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

Andrea Margheri, Massimiliano Masi, Rosario Pugliese and Francesco Tiezzi

Abstract—Access control systems are widely used means for the protection of computing systems. They are defined in terms of access control policies regulating the access to system resources. In this paper, we introduce a formally-defined, fully-implemented framework for specification, analysis and enforcement of attribute-based access control policies. The framework rests on FACPL, a language with a compact, yet expressive, syntax for specification of real-world access control policies and with a rigorously defined denotational semantics. The framework enables the automated verification of properties regarding both the authorisations enforced by single policies and the relationships among multiple policies. Effectiveness and performance of the analysis rely on a semantic-preserving representation of FACPL policies in terms of SMT formulae and on the use of efficient SMT solvers. Our analysis approach explicitly addresses some crucial aspects of policy evaluation, such as missing attributes, erroneous values and obligations, which are instead overlooked in other proposals. The framework is supported by Java-based tools, among which an Eclipse-based IDE offering a tailored development and analysis environment for FACPL policies and a Java library for policy enforcement. We illustrate the framework and its formal ingredients by means of an e-Health case study, while its effectiveness is assessed by means of performance stress tests and experiments on a well-established benchmark.

Index Terms—Attribute-based Access Control, Policy Languages, Policy Analysis, SMT

1 Introduction

Nowadays computing systems have pervaded every daily activity and prompted the proliferation of several 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 fostered an increasing research interest towards access control systems, which are the first line of defence for the protection of computing systems. They are defined by *rules* that establish under which conditions a subject's *request* for accessing a resource has to be permitted or denied. In practice, it amounts to restricting 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. From 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 [1]– or through predefined features, such as roles or groups –e.g., Role-Based Access Control (RBAC [2]). These approaches are however inadequate for dealing with modern distributed systems, as they suffer from scalability and interoperability issues, mainly due to the difficulty on defining granular access controls for each individual [3]. Moreover, they cannot easily encompass information representing the evaluation context, such as system status or current time. An alternative approach that permits to overcome these problems is Attribute-Based Access Control (ABAC) [4]. 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 [5]. 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) [3], sometimes used in place of ABAC.

Many languages have been proposed for the specification of access control policies (see, e.g., Han and Lei's survey [6]). Among the proposed languages, in the authors' knowledge, the OASIS standard eXtensible Access Control Markup Language (XACML) [7] 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 architectures. However, the management of real access control policies is in practice cumbersome and error-prone, and should be

- *A. Margheri is with University of Southampton, Electronics and Computer Science, Southampton, United Kingdom*
- *M. Masi is with Tiani "Spirit" GmbH, Wien, Austria*
- *R. Pugliese is with Università degli Studi di Firenze, Dipartimento di Statistica, Informatica, Applicazioni "G. Parenti", Firenze, Italy*
- *F. Tiezzi is with Università di Camerino, Scuola di Scienze e Tecnologie, Camerino, Italy*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2765640, IEEE Transactions on Software Engineering

supported by rigorous analysis techniques. Alas, the lack of a formally defined semantics for XACML [8], [9], [10], [11] hinders the specification and realisation of such techniques.

To tackle these difficulties, we introduce a formally-defined, fully-implemented framework based on the Formal Access Control Policy Language (FACPL), supporting developers in the specification, analysis and enforcement of access control policies.

This paper is a revised and extended version of previous works [12], [13]. Besides significant revisions and extensions of syntax and semantics of FACPL (we refer to Section 9 for a detailed comparison), we propose here a complete development methodology for access control policies. Most of all, differently from previous works, we introduce a constraint-based representation of FACPL policies enabling the efficient verification of various properties.

The FACPL-based Access Control Framework

The FACPL language defines a core, yet expressive, syntax for specification of high-level access control policies. It is inspired by XACML, with which it shares the main traits of the policy structure and some terminology. However, 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., *(i)* management of missing attributes, i.e., attributes controlled by a policy but not provided by the request to authorise, and *(ii)* formalisation of combining algorithms, i.e., strategies to resolve conflictual decisions that policy evaluation can generate.

The analysis functionalities provided by our framework enable static verification of two main groups of properties of FACPL policies. *Authorisation properties* permit reasoning on 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. *Structural properties*, instead, permit reasoning on the whole set of evaluations of policies and can be exploited, e.g., to implement maintenance and *change-impact analysis* [14] techniques.

The verification of these properties requires extensive checks on infinite requests, hence support through software tools is essential. As no off-the-shelf analysis tool directly takes FACPL specifications in input, our framework exploits a constraint formalism that permits uniformly representing policy elements and enabling automated analysis. The constraint formalism we introduce is based on Satisfiability Modulo Theories (SMT) formulae, that is formulae defining satisfiability problems involving multiple theories, such as 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 [15], even for access control policies [11], [16]. In practice, SMT-based approaches are more effective than many other ones, like decision diagrams [14] or description logic [17]. The soundness of our analysis techniques is guaranteed by the correspondence, which we formally prove, between the semantics of FACPL policies and that of their constraint-based representations.

Our framework is supported by a Java-based software *toolchain*. The key software tool is an Eclipse-based IDE that offers a tailored development and analysis environment for FACPL policies. Specifically, it helps access control policy developers in the tasks of specification, analysis and enforcement of policies by providing, e.g., static checks on the code and automatic generation of runnable SMT and Java code. The evaluation of the SMT code relies on the Z3 solver [18], while the policy enforcement relies on a Java library that we have specifically developed.

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 the methodology is formally presented in this paper, together with step-by-step examples and tool implementation. The tools allow access control system developers to use formally-defined functionalities without requiring them to be familiar with formal methods.

Our methodology enhances the proposals from the literature to different extents, for providing a single framework where all the relevant functionalities are expressed and formalised in a uniform manner. Indeed, XACML does not come with any formal specification and, hence, analysis; the formally-grounded proposals by Jajodia et al. [19], Crampton and Morisset [9], Ramli et al. [10], and Crampton and Williams [20] do not offer supporting tools; instead, the SMT-based analysis proposals by Arkoudas et al. [11] and Turkmen et al. [16] do not support such crucial features as, e.g., missing attributes and obligation instantiation. Comparisons with the relevant literature are reported in Section 9.

Our aim is to design an expressive language whose formal foundations enable tool-supported analysis, rather than to face XACML semantic issues or supersede it. Further specific contributions of the work reported in this paper are outlined below.

- The FACPL semantics manages missing attributes in a way similar to Crampton and Morisset [9] and Crampton and Williams [20], and extends their approaches with explicit error management.
- The formalisation of combining algorithms extends the work by Li et al. [21] with explicit combination of obligations and with different instantiation strategies.
- The authorisation properties are introduced to specifically take into account the ‘safety’ property of Tschantz and Krishnamurthi [22] by appropriately employing the request extensions set by Crampton et al. [23] for property formalisation.
- The main structural properties introduced by Fisler et al. [14], Kolovski et al. [17] and Arkoudas et al. [11] are uniformly formalised in terms of policy semantics.
- A constraint formalism is defined to provide a low-level, tool-independent representation of attribute-based policies that is capable to deal with all issues regarding policy evaluation.
- The validation of the proposal is carried out through experiments on a standard benchmark in the field of access control tools, i.e., the CONTINUE case study [24].

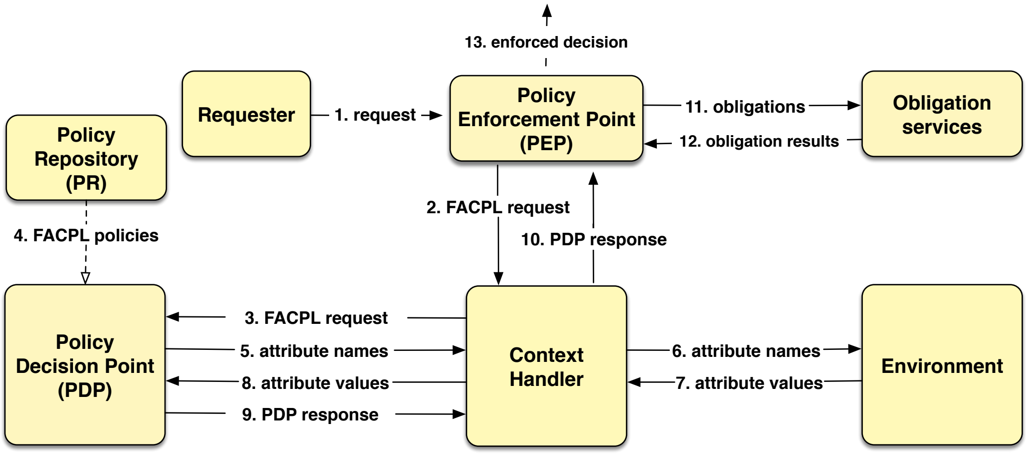


Figure 1. The FACPL evaluation process

The FACPL framework impacts on different phases of the software development process of policy-based systems:

Design: the framework supports the design of the access control module of the system via an Eclipse-based IDE, which permits to specify policies, automatically generate constraints, and analyse them through an SMT solver. Automatic translation of XACML policies into FACPL ones is also supported.

Implementation: the framework supports, via the automatic generation of runnable Java code from FACPL policies, the actual installation of an access control module. FACPL policies can also be installed as XACML ones by using the corresponding translator.

Maintenance: the loose coupling of the access control module with the rest of the system permits its dynamic update and replacement.

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 as a running example. 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 and present the validation results. 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 and B report, respectively, all the definitions for combining algorithms, and the proofs of the formal results.

2 The FACPL Evaluation Process

The FACPL evaluation process is shown in Figure 1. It assumes that system resources are paired with one or more FACPL policies, which define the credentials necessary to gain access to such resources. The evaluation process 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 the RFC 2753 [25] to define the evaluation process of policy-based systems. Since then, many policy languages, like XACML, have tailored them according to their specific features.

When the PEP receives an access request (step 1), the credentials contained in the request are encoded as a sequence of *attribute* elements forming a FACPL request and sent to the context handler (step 2). An attribute element is a name-value pair representing arbitrary information relevant for evaluating the access request. This encoding allows access requests, written in the format required by the controlled system, to be transparently evaluated by the FACPL process. Notably, PEP implementation depends on the specific system and can have different forms, such as a gateway or a Web server.

Then, the *context handler* sends the request to the PDP (step 3), by possibly adding environmental attributes, e.g. request receiving time, that may be used in the evaluation.

Upon request reception, the PDP retrieves (step 4) the FACPL policies to be currently enforced. Such policies are stored and supplied by the PR. The interaction between the PDP and the PR is denoted by a dashed arrow in the figure, as it is not further detailed here. In fact it can be realised in many different ways depending on the application. For example, the policies could be provided proactively, or upon a request by the PDP; moreover, they could be provided once and for all, or resent whenever dynamically updated.

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[1]. The meaning of the first two ones

[1] The FACPL supporting tools can handle the same extended indeterminate values dealt with by XACML, as discussed in Section 8. However, for the sake of presentation, in the formal specification of FACPL we only consider a single indeterminate value, rather than the whole set.

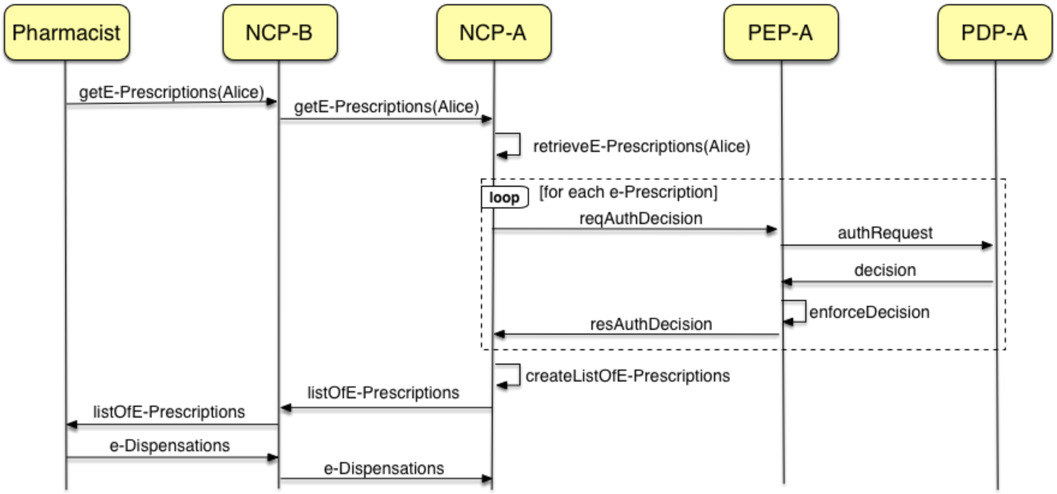


Figure 2. e-Prescription service protocol

is obvious, the third one means that there is no policy that applies to the request and the latter one means that an error has occurred during the evaluation. Policies can use operators that combine, according to different strategies, not-app and indet decisions with the others. In this way, combined policies can specify how the evaluation process has to manage, in an automatic fashion, sub-policies that do not apply to a request or whose evaluation raises errors.

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 result of obligations discharge. This decision could differ from that of the PDP and is the overall outcome of the evaluation process.

A main benefit of following the approach of the RFC 2753 is the separation of concerns among policies, their evaluation and the system itself. In particular, it means that: (i) different types of requests can be handled, as the PEP can appropriately encode them in the format required by the PDP; (ii) the PDP can be placed in any point of the system architecture, with the PEP acting as a gateway or a proxy; and (iii) the PR can be also instantiated to support dynamic, possibly regulated, modifications of policies.

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. In this context, we will show that access control policies expressed in FACPL can control accesses to health data in order to preserve data confidentiality and integrity.

The exchange of patients health data among European points of care, such as clinics, hospitals and pharmacies, has been pursued by the EU through the large scale pilot project epSOS [26]. The goal is to establish a suite of standardised data exchanging services for facilitating the cross-border interoperability [27] of the EU country healthcare systems and professionals, such as doctors, nurses and pharmacists, thus ultimately improving the effectiveness of healthcare treatments to EU citizens that are abroad. These services must respect a set of requirements in order to comply with country-specific legislations [28], [29] and to enforce the *patient informed consent*, i.e., the patients informed indications pertaining to personal data processing.

In this paper we focus on the *electronic prescription* (e-Prescription) service. This service allows EU patients, while staying in a foreign country B, 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 performed by means of 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)[2]. 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.

Starting from the epSOS specifications, we deduced a set of business requirements concerning the e-Prescription service. To streamline the presentation, we explicitly report in Table 2 all and only those requirements authorising some actions. Hence, every action not explicitly authorised is forbidden. For instance, it is not allowed to pharmacists to write e-Prescriptions, which is instead allowed to doctors exhibiting specific permissions. All the requirements are self-explanatory. We just want to point out that the first three requirements deal with access restrictions, while the other

[2] For the sake of presentation, we abstract from the activity carried out by the pharmacist to ascertain the patient identity.

Table 1
Syntax of FACPL

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

ones deal with additional functionalities that sophisticated access control systems, like the one we present, can provide.

Table 2
Requirements for the e-Prescription service

#	*Description*
1	Doctors with e-Pre-Read and e-Pre-Write permissions can write e-Prescriptions
2	Doctors with e-Pre-Read permission can read e-Prescriptions
3	Pharmacists with e-Pre-Read permission can read e-Prescriptions
4	Authorised user accesses must be recorded by the system
5	Patients must be informed of unauthorised access attempts
6	Exchanged data should be compressed

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 in Section 4.1. Then, in Section 4.2 we informally explain the semantics of its linguistic constructs and in Section 4.3 employ them to implement the access control system of the e-Health case study.

