, fully-implemented framework for the specification, analysis and enforcement of attribute-based access control policies. The framework rests on FACPL, a formal language with a compact, yet expressive, syntax that permits expressing real-world access control policies. By relying on the FACPL denotational semantics, we devise a constraint formalism that uniformly represents access control policies in terms of SMT formulae, whose solvers provide effective and efficient analysis. To this aim, we introduce and formalise a set of properties that permit assessing the authorisations enforced by policies and understanding the relationships among them. Our analysis approach explicitly addresses the role of missing attributes, erroneous values and obligations, that are crucial in policy evaluation and are instead overlooked in other proposals. The framework is supported by Java-based tools that allow access control system developers to use formally-defined functionalities without requiring them to be familiar with formal methods.

1. INTRODUCTION

Nowadays computing systems have pervaded every daily activity and prompted the proliferation of a variety of innovative services and applications. These modern distributed systems manage a huge amount of data that, due to its importance and societal impact, has brought out security issues of paramount importance. Controlling the access to system resources is thus crucial to prevent unauthorised accesses that could jeopardise trustworthiness of data.

This has prompted increasing research interest towards access control systems, which are the first line of defense for the protection of computing systems. They are defined by *rules* that establish under which conditions a subject's *request* for 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.

Access control is a broad field, covering several different approaches, using different technologies and involving various degrees of complexity. Since the first applications in operating systems, to the more recent ones in distributed systems, many access control approaches have been proposed. Traditional approaches are based on the identity of subjects, either directly – e.g., Access Control Matrix [Lampson 1974] – or through predefined features, such as roles or groups – e.g., Role-Based Access Control (RBAC [Ferraiolo and Kuhn 1992]). These approaches are however inadequate for dealing with modern distributed systems, as they suffer from scalability and interoperability issues. Moreover, they cannot easily encompass information representing the evaluation context, as e.g. system status or current time. An alternative approach that permits to overcome these problems is Attribute-Based Access Control (ABAC) [Hu et al. 2015]. Here, the rules are based on *attributes*, which represent 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 relevant to the rules at hand. Thus, ABAC permits defining fine-grained, flexible and context-aware access control rules that are expressive enough to uniformly represent all the other approaches [Jin et al. 2012]. Attribute-based rules are typically hierarchically structured and paired

with strategies for resolving possible conflicting authorisation results. These structured specifications are called *policies*; from this name derives the terminology Policy-Based Access Control (PBAC) [NIST 2009], sometimes used in place of ABAC.

Many languages have been proposed for the specification of access control policies (see, e.g., [Han and Lei 2012] for a survey). Among the proposed languages, in the authors' knowledge, the OASIS standard eXtensible Access Control Markup Language (XACML) [OASIS XACML TC 2013] is the best-known one. Due to its XML-based syntax and the advanced access control features it provides, XACML is commonly used in many real-world systems, e.g., in service-oriented ones. However, the management of access control policies is in practice cumbersome and error-prone, and should be supported by rigorous analysis techniques. Unfortunately, XACML is generally acknowledged as lacking of a formally defined semantics (see, e.g., [Rao et al. 2009; Crampton and Morisset 2012; Ramli et al. 2012; Arkoudas et al. 2014]), which makes it difficult the specification and realisation of analysis techniques.

To cope with these difficulties, we propose a full-fledged, formally-defined framework, based on the Formal Access Control Policy Language (FACPL), supporting developers in the specification, analysis and enforcement of access control policies.

The FACPL-based Access Control Framework

Our framework relies on the FACPL language and is built on the top of solid formal foundations. FACPL defines a core, yet expressive syntax for high-level 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. Evaluation of FACPL policies is formalised by a denotational semantics, which clarifies 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).

The analysis functionalities offered by our framework permits verifying *authorisation properties* and *structural properties*. The former properties permit to statically reason on the result of the evaluation of a policy with respect to a specific request, by also considering additional attributes that can be possibly introduced in the request at run-time and that might lead to unexpected authorisations. Instead, the latter properties permit to statically reason on the whole set of authorisations enforced by one or more policies and can be exploited, e.g., to implement maintenance and *change-impact analysis* [Fisler et al. 2005] techniques.

The verification of these properties requires extensive checks on large (possibly infinite) amounts of requests, hence tool support is essential. As no off-the-shelf analysis tool directly accepts FACPL specifications as an input, our framework exploits a constraint formalism that uniformly represents policy elements and enables automatic analysis. The constraint formalism we introduce is based on Satisfiability Modulo Theories (SMT) formulae, that is formulae defining satisfiability problems involving multiple theories, e.g. boolean and linear arithmetic ones. The relevant progress made in the development of automatic SMT solvers has led SMT to be extensively employed in diverse analysis applications [de Moura and Bjørner 2011], even for access control policies [Arkoudas et al. 2014; Turkmen et al. 2015]. In practice, SMT-based approaches are more effective than many other ones, like e.g. decision diagrams [Fisler et al. 2005] or description logic [Kolovski et al. 2007]. Notably, we formally prove the correspondence between the FACPL semantics and that of its constraint-based representation.

Our framework is equipped with a Java-based software *toolchain*. The key software tool is an Eclipse-based IDE that offers a tailored developing and analysis environment for FACPL-based policies. Specifically, it helps developers in the tasks of policy

specification, analysis and enforcement by providing, e.g., static checks on FACPL code and automatic generation of runnable SMT-based code and Java code. The evaluation of SMT-based code relies on the Z3 solver [de Moura and Bjørner 2008], while the enforcement of policies is made via an expressly developed Java library.

Contributions

The main contribution of this paper is the development of a comprehensive methodology supporting the whole life-cycle of access control policies, from their specification and analysis to their enforcement. Each ingredient of our methodology is formally introduced in this paper, together with its tools implementation. Our approach allows access control system developers to use formally-defined functionalities without requiring them to be familiar with formal methods. Indeed, our aim is to propose and deploy a compact, yet expressive, language whose formal foundations enable tool-supported analysis techniques, rather than to supersede XACML or face its semantic issues.

Additional contributions of this paper can be summarised as follows.

— The FACPL semantics manages missing attributes in a way similar to [Crampton and Morisset 2012] and extends it with explicit error management.
— The formalisation of combining algorithms extends that of [Li et al. 2009] with explicit combination of obligations and with different fulfilment strategies.
— The authorisation properties explicitly take into account the non-monotonicity issue of policy evaluation [Tschantz and Krishnamurthi 2006] by appropriately employing the request extensions set of [Crampton et al. 2015] for property formalisation.
— The main structural properties of [Fisler et al. 2005] and [Kolovski et al. 2007] are uniformly formalised in terms of policy semantics.
— The constraint formalism defines a low-level tool-independent representation of attribute-based policies that is capable to deal with all aspects of policy evaluation.

This paper is a revised and extended version of [Masi et al. 2012; Margheri et al. 2015]. Besides significant revisions and extensions of syntax and semantics of the policy language (we refer to Section 9 for a detailed comparison) this paper proposes a complete development methodology for access control policies. Most of all, differently from previous works, we introduce a constraint-based representation of policies enabling the verification of a variety of properties through SMT solvers.

Summary of the rest of the paper. In Section 2 we overview the FACPL evaluation process. In Section 3 we introduce an e-Health case study we use throughout the paper. In Section 4 we present the syntax of FACPL and its informal semantics, together with the FACPL-based specification of the case study. In Section 5 we formally define the FACPL semantics. In Section 6 we introduce the constraint formalism and the representation it enables of FACPL policies. In Section 7 we introduce various properties for access control policies and their verification via SMT solvers. In Section 8 we outline the Java-based software toolchain. In Section 9 we discuss the closest related work and, finally, in Section 10 we conclude and touch upon directions for future work. Appendixes A, B and C report, respectively, the definitions of combining algorithms and constraint combinations, and the proofs of the formal results.