4.1 Syntax

Intuitively, FACPL policies are hierarchically structured lists of elements containing controls on the value of attributes that should be provided by FACPL access requests. Together with permit or deny decisions, policies specify the combining algorithms to be used in their evaluation and the obligations for the enforcement process.

Formally, the syntax of FACPL is reported in Table 1. It is given through an EBNF-like grammar, where as usual the symbol $*$ stands for (possibly empty) sequences and $+$ for non-empty sequences.

A top-level term is a *Policy Authorisation System (PAS)* encompassing the specifications of a PEP and a PDP. The PEP is defined by the *enforcement algorithm* ($EnfAlg$) 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 (PDP) is instead defined by a policy ($Policy$), or by a sequence of policies ($Policy^+$) and an algorithm (Alg) for combining the results of the evaluation of these policies.

A *policy* is a sequence of labelled fields, as a sort of record. It can be either a basic authorisation *rule* ($Rule$) or a *policy set* collecting rules and other policy sets, so that policies can be hierarchically structured. A rule specifies an *effect* ($Effect$), which is the permit or deny decision returned when the rule is successfully evaluated, a *target*, which is an expression ($Expr$) indicating the set of access requests to which the rule applies, and a sequence of obligations ($Obligation^*$), that is actions to be discharged by the enforcement process. A policy set specifies a target, a sequence of enclosed policies along with an algorithm for combining the results of their evaluation, and two sequences of obligations, one to be discharged if the resulting effect is permit, the other if it is deny. Obligation sequences may be empty, while policy sequences cannot.

An attribute *name* ($Name$) refers to the literal value associated with the attribute. The name is structured in 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 usual categories of access control, i.e., *subject*, *resource* and *action*, to possibly application-dependent ones.

Expressions are built from attribute names and *literals* ($Value$), 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.

Combining algorithms offer different strategies to merge the decisions resulting from the evaluation of various policies; e.g., the p-over$_\delta$ algorithm states that decision permit

Table 3
Auxiliary syntax for FACPL responses

PDP responses	$PDPResponse ::= \langle\, Decision \quad IObligation^{*} \,\rangle$
Decisions	$Decision ::= \mathsf{permit} \mid \mathsf{deny} \mid \mathsf{not\text{-}app} \mid \mathsf{indet}$
Instantiated oblig.	$IObligation ::= [\, ObType \quad PepAction(\, Value^{*})\,]$

takes precedence over the others. They can be specialised by choosing different strategies (δ) for the instantiation of obligations; e.g., the greedy strategy states that only the obligations resulting from the actually evaluated policies are returned. In the algorithm names, p and d are shortcuts for permit and deny, respectively.

An *obligation* specifies a type ($ObType$), i.e., mandatory (m) or optional (o), and identifier and 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* ($Request$) consists of a (non-empty) sequence of *attributes*, i.e., name-value pairs, that enumerate request credentials in the form of literal values. *Multivalued attributes*, i.e., names associated with a set of values, are rendered as multiple attributes sharing the same name.

The responses resulting from the evaluation of FACPL requests are written using the auxiliary syntax reported in Table 3. The two-stage evaluation process described in Section 2 produces two different kinds of responses: *PDP responses* ($PDPResponse$) and *decisions* ($Decision$), i.e., responses by the PEP. The former ones, in case of decision permit and deny, pair the decision with a (possibly empty) sequence of instantiated obligations ($IObligation^{*}$). An *instantiated obligation* is a pair made of a type (i.e., m or o) and an action whose arguments are values.

To simplify notations, in the sequel we will omit the label preceding a sub-term generated by the grammar in Table 1 whenever the sub-term is missing or is the expression true. Thus, e.g., the rule (deny target : true obl :) will be simply written as (deny). Moreover, when in the $PDPResponse$ the sequence of instantiated 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 refer to Section 5 for the formal characterisation. 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 instantiation 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, the target of enclosed policies may refine that of the enclosing ones, while a policy with target expression true (resp., false) applies to all (resp., no) requests.

Evaluating expressions amounts to applying operators and to *resolve* the attribute names occurring within, that is to determine the value corresponding to each such name. This value can either be directly contained in the request or retrieved from the environment by the context handler (steps 5-8 in Figure 1). If name resolution fails, the special value $\perp$ is returned. This enables precise management of those requests containing less attributes than expected. From the analysis point of view, this also enables reasoning on the role of missing attributes in policy evaluation; further details are given in Section 7.

The syntax of policies, and in particular that of attribute names and expressions, does not consider types. Indeed, we want a policy to provide a response to any request, not only to those complying with the expected type of the values referred by the attribute names controlled by the policy. Since we do not filter requests on the base of the type of their attributes, we cannot in general statically ensure that expressions within policies are well-typed. Consequently, errors will be generated, and possibly managed, at evaluation-time when expression operators are applied to arguments of unexpected type.

Indeed, 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; otherwise, if at least one argument is error, error is returned, else (that is at least one argument is $\perp$ and none is error) $\perp$ is returned. The operators and and or implement a different treatment of these special values. Specifically, the operator and returns: (i) true if both operands are true; (ii) false if at least one operand is false; (iii) $\perp$ if at least one operand is $\perp$ and none is false or error; (iv) error otherwise, e.g., when an operand is not a boolean value. The operator or is the dual of and. Hence, the operators 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 rule ends with the instantiation of all the enclosed obligations, while that of a policy set ends with the instantiation of all the obligations in the sequence corresponding to the decision calculated for the policy. The instantiation of an obligation consists in evaluating every expression argument of the enclosed action. If an error occurs, the policy decision is changed to indet. Otherwise, the instantiated obligations are paired with the policy decision to form the PDP response.

Evaluating a policy set requires the application of the specified algorithm for combining the decisions resulting from the evaluation of various policies and, thus, resolving possible conflicts, e.g., whenever both decisions permit and deny occur. 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, prevails.
- 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 either according to a given precedence criterium or to policy applicability. The last two algorithms, instead, are borrowed from the work by Li et al. [21] 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 instantiated obligations according to the chosen instantiation strategy δ. There are two possible strategies. The all strategy requires evaluation of all policies in the input sequence and returns the instantiated 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 instantiated. Therefore, the instantiation strategies only affect the amount of instantiated obligations possibly returned. The greedy strategy, that reflects the management of obligations in XACML, may significantly improve the policy evaluation performance. Instead, the all strategy may require additional workload but, on the other hand, ensures that all the policies and their obligations are always taken into account.

As last step, the calculated PDP response is sent to the PEP for the enforcement. To this aim, the PEP must discharge all obligations and decide, by means of the chosen enforcement algorithm, how to enforce decisions not-app and indet. The algorithms are those popularised by XACML and, in particular, the deny-biased (resp., permit-biased) one 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 instantiation, but also the enforcement one. However, errors caused by optional obligations, that is 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 2. 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., a rule with effect permit) as follows

$$
\begin{aligned}
(\,\mathsf{permit}\ \ \mathsf{target} : {} & \mathsf{equal}(\mathsf{subject/role}, \text{“doctor”}\,) \\
& \mathsf{and}\ \mathsf{equal}(\mathsf{action/id}, \text{“write”}\,) \\
& \mathsf{and}\ \mathsf{in}(\,\text{“e-Pre-Write”}, \mathsf{subject/permission}) \\
& \mathsf{and}\ \mathsf{in}(\,\text{“e-Pre-Read”}, \mathsf{subject/permission}))
\end{aligned}
$$

The rule target[3] checks if the requester role is doctor, if the action is write, and if the subject's permissions include those for writing and reading an e-Prescription. The control that the resource type is equal to e-Prescription will be performed 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) is similarly expressed: the corresponding rule differs from the previous one for the action and the permissions. Instead, the rule corresponding to Requirement (3) only differs from the second one for the role.

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; again, to improve code readability, we use textual encoding for resources. Since all granted requests are explicitly authorised, choosing the p-over$_{\mathsf{all}}$ algorithm as combining strategy

[3] 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, such as HL7 [30], should be used for defining role checks.

seems a natural choice. Let thus Policy (P1) 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”)          (P1)
          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-p : [ m  log(system/time, resource/type,
                                     subject/id, action/id) ] }
```

Policy (P1) reports not only access controls but also an obligation formalising Requirement (4) about the logging of each authorised access. The arguments of the obligation action are separated by commas to increase their readability.

Let us now consider a FACPL request and evaluate it with respect to Policy (P1). For the sake of presentation, hereafter we write $n \triangleq t$ to assign the symbolic name n to the term t. Let us suppose that doctor Dr. House wants to write an e-Prescription; the corresponding request is defined as follows

```
req1 ≜ (subject/id, “Dr. House”) (subject/role, “doctor”)
       (action/id, “write”) (resource/type, “e-Prescription”)
       (subject/permission, “e-Pre-Read”)
       (subject/permission, “e-Pre-Write”)  ...
```

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 forms 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, which is a permit including a log obligation, is defined, e.g., as follows

```
⟨ permit  [ m  log(2016-10-22 10:15:12,
               “e-Prescription”, “Dr. House”, “write”) ]⟩
```

The instantiated obligation indicates that the PDP succeeded in retrieving and evaluating all the attributes occurring within the arguments of the action; run-time 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 (P1), 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 by a pharmacist for a write action on an e-Prescription is not explicitly allowed by the requirements in Table 2; hence, it should be forbidden. However, the corresponding request

```
req2 ≜ (subject/id, “Dr. Wilson”) (subject/role, “pharmacist”)
       (action/id, “write”) (resource/type, “e-Prescription”)
       (subject/permission, “e-Pre-Read”)  ...
```

would evaluate to not-app. In fact, all enclosed rules do not apply, because their targets do not match, and the resulting not-app decisions are combined by the p-over$_{\text{all}}$ 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, like in 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 (P1) with the d-unless-p$_{\text{all}}$ one. This implies that deny is taken as the default decision and is returned whenever no rule returns permit. Alternatively, we can achieve the same by using a policy set defined as the combination, through the p-over$_{\text{all}}$ algorithm, of Policy (P1) 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 (P2) be defined as

```
{ p-over_all
  policies : { ... Policy (P1) ... }  (deny)
  obl-p : [ o  compress( ) ]                                  (P2)
  obl-d : [ m  mailTo(resource/patient-mail,
               “Data request by unauthorised subject”) ] }
```

Policy (P2) reports two obligations formalising, respectively, the last two requirements of Table 2: (i) if possible, data are exchanged in compressed form by means of an obligation for the effect permit and (ii) a patient is informed about unauthorised attempts to access her data by means of an obligation for the effect deny. The type ‘optional’ is exploited so that compressed exchanges are not strictly required but, e.g., only whenever the corresponding service is available.

Policy (P2) 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 to Policy (P2) a check on the patient identifier to which the policy applies, such as target : equal(“Alice”, resource/patient-id). 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, in which for example 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 [31].

As shown before, it could be challenging to identify unexpected authorisations and to determine whether policy fixes affect decisions 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*, which 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 semantics 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 domain to a specific semantic domain. These functions are inductively defined on the FACPL syntax through appropriate semantic clauses following a 'point-wise' style. For instance, on the syntactic domain *Policy* representing all FACPL policies, we formalise the function $\mathcal{P}$ that defines a semantic domain mapping FACPL requests to PDP responses.

In the sequel, we convene that the application of the semantic functions is left-associative, omit parenthesis whenever possible, and surround syntactic objects with the 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 1 and 3 defining the FACPL syntax denominates the set of constructs of the syntactic category defined by the corresponding EBNF rule. For example, the nonterminal *Policy* identifies the set of all FACPL policies. The used notations are summarised in Table 4; the missing semantic domains coincide with the corresponding syntactic ones.

In the rest of this section, we detail the semantics of requests in Section 5.1, PDP in Sections 5.2 and 5.3, PEP in Section 5.4, and PAS in Section 5.5. Moreover, we present some key properties of the semantics in Section 5.6 and an application of the semantics to the e-Health case study in Section 5.7.

5.1 Semantics of Requests

The meaning of a request[4] is an element of the set $R \triangleq \mathit{Name} \rightarrow (\mathit{Value} \cup 2^{\mathit{Value}} \cup \{\bot\})$, i.e., 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 $\mathcal{R} : \mathit{Request} \rightarrow R$, defined as follows:

$$\mathcal{R}[\![(n', v')]\!]n = \begin{cases} v' \ \texttt{if}\ n = n' \\ \bot \ \texttt{otherwise} \end{cases}$$

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

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 $\uplus$, which is straightforwardly defined by case analysis on the first argument as follows

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

where we let $V \in 2^{\mathit{Value}}$.

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

The first row of the table contains the clauses for basic expressions, which are attribute names and literal values. The semantics of the expression formed by a name n is a function that, given a semantic request r in input, returns the value that r associates to n. Similarly, the semantics of a value is a function that always returns the value itself.

The remaining clauses, one for each operator, present the semantics of expression operators. In particular, each clause uses straightforward semantic operators for composing denotations (e.g., $=$ corresponds to equal), and implements the management strategy for the special values $\bot$ and error. The clauses 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 one argument is $\bot$ and there is no error. The clauses of operators and and or possibly mask these special values by implementing the behaviour informally described in Section 4.2. It is worth noting that the explicit management of missing attributes and evaluation errors ensures a full account of crucial aspects of access control policy evaluation, usually neglected by other proposals from the literature [10], [11], [19]. The only proposals considering the role of missing attributes are those by Crampton and Morisset [9] and Crampton and Williams [20], but they only consider a simplified policy language and assume that expressions cannot generate errors.

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

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

[4] 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 from the environment by the context handler (steps 5-8 in Figure 1).

Table 4
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	$pepAct$		$PepAction$	
Instantiated obligations	io		$IObligation$	
Obligations	o	$\mathcal{O}$	$Obligation$	$R \rightarrow IObligation \cup \{\mathsf{error}\}$
PDP Responses	res		$PDPReponse$	
Policies	p	$\mathcal{P}$	$Policy$	$R \rightarrow PDPReponse$
Policy Decision Points	pdp	$\mathcal{PDP}$	PDP	$R \rightarrow PDPReponse$
Combining algorithms	a	$\mathcal{A}$	$Alg \times Policy^{+}$	$R \rightarrow PDPReponse$
Decisions	dec		$Decision$	
Enforcement algorithms	ea	$\mathcal{EA}$	$EnfAlg$	$PDPReponse \rightarrow Decision$
Policy Auth. System	pas	$\mathcal{PAS}$	PAS	$Request \rightarrow Decision$

Table 5
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 $i \neq j$

$$\mathcal{E}\llbracket n \rrbracket r = r(n)$$

$$\mathcal{E}\llbracket v \rrbracket r = v$$

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

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

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

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

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

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

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 the sequence contains error or $\bot$. Therefore, the evaluation of $\mathcal{E}\llbracket expr^* \rrbracket r$ fails if any of the expressions forming $expr^*$ evaluates to error or $\bot$.

The semantics of obligations corresponds to their instantiation. Formally, it is given by the function $\mathcal{O} : Obligation \rightarrow (R \rightarrow IObligation \cup \{\mathsf{error}\})$, whose definition clause is

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

where w stands for a literal value or a set of literal values. Thus, given a request, the instantiation of an obligation succeeds if 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

$$\mathcal{O}\llbracket \epsilon \rrbracket r = \epsilon \qquad \mathcal{O}\llbracket o' \, o^* \rrbracket r = \mathcal{O}\llbracket o' \rrbracket r \bullet \mathcal{O}\llbracket o^* \rrbracket r \qquad \text{(S-3b)}$$

Notably, a sequence of instantiated obligations is returned only if every obligation in the sequence is successfully instantiated; 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 instantiated obligations. Formally, it is given by the function $\mathcal{P} : Policy \rightarrow (R \rightarrow PDPReponse)$ that has two defining clauses: one for rules and one for policy sets. The clause for rules is

$$\mathcal{P}\llbracket (e\ \ \mathsf{target} : expr\ \ \mathsf{obl} : o^*\ ) \rrbracket r = \begin{cases} \langle e\ \ io^* \rangle & \texttt{if } \mathcal{E}\llbracket expr \rrbracket r = \mathsf{true}\ \wedge\ \mathcal{O}\llbracket o^* \rrbracket r = io^* \\ \mathsf{not\text{-}app} & \texttt{if } \mathcal{E}\llbracket expr \rrbracket r = \mathsf{false}\ \vee\ \mathcal{E}\llbracket expr \rrbracket r = \bot \\ \mathsf{indet} & \texttt{otherwise} \end{cases} \qquad \text{(S-4a)}$$

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 are successfully instantiated. In this case, the instantiated 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 instantiating the obligations.

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 and sequence of policies to a function from requests to PDP responses. The clause for policy sets is then

$$\mathcal{P}[\![\{a\ \ \mathsf{target}: expr\ \ \mathsf{policies}: p^+\ \mathsf{obl\text{-}p}: o_p^*\ \ \mathsf{obl\text{-}d}: o_d^*\ \}]\!]r =$$

$$\begin{cases} \langle \mathsf{permit}\ \ io_1^* \bullet io_2^* \rangle & \texttt{if}\ \mathcal{E}[\![expr]\!]r = \mathsf{true} \\ & \quad \wedge\, \mathcal{A}[\![a, p^+]\!]r = \langle \mathsf{permit}\ io_1^* \rangle \\ & \quad \wedge\, \mathcal{O}[\![o_p^*]\!]r = io_2^* \\ \langle \mathsf{deny}\ \ io_1^* \bullet io_2^* \rangle & \texttt{if}\ \mathcal{E}[\![expr]\!]r = \mathsf{true} \\ & \quad \wedge\, \mathcal{A}[\![a, p^+]\!]r = \langle \mathsf{deny}\ io_1^* \rangle \\ & \quad \wedge\, \mathcal{O}[\![o_d^*]\!]r = io_2^* \\ \mathsf{not\text{-}app} & \texttt{if}\ \mathcal{E}[\![expr]\!]r = \mathsf{false} \\ & \quad \vee\, \mathcal{E}[\![expr]\!]r = \bot \\ & \quad \vee\, (\mathcal{E}[\![expr]\!]r = \mathsf{true} \\ & \qquad\qquad \wedge\, \mathcal{A}[\![a, p^+]\!]r = \mathsf{not\text{-}app}) \\ \mathsf{indet} & \texttt{otherwise} \end{cases} \tag{S-4b}$$

Thus, the policy set applies to a request when the target evaluates to true, the semantics of the combining algorithm a returns an effect e and a sequence of instantiated obligations io_1^*, and all the enclosed obligations for the effect e are successfully instantiated and return a sequence io_2^*. In this case, the PDP response contains e and the concatenation of the sequences io_1^* and io_2^*. Instead, if the target evaluates to false or to $\bot$, or the combining algorithm returns $\mathsf{not\text{-}app}$, the policy set does not apply to the request. The response is indet in the remaining cases, i.e., when an error occurred in the evaluation of the target or of the obligations, or when the evaluation of the combining algorithm returned indet.

Finally, the semantics of a PDP is a function from requests to PDP responses defined by the following clauses:

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

When the PDP is a single policy, its semantics is given in terms of the function $\mathcal{P}$, otherwise it is given in terms of the function $\mathcal{A}$.

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, such as $\mathsf{p\text{-}over}$ or $\mathsf{d\text{-}over}$; the corresponding semantic operator is identified as $\otimes\mathsf{alg}$ and is defined by means of a two-dimensional matrix that, given two PDP responses, calculates the resulting combined response. For instance, Table 6(a) reports the combination matrix for the $\otimes\mathsf{p\text{-}over}$ operator. Basically, the matrix specifies the precedences among the permit, deny, $\mathsf{not\text{-}app}$ and indet decisions, and shows how the resulting sequence of instantiated obligations is obtained, that is by concatenating the instantiated obligations of the responses whose decision matches the combined one. The binary operators shown in Appendix A define all the other combining algorithms in the same manner. We convene that, when applied to single policies, operators $\otimes\mathsf{p\text{-}unless\text{-}d}$ and $\otimes\mathsf{d\text{-}unless\text{-}p}$ turn the $\mathsf{not\text{-}app}$ and indet responses into, respectively, $\langle\mathsf{permit}\ \ \epsilon\rangle$ and $\langle\mathsf{deny}\ \ \epsilon\rangle$, while the remaining operators leave them unchanged.

The semantics of the combining algorithms can be now formalised by the function $\mathcal{A} : Alg \times Policy^+ \to (R \to PDPReponse)$. This function is defined inductively on the structure of the input policy sequence by means of two pairs of clauses, one for each instantiation strategy. If the all strategy is adopted, the definition clauses are then

$$\begin{aligned} \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p]\!]r &= \otimes\mathsf{alg}(\mathcal{P}[\![p]\!]r) \\ \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p^+p']\!]r &= \otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p^+]\!]r, \mathcal{P}[\![p']\!]r) \end{aligned} \tag{S-6a}$$

Notably, the all strategy always requires evaluation of all input policies. Instead, the definition clauses for the greedy strategy are

$$\begin{aligned} &\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p]\!]r = \otimes\mathsf{alg}(\mathcal{P}[\![p]\!]r) \\ &\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p^+p']\!]r = \\ &\begin{cases} \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}},p^+]\!]r & \texttt{if}\ isFinal_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}},p^+]\!]r) \\ \otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}},p^+]\!]r,\mathcal{P}[\![p']\!]r) & \texttt{otherwise} \end{cases} \end{aligned} \tag{S-6b}$$

Differently from the all strategy, the greedy one halts the evaluation of the input policy sequence as soon as a final decision is determined, without necessarily evaluating all the policies in the sequence. Indeed, given a response in input, the auxiliary predicates $isFinal_{\mathsf{alg}}$, one for for each combining algorithm alg, check if the response decision is final according to the algorithm alg, that is if such decision cannot change due to further combinations. Predicates $isFinal_{\mathsf{alg}}$ are defined in Table 6(b); 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 $\mathsf{p\text{-}over}$ 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 $\mathsf{first\text{-}app}$ algorithm, instead, all decisions except $\mathsf{not\text{-}app}$ are final, since they represent the fact that the first applicable policy has been already found. Both consensus algorithms have indet as final decision, because no form of consensus can be reached once an indet is obtained. Similarly, the $\mathsf{one\text{-}app}$ algorithm has indet as final decision.

We conclude with an example illustrating the differences between the two instantiation strategies. Let us consider the sequence of policies $p_1\ p_2\ p_3$. If it is given in input to a combining algorithm using the all instantiation strategy, then we get the following unfolding of the inductive clauses (S-6a):

$$\begin{aligned} &\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1\ p_2\ p_3]\!]r \\ &\quad = \otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1\ p_2]\!]r, \mathcal{P}[\![p_3]\!]r) \\ &\quad = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1]\!]r, \mathcal{P}[\![p_2]\!]r), \mathcal{P}[\![p_3]\!]r) \\ &\quad = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r), \mathcal{P}[\![p_2]\!]r), \mathcal{P}[\![p_3]\!]r) \end{aligned}$$

which means that all the three input policies are evaluated.

Instead, if the algorithm uses the greedy strategy, then the unfolding of the inductive clauses (S-6b) for the example sequence of policies is shown in Table 7. Now, if the evaluation of policy p_1 returns a final decision, that is $isFinal_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1]\!]r)$ holds, then we have

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r = \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1]\!]r = \otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r)\,.$$

Table 6
Auxiliary definitions for the semantics of combining algorithms: (a) combination matrix for the $\otimes$p-over operator, where res_1 and res_2 indicate the first and the second argument, respectively; (b) definition of the $isFinal_{\mathsf{alg}}(res)$ predicates

(a)

$_{res_1}\backslash^{res_2}$	$\langle\mathsf{permit}\ \ io_2^*\rangle$	$\langle\mathsf{deny}\ \ io_2^*\rangle$	not-app	indet
$\langle\mathsf{permit}\ \ io_1^*\rangle$	$\langle\mathsf{permit}\ \ io_1^* \bullet io_2^*\rangle$	$\langle\mathsf{permit}\ \ io_1^*\rangle$	$\langle\mathsf{permit}\ \ io_1^*\rangle$	$\langle\mathsf{permit}\ \ io_1^*\rangle$
$\langle\mathsf{deny}\ \ io_1^*\rangle$	$\langle\mathsf{permit}\ \ io_2^*\rangle$	$\langle\mathsf{deny}\ \ io_1^* \bullet io_2^*\rangle$	$\langle\mathsf{deny}\ \ io_1^*\rangle$	indet
not-app	$\langle\mathsf{permit}\ \ io_2^*\rangle$	$\langle\mathsf{deny}\ \ io_2^*\rangle$	not-app	indet
indet	$\langle\mathsf{permit}\ \ io_2^*\rangle$	indet	indet	indet

(b)

$$isFinal_{\textsf{p-over}}(res) = \begin{cases} \mathsf{true} & \mathtt{if}\ res.dec = \mathsf{permit} \\ \mathsf{false} & \mathtt{otherwise} \end{cases}$$

$$isFinal_{\textsf{d-over}}(res) = \begin{cases} \mathsf{true} & \mathtt{if}\ res.dec = \mathsf{deny} \\ \mathsf{false} & \mathtt{otherwise} \end{cases}$$

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

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

$$isFinal_{\textsf{first-app}}(res) = \begin{cases} \mathsf{false} & \mathtt{if}\ res.dec = \textsf{not-app} \\ \mathsf{true} & \mathtt{otherwsise} \end{cases}$$

$$isFinal_{\textsf{one-app}}(res) = \begin{cases} \mathsf{true} & \mathtt{if}\ res.dec = \mathsf{indet} \\ \mathsf{false} & \mathtt{otherwsise} \end{cases}$$

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

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

Table 7
Unfolding of clause (S-6b) for the example sequence of policies $p_1\ p_2\ p_3$

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2\ p_3]\!]r = \begin{cases} \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r = & \mathtt{if}\ isFinal_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r) \\ \quad \begin{cases} \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1]\!]r = \otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r) & \mathtt{if}\ isFinal_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1]\!]r) \\ \otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1]\!]r, \mathcal{P}[\![p_2]\!]r) & \mathtt{otherwise} \end{cases} & \\ \otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r, \mathcal{P}[\![p_3]\!]r) & \mathtt{otherwise} \end{cases}$$

Hence, $isFinal_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r)$ holds too, thus we get

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2\ p_3]\!]r = \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r = \otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r)$$

which means that only the first policy is evaluated. Otherwise, if the evaluation of policy p_1 does not return a final decision, then we have

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r), \mathcal{P}[\![p_2]\!]r)\,.$$

Now, if this evaluation returns a final decision, then we get

$$\begin{aligned} \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2\ p_3]\!]r &= \mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r \\ &= \otimes\mathsf{alg}(\otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r), \mathcal{P}[\![p_2]\!]r) \end{aligned}$$

which means that only the first two policies are evaluated. Instead, if $\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2]\!]r$ does not return a final decision, then all the three policies are evaluated and we get

$$\begin{aligned} &\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1\ p_2\ p_3]\!]r \\ &\quad = \otimes\mathsf{alg}(\otimes\mathsf{alg}(\otimes\mathsf{alg}(\mathcal{P}[\![p_1]\!]r), \mathcal{P}[\![p_2]\!]r), \mathcal{P}[\![p_3]\!]r) \end{aligned}$$

as if the all strategy is used.

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 $(\!(\ )\!) : IObligation^* \to \{\mathsf{true}, \mathsf{false}\}$ that, given a sequence of instantiated obligations, executes such obligations and returns a boolean value that indicates whether the evaluation has successfully completed. Since failures caused by optional obligations can be safely ignored by the PEP, only failures of mandatory obligations have to be taken into account. The function is thus defined as follows

$$(\!(\epsilon)\!) = \mathsf{true}$$

$$(\!([\,\mathsf{o}\ \ pepAct(w^*)\,] \bullet io^*)\!) = (\!(io^*)\!)$$

$$(\!([\,\mathsf{m}\ \ pepAct(w^*)\,] \bullet io^*)\!) = \begin{cases} (\!(io^*)\!) & \mathtt{if}\ pepAct(w^*) \Downarrow \mathsf{ok} \\ \mathsf{false} & \mathtt{otherwise} \end{cases}$$

where $\Downarrow$ok denotes if 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 the predicate $\Downarrow$ok is hence unspecified too; we just assume that it is total and deterministic. In other words, the syntactic domain $PepAction$ is a parameter of the syntax, while the predicate $\Downarrow$ok is a parameter of the semantics. The latter parameter could be refined to deal with, e.g., obligations to be enforced after the decision releasing (see Section 10). For example, discharging obligations could simply refer to the fact that the system has taken charge of their execution, rather than to the fact that they have been completely executed.

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

$$\mathcal{EA}[\![\textsf{deny-biased}]\!]res = \begin{cases} \mathsf{permit} & \mathtt{if}\ res.dec = \mathsf{permit}\ \wedge\ (\!(res.io)\!) \\ \mathsf{deny} & \mathtt{otherwise} \end{cases} \qquad \text{(S-7a)}$$