2. THE FACPL EVALUATION PROCESS

The FACPL evaluation process of (access control) policies and requests is shown in Figure 1. It defines the interactions, leading to the final authorisation decision, among three key components: the Policy Repository (PR), the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP). These entities and their interactions were introduced in [Yavatkar et al. 2000] to define the evaluation process of policy-based

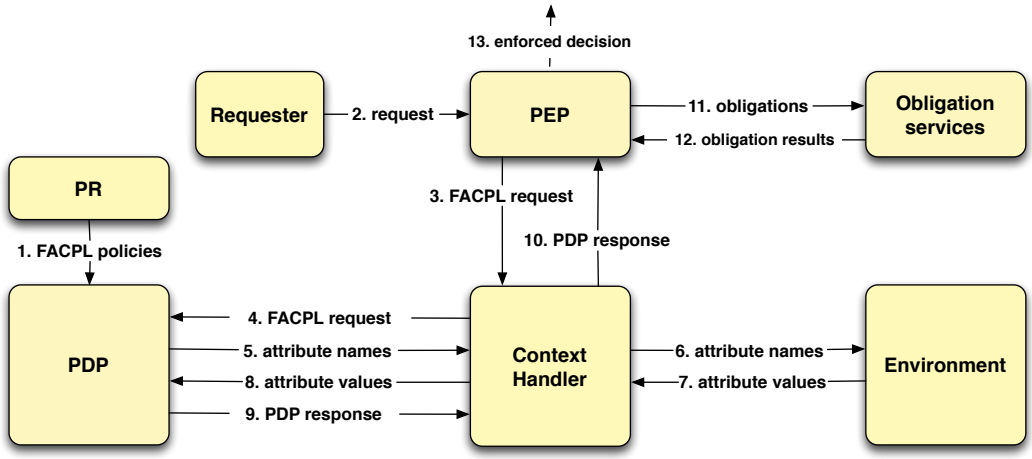

Fig. 1. The FACPL evaluation process

systems. Each policy language, e.g. XACML, has then tailored them according to its specific features.

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), which then decides if the access can be granted.

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 representing arbitrary information relevant for evaluating the access request) forming a FACPL request (step 3). PEPs can have many different forms, e.g. a gateway or a Web server. Therefore, this encoding allows policies and requests to be written and evaluated independently from their specific nature.

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. The PDP response (steps 9-10) contains an authorisation *decision* and possibly some *obligations*.

The decision is one among permit, deny, not-app and indet. The meaning of the first two decisions is obvious, the third one means that there is no policy that applies to the request and the latter one means that some errors have occurred during the evaluation. Policies can automatically manage these errors by using operators that combine, according to different strategies, indet decisions with the others.

Obligations are instead additional actions connected to the access control system that must be discharged by the PEP through appropriate *obligation services* (steps 11-12). 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.

3. AN E-HEALTH CASE STUDY

The case study we consider throughout this paper concerns the provision of e-Health services for exchanging private health data. To improve the effectiveness of healthcare, e-Health services aim at allowing healthcare professionals (such as doctors, nurses,

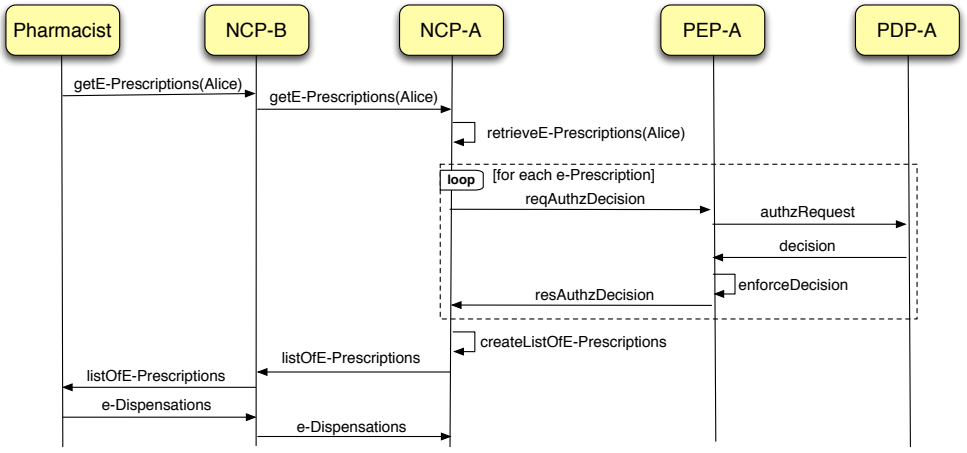

Fig. 2. e-Prescription service protocol

pharmacists, etc.) to remotely access patients data. In order to preserve confidentiality and integrity of such data, we control these accesses by means of FACPL policies.

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 epSOS[1], 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 [European Parliament and Council 1995; The Article 29 Data Protection WP 2013] 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 [Kovac 2014]. As a case study for this paper, 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. The e-Prescription service helps pharmacists in country B to retrieve (and properly convert) e-Prescriptions from country A; this is due to 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 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.

By looking at the epSOS specifications, we can deduce a set of business requirements concerning 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 I, we report in a *closed-world* form, i.e. everything not re-

[1]The Large scale pilot epSOS (Smart Open Services for European Patients), http://www.epsos.eu

Table I. Requirements for the e-Prescription service

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

ported has to be forbidden, the self-explanatory requirements we focus on in the rest of the paper when dealing with the case study. The first three requirements deal with access restrictions, while the others deal with additional functionalities that sophisticated access control systems, like the one we introduce, can provide.

4. THE FACPL LANGUAGE

In this section we present FACPL, the language we propose for defining high-level access control policies and requests. First, we introduce its syntax (Section 4.1). Then, we informally explain the semantics of its linguistic constructs (Section 4.2) and employ them to implement the access control system of the e-Health case study (Section 4.3).

4.1. Syntax

The syntax of FACPL is reported in Table II. 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.

Table II. Syntax of FACPL

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

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.

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. For simplicity sake, the expression syntax does not take types explicitly into account (because they are not relevant in this setting and, statically, their treatment would be standard). However, at evaluation-time an error will be returned when operator arguments are of unexpected types; there are then specific operators that can manage and mask errors. Notably, FACPL supporting tools implement a type inference system that statically avoids writing expressions that always produce errors. Moreover, the syntax of expressions accepted by the tools can be extended with additional operators (see Section 8 for further details).

An $attribute\ name$ indicates 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 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.

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 (as e.g. the p-over$_\delta$ algorithm stating that 'decision permit takes precedence over the others') and can be specialised by choosing different strategies for the fulfilment of obligations (as 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* $[\, Effect \;\; ObType \;\; PepAction(Expr^*)\,]$ 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 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. *Multivalued 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 III.

The two-stage evaluation process described in Section 2 produces two different kinds of responses: *PDP responses* and *decisions* (i.e. responses by the PEP). The former ones,

Table III. Auxiliary Syntax for FACPL responses

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

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 II 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 $\langle Decision \rangle$.

4.2. Informal Semantics

We now informally explain how the FACPL linguistic constructs are dealt with in the evaluation process of access requests described in Section 2. 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 $\perp$ is returned. This value can be exploited to enforce different strategies for managing the absence of attributes. In details, dealing with $\perp$ 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 $\perp$ in a way similar to value false allows attributes to be missing without always generating errors.

The evaluation of expressions takes into account the types of the operators' arguments, and possibly returns the special values $\perp$ 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 $\perp$ and none is error, $\perp$ 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, $\perp$ if at least one operand is $\perp$ 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 $\perp$ and error. Instead, the unary operator not only swaps values true and false and leaves $\perp$ 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 sequence 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 [Li et al. 2009] 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 evaluation performance of a sequence of several policies.

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.

4.3. Policies for the e-Health case study

We now use the FACPL linguistic abstractions to formalise the requirements for the e-Health case study reported in Table I. These rules are meant to prevent unauthorised access to patient data and hence to ensure their confidentiality and integrity. The specification of this 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 (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. This, because of the hierarchical processing of FACPL elements, is enough to ensure that the rule will only be applied to e-Prescriptions.

Requirement (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 (3) only differs from the second one for the role value.

These three rules, modelling Requirements (1), (2) and (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 $\text{p-over}_{\text{all}}$ algorithm as combining strategy seems a natural choice. Let thus Policy (1) 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) ] }
```
(1)

Policy (1) reports not only access controls but also an obligation formalising Assumption (4) about the logging of each authorised access, i.e. all the permit ones. The arguments of the obligation action are separated by commas to increase their readability.

[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.

Let us now consider a FACPL request and evaluate it with respect to Policy (1). 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 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 multivalued 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 (1), 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 I), 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 (1) 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 (1) 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 (2) be defined as

$$\begin{aligned}&\{\,\mathsf{p\text{-}over}_{\mathsf{all}} \\ &\quad \mathsf{policies}: \{\ldots \mathit{Policy}\ (1) \ldots\}\ \ (\mathsf{deny}) \\ &\quad \mathsf{obl}: [\,\mathsf{deny}\ \ \mathsf{m}\ \ \mathsf{mailTo}(\mathsf{resource/patient\text{-}mail}, \text{“Data request by unauthorised subject”})\,] \\ &\qquad\ [\,\mathsf{permit}\ \ \mathsf{o}\ \ \mathsf{compress}()\,]\,\}\end{aligned} \tag{2}$$

Policy (2) reports two obligations formalising, respectively, the last two requirements of Table I: (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 (2) can be used as a basis for the definition of the *patient informed consent* (see Section 3). 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 (2), 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 (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].

As shown before, it could be challenging to identify unexpected authorisations and to determine whether policy fixes affect authorisations that should not be altered. The combination of a large number of complex policies is indeed an error-prone task that has to be supported with effective analysis techniques. Therefore we equip FACPL with a formal semantics and then define a constraint-based analysis providing effective supporting techniques for the verification of properties on policies.

5. FACPL FORMAL SEMANTICS

In this section, we present the formal semantics of FACPL by formalising the evaluation process introduced in Section 2 and detailed in Section 4.2. 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.

To this purpose, we specify a family of semantic functions mapping each syntactic category to a specific semantic domain. These functions are inductively defined on the FACPL syntax through appropriate semantic clauses following a 'point-wise' style.

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 II and III (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 IV (the missing semantic domains coincide with the corresponding syntactic ones).

In the rest of this section we detail the semantics of requests (Section 5.1), PDP (Sections 5.2 and 5.3), PEP (Section 5.4), Policy Authorisation System (Section 5.5) and we conclude with some properties of the semantics (Section 5.6).

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

Table IV. Correspondence between syntactic and semantic domains

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

5.1. Semantics of Requests

The meaning of a request[5] is a function of the set $R \triangleq Name \rightarrow (Value \cup 2^{Value} \cup \{\bot\})$, that is a total function that maps attribute names to either a literal value, or a set of values (in case of multivalued 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 function $\mathcal{R} : Request \rightarrow R$, 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 multivalued 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 $V \in 2^{Value}$.

5.2. Semantics of the 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 V we report (an excerpt of) the clauses defining the function $\mathcal{E} : Expr \rightarrow (R \rightarrow Value \cup 2^{Value} \cup \{\mathsf{error}, \bot\})$ modelling the semantics of expressions. This means that the semantics of an expression is a function of the form $R \rightarrow Value \cup 2^{Value} \cup \{\mathsf{error}, \bot\}$ 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 incorrect type).

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

[5]For simplicity 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 (steps 5-8 in Figure 1).

Table V. 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 $i \neq j$)

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

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

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

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

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

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

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

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

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 V 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, the clauses of operators and and or possibly mask these special values by implementing the behaviour informally described in Section 4.2.

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 $\mathcal{O} : Obligation \rightarrow (R \rightarrow FObligation \cup \{\mathsf{error}\})$ defined by the following clause

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

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

$$\epsilon|_e \ = \ \epsilon$$

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

The semantics of policy sets relies on the semantics of combining algorithms. Indeed, as detailed in Section 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

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

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, i.e.

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

5.3. Semantics of 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

Table VI. Combination matrix for the $\otimes$p-over operator (res_1 and res_2 indicate the first and the second argument, respectively)

$res_1 \backslash^{res_2}$	$\langle$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

corresponding semantic operator is identified as $\otimes$alg and is defined by means of a two-dimensional matrix that, given two PDP responses, calculates the resulting combined response. For instance, Table VI reports the combination matrix for the $\otimes$p-over operator. Basically, the matrix specifies the precedences among the permit, deny, not-app and indet decisions, and shows 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. All other combining algorithms described in Section 4.2, and possibly many others, can be defined in the same manner (see Appendix A). 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

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

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 VII; 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. In case of the first-app algorithm, instead, all decisions except not-app are final since they represent the fact that the first applicable policy has been already found. Both consensus

[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.

Table VII. Definition of the $isFinal_{\mathsf{alg}}(res)$ predicate

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

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

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

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

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.

5.4. Semantics of the 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 $(\!(\ )\!) : FObligation^* \to \{\mathsf{true}, \mathsf{false}\}$ 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

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

where $\Downarrow \mathsf{ok}$ denotes that the discharge of the action $pepAct(w^*)$ succeeded. Since the set of action identifiers is intentionally left unspecified (see Section 4.1), the definition of predicate $\Downarrow \mathsf{ok}$ is hence unspecified too (in other words, the syntactic domain $PepAction$ is a parameter of the syntax, while $\Downarrow \mathsf{ok}$ 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

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

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

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

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

5.5. Semantics of the Policy Authorisation System

The semantics of a Policy Authorisation System 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 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 Policy Authorisation System when given the request req in input.