where $res.dec$ and $res.io$ indicate the decision and the sequence of instantiated obligations of the response res,

Table 8
Definition of the `toAll()` function

$$\mathtt{toAll}((e\ \ \mathsf{target} : expr\ \ \mathsf{obl} : o^*\ )) = (e\ \ \mathsf{target} : expr\ \ \mathsf{obl} : o^*\ )$$

$$\mathtt{toAll}(\{\mathsf{alg}_\delta\ \ \mathsf{target} : expr\ \ \mathsf{policies} : p^+\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}) = \{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target} : expr\ \ \mathsf{policies} : \mathtt{toAll}(p^+)\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}$$

$$\mathtt{toAll}(p'\ p^+) = \mathtt{toAll}(p')\ \mathtt{toAll}(p^+)$$

respectively. 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 of the previous one, whereas the clause for the base algorithm is

$$\mathcal{EA}[\![\mathsf{base}]\!]res = \begin{cases} \mathsf{permit} & \mathtt{if}\ res.dec = \mathsf{permit}\ \wedge\ (\!(res.io)\!) \\ \mathsf{deny} & \mathtt{if}\ res.dec = \mathsf{deny}\ \wedge\ (\!(res.io)\!) \\ \mathsf{not\text{-}app} & \mathtt{if}\ res.dec = \mathsf{not\text{-}app} \\ \mathsf{indet} & \mathtt{otherwise} \end{cases} \qquad \text{(S-7b)}$$

Both decisions permit and deny are enforced only if all obligations in the PDP response are successfully discharged, otherwise indet is enforced. Instead not-app and indet decisions are enforced without modifications. Therefore, a PEP using the base algorithm never changes PDP decisions, unless unsuccessful discharge of obligations. The rationale behind this behaviour is twofolds: it permits, on the one hand, debugging the obligation discharging process of PEP, on the other hand, returning the decisions as-is to the controlled system, which can then manage the two forms of error in different ways.

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 $\mathcal{PAS} : PAS \rightarrow (Request \rightarrow Decision)$ defined by

$$\mathcal{PAS}[\![\{\ \mathsf{pep} : ea\ \ \mathsf{pdp} : pdp\ \}, req]\!] = \mathcal{EA}[\![ea]\!](\mathcal{PDP}[\![pdp]\!](\mathcal{R}[\![req]\!])) \qquad \text{(S-8)}$$

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, that is $\mathcal{PDP}[\![pdp]\!]$, which returns a response that is passed to the semantics of the PEP, that is $\mathcal{EA}[\![ea]\!]$. The latter function returns the final decision of the Policy Authorisation System when given the request req in input.

5.6 Properties of the Semantics

We present in this section some key properties and results regarding the FACPL semantics.

The main result is that the semantics is *total* and *deterministic*. This means that it is defined for all possible input pairs consisting of a FACPL specification, that is a Policy Authorisation System, and a request, and that it always returns the same decision when applied to a specific pair.

Theorem 5.1 (Total and Deterministic Semantics)**.**

1) *For all $pas \in PAS$ and $req \in Request$, there exists a $dec \in Decision$, such that $\mathcal{PAS}[\![pas, req]\!] = dec$.*
2) *For all $pas \in PAS$, $req \in Request$ and $dec, dec' \in Decision$, it holds that*
$$\mathcal{PAS}[\![pas, req]\!] = dec\ \ \wedge\ \ \mathcal{PAS}[\![pas, req]\!] = dec' \Rightarrow\ \ dec = dec'\,.$$

Proof (sketch). It boils down to showing that $\mathcal{PAS}$ is a total and deterministic function (see Appendix B.1). □

The following result states that the semantics of a policy does agree, for what concerns the resulting decision, with that of the policy obtained by replacing everywhere the greedy instantiation strategy with the all one. Of course, the semantics of the two policies may differ for what concerns the returned instantiated obligations. To formally express such result, we introduce the auxiliary function `toAll()` taking care of replacing the instantiation strategy used by the algorithms enclosed within a policy. Its straightforward inductive definition is given in Table 8.

Theorem 5.2 (Instantiation Strategy Correspondence)**.** *For all $p \in Policy$ and $r \in R$, it holds that*

$$\mathcal{P}[\![p]\!]r = \langle dec\ io_1^* \rangle\ \ \Leftrightarrow\ \ \mathcal{P}[\![\mathtt{toAll}(p)]\!]r = \langle dec\ io_2^* \rangle\ .$$

Proof (sketch). The proof (see Appendix B.1) is by induction on the *depth*, which is the nesting level, of p. □

This theorem ensures that the analysis for policies enclosing algorithms using the greedy instantiation strategy can be soundly carried out by verifying the corresponding policies only using the all strategy, as stated by Theorem 6.2.

Another consequence of the previous theorem regards the application of a combining algorithm to a policy sequence. In this case, while evaluating the input policies, once the decision calculated by the algorithm with the greedy strategy on the whole input sequence has been determined, then it cannot change due to evaluation of the remaining policies. This means that obligations considered by the all strategy but not by the greedy one, and in particular the instantiation errors they may raise, do not affect the calculated decision. This property is formalised by the following corollary.

Corollary 5.3. *If* $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1^+\ \ p_2^*]\!]r).dec = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1^+]\!]r).dec$ *then* $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1^+\ \ p_2^*]\!]r).dec = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1^+]\!]r).dec.$

Proof. From Theorem 5.2 we have $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p_1^+\ p_2^*]\!]r).dec = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1^+\ p_2^*]\!]r).dec$. Then the thesis directly follows from the hypothesis. □

We now consider the so-called *reasonability properties* of Tschantz and Krishnamurthi [22] that precisely characterise the expressiveness of a policy language. FACPL enjoys the property called *independent composition* of policies, which means that 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 other hand, FACPL in general ensures neither *safety*, that is a granted request may not be granted anymore if it is extended with new attributes, nor *monotonicity*, that is the introduction of a new policy in a combination of policies may change a permit decision to a different one. This should be somehow expected as these latter two properties are enjoyed neither by XACML nor by any other policy language featuring deny rules and combining algorithms similar to those we have presented.

We conclude by highlighting the relationship between attribute names occurring in a policy and names defined by requests. By letting $Names(p)$ indicate the set of attribute names occurring in the expressions within p, we can state the following result which has important practical implications on the feasibility of the automated analysis.

Lemma 5.4 (Policy relevant attributes)**.** *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 (sketch). The property straightforwardly derives from the semantics of FACPL expressions and from Theorem 5.1 (see Appendix B.1). □

5.7 Semantics of a policy for the e-Health case study

In this section, we illustrate how to apply the clauses defining the semantics of the FACPL constructs for deriving the semantics of the policy (P1) of the e-Health case study. For the sake of presentation, we directly consider the application of the resulting semantic function to the request $\mathsf{req1}$ (at page 8). Formally, this amounts to evaluating $\mathcal{P}[\![\,(\mathrm{P1})\,]\!]\mathsf{req1}$.

According to clause (S-4b), we have first to evaluate the target. By applying the clauses in Table 5, we get

$$\begin{aligned}
&\mathcal{E}[\![\mathsf{equal}(\mathsf{resource/type}, \text{“e-Prescription”}))]\!]\mathsf{req1}\\
&\quad = (\mathcal{E}[\![\mathsf{resource/type}]\!]\mathsf{req1} = \mathcal{E}[\![\text{“e-Prescription”}]\!]\mathsf{req1})\\
&\quad = (\mathsf{req1}(\mathsf{resource/type}) = \text{“e-Prescription”})\\
&\quad = (\text{“e-Prescription”} = \text{“e-Prescription”})\\
&\quad = \mathsf{true}
\end{aligned}$$

Due to the result of the target evaluation, we are falling within one of the first two cases of clause (S-4b). Hence we proceed by evaluating $\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, (\mathrm{R1})\ (\mathrm{R2})\ (\mathrm{R3})]\!]\mathsf{req1}$, where (Ri) stands for the i-th rule enclosed within (P1). According to clauses (S-6a), we have to evaluate all rules (Ri). Since the target of rule (R1) evaluates to true, while the targets of the other rules evaluate to false, from clause (S-4a), we have

$$\begin{aligned}
\mathcal{P}[\![(\mathrm{R1})]\!]\mathsf{req1} &= \mathsf{permit}\\
\mathcal{P}[\![(\mathrm{R2})]\!]\mathsf{req1} &= \mathsf{not\text{-}app}\\
\mathcal{P}[\![(\mathrm{R3})]\!]\mathsf{req1} &= \mathsf{not\text{-}app}
\end{aligned}$$

Therefore, by applying clauses (S-6a), we get:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, (\mathrm{R1})\ (\mathrm{R2})\ (\mathrm{R3})]\!]\mathsf{req1}\\
&\quad = \otimes\mathsf{p\text{-}over}(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, (\mathrm{R1})\ (\mathrm{R2})]\!]\mathsf{req1}, \mathcal{P}[\![(\mathrm{R3})]\!]\mathsf{req1})\\
&\quad = \otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}},(\mathrm{R1})]\!]\mathsf{req1},\mathcal{P}[\![(\mathrm{R2})]\!]\mathsf{req1}),\\
&\qquad\qquad \mathcal{P}[\![(\mathrm{R3})]\!]\mathsf{req1})\\
&\quad = \otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\mathcal{P}[\![(\mathrm{R1})]\!]\mathsf{req1}),\mathcal{P}[\![(\mathrm{R2})]\!]\mathsf{req1}),\\
&\qquad\qquad \mathcal{P}[\![(\mathrm{R3})]\!]\mathsf{req1})\\
&\quad = \otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\mathsf{permit}), \mathsf{not\text{-}app}), \mathsf{not\text{-}app})\\
&\quad = \otimes\mathsf{p\text{-}over}(\otimes\mathsf{p\text{-}over}(\mathsf{permit}, \mathsf{not\text{-}app}), \mathsf{not\text{-}app})\\
&\quad = \otimes\mathsf{p\text{-}over}(\mathsf{permit}, \mathsf{not\text{-}app})\\
&\quad = \mathsf{permit}
\end{aligned}$$

This means that we are falling within the first case of clause (S-4b). Therefore, we need now to evaluate the obligations to be discharged if the resulting effect is permit. According to clause (S-3a), we have

$$\begin{aligned}
&\mathcal{O}[\![[\mathsf{m}\ \ \mathsf{log}(\mathsf{system/time}, \mathsf{resource/type},\\
&\qquad\qquad\qquad\qquad \mathsf{subject/id}, \mathsf{action/id})]]\!]\mathsf{req1}\\
&\qquad = [\,\mathsf{m}\ \ \mathsf{log}(\mathsf{2016\text{-}10\text{-}22\ 10{:}15{:}12},\\
&\qquad\qquad\qquad \text{“e-Prescription”}, \text{“Dr. House”}, \text{“write”})\,]
\end{aligned}$$

where basically the evaluation is obtained by replacing each attribute name by the corresponding value in $\mathsf{req1}$. All values, but that relative to system/time, were explicitly present in the original request; the current time is a runtime information retrieved from the environment through the context handler.

By concluding, we put together, according to clause (S-4b), the results of the evaluation of the combining algorithm and of the instantiated obligation, thus getting

$$\begin{aligned}
&\mathcal{P}[\![\,(\mathrm{P1})\,]\!]\mathsf{req1}\\
&\qquad = \langle\ \mathsf{permit}\ \ [\,\mathsf{m}\ \ \mathsf{log}(\mathsf{2016\text{-}10\text{-}22\ 10{:}15{:}12},\\
&\qquad\qquad\qquad \text{“e-Prescription”}, \text{“Dr. House”}, \text{“write”})\,]\rangle
\end{aligned}$$

6 FACPL Constraint-based Representation

The analysis of access control policies provides effective means for ensuring the correctness of policy enforcement and, consequently, confidentiality and integrity of system resources. In the case of FACPL, the analysis is made difficult by the hierarchical structure of policies, the presence of conflict resolution strategies and the intricacies deriving from the many involved controls. Moreover, no off-the-shelf analysis tool directly takes FACPL specifications as input. Hence, for enabling the analysis of FACPL policies through well-established and efficient software tools, we introduce a constraint formalism that permits, on the one hand, to uniformly represent policies and, on the other hand, to perform extensive checks of requests.

The constraint-based representation we propose specifies satisfaction problems in terms of formulae based on multiple theories, e.g., boolean and linear arithmetics. Such SMT formulae are supported by many automatic solvers, such as Z3 [18], CVC4 [32], and Yices [33]. This has paved the way to an extensive employment of SMT formulae in diverse analysis applications [15].

This section introduces our constraint-based representation of FACPL policies, while the analysis it enables is presented in Section 7. We first present the constraint formalism

Table 9
Syntax of constraints; $Value$ and $Name$ are defined in Table 1

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

in Section 6.1. Then we introduce the constraint representation of FACPL policies in Section 6.2 and some crucial results stating that it is a semantic-preserving representation in Section 6.3. Finally, we show some examples of constraints obtained from our e-Health case study in Section 6.4.

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 through some conditions on a set of attribute names[5]. An assignment of values to attribute names satisfies a constraint if all constraint conditions are matched. Our formalism, besides usual operators and values, explicitly considers the role of missing attributes, by assigning $\bot$ to attribute names, and of run-time errors, which correspond to type mismatches in constraint evaluations. In fact, according to the usually accepted semantics of access control policies (besides XACML, see the works by Crampton and Morisset [9] and Crampton and Williams [20]), a condition involving a missing attribute should not be evaluated to false by default.

Syntax. Constraints are written according to the grammar in Table 9. Thus, a constraint can be a literal value, an attribute name, or a more complex constraint obtained through predicates $\texttt{isMiss}()$, $\texttt{isErr}()$ and $\texttt{isBool}()$, or through boolean, comparison and arithmetic operators. The operators $\neg$, $\wedge$ and $\vee$ are the usual boolean ones, while $\dot{\neg}$, $\dot{\wedge}$ and $\dot{\vee}$ correspond to the 4-valued ones of FACPL expressions which implement the special management of $\bot$ and error values.

In the sequel, in addition to the notations of Table 4, 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 $\mathcal{C} : Constr \to (R \to Value \cup 2^{Value} \cup \{\mathsf{error}, \bot\})$ inductively defined by the clauses in Table 10, where 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, returns a literal value or a set of literal values or one of the special values $\bot$ and error.

The semantics of constraints, except for the cases of predicates and usual boolean operators, mimics the semantic definitions of the corresponding FACPL expression operators defined in Table 5. For instance, 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 usual boolean operators are instead defined by 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. The 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 a FACPL policy is a logical combination of the constraints representing targets, obligations and combining algorithms occurring within the policy. The translation is formally, and *compositionally*, defined by a family of translation functions $\mathcal{T}_{\cdot}$, that return the constraints representing the different FACPL terms. We use the brackets $\{\!|$ and $|\!\}$ to represent the application of a translation function to a syntactic term.

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

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

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

$$\begin{array}{ll}
\texttt{getCop}(\mathsf{and}) = \dot{\wedge} & \texttt{getCop}(\mathsf{or}) = \dot{\vee}\\
\texttt{getCop}(\mathsf{equal}) = \ = & \texttt{getCop}(\mathsf{in}) = \in\\
\texttt{getCop}(\mathsf{greater\text{-}than}) = \ > & \texttt{getCop}(\mathsf{add}) = \ +
\end{array}$$

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

$$\begin{aligned}
&\mathcal{T}_{Ob}\{\!|\epsilon|\!\} = \mathsf{true}\\
&\mathcal{T}_{Ob}\{\!|o'\ o^*|\!\} = \mathcal{T}_{Ob}\{\!|o'|\!\} \wedge \mathcal{T}_{Ob}\{\!|o^*|\!\}\\
&\mathcal{T}_{Ob}\{\!|[t\ PepAction(expr^*)]|\!\} =\\
&\quad \textstyle\bigwedge_{expr' \in expr^*} \neg\texttt{isMiss}(\mathcal{T}_E\{\!|expr'|\!\}) \wedge \neg\texttt{isErr}(\mathcal{T}_E\{\!|expr'|\!\})
\end{aligned} \tag{T-2}$$

Hence, a sequence of obligations corresponds to the conjunction of the constraints representing each obligation. When translating a single obligation, predicates $\texttt{isMiss}()$ and $\texttt{isErr}()$ are used to check the instantiation conditions,

[5] 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 10
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 $i \neq j$

$$\mathcal{C}\llbracket n \rrbracket r = r(n) \qquad \mathcal{C}\llbracket v \rrbracket r = v$$

$$\mathcal{C}\llbracket \mathtt{isMiss}(c) \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \perp \\ \mathsf{false} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket \mathtt{isErr}(c) \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \mathsf{error} \\ \mathsf{false} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket \mathtt{isBool}(c) \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r \in \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{false} & \mathtt{otherwise} \end{cases}$$

$$\mathcal{C}\llbracket \dot{\neg}\ c \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \mathsf{false} \\ \mathsf{false} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \mathsf{true} \\ \perp & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \perp \\ \mathsf{error} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket c_1\ \dot{\wedge}\ c_2 \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{true} \\ \mathsf{false} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathsf{false}\ \mathtt{or}\ \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{false} \\ \perp & \mathtt{if}\ \mathcal{C}\llbracket c_i \rrbracket r = \perp\ \mathsf{and}\ \mathcal{C}\llbracket c_j \rrbracket r \in \{\mathsf{true}, \perp\} \\ \mathsf{error} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket c_1\ \dot{\vee}\ c_2 \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathsf{true}\ \mathtt{or}\ \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{true} \\ \mathsf{false} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{false} \\ \perp & \mathtt{if}\ \mathcal{C}\llbracket c_i \rrbracket r = \perp\ \mathsf{and}\ \mathcal{C}\llbracket c_j \rrbracket r \in \{\mathsf{false}, \perp\} \\ \mathsf{error} & \mathtt{otherwise} \end{cases}$$

$$\mathcal{C}\llbracket \neg\ c \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c \rrbracket r = \mathsf{false} \\ & \quad \mathtt{or}\ \mathcal{C}\llbracket c \rrbracket r = \perp \\ \mathsf{false} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket c_1\ \wedge\ c_2 \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathsf{true}\ \mathtt{and}\ \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{true} \\ \mathsf{false} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket c_1\ \vee\ c_2 \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathsf{true}\ \mathtt{or}\ \mathcal{C}\llbracket c_2 \rrbracket r = \mathsf{true} \\ \mathsf{false} & \mathtt{otherwise} \end{cases}$$

$$\mathcal{C}\llbracket c_1\ =\ c_2 \rrbracket r = \begin{cases} \mathsf{true} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r, \mathcal{C}\llbracket c_2 \rrbracket r \in T\ \mathsf{and}\ \mathcal{C}\llbracket c_1 \rrbracket r = \mathcal{C}\llbracket c_2 \rrbracket r \\ \mathsf{false} & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r, \mathcal{C}\llbracket c_2 \rrbracket r \in T\ \mathsf{and}\ \mathcal{C}\llbracket c_1 \rrbracket r \neq \mathcal{C}\llbracket c_2 \rrbracket r \\ \perp & \mathtt{if}\ \mathcal{C}\llbracket c_i \rrbracket r = \perp\ \mathsf{and}\ \mathcal{C}\llbracket c_j \rrbracket r \neq \mathsf{error} \\ \mathsf{error} & \mathtt{otherwise} \end{cases} \qquad \mathcal{C}\llbracket c_1\ +\ c_2 \rrbracket r = \begin{cases} \mathcal{C}\llbracket c_1 \rrbracket r + \mathcal{C}\llbracket c_2 \rrbracket r & \mathtt{if}\ \mathcal{C}\llbracket c_1 \rrbracket r, \mathcal{C}\llbracket c_2 \rrbracket r \in \mathit{Double} \\ \perp & \mathtt{if}\ \mathcal{C}\llbracket c_i \rrbracket r = \perp\ \mathsf{and}\ \mathcal{C}\llbracket c_j \rrbracket r \neq \mathsf{error} \\ \mathsf{error} & \mathtt{otherwise} \end{cases}$$

i.e., that the occurring expressions cannot evaluate to $\perp$ or error. The n-ary conjunction operator returns true if the considered obligation contains no expression, i.e., $expr^* = \epsilon$.