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 [Tschantz and Krishnamurthi 2006] that precisely characterise the expressiveness of a policy language.

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

THEOREM 5.1 (DETERMINISTIC AND TOTAL SEMANTICS). *For all* $pas \in PAS$, $req \in Request$ *and* $dec, dec' \in Decision$, *it holds that*

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

PROOF. It boils down to show that $\mathcal{P}as$ is a total function (see Appendix C.1). □

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*, e.g. a request that is granted may not be granted anymore if new attributes are introduced in the request, nor *monotonicity*, e.g. the introduction of a new policy in a combination may modify a permit decision to a different one. These properties are ensured neither by XACML nor by other policy languages featuring deny rules and combining algorithms like those we have shown.

Finally, we 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 5.2. *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 straightforwardly derives from the semantics of FACPL expressions and from Theorem 5.1 (see Appendix C.1). □

6. FACPL CONSTRAINT-BASED REPRESENTATION

The analysis of access control policies, like those expressible in FACPL, is essential for ensuring confidentiality and integrity of system resources. However, the hierarchical structure of policies, the presence of conflict resolution strategies and the intricacies deriving from the many involved controls complicate the analysis. From a more practical point of view, no off-the-shelf analysis tool directly accepts a FACPL specification

Table VIII. Constraints syntax

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

in input. Therefore, to enable the analysis of FACPL policies through well-established and efficient tools, we propose and exploit a constraint formalism that permits, on the one hand, to uniformly represent policies and, on the other hand, to perform extensive checks of (a possibly infinite number of) requests.

The constraint-based representation we propose 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. The SMT-based approach is supported by the relevant progress made in the development of automatic SMT solvers (e.g., Z3 [de Moura and Bjørner 2008], CVC4 [Barrett et al. 2011], Yices [Dutertre 2014]), which make SMT formulae to be extensively employed in diverse analysis applications [de Moura and Bjørner 2011].

This section introduces our constraint-based representation of FACPL policies, while the analysis it enables is presented in Section 7. In particular, we first introduce the constraint formalism (Section 6.1), then we formalise how to generate constraints from FACPL policies (Section 6.2) and we finally conclude with some examples of constraints obtained from our e-Health case study (Section 6.3).

6.1. 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[7]. 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 $\perp$ 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 VIII (the nonterminals $Value$ and $Name$ are defined in Table II). 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 IV, 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 $\mathcal{C} : Constr \to (R \to Value \cup 2^{Value} \cup \{\mathsf{error}, \perp\})$ inductively defined by the clauses in Table IX (the clauses for $>$, $\in$, $-$, $*$ 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 $\perp$ 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

[7]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 paper, we refer to constraint variables as attribute names.

Table IX. 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 $i \neq j$)

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

$$\mathcal{C}[\![\texttt{isMiss}(c)]\!]r = \begin{cases} \textsf{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \bot \\ \textsf{false} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![\texttt{isErr}(c)]\!]r = \begin{cases} \textsf{true} & \texttt{if } \mathcal{C}[\![c]\!]r = \textsf{error} \\ \textsf{false} & \texttt{otherwise} \end{cases}$$

$$\mathcal{C}[\![\texttt{isBool}(c)]\!]r = \begin{cases} \textsf{true} & \texttt{if } \mathcal{C}[\![c]\!]r \in \{\textsf{true}, \textsf{false}\} \\ \textsf{false} & \texttt{otherwise} \end{cases}$$

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

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

$$\mathcal{C}[\![c_1\ \wedge\ c_2]\!]r = \begin{cases} \textsf{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \textsf{true} \texttt{ and } \mathcal{C}[\![c_2]\!]r = \textsf{true} \\ \textsf{false} & \texttt{otherwise} \end{cases}$$

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

$$\mathcal{C}[\![c_1\ \vee\ c_2]\!]r = \begin{cases} \textsf{true} & \texttt{if } \mathcal{C}[\![c_1]\!]r = \textsf{true} \texttt{ or } \mathcal{C}[\![c_2]\!]r = \textsf{true} \\ \textsf{false} & \texttt{otherwise} \end{cases}$$

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

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

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

operators defined in Table V (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.

6.2. 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 $\mathcal{T}.$, 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

$$\begin{array}{lll} \mathcal{T}_E\{\!|v|\!\} = v & \mathcal{T}_E\{\!|n|\!\} = n & \mathcal{T}_E\{\!|\mathsf{not}(expr)|\!\} = \dot{\neg}\mathcal{T}_E\{\!|expr|\!\} \\ \multicolumn{3}{l}{\mathcal{T}_E\{\!|\mathsf{op}(expr_1, expr_2)|\!\} = \mathcal{T}_E\{\!|expr_1|\!\}\ \ \mathtt{getCop}(\mathsf{op})\ \ \mathcal{T}_E\{\!|expr_2|\!\}} \end{array} \tag{T-1}$$

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 $\mathtt{getCop}()$, 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

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

Hence, a sequence of obligations corresponds to the conjunction of the constraints representing each obligation. When translating a single obligation, predicates $\mathtt{isMiss}()$ and $\mathtt{isErr}()$ 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-tuples of constraints 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

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

(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 $\mathsf{not\text{-}app}$ constraint is satisfied when the constraint corresponding to the target expression evaluates to false or to $\bot$. Instead, the negation

Table X. Constraint combination strategy for the p-over algorithm

$$
\begin{array}{rcl}
\mathsf{p\text{-}over}(A,B) & = & \langle \mathsf{permit} : A\downarrow_p \vee B\downarrow_p \\
& & \quad \mathsf{deny} : (A\downarrow_d \wedge B\downarrow_d) \vee (A\downarrow_d \wedge B\downarrow_n) \vee (A\downarrow_n \wedge B\downarrow_d) \\
& & \quad \mathsf{not\text{-}app} : A\downarrow_n \wedge B\downarrow_n \\
& & \quad \mathsf{indet} : (A\downarrow_i \wedge \neg B\downarrow_p) \vee (\neg A\downarrow_p \wedge B\downarrow_i)\rangle
\end{array}
$$

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 $\neg\ \mathtt{isBool}(\mathcal{T}_E\{\!|expr|\!\})$, we introduce $\neg\ \mathtt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})$ because we want to exclude that $\mathcal{T}_E\{\!|expr|\!\} = \perp$ (otherwise, we would fall in the case of decision not-app).

The clause for policy sets is as follows

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

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.

Combining algorithms are dealt with by the function $\mathcal{T}_A : Alg \times Policy^+ \rightarrow PCT$ 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

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

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. By way of example, Table X reports the combination of two constraint tuples, say A and B, according to the p-over algorithm. The combinations for the remaining algorithms are in Appendix B. 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. $\{Alg\ \ \mathsf{target} : \mathsf{true}\ \ \mathsf{policies} : Policy^+\ \}$.

We conclude by presenting the key result 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 this result, we prove that the constraint semantics is deterministic and total.

THEOREM 6.1 (DETERMINISTIC AND 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. By structural induction on the syntax of c (see Appendix C.2). □

THEOREM 6.2 (POLICY SEMANTIC CORRESPONDENCE). *For all* $p \in Policy$ *enclosing combining algorithms only using* all *as fulfilment strategy, and* $r \in R$, *it holds that*

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

PROOF. The proof (see Appendix C.2) is by induction on the *depth*, i.e. the nesting level, of p and relies on three auxiliary correspondence results regarding expressions (Lemma C.1), obligations (Lemma C.2) and combining algorithms (Lemma C.3). □

The latter theorem means that the properties verified over constraints would return the same results if they were directly proven on FACPL policies. Thus, it ensures that the analysis we present in Section 7 is sound. Notice also that this result 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 5.1 to constraint tuples.

COROLLARY 6.3 (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*

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

PROOF. The thesis immediately follows from Theorems 5.1 and 6.2. □

6.3. Constraint-based Representation of the e-Health case study

We now apply the translation functions just introduced to (a part of) the considered case study. For the sake of presentation, we shorten the attribute names used within policies. For instance, the rule addressing Requirement (1) becomes as follows

(permit target : equal(sub/role, "doctor") and equal(act/id, "write")
and in("e-Pre-Write", sub/perm) and in("e-Pre-Read", sub/perm))

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

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

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

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

The tuples for the rules addressing Requirements (2) and (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 (1). Besides the target expression, which is straightforwardly translated to the constraint $c_{trgP} \triangleq$

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 X. 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

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

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

Finally, the constraint tuple of Policy (1) generated by function $\mathcal{T}_P$ is as follows

$$\begin{array}{l}\langle \mathsf{permit}:\ c_{trgP} \wedge ((c_{trg1} \wedge \mathsf{true}) \vee (c_{trg2} \wedge \mathsf{true}) \vee (c_{trg3} \wedge \mathsf{true})) \wedge c_{obl\_p} \\ \ \mathsf{deny}:\ c_{trgP} \wedge ((((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \mathsf{false})) \wedge \mathsf{false}) \\ \qquad \vee(((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \mathsf{false})) \wedge \neg c_{trg3}) \\ \qquad \vee((\neg c_{trg1} \wedge \neg c_{trg2}) \wedge \mathsf{false})) \wedge\ \mathsf{true} \\ \ \mathsf{not\text{-}app}:\ \neg c_{trgP} \vee (c_{trgP} \wedge (\neg c_{trg1} \wedge \neg c_{trg2} \wedge \neg c_{trg3})) \\ \ \mathsf{indet}:\ \neg(\mathtt{isBool}(c_{trgP}) \vee\ \mathtt{isMiss}(c_{trgP})) \\ \vee(c_{trgP} \wedge (((\neg(\mathtt{isBool}(c_{trg1}) \vee\ \mathtt{isMiss}(c_{trg1})) \vee (c_{trg1} \wedge \neg\mathsf{true})) \wedge \neg(c_{trg2} \wedge \mathsf{true})) \\ \ \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}))) \wedge \neg(c_{trg3} \wedge \mathsf{true})) \\ \ \vee (\neg((c_{trg1} \wedge \mathsf{true}) \vee (c_{trg2} \wedge \mathsf{true})) \wedge (\neg(\mathtt{isBool}(c_{trg3}) \vee\ \mathtt{isMiss}(c_{trg3})) \vee (c_{trg3} \wedge \neg\mathsf{true}))) \\ \vee(c_{trgP} \wedge ((c_{trg1} \wedge \mathsf{true}) \vee (c_{trg2} \wedge \mathsf{true}) \vee (c_{trg3} \wedge \mathsf{true})) \wedge \neg c_{obl\_p}) \\ \vee(c_{trgP} \wedge ((((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \mathsf{false})) \wedge \mathsf{false}) \\ \ \vee(((\mathsf{false} \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \neg c_{trg2}) \vee (\neg c_{trg_1} \wedge \mathsf{false})) \wedge \neg c_{trg3}) \\ \ \vee((\neg c_{trg1} \wedge \neg c_{trg2}) \wedge \mathsf{false})) \wedge \neg\mathsf{true})\ \rangle\end{array}$$

As this example demonstrates, 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.

7. ANALYSIS OF FACPL POLICIES

The analysis of FACPL policies we propose aims at verifying different types of properties by exploiting the constraint-based representation of policies. We first formalise a relevant set of properties in terms of expected authorisations for requests, and then we define the strategies for their automatic verification by means of constraints. Recall that the pure dynamic aspects of FACPL are not addressed by constraints.

Furthermore, since FACPL does not enjoy the *safety* property (see Section 5.6), the analysis investigates how the extension of a request through the addition of further attributes might change its authorisation in a possibly unexpected way. Intuitively,

it is important to consider the authorisation decisions not only of specific requests, but also of their extensions because, e.g., a malicious user could try to exploit them to circumvent the access control system. This analysis approach is partially inspired by the probabilistic analysis on missing attributes introduced in [Crampton et al. 2015].

In the following, we first formalise the proposed properties (Section 7.1) and present some concrete examples of them from the case study (Section 7.2). Afterwards, we show how to express the constraint formalism into a tool-accepted specification (Section 7.3) and exploit it to automatically verify the properties with an SMT solver (Section 7.4).

7.1. Formalisation of Properties

We consider both properties that refer to the expected authorisation of single requests, i.e. *authorisation properties* (Section 7.1.1), and to the relationships among policies with respect to the authorisations they enforce, i.e. *structural properties* (Section 7.1.2).

7.1.1. Authorisation Properties. To formalise some of these properties, we introduce the notion of *request extension set* of a given request r. It 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.

Evaluate-To. This property, written r $\texttt{eval}$ dec, requires the policy under examination to evaluate the request r to decision dec. The satisfiability, written $\texttt{sat}$, 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 analysis approach based on request extensions is practically feasible, although such sets might be infinite. Indeed, Lemma 5.2 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.

7.1.2. Structural Properties. A structural property aims at characterising the relationships among the authorisations enforced by one or multiple policies. Different structural properties have been proposed in the literature (e.g., in [Fisler et al. 2005] and [Kolovski et al. 2007]) 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 reason on the relationships among policies and support system designers in developing and maintaining policies. One technique they enable is the *change-impact analysis* [Fisler et al. 2005]. This analysis examines the effect of policy modifications for discovering unintended consequences of such changes.

7.2. 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 4.3, i.e. Policies (1) and (2), 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[8] as follows

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

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

[8]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.

e-Prescription can be evaluated to not-app. This corresponds to the *May-Evaluate-To* property defined as follows

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

The verification of these properties with respect to Policy (1) results in

$$Policy\ (1)\ \mathtt{unsat}\ \ (Pr1) \qquad\qquad Policy\ (1)\ \mathtt{sat}\ \ (Pr2)$$

where unsat indicates that the policy does not satisfy the property. Indeed, as already discussed in Section 4.3, 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 (2) results in

$$Policy\ (2)\ \mathtt{sat}\ \ (Pr1) \qquad\qquad Policy\ (2)\ \mathtt{unsat}\ \ (Pr2)$$

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

$$Policy\ (1)\ \mathtt{unsat\ complete} \qquad Policy\ (2)\ \mathtt{sat\ complete}$$

As expected, Policy (1) does not satisfy completeness, i.e. there is at least one request that evaluates to not-app, whereas Policy (2) is complete. Instead, we can check if Policy (2) correctly refines Policy (1) by simply verifying coverage. We get

$$Policy\ (2)\ \mathtt{sat\ cover}\ Policy\ (1)$$

This follows from the fact that Policy (2) evaluates to permit the same set of requests as Policy (1) and that Policy (1) 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).

7.3. Expressing Constraints with SMT-LIB

Property verification 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 constraints defined in Section 6 by means of the SMT-LIB language (http://smtlib.cs.uiowa.edu/), 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. 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 function and array theories. In the following, we provide 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.

Table XI. 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

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

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

$$\frac{\Gamma \vdash expr_1 : U_1 \mid C_1 \qquad \Gamma \vdash expr_2 : U_2 \mid C_2}{\Gamma \vdash \mathsf{eop}(expr_1, expr_2) : Bool \mid C_1 \wedge C_2 \wedge U_1 = Bool \wedge U_2 = Bool} \ \mathsf{eop} \in \{\mathsf{and}, \mathsf{or}\}$$

$$\frac{\Gamma \vdash expr_1 : U_1 \mid C_1 \qquad \Gamma \vdash expr_2 : U_2 \mid C_2}{\Gamma \vdash \mathsf{equal}(expr_1, expr_2) : Bool \mid C_1 \wedge C_2 \wedge U_1 = U_2} \qquad \frac{\Gamma \vdash expr_1 : U_1 \mid C_1 \qquad \Gamma \vdash expr_2 : 2^{U_2} \mid C_2}{\Gamma \vdash \mathsf{in}(expr_1, expr_2) : Bool \mid C_1 \wedge C_2 \wedge U_1 = U_2}$$

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 XI. The rules are straightforward and infer the judgment $\Gamma \vdash expr : U \mid C$ 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 then used to instantiate the type parameters of the SMT-LIB constraints representing well-typed policies.

The type inference system aims at statically discarding all 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 $\mathsf{cat/id}$ 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 erroneous attribute values, even though we discard policies that are not well-typed.

On top of the `TValue` datatype we build the uninterpreted functions expressing the constraint operators of Table VIII. 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 IX, so that different `TValue` records are returned according to the input. Notably, 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 multivalued 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 multivalued 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[9]. 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 6.3) is

[9]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.