The translation function for policies, $\mathcal{T}_P$, exploits the translation functions previously introduced, as well as a function $\mathcal{T}_A$ representing the result 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 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 constraint c_l; e.g., $\downarrow_p$ returns the permit constraint c_p.

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

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

The clause takes into account the rule constituent parts and combines them according to the rule semantics (see clause (S-4a)). Because of the semantics of the constraint operator $\neg$, the not-app constraint is satisfied when the constraint corresponding to the target expression evaluates to false or to $\perp$. Instead, the negation of a constraint corresponding to a sequence of obligations represents the failure of their instantiation. 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. It is worth noting that this constraint is expressed in terms of predicates $\mathtt{isBool}()$, $\mathtt{isMiss}()$ and $\mathtt{isErr}()$, so to capture the different causes of decision indet. The clause for rules with effect deny is omitted, as it only differs from clause (T-3a) because the permit and deny constraints are swapped.

The clause for policy sets is

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

With respect to the clauses for rules, it additionally takes into account the result of the application of the combining algorithm according to the policy set semantics (see clause (S-4b)). 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 $\mathcal{T}_A : Alg \times Policy^+ \to PCT$ that, given an algorithm and a sequence of policies, returns a constraint tuple representing the result of the algorithm application. Its inductive definition is

$$\begin{aligned} \mathcal{T}_A\{\!|\mathsf{alg}_\delta, p|\!\} &= \mathsf{alg}(\mathcal{T}_P\{\!|p|\!\}) \\ \mathcal{T}_A\{\!|\mathsf{alg}_\delta, p^+p'|\!\} &= \mathsf{alg}(\mathcal{T}_A\{\!|\mathsf{alg}_\delta, p^+|\!\}, \mathcal{T}_P\{\!|p'|\!\}) \end{aligned} \tag{T-4}$$

Notably, the translation does not depend on the strategy δ. 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, the combination of two constraint tuples, say A and B, according to the $\mathsf{p\text{-}over}$ algorithm, is defined as follows

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{p\text{-}over}(A, B) =} \\
\langle\mathsf{permit}: & A \downarrow_p \vee B \downarrow_p \\
\;\mathsf{deny}: & (A \downarrow_d \wedge B \downarrow_d) \vee (A \downarrow_d \wedge B \downarrow_n) \vee (A \downarrow_n \wedge B \downarrow_d) \\
\;\mathsf{not\text{-}app}: & A \downarrow_n \wedge B \downarrow_n \\
\;\mathsf{indet}: & (A \downarrow_i \wedge \neg B \downarrow_p) \vee (\neg A \downarrow_p \wedge B \downarrow_i)\rangle
\end{array}$$

The combinations for the remaining algorithms are reported in Appendix A. In case of a single tuple in input, all the algorithms leave the tuple unchanged, but $\mathsf{p\text{-}unless\text{-}d}$, which given an input tuple A returns the tuple

$$\begin{array}{ll}
\langle\, \mathsf{permit} : A \downarrow_p \vee A \downarrow_n \vee A \downarrow_i & \mathsf{deny} : A \downarrow_d \\
\;\; \mathsf{not\text{-}app} : \mathsf{false} & \mathsf{indet} : \mathsf{false}\, \rangle
\end{array}$$

and $\mathsf{d\text{-}unless\text{-}p}$, 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, that is $\{Alg\ \ \mathsf{target}: \mathsf{true}\ \ \mathsf{policies}: Policy^{+}\}$.

6.3 Properties of the Translation

The key result regarding the translation is that the semantics of the constraint-based representation of a policy and the semantics of the policy itself do agree. Before presenting this result, we show for the constraint semantics a result analogous to Theorem 5.1.

Theorem 6.1 (Total and Deterministic Constraint Semantics)**.**

1) *For all $c \in Constr$ and $r \in R$, there exists an $el \in (Value \;\cup\; 2^{Value} \cup \{\mathsf{error}, \bot\})$, such that $\mathcal{C}[\![c]\!]r = el$.*

2) *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 (sketch). By structural induction on the syntax of c (see Appendix B.2). □

Theorem 6.2 (Policy Semantic Correspondence)**.** *For all $p \in Policy$ and $r \in R$, it holds that*

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

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

This theorem implies that the properties verified over the constraints resulting from the translation of a FACPL policy would return the same results as if they were directly proven on the FACPL policy itself. Thus, it ensures that the analysis we present in Section 7 is sound.

From the previous theorems it follows that policy constraint tuples partition the set of input requests, in other words each access request satisfies only one of the constraints of a policy constraint tuple. Essentially, the following corollary extends Theorem 6.1 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*

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

Proof. The thesis immediately follows from Theorems 6.1 and 6.2. □

Finally, it is worth noting that the translation does not depend on the algorithm instantiation strategy, as formally stated by the following lemma. The proof is omitted as it trivially follows from the definition of the translation; the key point is that in the algorithm translation clause (T-4) the instantiation strategy δ plays no role.

Lemma 6.4. *For all $p \in Policy$, it holds that*

$$\mathcal{T}_P\{\!|p|\!\} = \mathcal{T}_P\{\!|\mathtt{toAll}(p)|\!\}\ .$$

6.4 Constraint-based Representation of the e-Health case study

We now apply the translation functions introduced in Section 6.2 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

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

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

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

The translation proceeds by considering obligations; in this case they are missing hence the constraint true is obtained. Function $\mathcal{T}_P$ finally defines the constraint tuple for the rule as follows

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

The tuples for the rules addressing Requirements (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 (P1). Besides the target expression, which is straightforwardly translated to the constraint $c_{trgP} \triangleq \mathsf{res/typ} = \text{“e-Pre”}$, the constraint tuple is built up from the result of function $\mathcal{T}_A$ representing the application of the algorithm $\mathsf{p\text{-}over}$. Specifically, the constraint tuples of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2765640, IEEE Transactions on Software Engineering

rules are iteratively combined according to the definition of $\mathsf{p\text{-}over}(A, B)$ previously reported. For example, the combination of the first two rules generates the following tuple

$$\begin{aligned}
\langle\ & \mathsf{permit}:\ (c_{trg1} \land \mathsf{true}) \lor (c_{trg2} \land \mathsf{true})\\
& \mathsf{deny}:\ (\mathsf{false} \land \mathsf{false}) \lor (\mathsf{false} \land \neg c_{trg2}) \lor (\neg c_{trg_1} \land \mathsf{false})\\
& \mathsf{not\text{-}app}:\ \neg c_{trg1} \land \neg c_{trg2}\\
& \mathsf{indet}: ((\neg(\mathtt{isBool}(c_{trg1}) \lor \mathtt{isMiss}(c_{trg1}))\\
& \qquad\qquad \lor(c_{trg1} \land \neg\mathsf{true})) \land \neg(c_{trg2} \land \mathsf{true}))\\
& \qquad \lor(\neg(c_{trg1} \land \mathsf{true}) \land\ (\neg(\mathtt{isBool}(c_{trg2})\\
& \qquad\qquad \lor \mathtt{isMiss}(c_{trg2})) \lor (c_{trg2} \land \neg\mathsf{true})))\ \rangle
\end{aligned}$$

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 for the effect permit, whose corresponding constraint is

$$\begin{aligned}
& c_{obl\_p} \triangleq\\
& \quad \textstyle\bigwedge_{n \in \{\mathsf{sys/time}, \mathsf{res/typ}, \mathsf{sub/id}, \mathsf{act/id}\}} \neg\mathtt{isMiss}(n) \land \neg\mathtt{isErr}(n)
\end{aligned}$$

The constraint corresponding to obligations for the effect deny, which are missing, is instead true.

Finally, the constraint tuple of Policy (P1) generated by function $\mathcal{T}_P$ is

$$\begin{aligned}
\langle & \mathsf{permit}:\ c_{trgP}\\
& \qquad \land ((c_{trg1} \land \mathsf{true}) \lor (c_{trg2} \land \mathsf{true}) \lor (c_{trg3} \land \mathsf{true}))\\
& \qquad \land\ c_{obl\_p}\\
& \mathsf{deny}:\ c_{trgP}\\
& \land ((((\mathsf{false} \land \mathsf{false}) \lor (\mathsf{false} \land \neg c_{trg2}) \lor (\neg c_{trg_1} \land \mathsf{false}))\\
& \qquad \land \mathsf{false})\\
& \quad \lor(((\mathsf{false} \land \mathsf{false}) \lor (\mathsf{false} \land \neg c_{trg2}) \lor (\neg c_{trg_1} \land \mathsf{false}))\\
& \qquad \land\neg c_{trg3})\\
& \quad \lor((\neg c_{trg1} \land \neg c_{trg2}) \land \mathsf{false}))\\
& \land \mathsf{true}\\
& \mathsf{not\text{-}app}:\ \neg c_{trgP} \lor (c_{trgP} \land (\neg c_{trg1} \land \neg c_{trg2} \land \neg c_{trg3}))\\
& \mathsf{indet}:\ \neg(\mathtt{isBool}(c_{trgP}) \lor \mathtt{isMiss}(c_{trgP}))\\
& \lor (c_{trgP} \land (((\neg(\mathtt{isBool}(c_{trg1}) \lor \mathtt{isMiss}(c_{trg1}))\\
& \qquad \lor(c_{trg1} \land \neg\mathsf{true})) \land \neg(c_{trg2} \land \mathsf{true}))\\
& \lor \neg((c_{trg1} \land \mathsf{true}) \land (\neg(\mathtt{isBool}(c_{trg2}) \lor \mathtt{isMiss}(c_{trg2}))\\
& \qquad \lor(c_{trg2} \land \neg\mathsf{true}))) \land \neg(c_{trg3} \land \mathsf{true}))\\
& \lor (\neg((c_{trg1} \land \mathsf{true}) \lor (c_{trg2} \land \mathsf{true})) \land (\neg(\mathtt{isBool}(c_{trg3})\\
& \qquad \lor \mathtt{isMiss}(c_{trg3})) \lor (c_{trg3} \land \neg\mathsf{true})))\\
& \lor (c_{trgP} \land ((c_{trg1} \land \mathsf{true}) \lor (c_{trg2} \land \mathsf{true})\\
& \qquad \lor(c_{trg3} \land \mathsf{true})) \land \neg c_{obl\_p})\\
& \lor (c_{trgP} \land ((((\mathsf{false} \land \mathsf{false}) \lor (\mathsf{false} \land \neg c_{trg2})\\
& \qquad \lor(\neg c_{trg_1} \land \mathsf{false})) \land \mathsf{false})\\
& \lor (((\mathsf{false} \land \mathsf{false}) \lor (\mathsf{false} \land \neg c_{trg2}) \lor (\neg c_{trg_1} \land \mathsf{false}))\\
& \qquad \land \neg c_{trg3})\\
& \lor ((\neg c_{trg1} \land \neg c_{trg2}) \land \mathsf{false})) \land \neg\mathsf{true})\ \rangle
\end{aligned}$$

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 evident that, due to constraints complexity, their evaluation, as well as their generation, is error-prone. Thus, constraints-based analysis should not be done manually, but with the support of automatic tools, as presented hereafter.

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. In Section 7.1, we formalise a relevant set of properties in terms of expected authorisations for requests, and present some concrete examples of them regarding the case study. Then, in Section 7.2 we show how to express the constraint formalism into a tool-accepted specification and in Section 7.3 we exploit it to automatically verify the properties with an SMT solver.

7.1 Formalisation of Properties

We consider both properties that refer to the expected authorisation of single requests, called *authorisation properties*, and to the relationships among policies on the base of the whole set of authorisations they establish, called *structural properties*; afterwards we comment on their automated verification.

7.1.1 Authorisation Properties

The analysis carried out through authorisation properties aims at investigating how the extension of a request through the addition of further attributes might change its authorisation in a possibly unexpected way. In fact, as remarked in Section 5.6, FACPL does not enjoy the *safety* property. Thus, it is important for system designers to consider the authorisation decisions not only of specific requests, but also of their extensions since, e.g., a malicious user could try to exploit them to circumvent the access control system. This approach is partially inspired by the probabilistic analysis on missing attributes introduced by Crampton et al. [23].

To formalise the authorisation properties, we introduce the notion of *request extension set* of a given request r. It is defined as

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

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

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

$$p\ \mathtt{sat}\ r\ \mathtt{eval}\ dec \qquad \mathit{iff} \qquad \mathcal{P}[\![p]\!]r = \langle dec\ \ io^*\rangle$$

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

May-Evaluate-To. This property, written $r\ \mathtt{eval_{may}}\ dec$, 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

$$\begin{aligned}
& p\ \mathtt{sat}\ r\ \mathtt{eval_{may}}\ dec \quad \mathit{iff}\\
& \qquad \exists\, r' \in Ext(r)\ :\ \mathcal{P}[\![p]\!]r' = \langle dec\ \ io^*\rangle
\end{aligned}$$

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

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

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

We consider here some examples of properties regarding the patient consent policies in Section 4.3, namely Policies (P1) and (P2). The properties we are going to introduce allow us to verify whether the policies disallow the access to a pharmacist who wants to write an e-Prescription. To this aim, we define an *Evaluate-To* property[6] as

$$\begin{array}{l}(\mathsf{sub/role}, \text{“}\mathsf{pharmacist}\text{”})(\mathsf{act/id}, \text{“}\mathsf{write}\text{”})\\ (\mathsf{res/typ}, \text{“}\mathsf{e\text{-}Pre}\text{”})\end{array} \ \texttt{eval} \ \ \mathsf{deny} \qquad (\mathrm{Pr1})$$

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

$$\begin{array}{l}(\mathsf{sub/role}, \text{“}\mathsf{pharmacist}\text{”})\\ (\mathsf{res/typ}, \text{“}\mathsf{e\text{-}Pre}\text{”})\end{array} \ \texttt{eval}_{\texttt{may}} \ \ \mathsf{not\text{-}app} \qquad (\mathrm{Pr2})$$

The verification of these properties with respect to Policy (P1) results in

$$Policy\,(P1) \texttt{ unsat } (Pr1) \qquad\qquad Policy\,(P1) \texttt{ sat } (Pr2)$$

where unsat indicates that the policy does not satisfy the property. Indeed, as already pointed out 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 (P2) results in

$$Policy\,(P2) \texttt{ sat } (Pr1) \qquad\qquad Policy\,(P2) \texttt{ unsat } (Pr2)$$

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

7.1.2 Structural Properties

This group of properties refers to the structure of the sets of authorisations established by one or multiple policies. In case of multiple policies, the properties aim at characterising the relationships among the policies. Different structural properties have been proposed in the literature by pursuing different approaches for their definition and verification [11], [14], [17]. Here, we consider a set of commonly addressed properties and provide a uniform characterisation thereof in terms of requests and policy semantics. These properties support system designers in developing and maintaining policies. For instance, they enable *change-impact analysis* [14], which examines policy modifications for discovering unintended consequences of such changes.

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

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

Essentially, we require that the policy applies to any request, that is 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 a decision considered *admissible*, i.e., permit or deny. The satisfiability of the *Disjointness* property by a policy p is defined as

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

It is worth noting that disjoint polices can be combined with the assurance that the allowed or forbidden authorisations established 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 establishes 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, the policy under examination evaluates to the same decision. The satisfiability of the *Coverage* property by a policy p is defined as

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

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

We conclude by considering some structural properties of the patient consent policies. Specifically, by verifying completeness, we can check if there is a request that evaluates to not-app. We get

$$Policy\,(P1) \texttt{ unsat complete} \quad Policy\,(P2) \texttt{ sat complete}$$

As expected, Policy (P1) does not satisfy completeness, because there is at least one request that evaluates to not-app, whereas Policy (P2) is complete. Instead, we can check if Policy (P2) correctly refines Policy (P1) by simply verifying coverage. We get

$$Policy\ (P2) \texttt{ sat cover } Policy\ (P1)$$

[6] For the sake of presentation, we write requests as sequences of attributes according to the FACPL syntax rather than using their semantical functional representation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2765640, IEEE Transactions on Software Engineering

This follows from the fact that Policy (P2) evaluates to permit the same set of requests as Policy (P1) and that Policy (P1) never returns deny; clearly, the opposite coverage property does not hold. It should be also noted that the two policies are not disjoint as they share the same set of permitted requests.

7.1.3 Towards Automated Verification

The analysis approach we propose is feasible in practice, although the involved sets of requests, such as the request extension set of a given request and the set of all possible requests, might be infinite. Indeed, Lemma 5.4 implies that only the attribute names occurring within the policies of interest are relevant for their analysis, and these are finite in number; any other name cannot affect policy evaluation. Thus, for instance, to analyse a policy p, we must not consider the set R of all possible requests, but only the set of those requests whose domain is $Names(p)$, which is the finite set of attribute names occurring in p. This property paves the way for carrying out automated property verification by means of SMT solvers as described in Section 7.3.

7.2 Expressing Constraints with SMT-LIB

In order to be practically effective, tool support is essential for property verification. To this aim, we express the constraints defined in Section 6 by means of the SMT-LIB language [34], which is a standardised constraint language accepted by most 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, such as uninterpreted function and array theories. We now 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.

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 reported in Table 11. 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 basically consists 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 getting rid of all those 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 $X_{\mathsf{cat/id}} = Double \wedge X_{\mathsf{cat/id}} = Bool$, where we have omitted conditions trivially satisfied. This constraint is clearly unsatisfiable as attribute cat/id cannot simultaneously be a double and a boolean. Hence, a policy containing such expression is not well-typed and would be statically discarded. Although policies are well-typed, errors can arise during their evaluation due to requests assigning values of unexpected type to some attributes. In our analysis, the field err is used to address the impact of these errors. It is indeed crucial to analyse also wrongly typed requests, as they might end up granting unwanted accesses to careless or even malicious users.

On top of the TValue datatype we build the uninterpreted functions expressing the operators of the proposed constraint formalism. By way of example, the 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 10, so that different TValue records are returned according to the input. The function isFalse (resp. isTrue) is used to compactly check that all fields of the record are false (resp. only the field val is true). All the other constraint operators, except $\in$, are defined similarly.

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

Table 11
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

$$\frac{v \in Bool}{\Gamma \vdash v : Bool \mid \mathsf{true}} \qquad \frac{v \in Double}{\Gamma \vdash v : Double \mid \mathsf{true}} \qquad \frac{v \in String}{\Gamma \vdash v : String \mid \mathsf{true}} \qquad \frac{v \in Date}{\Gamma \vdash v : Date \mid \mathsf{true}} \qquad \frac{v \in 2^{Value}}{\Gamma \vdash v : 2^{Value} \mid \mathsf{true}} \qquad \frac{\Gamma(n) = X}{\Gamma \vdash n : X \mid \mathsf{true}}$$

$$\frac{\Gamma \vdash expr : U \mid C}{\Gamma \vdash \mathsf{not}(expr) : Bool \mid C \wedge U = Bool} \qquad \frac{\Gamma \vdash expr_1 : U_1 \mid C_1 \qquad \Gamma \vdash expr_2 : U_2 \mid C_2}{\Gamma \vdash \mathsf{eop}(expr_1, expr_2) : Bool \mid C_1 \wedge C_2 \wedge U_1 = Bool \wedge U_2 = Bool} \ \mathsf{eop} \in \{\mathsf{and}, \mathsf{or}\}$$

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