```
(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 (1) can be found at http://facpl.sf.net/eHealth/index.html.

7.4. Automated Properties Verification

The SMT-LIB coding permits using SMT solvers to automatically verify the properties of Section 7.1. Below, we present the verification strategies to follow.

Authorisation Properties. The automated verification of authorisation properties requires to modify, according to the considered property, the policy constraint modelling the decision of interest and then to check its satisfiability.

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 $\perp$. Given the property r `eval` dec, we explicitly represent the request r in terms of additional assertions for the constraint modelling the decision dec of the considered policy. In details, for each attribute in r, say $(attr1, v1)$, we insert

```
(assert (= (val attr1) v1))
(assert (and (not (miss attr1)) (not (err attr1))))
```

and, for any other attribute name, say *attr2*, not explicitly assigned by r but used within the constraint, we insert

```
(assert (miss attr2))
```

to assert that *attr2* is $\perp$. The satisfiability of the property thus corresponds to that of the resulting constraint.

To verify the *May-Evaluate-To* property, since we have to consider request extensions, we do not assert the unassigned names to $\perp$. Thus, property r $\texttt{eval}_{\texttt{may}}$ dec is satisfied by a policy if the constraint modelling the decision dec, to which we add assertions representing the request r, is satisfiable.

The *Must-Evaluate-To* property requires instead a different verification approach. In fact, given the property r $\texttt{eval}_{\texttt{must}}$ dec, we take the constraint modelling dec, we represent the request r as in the *May-Evaluate-To* property, and we have to prove the *validity* of the resulting constraint, i.e. that it is satisfied by all assignments for attribute names. This amounts to check if the negation of the resulting constraint 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 validity of combinations of constraints. 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 other properties require multiple checks.

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. 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. If this holds for the two conjunctions separately, the property holds.

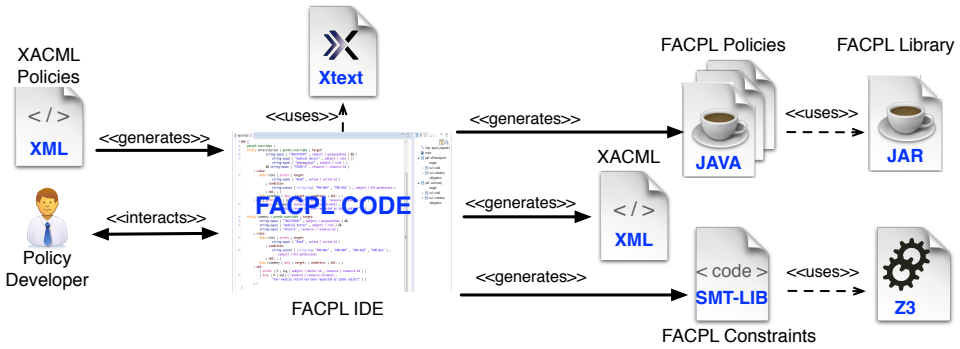

Fig. 3. The FACPL toolchain

Finally, it is worth noticing that we are not considering the set R of all possible requests because, due to Lemma 5.2, only the attribute names occurring in the policies of interest are relevant for the analysis; any other name cannot affect policy evaluation.

8. THE FACPL TOOLCHAIN

The coding, analysis and enforcement tasks pursued in the development of FACPL specifications are fully supported by a Java-based software toolchain[10], graphically depicted in Figure 3. 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 and SMT-LIB code. An expressly developed Java library is used to compile and execute the Java code, while the analysis of SMT-LIB code exploits the Z3 solver.

To provide interoperability with the standard XACML and the variety of available tools supporting it (e.g., XCREATE [Bertolino et al. 2012], Margrave [Fisler et al. 2005] and Balana [WSO2 2015]), the IDE automatically translates FACPL code into XACML one and vice-versa. Because of slightly different expressivity, there are some limitations in FACPL and XACML interoperability (see Section 9.1 for further details).

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, the web interface reachable from http://facpl.sf.net/eHealth/demo.html shows a proof-of-concept demo on how a FACPL-based access control system can be exploited for providing e-Health services.

In the rest of this section, we detail the FACPL Java library and IDE, while Section 9.4 reports performance and functionality comparisons with other similar tools.

8.1. 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 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.

[10]The FACPL supporting tools are freely available and open-source; binary files, source files, unit tests and documentation can be found at the FACPL website http://facpl.sf.net.

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 introduced by dynamically providing their implementation classes to the PEP initialisation method.

By way of example, we report here an excerpt of the Java code of Policy (1).

```
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")));
      addRule(new rule1());
      addRule(new rule2());
      addRule(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 rule1 extends Rule{
      rule1 (){
        addEffect(Effect.PERMIT);
        addTarget(...new ExpressionFunction(In.class,
            new AttributeName("subject","permission"),"e-Pre-Write"),...);
      }
   }
   private class rule2 extends Rule{ rule2 (){...} }
   private class rule3 extends Rule{ rule3 (){...} }
}
```

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.

8.2. The FACPL IDE

The FACPL IDE is developed as an Eclipse plug-in and aims at bringing together the available functionalities and tools. Indeed, it fully supports writing, evaluating and analysing of FACPL specifications. The plug-in has been implemented by means of Xtext, that is a framework to design and deploy 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. It offers graphical features (e.g., keywords highlighting, code suggestion and navigation within and

among files), static controls on FACPL code (e.g., uniqueness of identifiers and type checking), and automatic generation of Java, XACML, and SMT-LIB code. To configure all the required libraries, a dedicated wizard creates a FACPL-type project.

To facilitate the analysis of FACPL policies, the plug-in also provides a simple interface allowing policy developers to specify the authorisation and structural properties to be verified on a certain policy. Thus, the plug-in automatically generates the corresponding SMT-LIB files according to the strategies reported in Section 7.4; an execution script for the Z3 solver is also generated. Notably, the SMT-LIB files can be also evaluated by any other solver accepting SMT-LIB and supporting the theories we use.

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 and SMT-LIB implementation is automatically generated. The actual implementation of the Java class, as well as of the SMT-LIB function, is left to the developer.

9. RELATED WORK

A preliminary version of FACPL was introduced in [Masi et al. 2012] with the aim of formalising 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); 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. [Margheri et al. 2013] for a practical example of a policy-based manager for a Cloud platform). Additional significative differences concern the definition of the policy semantics: in [Masi et al. 2012] 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 this work is significantly different: we do not only propose a different language, but we provide a complete methodology that encompasses all phases of policy lifecycle, i.e. specification, analysis and enforcement. Concerning the analysis, we define a set of relevant authorisation and structural properties (whose preliminary definition is given in [Margheri et al. 2015]) characterised in terms of sets of requests. We then introduce a constraint-based representation of policies and an SMT-based approach for mechanically verifying properties on top of constraints. To effectively support the functionalities, we provide a fully-integrated software toolchain.

In the rest of this section we survey more closely related work. First, we comment on differences and interoperability of FACPL with the already mentioned standard XACML (Section 9.1). Then, we discuss other relevant policy languages (Section 9.2), and approaches to the analysis of (access control) policies (Section 9.3). Finally, we compare supporting tools (Section 9.4).

9.1. FACPL vs XACML

XACML [OASIS XACML TC 2013] is a well-established standard for the specification of attribute-based access control policies and requests. It has an XML-based syntax and an evaluation process defined in accordance with [Yavatkar et al. 2000] (hence

Table XII. FACPL vs. XACML on the e-Health case study

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%

similar to the FACPL one). As a matter of notation, hereafter the words emphasised in sans-serif, e.g. Rule, are XML elements, while element attributes are in italics.

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 are hardly of immediate comprehension for developers and are not suitable for formally defining semantics and analysis techniques. Table XII 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.sf.net/eHealth/index.html).

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.

A XACML Target is made of Match elements defining basic comparison functions on request attributes. The elements are then organised in terms of the tag structure AnyOf-AllOf-Match. This structure 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. 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, a 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 imposes 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 sub-decisions: due to space limitations, we have not considered them here but they are supported by the FACPL library.

Finally, 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.

Table XIII. Comparison of a relevant set of policy languages (where ✓* means that user encoding are required)

Features	**XACML**	**Ponder**	**ASL**	**PTaCL**	[Rao et al. 2009]	[Arkoudas et al. 2014]	**FACPL**
Rule-based	✓	✓					✓
Logic-based			✓	✓	✓	✓	
Mnemonic spec.		✓					✓
Comb. algorithms	✓		✓*	✓*	✓	✓*	✓
Obligations	✓	✓					✓
Missing attributes	✓			✓			✓
Error handling	✓						✓

9.2. Policy Languages for Access Control

Policy languages have recently been the subject of extensive research, both by industry and academia. Indeed, policies permit managing different important aspects of system behaviours, ranging from access control to adaptation and emergency handling. We compare in the following the main policy languages devoted to access control, which is our focus; Table XIII summarises the comparison.