```
(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`. For example, `(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, we have

```
(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 nonlinear arithmetic, specifically multiplication, it is possible to define constraints that a solver cannot handle. Anyway, modern constraint solvers are actually able to resolve nontrivial nonlinear problems that, for what concerns access control policies, should prevent any undefined evaluation[7]. 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 [35]. 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} introduced in Section 6.4 is

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

where identifiers starting with `n_` (resp. `cst_`) represent attribute names (resp. literals) of the represented expression. The whole SMT-LIB code for Policy (P1) can be found at [36].

7.3 Automated Properties Verification

The SMT-LIB coding permits using SMT solvers to automatically verify the properties formalised in Section 7.1. In the following, given a FACPL policy p, we denote by $\langle \mathsf{permit} : \textit{smtlib-}c_p \ \mathsf{deny} : \textit{smtlib-}c_d \ \mathsf{not\text{-}app} : \textit{smtlib-}c_n \ \mathsf{indet} : \textit{smtlib-}c_i \rangle$ the tuple of SMT-LIB codes representing the formal constraints $\mathcal{T}_P\{\!|p|\!\} = \langle \mathsf{permit} : c_p \ \mathsf{deny} : c_d \ \mathsf{not\text{-}app} : c_n \ \mathsf{indet} : c_i \rangle$. Hereafter, we present first the strategies to follow for verifying the authorisation properties, then those for verifying the structural properties.

7.3.1 Authorisation Properties

The verification of authorisation properties requires: (i) to define the SMT-LIB coding of the chosen property, given the request defined by the property and the policy constraint *smtlib-c* of interest; and (ii) to check the satisfiability (or validity) of the resulting constraint.

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

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

[7] It should be noted that if at least one argument of each occurrence of the multiply operator is a numeric constant, then the resulting nonlinear arithmetic constraints are decidable.

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

$$\overline{r_{smtlib}(p)} \triangleq \\ \{\ (\texttt{assert}\ (\texttt{miss}\ \ n))\ \mid n \in Names(p)\ \land\ r(n) = \perp\ \}$$

where, as in Section 5.6, $Names(p)$ indicates the set of attribute names occurring in p. Thus, all the names n that occur in p and are not assigned to a value in r are asserted to be missing attributes. The overbar notation hints that the assertion concerns the complement of the attributes in r.

By exploiting this SMT-LIB coding of requests, we can now use an SMT solver to automate the verification of the authorisation properties as follows

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

where $\circ$ indicates the concatenation of SMT-LIB code[8] and valid means that the corresponding SMT-LIB code is a valid set of assertions, i.e., satisfied by all the assignments for the attribute names.

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 $\perp$. This means that the request r is coded in SMT-LIB with r_{smtlib} and $\overline{r_{smtlib}(p)}$. The satisfiability of the property thus corresponds to that of the resulting SMT-LIB code.

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

Finally, to verify the *Must-Evaluate-To* property, we code again the request with r_{smtlib} only, but we check the validity of the resulting SMT-LIB code. This amounts to checking if the negation of the resulting SMT-LIB code is not satisfiable, in which case the property holds.

7.3.2 Structural Properties

The verification of structural properties does not require to modify policy constraints, but rather to check the unsatisfiability of combinations of constraints. Indeed, we have

$$\begin{array}{lll}
p\ \texttt{sat complete} & \texttt{iff} \quad smtlib\text{-}c_n & \texttt{is unsat} \\
p\ \texttt{sat disjoint}\ p' & \texttt{iff} & \\
\multicolumn{3}{l}{\quad \left\{ \begin{array}{ll}
smtlib\text{-}c_p\ \circ\ smtlib\text{-}c'_p & \texttt{is unsat} \\
smtlib\text{-}c_p\ \circ\ smtlib\text{-}c'_d & \texttt{is unsat} \\
smtlib\text{-}c_d\ \circ\ smtlib\text{-}c'_p & \texttt{is unsat} \\
smtlib\text{-}c_d\ \circ\ smtlib\text{-}c'_d & \texttt{is unsat}
\end{array} \right.} \\
p\ \texttt{sat cover}\ p' & \texttt{iff} & \\
\multicolumn{3}{l}{\quad \left\{ \begin{array}{ll}
\neg\ smtlib\text{-}c_p\ \circ\ smtlib\text{-}c'_p & \texttt{is unsat} \\
\neg\ smtlib\text{-}c_d\ \circ\ smtlib\text{-}c'_d & \texttt{is unsat}
\end{array} \right.}
\end{array}$$

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

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

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

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

8 The FACPL Toolchain

The coding, analysis and enforcement tasks pursued in the development of FACPL specifications are fully supported by the Java-based software toolchain[9] graphically depicted in Figure 3. The key element of the toolchain is an Eclipse-based IDE that provides features like, e.g., static code checks and automatic generation of runnable Java and SMT-LIB code. A dedicated 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 XACML v3.0 and the various available tools supporting it (e.g., XCREATE [38], Margrave [14] and Balana [39]), the IDE automatically translates FACPL code into XACML and vice-versa. Since the two languages slightly differ in terms of expressiveness, there are some limitations in FACPL and XACML interoperability as detailed in Section 9.1.

Furthermore, to allow newcomers to directly experiment with FACPL, the web application "Try FACPL in your Browser" reachable from the FACPL website offers an online editor and an evaluation engine for FACPL policies, with the e-Health case study as a running example. Additionally, a web interface [36] shows a proof-of-concept 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 in Sections 8.1 and 8.2, respectively, and present the performance evaluation of our tools in Section 8.3.

8.1 The FACPL Library

Our Java library permits evaluating FACPL policies, hence 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, like expressions and combining algorithms, with respect to its formal specification.

[8] Notably, checking the satisfiability of the SMT-LIB code resulting from the concatenation of two sets of SMT-LIB assertions amounts to checking if both the assertions hold at the same time.

[9] The FACPL supporting tools are freely available and open-source; binary files, source files, unit tests, case study implementations and documentation can be found at the FACPL website [37].

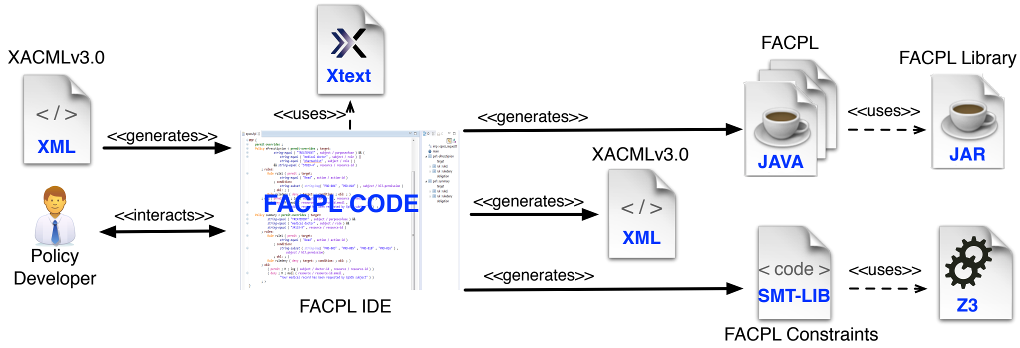


Figure 3. The FACPL toolchain

For each element of the language, the library contains an abstract class that provides its evaluation method. In practice, a FACPL policy, as well as a PDP, is translated into a Java class that extends 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 invoking the evaluation method of a policy, which coordinates the evaluation of its enclosed elements in compliance with its formal specification. In addition to the authorisation process, the library supports the enforcement process by defining the three enforcement algorithms and a minimal set of pre-defined PEP actions, i.e., `log`, `mailTo` and `compress`. Additional actions can be dynamically introduced by providing their implementation classes to the PEP initialisation method.

By way of example, we report in the listing in Figure 4 an excerpt of the Java code of Policy (P1) introduced in Section 4.3. Besides the specific methods used for adding policy elements, the Java code highlights the use of class references for selecting expression operators and combining algorithms. This design choice, together with the use of best-practices of object-oriented programming, allows the library to be easily extended with, e.g., new expression operators, combining algorithms and enforcement actions. Further details are given in Section 10.1. Notice that rules are private inner classes, because they cannot be referred outside the enclosing policy sets.

Besides the four-valued decisions considered so far, the FACPL library also supports the extended indeterminate values used by XACML v3.0, i.e., indetP, indetD and indetDP. They can be used to specify the potential decision (permit, deny and both, respectively) that should have been returned by the evaluation of a policy if an error would not have occurred. Extended indeterminate values allow the PDP to obtain additional information about policy evaluation, which can be exploited, e.g., during policy debugging for improving the treatment of errors. However, their usage may require additional workload. In fact, it establishes that if the target of a policy set evaluates to error, rather than stopping and returning indet, the evaluation process continues the computation by processing the enclosed policies and using the decision resulting from the application of the combining algorithm to calculate an extended indeterminate value. Thus, e.g., if the combining algorithm returns the decision permit, the evaluation of the policy returns indetP. For all these reasons, we have chosen to support the extended indeterminate values by means of a boolean parameter of the method `doAuthorisation` whose setting can enable or disable their use at each PDP invocation.

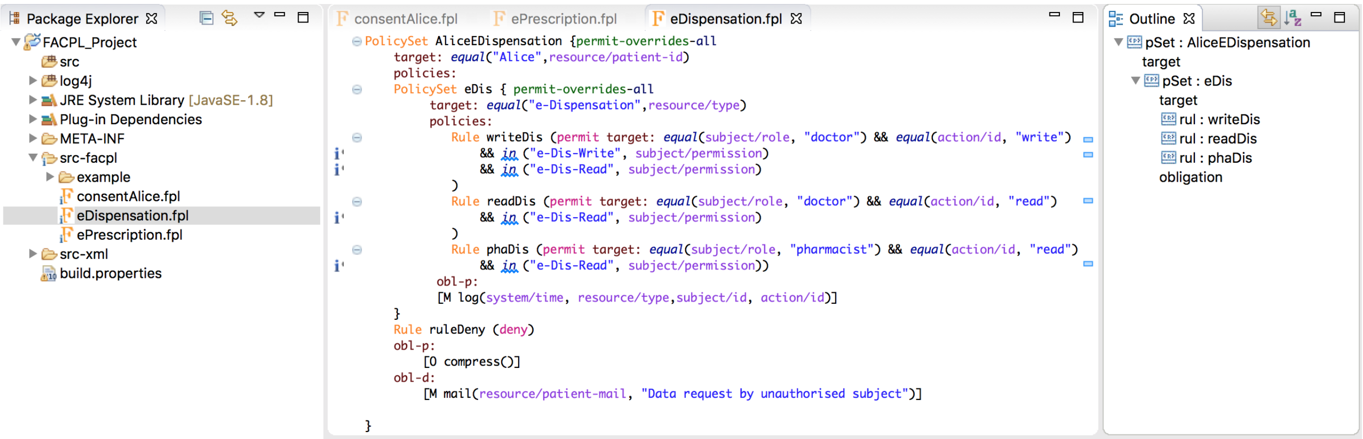
```
public class PolicySet_e-Prescription extends PolicySet{
  public PolicySet_e-Prescription(){
    addCombiningAlg(PermitOverrides.class);
    addTarget(new ExpressionFunction(Equal.class, "e-
         Prescription",
      new AttributeName("resource","type")));
    addRule(new rule1());
    addRule(new 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 (){...} }}
```

Figure 4. Java code of Policy (P1)

8.2 The FACPL IDE

The FACPL IDE is an Eclipse plug-in implemented by means of Xtext [40]. It aims at bringing together the available FACPL functionalities and tools. Indeed, it fully supports writing, evaluating and analysing FACPL specifications. A screenshot of the IDE is shown in Figure 5.

The IDE 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. In order to ease the organisation of large policy specifications, the plug-in supports modularisation of files