Among the many proposed policy languages, we can identify two main specification approaches: *rule-based*, as e.g. the XACML standard and Ponder [Damianou et al. 2001; Twidle et al. 2009], and *logic-based*, as e.g. ASL [Jajodia et al. 1997], PTaCL [Crampton and Morisset 2012] and the logical frameworks in [Arkoudas et al. 2014]. Many other works, as e.g. [Li et al. 2009; Rao et al. 2009; Ramli et al. 2014], 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 [Sloman 1994] 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 Event-Condition-Action rules triggering the enforcement of adaptation actions. This setting is at the basis of Ponder.

Ponder 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 conflicts from occurring, 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 [Ceri et al. 1989] (see, e.g., [Jajodia et al. 1997; Hashimoto et al. 2009; DeTreville 2002]), 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 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 fea-

tures like, e.g., conflict resolution strategies. A suitable use of policies hierarchisation enables propagation of access rights also in FACPL specifications.

PTaCL 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 only partially address errors. 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 [Tschantz and Krishnamurthi 2006].

A similar study, but more focussed on the distinguishing features of XACML, is reported in [Ramli et al. 2014]. It introduces a formalisation of XACML in terms of multi-valued logics, by first considering 4-valued decisions and then 6-valued ones. 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 have instead a crucial role in XACML policy evaluation.

Another logic-based language is presented in [Arkoudas et al. 2014]. 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 [Rao et al. 2009] 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, [Li et al. 2009] 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.

Moreover, in [Li et al. 2009] nonlinear constraints are used for the specification of combining algorithms which 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 application of some binary operators.

9.3. Analysis of Access Control Policies

The increasing spread of policy-based specifications has prompted the development of many verification techniques like, e.g., property checking and behavioural characterisations. Such techniques have been implemented by means of different formalisms, ranging from SMT formulae to multi-terminal binary decision diagrams (MTBDD), including different kinds of logics. Below, we review the more relevant ones.

The works concerning policy analysis that are closer to our approach are of course those exploiting SMT formulae. In [Turkmen et al. 2015], 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 reals, 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 approach shown in Appendix B.

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, 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. The SMT-based framework of [Arkoudas et al. 2014], introduced in Section 9.2, suffers from similar drawbacks.

The only analysis approach that takes missing attributes into account is presented in [Crampton et al. 2015]. The analysis is based on a notion of request extension, as we have done in Section 7. Differently from our approach, this analysis aims at quantifying the impact of possibly missing attributes on policy evaluations.

The change-impact analysis of XACML policies presented in [Fisler et al. 2005] 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 [Arkoudas et al. 2014], an SMT-based approach like ours scales 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 SMT-based approaches. 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 [Kolovski et al. 2007] as a target formalism for representing a part of XACML. The approach does not take into account many combining algorithms and the decisions not-app and indet. Thus, it only permits reasoning on a set of properties significantly reduced with respect to that supported by our 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 [Ahn et al. 2010; Ramli et al. 2012] for encoding XACML and enabling verification of structural properties that are similar to the `complete` one defined in Section 7.1.2. This approach however suffers from some drawbacks due to the nature of ASP. In fact, differently from SMT, ASP does not support quantifiers and multiple 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 Z3 is available. Similarly, the work in [Hughes and Bultan 2008] exploits the SAT-based tool Alloy [Jackson 2002] to detect inconsistencies in XACML policies. However, as outlined in [Arkoudas et al. 2014] and [Fisler et al. 2005], Alloy is not able to manage even quite small policies and, more importantly, it cannot reason on arithmetic or any additional theory.

Finally, it is worth noticing that various analysis approaches using SAT-based tools have been developed for the Ponder language, see e.g. [Bandara et al. 2003]. These approaches, however, cannot actually be compared with ours due to the consistent differences among Ponder and FACPL. Furthermore, many other works deal with the analysis of access control policies by using, e.g., process algebra and model checking techniques. However, they approach only a limited part of access control policy aspects and suffer from scalability issues with respect to SMT-based tools.

In summary, all the approaches to the analysis of access control policies mentioned above are deficient in several respects. 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.

9.4. Performance and Functionalities of Supporting Tools

The effectiveness of supporting tools is a crucial point for the usability of a policy language. In the following, we thus compare the performance of 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 [Krishnamurthi 2003], which has been adopted as a standard benchmark in the field of access control tools[11].

The XACML standard is by now the point-of-reference for industrial access control. In the authors' knowledge, the most up-to-date, freely available XACML tool is Balana [WSO2 2015]. 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 [Arkoudas et al. 2014], which relies on the SMT solver Yices [Dutertre 2014]. Differently from Z3, Yices does not support datatype theory, which is instead crucial to deal 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[12], while our Z3-based tool requires around 120ms. Notably, other not SMT-based tools, like, e.g., Margrave, have significantly lower performance when policies scale. In fact, as reported in [Arkoudas et al. 2014], 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 close to the ours. To the best of our knowledge, the only similar (freely available) IDEs are the ALFA Eclipse plugin 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.

10. CONCLUDING REMARKS AND FUTURE WORK

We have described a full-fledged approach for the specification, analysis and enforcement of access control policies, which is based on the FACPL language and its tools. The FACPL formal semantics provides a formalisation of complex access control features —including obligations and missing attributes, which are instead overlooked by many other proposals— and lays the basis for developing analysis techniques and tools. Indeed, we have shown that FACPL policies can be represented in terms of SMT-based formulae, whose automated evaluation permits verifying various authorisation and structural properties. We have demonstrated feasibility and effectiveness of our approach by means of an e-Health case study for the provision of exchanging services of medical data across European countries. We have also shown that the use of SMT solvers provides us with stable and efficient tools, ensuring better performance than many other approaches from the literature.

[11]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.sf.net/continue/.
[12]This value is taken for granted from [Arkoudas et al. 2014], because the provided CONTINUE implementation only runs on Windows machines. Anyway, their hardware configuration is similar to ours.

In a general perspective, our approach brings together the benefits deriving from using a high-level, mnemonic rule-based language with the rigorous means provided by denotational semantics and constraints. Additionally, the supporting tools we implemented allow access control system developers to use any of the formally-defined functionalities provided by our framework, without the need that they be familiar with formal methods.

In the next future, we want to address continuous controls while accesses are in progress. This will require to provide a formal representation of access history and exploit it in our analysis approach. Moreover, we plan to study properties that take into account obligations. Specifically, we want to define properties on conflicts and dependencies among obligations and to devise appropriate analysis strategies.

References

G. J. Ahn, H. Hu, J. Lee, and Y. Meng. 2010. Representing and Reasoning about Web Access Control Policies. In *COMPSAC*. IEEE Computer Society, 137–146.

K. Arkoudas, R. Chadha, and C.-Y. J. Chiang. 2014. Sophisticated Access Control via SMT and Logical Frameworks. *ACM Trans. Inf. Syst. Secur.* 16, 4 (2014), 17.

A. K. Bandara, E. Lupu, and A. Russo. 2003. Using Event Calculus to Formalise Policy Specification and Analysis. In *POLICY*. IEEE, 26.

C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. 2011. CVC4. In *Proc. of CAV (LNCS)*, Vol. 6806. Springer, 171–177.

A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. 2012. The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies. In *WEBIST*. SciTePress, 155–160.

S. Ceri, G. Gottlob, and T. Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166.

J. Crampton and C. Morisset. 2012. PTaCL: A Language for Attribute-Based Access Control in Open Systems. In *POST (LNCS)*, P. Degano and J. D. Guttman (Eds.), Vol. 7215. Springer, 390–409.

J. Crampton, C. Morisset, and N. Zannone. 2015. On Missing Attributes in Access Control: Non-deterministic and Probabilistic Attribute Retrieval. In *SACMAT*. ACM, 99–109.