```
PolicySet AliceEDispensation {permit-overrides-all
    target: equal("Alice",resource/patient-id)
    policies:
    PolicySet eDis { permit-overrides-all
        target: equal("e-Dispensation",resource/type)
        policies:
            Rule writeDis (permit target: equal(subject/role, "doctor") && equal(action/id, "write")
                && in ("e-Dis-Write", subject/permission)
                && in ("e-Dis-Read", subject/permission)
            )
            Rule readDis (permit target: equal(subject/role, "doctor") && equal(action/id, "read")
                && in ("e-Dis-Read", subject/permission)
            )
            Rule phaDis (permit target: equal(subject/role, "pharmacist") && equal(action/id, "read")
                && in ("e-Dis-Read", subject/permission))
         obl-p:
         [M log(system/time, resource/type,subject/id, action/id)]
    }
    Rule ruleDeny (deny)
    obl-p:
        [O compress()]
    obl-d:
        [M mail(resource/patient-mail, "Data request by unauthorised subject")]

}
```

Figure 5. The FACPL IDE

and import commands extending file scopes. Also, to ensure interoperability from XACML to FACPL, it relies on a higher-order function `map` that, given a function and a set of elements, applies the function to each element of the set.

The development environment provided by the plug-in is standard. It offers graphical features, like keywords highlighting, code suggestion and navigation within and among files, static controls on FACPL code, like uniqueness of identifiers and type checking, and automatic generation of Java, XACML, and SMT-LIB code.

To facilitate the analysis of FACPL policies, the plug-in 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 7.3; an execution script for the Z3 solver is also generated. Of course, the SMT-LIB files can be also evaluated by any other solver accepting SMT-LIB code 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, that 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 that of the SMT-LIB function, is left to the developer.

FACPL-type projects are created through a dedicated wizard. By construction, they include all the libraries needed for coding and evaluation tasks. A project consists of text files, with extension *.fpl*, containing FACPL code. Similarly to an Eclipse Java project, our IDE supports writing of these files by means of a dedicated text editor, an outline view and contextual menus. Functionalities supporting code development, like code suggestion and auto-completion, are available via the usual Eclipse shortcuts and menus. In particular, from either the toolbar menu or the right-click editor menu, the developer can find a set of pre-defined commands to generate Java, XACML and SMT-LIB code, or to open a step-by-step wizard for the definition of authorisation and structural properties. Besides coding from scratch, FACPL policies could be produced by exploiting the automatic translation from XACML policies. Additional usage information are available in the FACPL tools' guide [37].

To conclude we want here to mention two other freely available IDEs for access control: the ALFA Eclipse plugin by Axiomatics [41] and the graphical editor of the Balana-based framework [42]. Differently from our IDE, their functionalities are mainly limited to the editing of XACML policies. In particular, ALFA does not allow request evaluation, since the Axiomatics engine is a proprietary software.

8.3 Performance evaluation

The effectiveness of supporting tools is a crucial point for the usability of a policy language. Therefore, hereafter we assess the performance of both the FACPL Java library and the SMT-based automated analysis, also in comparison with closely related tools.

Concerning the library, we conducted two different tests[10]: (i) a performance comparison with a state-of-the-art XACML tool on the CONTINUE case study [24], partially analysed by Fisler et al. [14]; and (ii) a performance stress test on a large set of randomly generated policies, thus to analyse the library scalability. We present below the most relevant test results; the suites of policies and requests, as well as all the test results, are available on-line [43].

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 implementation is Balana [39]. Differently from our framework that represents FACPL policies as Java classes, Balana manages XACML policies directly in XML by exploiting a DOM representation of the XML files and evaluating XACML requests through a visit of the DOM representing the policy. We have compared the evaluation of more than 1.500 requests and obtained the results reported in Figure 6; for the sake of readability only the results concerning 700 requests are reported. The mean request evaluation time is 0.49ms for FACPL and 1.27ms for Balana: this implies that evaluating a Java class ensures higher performance than navigating the DOM. Additionally, Balana requires an initial set-up time of 770ms to create the DOM.

[10] Both tests were conducted on a MacBook Pro, 3.1 GHz Intel i7, 16 Gb RAM, running OS X Sierra.

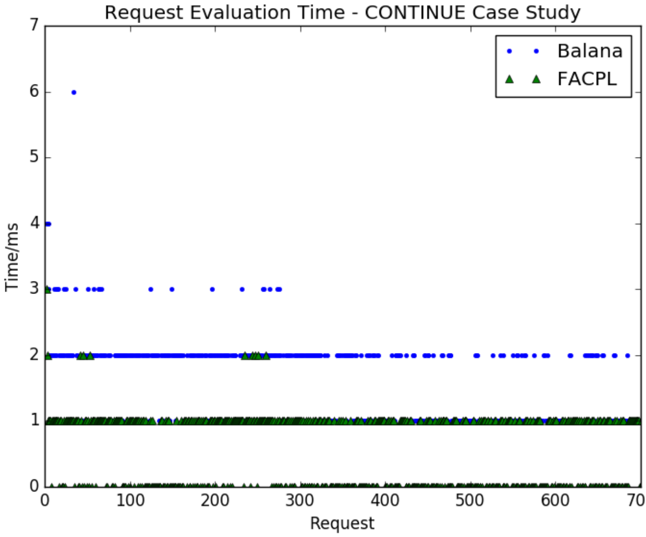


Figure 6. FACPL vs. Balana performance evaluation

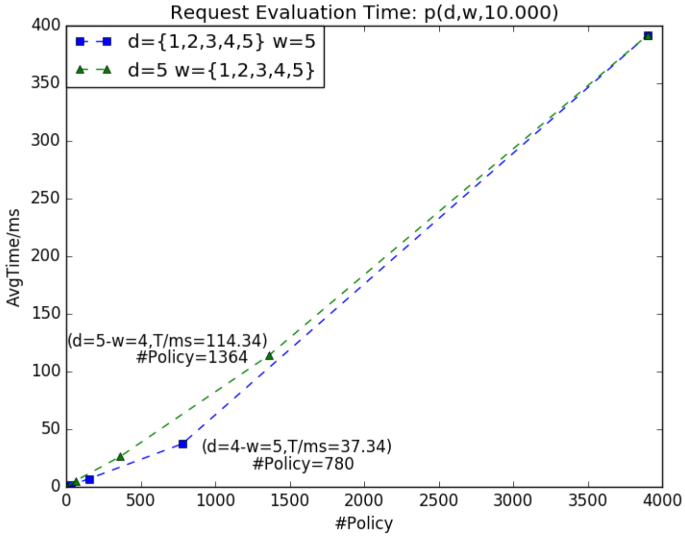


Figure 7. FACPL performance stress test (when $a = 10.000$)

The CONTINUE case study is by now used as a standard benchmark in the field of access control tools. However it is relatively small: it is made of 24 policies controlling 14 attributes. All policies are combined through the first-app algorithm thus, as soon as a policy applies, the evaluation stops. Therefore, for evaluating performance and scalability of the FACPL Java library, we have also considered a set of large randomly-generated policies. We generated the policies according to the following criteria: (i) a variable number of occurring *attribute names*, that is 10, 100, 1.000 or 10.000; (ii) a policy *depth* ranging from 1 to 5; (iii) a policy *width* ranging from 1 to 5; (iv) only the all instantiation strategy is used so to always require the evaluation of all the occurring policies. The combinations of the previous criteria give rise to a test-bed of 100 policies, formed by a distinct number of differently structured sub-policies featuring a different number of attribute names. More specifically, given a number of attribute names a, depth d and width w, the policy $p(d, w, a)$ is generated according to the template in Table 12a. Namely, d corresponds to the nesting levels in the policy hierarchy, while w corresponds to the number of policies that each policy (set) in the hierarchy contains. The total number of sub-policies contained by every policy $p(d, w, a)$ is summarised in Table 12b. For each of the 25 generated policies, there are 4 different versions, one for each value that a can take.

Table 12
(a) Structure of each of policy $p(d, w, a)$; (b) Total number of sub-policies for each combination of d and w

(a)

	$p(d, w, a)$
$\#w^1$	$p_1^1 \quad \cdots \quad p_w^1$
$\#w^2$	$p_1^2 \cdots p_w^2 \quad \cdots \quad p_{w+1}^2 \cdots p_{w^2}^2$
$\vdots$	$\vdots \quad \vdots \quad \vdots$
$\#w^d$	$p_1^d \cdots p_w^d \quad \cdots p_{w^d-w+1}^d \cdots p_{w^d}^d$

(b)

$_d\backslash^w$	1	2	3	4	5
1	1	2	3	4	5
2	2	6	12	20	30
3	3	14	39	84	155
4	4	30	120	340	780
5	5	62	363	1364	3905

The generated test-bed has been used to perform the stress test on the FACPL library. The results, when a is set to 10.000, are summarised in Figure 7. The graphs show how the performance changes as a function of the policy structure, thus depending on d and w. Better performances are obtained by structuring policies in terms of larger width values (marked by the blue square), rather than larger depth values (marked by the green triangle). Namely, the average evaluation time increases more by increasing policy depth than width.

Concerning the automated analysis, the tool closer to ours [11] relies on the SMT solver Yices [33]. Differently from Z3, Yices does not support datatype theory, which is crucial for dealing with a wide range of policy aspects, like missing and erroneous attributes. To analyse the completeness of the CONTINUE policies, the Yices-based tool requires around 570ms, while our Z3-based tool requires around 120ms[11]. To further evaluate the analysis performance, we also report in Figure 8 the time required to verify the satisfiability of the not-app constraint, that is the verification of the `complete` property (marked by the blue square), and the permit constraint (marked by the green triangle) of the 3905 policies of the form $p(5, 5, a)$ obtained by varying the number a of attributes. Namely, the `complete` property is always verified in less than one second, despite the increasing number of attributes. Instead, the verification time for the satisfiability of the permit constraint increases by rising the number of attribute names, but the increments are significantly lower than the attribute name variations, that is from $199s$ with 1.000 attribute names, to $394s$ with 10.000 ones. The difference between the two cases is due to the policy semantics: while a policy evaluates to not-app as soon as its target is not-app, to obtain a permit decision it is necessary to apply the combining algorithm to the enclosed policies. It is finally worth noting that the considered policies have a limited number of attribute names representing sets of values. In

[11] The Yices value is taken directly from the work by Arkoudas et al. [11], since the provided CONTINUE implementation only runs on Windows machines. Therefore, we ran this Z3 analysis on an older comparable hardware configuration (with the current configuration it takes only 60ms).

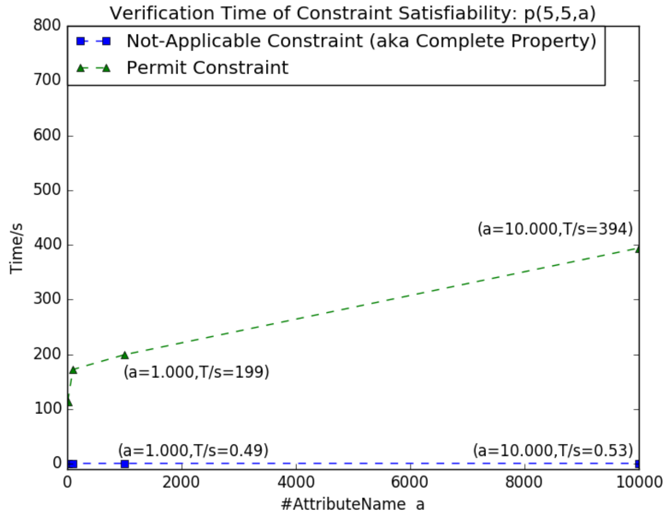


Figure 8. FACPL analysis performance (when $d = w = 5$)

fact, according to the definition of the `inInt` function in Section 7.2, a higher number of multivalued attributes would require many satisfiability checks of existential quantifiers along with the 3905 policies. In this case, the analysis may require hours instead of minutes. For example, it lasts two hours for the case of 100 attribute names and 20 multivalued attributes, while it cannot complete for the case of 1.000 attribute names and 200 multivalued attributes.

To sum up, the extensive performance evaluation of the FACPL Java library has pointed out that (i) the request evaluation time on the CONTINUE case study is definitely better than the DOM-based approach of Balana; (ii) the performance are affected by the structure of the policy since the more the policy depth, the longer the evaluation time; (iii) the library can handle a substantial number of policies built upon up to 10.000 attributes. The performance of the analysis tool has instead pointed out that (i) the performance on CONTINUE are definitely better than the approach by Arkoudas et al. [11]; (ii) the SMT-based approach provides good performance also when policies scale. The latter point is critical for other tools not based on SMT. In fact, as reported by Arkoudas et al. [11], the increment of possible values for the attributes occurring in the CONTINUE policies prevents Margrave to accomplish the analysis. On the contrary, the results of tests conducted on the test-bed, and shown in Figure 8, witness that our tool can deal with infinite sets of attribute values, e.g. integers, on significantly large policies.

9 Related Work

A preliminary version [12] of FACPL aimed at 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. For instance, a practical example of a policy-based manager for a Cloud platform is given by Margheri et al. [44]. Additional important differences concern the definition of the policy semantics. Masi et al. [12] define it in terms of partitions of the set of all possible requests. Here, instead, policy semantics 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 instantiation 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 an enhanced language, but we provide a complete methodology that encompasses all phases of policy life-cycle, i.e., specification, analysis and enforcement. Concerning the analysis, we characterise in terms of sets of requests a number of relevant authorisation and structural properties, preliminarily introduced by Margheri et al. [13]. We then present a constraint-based representation of policies and an SMT-based approach for verifying properties on top of constraints. To effectively support these functionalities, we provide a fully-integrated software toolchain.

Table 13
FACPL vs. XACML on the e-Health case study

Policy	Number of lines		Saved	Number of types		Saved
	XACML	FACPL	lines	XACML	FACPL	types
e-Prescription	239	24	89,95%	10.656	894	91,61%
e-Dispensation	239	24	89,95%	10.674	914	91,43%
Patient Consent	423	38	91,01%	19.195	1.558	91,88%

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

9.1 FACPL vs XACML

XACML [7] 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 the RFC 2753 [25], hence similar to the FACPL one. As a matter of notation, hereafter the words emphasised in sans-serif, like 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 that 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 13 exemplifies a lexical comparison between the FACPL policies for the e-Health case study and the corresponding XACML ones[12].

[12] All the considered XACML policies, together with the corresponding FACPL ones, are available on-line [36].

Although FACPL and XACML policies have a similar structure, there are quite a number of semantic differences, as we outline below. These differences are however overcome by the translators presented in Section 8, which ensure partial interoperability between the two languages.

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; for example, an AttributeSelector with category $subject$ and an XPath expression like $type/id/text()$ correspond to subject/type.id. Notably, differently from FACPL, AttributeDesignator and AttributeSelector always return sets of values, so-called *bag*s. These bags are dealt with as indivisible values in the evaluation of all policy elements except for Targets, where they are dealt with as explained below.

A XACML Target consists 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. The evaluation of a Match element is defined by separately applying the enclosed comparison function to each value of the bag resulting from the evaluation of the enclosed AttributeDesignator or AttributeSelector. In particular, the evaluation returns true (resp., false), if at least one (resp., all) application of the function to a (resp., all) value of the bag returns true (resp., false); otherwise indet is returned. The XACML-to-FACPL translator preserves this behaviour in FACPL by exploiting the auxiliary function map mentioned in Section 8.

Another semantic difference in the target evaluation is due to the management of errors and missing attributes. Indeed, while evaluating the target, when a value is missing, the XACML semantics of Match elements returns error or false according to the setting of the boolean parameter MustBePresent, whereas the FACPL semantics of target elements uses the special value $\bot$ to manage the case of missing attributes in a distinguished way. The same occurs for evaluation errors. In this way, the FACPL semantics provides a more fine-grained management of missing attributes and evaluation errors than the XACML's one. The issue is dealt with by the XACML-to-FACPL translator through a function which converts $\bot$ to false.

For the reasons above, there are some limitations when translating FACPL policies into XACML ones. Indeed, the case of missing attributes cannot be rendered in XACML. Similarly, also multivalued attributes cannot be dealt with due to the semantic differences in XACML between Target and Condition elements.

Except for target evaluation, the semantics of XACML and FACPL policies mainly comply with each other. However, the specification approach fostered by FACPL is more generic and poses less constraints on the policy structure. In particular, XACML prescribes a policy structure based on Policys, which are collections of Rules, and PolicySets, which are collections of Policys and/or PolicySets. Most of all, XACML forces specific constraints on targets of Policy and Policy Set: they can only contain comparison functions and each comparison can only contain one attribute name. Moreover, XACML supports fewer combining algorithms than FACPL and provides only the greedy instantiation strategy. Additionally, as previously pointed out, XACML specialises the decision indet into three sub-decisions: for the sake of presentation, we have not considered them in the formal development of FACPL, but they are fully supported by the FACPL library (see Section 8).

FACPL and XACML share the same management of obligations, although in FACPL this process is specified in a more precise manner, through the instantiation strategies and the binary combining operators. It is also worth noting that the XACML 'obligation fulfilment' is termed 'obligation instantiation' in FACPL, since indeed the evaluation of obligations by the PDP does not carry out any task beyond its mere instantiation.