N. Damianou, N. Dulay, E. Lupu, and M. Sloman. 2001. The Ponder Policy Specification Language. In *POLICY (LNCS 1995)*. Springer, 18–38.

L. M. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

L. M. de Moura and N. Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

J. DeTreville. 2002. Binder, a Logic-Based Security Language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02)*. IEEE Computer Society, Washington, DC, USA, 105–113.

B. Dutertre. 2014. Yices 2.2. In *Proc. of CAV (LNCS)*, Vol. 8559. Springer, 737–744.

European Parliament and Council. 1995. 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.

D. F. Ferraiolo and D. R. Kuhn. 1992. Role-Based Access Control. In *NIST-NCSC National Computer Security Conference*. 554–563.

K. Fisler, S. Krishnamurthi, L.A. Meyerovich, and M.C. Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *ICSE*. ACM, 196–205.

W. Han and C. Lei. 2012. A survey on policy languages in network and security management. *Computer Networks* 56, 1 (2012), 477–489.

M. Hashimoto, M. Kim, H. Tsuji, and H. Tanaka. 2009. Policy Description Language for Dynamic Access Control Models. In *DASC*. IEEE, 37–42.

V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. 2015. Attribute-Based Access Control. *IEEE Computer* 48, 2 (2015), 85–88.

G. Hughes and T. Bultan. 2008. Automated verification of access control policies using a SAT solver. *STTT* 10, 6 (2008), 503–520.

D. Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

S. Jajodia, P. Samarati, and V. S. Subrahmanian. 1997. A Logical Language for Expressing Authorizations. In *Symposium On Security And Privacy*. IEEE, 31–42.

X. Jin, R. Krishnan, and R. S. Sandhu. 2012. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *DBSec*. Springer, 41–55.

V. Kolovski, J. A. Hendler, and B. Parsia. 2007. Analyzing web access control policies. In *WWW*. ACM, 677–686.

M. Kovac. 2014. E-Health Demystified: An E-Government Showcase. *IEEE Computer* 47, 10 (2014), 34–42. http://dx.doi.org/10.1109/MC.2014.282

S. Krishnamurthi. 2003. The CONTINUE Server (or, How I Administered PADL 2002 and 2003). In *PADL (LNCS)*, Vol. 2562. Springer, 2–16.

B. W. Lampson. 1974. Protection. *Operating Systems Review* 8, 1 (1974), 18–24.

N. Li, Q. Wang, W. H. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. 2009. Access control policy combining: theory meets practice. In *SACMAT*. ACM, 135–144.

A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. 2013. Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation. A Practical Approach. In *WSFM (LNCS)*, Vol. 8379. Springer, 85–105.

A. Margheri, R. Pugliese, and F. Tiezzi. 2015. On Properties of Policy-Based Specifications. In *WWV (EPTCS)*, Vol. 188. 33–50.

M. Masi, R. Pugliese, and F. Tiezzi. 2012. Formalisation and Implementation of the XACML Access Control Mechanism. In *ESSoS (LNCS 7159)*. Springer, 60–74.

NIST. 2009. A survey of access control models. (2009). http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.

OASIS XACML TC. 2013. eXtensible Access Control Markup Language (XACML) version 3.0 . (January 2013). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

C. D. P. K. Ramli, Nielson H. Riis, and F. Nielson. 2014. The logic of XACML. *Sci. Comput. Program.* 83 (2014), 80–105.

C. D. P. K. Ramli, H. Riis Nielson, and F. Nielson. 2012. XACML 3.0 in Answer Set Programming. In *LOPSTR (LNCS)*, Vol. 7844. Springer, 89–105.

P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. 2009. An algebra for fine-grained integration of XACML policies. In *SACMAT*. ACM, 63–72.

M. Sloman. 1994. Policy Driven Management for Distributed Systems. *J. Network Syst. Manage.* 2, 4 (1994), 333–360.

The Article 29 Data Protection WP. 2013. (2013). http://ec.europa.eu/justice/data-protection/article-29/.

M. C. Tschantz and S. Krishnamurthi. 2006. Towards reasonability properties for access-control policy languages. In *SACMAT*. ACM, 160–169.

F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. 2015. Analysis of XACML Policies with SMT. In *POST (LNCS)*, Vol. 9036. Springer, 115–134.

K. P. Twidle, N. Dulay, E. Lupu, and M. Sloman. 2009. Ponder2: A Policy System for Autonomous Pervasive Environments. In *ICAS*. IEEE, 330–335.

WSO2. 2015. Balana: Open source XACML implementation. (2015). https://github.com/wso2/balana.

R. Yavatkar, D. Pendarakis, and R. Guerin. 2000. A Framework for Policy-based Admission Control. RFC 3060 (Proposed Standard). (2000). https://tools.ietf.org/html/rfc2753

A. MATRICES FOR COMBINING OPERATORS

$\otimes$p-over	$\langle$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	$\langle$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	$\langle$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	$\langle$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	$\langle$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	$\langle$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	$\langle$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	$\langle$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

B. CONSTRAINTS COMBINATIONS FOR COMBINING OPERATORS

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

C. PROOFS OF RESULTS

To prove some of the results, we will reason by induction on the *depth* of policies, i.e. the number of nested policies, which is defined by induction on the syntax of policies as follows

$$\begin{aligned}
& depth((e \;\; \text{target} : expr \;\; \text{obl} : o^*)) = 0 \\
& depth(\{a \;\; \text{target} : expr \;\; \text{policies} : p^+ \; \text{obl} : o^*\}) = 1 + max\{depth(p) \mid p \in p^+\}
\end{aligned}$$

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.

C.1. Proofs of Results in Section 5

THEOREM 5.1 [Deterministic and Total Semantics] For all $pas \in PAS$, $req \in Request$ and $dec, dec' \in Decision$, it holds that

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

PROOF. The proof reduces to showing that $\mathcal{P}as$ is a *total function*, i.e. it uniquely associates a decision to each input of the form (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 V, 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.

□

LEMMA 5.2. 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

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

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 $(e \ \mathsf{target} : expr \ \mathsf{obl} : o^*)$, 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 $\{a \ \mathsf{target} : expr \ \mathsf{policies} : (p^+)^n \ \mathsf{obl} : o^*\}$, 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).

□

C.2. Proofs of results in Section 6

THEOREM 6.1 [Deterministic and 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 not hard to believe 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.

□

The proof of Theorem 6.2 relies on the following three auxiliary results.

LEMMA C.1. *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 IX and that of the operator not in Table V.

($expr = \mathsf{op}(expr_1, expr_2)$). Since $\mathcal{T}_E\{\!|expr|\!\} = \mathcal{T}_E\{\!|expr_1|\!\}\ \mathsf{getOp}(\mathsf{op})\ \mathcal{T}_E\{\!|expr_2|\!\}$ 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 IX and that of the constraint operator $\mathsf{getOp}(\mathsf{op})$ in Table V.

□

LEMMA C.2. *For all $o \in Obligation$ and $r \in R$ it holds that*

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

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

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

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 C.1, 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 C.1, 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 C.3. *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 the tables in Appendixes A and B.

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 6.2 [Policy Semantic Correspondence] For all $p \in Policy$ enclosing combining algorithms only using all as fulfilment strategy, and $r \in R$, it holds that

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

PROOF. The proof proceeds by induction on the depth i of p.

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 C.1, it follows that

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

and, by Lemma C.2 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 C.1 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 C.1 and C.2 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 C.1, it follows that

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

and, by Lemma C.2 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

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

then, by Lemma C.3, 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

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

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

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

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

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

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

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

The thesis then directly follows by Lemmas C.1 and C.3, 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

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

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

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

$$
\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 = \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 C.1, C.2 and C.3, due to the induction hypothesis and the definition of $\mathcal{C}$.

□