Also, 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.

To sum up, except for minor differences on tangled XACML aspects mainly concerning the management of missing attributes and evaluation errors, FACPL subsumes XACML policies not containing XML raw data. At the same time, FACPL offers a higher flexibility in the policy specification approach and a richer set of combining algorithms. Most of all, FACPL provides a formal semantics that supports formally-based analysis and drives its implementation.

From a more practical perspective, FACPL tools can support existing XACML-based systems in different ways. For example, given a PEP generating XACML requests, one can use our XACML-to-FACPL translator to transform the requests in FACPL format. Since we are translating requests, this further step does not significantly affect the performance of the evaluation process. The resulting requests can be then redirected for evaluation to a FACPL PDP by simply invoking the corresponding Java method. To facilitate this integration, besides the XACML-to-FACPL translator available within the IDE, we also provide a standalone version of the translator. As another example, given a XACML PDP, our XACML-to-FACPL translator can be used to generate, with the aforementioned limitations, the corresponding FACPL policies. This transformation enables both policy analysis and generation of Java code to possibly replace the XACML PDP.

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 14 summarises the comparison.

Among the many proposed policy languages, we can identify two main specification approaches: *rule-based*, such as the XACML standard and Ponder [45], [46], and *logic-*

Table 14
Comparison of some relevant policy languages (✓* means that user encoding is required)

Features	XACML	Ponder	ASL	PTaCL	[8]	[11]	FACPL
Rule-based	✓	✓					✓
Logic-based			✓	✓	✓	✓	
Mnemonic spec.		✓					✓
Comb. algorithms	✓		✓*	✓*	✓	✓*	✓
Obligations	✓	✓					✓
Missing attributes	✓			✓			✓
Error handling	✓						✓

based, such as ASL [19], PTaCL [9] and the logical frameworks by Arkoudas et al. [11]. In the literature, several authors, such as Li et al. [21], Rao et al. [8], and Ramli et al. [47], 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 by Sloman [48] 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 [19], [49], [50] are based on Datalog [51], which implies that the access rules are defined as first order logic predicates. In general, these proposals offer valuable means for a low-level design of rules, but the lack of high-level features, like combining algorithms or obligations, prevent them from representing policies like those of FACPL.

ASL, one of the firstly defined logic-based languages, expresses authorisation policies based on user identity credentials and authorisation privileges, and supports hierarchy and propagation of access rights among roles and groups of users. Additional predicates enable the definition of a posteriori integrity checks on authorisation decisions, like conflict resolution strategies. Differently from ASL, FACPL provides high-level constructs and offers by-construction many not straightforward features like, e.g., conflict resolution strategies. The use of a suitable policy hierarchy enables access right propagation in FACPL policies as well.

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 formalising the *non-monotonicity* and *safety* properties of attribute-based policies introduced by Tschantz and Krishnamurthi [22]. The PTaCL extension by Crampton and Williams [20] introduces obligations and their instantiation, but it still lacks error handling.

A similar study, but more focussed on the distinguishing features of XACML, is done by Ramli et al. [47]. 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 due to Arkoudas et al. [11]. 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, like 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* by Rao et al. [8] 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. [21] explicitly introduce 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 instantiation strategies, which require different iterative applications of operators. Moreover, in the work by Li et al. [21] 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.

Finally, we want to point out that the term *obligation* is used with different meaning in the literature. Some authors, among which Hilty et al. [52] and Bettini et al. [53], use the term to refer to actions that must be performed in the future, after the access is granted. In this sense, obligations are concerned with commitments of the involved parties that cannot be guaranteed at the moment of granting access and have thus to be checked afterwards by means of suitable monitoring mechanisms. This approach is also followed by Pretschner et al. [54] that, for better controlling the usage of accessed data, refine the notion of obligation enforceability in order to distinguish between controllable, observable and non-observable obligations. Differently from these authors, we are not concerned with what happens after access has been granted since FACPL obligations, like XACML ones, refer to actions to be performed before an access is authorised.

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. Hereafter we review the more relevant ones.

The works concerning policy analysis that are closer to our approach are of course those exploiting SMT formulae. Turkmen et al. [16] introduce a strategy for representing XACML policies in terms of SMT formulae. The representation, which is based on an informal semantics of XACML, supports integers, booleans and doubles, while the representation of sets of values and strings is only sketched. The combining algorithms are modelled as conjunctions and disjunctions of formulae representing the policies to be combined, in a form similar to the approach shown in Appendix A. 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. 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 by Arkoudas et al. [11], mentioned in Section 9.2, suffers from similar drawbacks.

The only analysis approach that takes missing attributes into account is due to Crampton et al. [23]. The analysis is based on a notion of request extension, as we have done in Section 7. This analysis aims at quantifying the impact of possibly missing attributes on policy evaluations. Specifically, the PTaCL language is used to model policy target expressions, and PRISM [55] is then used to experimentally study the probability of evaluation errors due to missing attributes. Differently from our approach, this analysis does not support the specification and evaluation of properties on calculated decisions.

The change-impact analysis of XACML policies by Fisler et al. [14] 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 by Arkoudas et al. [11], an SMT-based approach like ours scales significantly better than the MTBDD one.

Datalog-based languages, like 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 a high abstraction level, but they neglect many of the advanced features of modern access control systems.

Description Logic (DL) is used by Kolovski et al. [17] 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 by Ahn et al. [56] and Ramli et al. [10] 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, Hughes and Bultan [57] exploit the SAT-based tool Alloy [58] to detect inconsistencies in XACML policies. However, as outlined by Arkoudas et al. [11] and Fisler et al. [14], 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 noting that various analysis approaches using SAT-based tools have been developed for the Ponder language [59]. These approaches, however, cannot actually be compared with ours due to the numerous 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 consider only a limited part of access control policy aspects and suffer from scalability issues when compared to SMT-based tools.

In summary, all the approaches to the analysis of access control policies mentioned above are deficient in several respects when compared with ours. Those based on SMT formulae do not address relevant aspects, like missing attributes, while the other ones do not enjoy the benefits of using SMT, in particular support of multiple theories and scalable performance.

10 Concluding Remarks and Future Work

We have described a full-fledged framework for the specification, analysis and enforcement of access control policies. Our framework relies on FACPL and is built on top of solid formal foundations. The FACPL semantics provides a formalisation of complex access control features —including obligations and missing attributes, which are overlooked by many other proposals— and lays the basis for developing analysis techniques and tools. We have shown that FACPL policies can be represented in terms of specific SMT

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2765640, IEEE Transactions on Software Engineering

formulae, whose automatic evaluation permits verifying various authorisation and structural properties. We have demonstrated feasibility and effectiveness of our approach by means of a case study from the e-Health application domain, which is currently one of the most critical application domains of access control systems. 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.

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. Most of all, 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.

We conclude by first enlighting some distinguishing traits of FACPL in Section 10.1, then by pointing out some future research directions in Section 10.2.

10.1 Discussion

We want here to recap and reflect on a few characteristics of FACPL, and its framework, and the design choices that underlie them.

Expressiveness. The access control systems expressible by FACPL are those expressible by XACML, but not dealing with XML raw data, with in addition the possibility of using consensus-based combining algorithms and the all instantiation strategy for obligations. FACPL access control systems are systematically more compact and feature a smoother management of errors and missing attributes. This latter characteristic, together with the fact that we decided not to introduce any static check on the type of requests, permits to accurately deal with every access control request. Alternatively, we could have defined a type inference system in order to statically check the policies and infer the expected type of any attribute name occurring within. Then, we could have reserved evaluation for only those requests whose attribute names comply with their expected type, while we could have directly returned the indet decision for all the other requests. By pursuing such an approach, however, we would have lost expressiveness, since policies could not be able anymore to automatically manage errors due to unexpected attribute values and possibly mask them by using operators that combine, according to different strategies, indet decisions with the others.

Besides the definition of access controls, FACPL permits defining obligations, which are a key ingredient to enhance the expensiveness of access control systems. As exemplified in the definition of the e-Health case study in Section 4.3, the instantiation of obligations permits defining context-dependent actions to be enforced at run-time in the controlled system. Indeed, the side-effects of policy evaluation are not only the enforcement of access decisions, but also the enforcement of dynamically instantiated actions. FACPL obligations permit enforcing, e.g., resource usage, adaptation and emergency handling strategies. For example, in the application of the Ponder language by Sloman and Lupu [60] and in the preliminary version of FACPL by Margheri et al. [44], [61], obligations are used to enforce self-adaptation strategies in autonomic computing systems. Instead, in the context of emergency handling, an obligation-based approach is proposed by Brucker and Petritsch [62] and Marinovic et al. [63] to enforce the principle known as 'break the glass', which means that authorisation controls can be bypassed in case of emergency.

We intentionally abstract from the actual syntax of obligations. They are simply intended to be actions executed at run-time. From time to time, they can be chosen to more adequately express the access control system at hand. We also abstract from the actual obligation semantics: the discharging of obligations done by the PEP simply refers to the fact that the system has taken charge of their execution, which has to complete by the conclusion of the PEP enforcement process. However, the possibility of enforcing some obligations after releasing the decision and granting the access is a topic worth to be studied. This is indeed one of the future research directions we want to pursue.

Validation. Our framework has essentially three constituent elements: (i) the linguistic constructs together with their denotational semantics; (ii) the constraint formalism and the semantic-preserving translation; (iii) the Java-based supporting tools. For each of them we have presented different validation results, both theoretical and empirical.

The linguistic constructs are validated with respect to their expressiveness. This is done, on the one hand, by modelling a real-world case study from the e-Health application domain, on the other hand, by comparing FACPL with XACML, which is the state-of-the-art OASIS standard for attribute-based access control systems. FACPL formal semantics is validated according to the so-called *reasonability properties* by Tschantz and Krishnamurthi [22] that precisely characterise the expressiveness of a policy language. Besides these properties, we show that the semantics is well-defined (Theorem 5.1) and precisely characterise the attributes that are relevant for policy evaluation (Lemma 5.4); this important result, as pointed out in Section 7.1.3, underlies the automatic property verification. All the results are presented in Section 5.6, while their proofs are relegated to Appendix B.1.

Similarly, the constraint formalism and the semantic-preserving translation of FACPL policies into SMT formulae are validated by the theoretical results presented in Section 6.3 and proved in Appendix B.2. All together these results ensure that the approach to the analysis of FACPL policies presented in Section 7 is sound.

The software tools are validated by empirically examining their performance and functionalities. The obtained results are reported in Section 8.3.

Exploitation. The FACPL framework is a production-level software that is also used in industry. Indeed, since its preliminary version, FACPL has been used by Tiani Spirit [64] instead of XACML to carry out design and automated analysis of access control policies. In particular, the FACPL access control engine has been used as XACML reference implementation in several projects. Furthermore, FACPL was used for team works in a PhD school on engineering Collective Autonomic Systems [65] and has been used in many bachelor and master thesis (further details can be found at the FACPL web-site). These practical exploitations have highlighted that its compact, mnemonic syntax requires very short learning time, even to undergraduate students. The users have also appreciated the flexibility of the IDE, which can be smoothly integrated within other development environments.

Extendability. The proposed framework offers a wide range of constructs, ranging from expression operators to combining algorithms, for defining access controls. Anyway, to better suit any need, as reported in Section 8, both the Java library and the IDE can be easily extended with the introduction of, e.g., new expression operators. This approach supports writing customised FACPL specifications. These specifications can then be translated, in accordance with the user's definition of the added constructs, to Java and SMT-LIB code that can be evaluated and analysed, respectively. The formal assurance of semantic preservation (Theorem 6.2) can be easily tailored for encompassing the user's extensions. For instance, in case of addition of a new expression operator, it only requires devising a constraint operator, or a combination thereof, that faithfully represents the semantics of the new operator.

10.2 Future Work

We plan to address the issue of controlling the access while it is in progress. In this sort of 'continuos' access control, the challenge is to ensure guarantees on how granted accesses are used. This model is usually referred to as Usage Control [66] in the literature and has been recently studied by various researchers. To deal with usage control, temporal aspects are of paramount importance, both to refer to ongoing accesses and to enforce obligations after releasing access decisions. To this aim, along the lines of the work by Carniani et al. [67], we will provide a FACPL-based solution for usage control that, by relying on the already available context-dependent authorisation process, can control ongoing accesses and instantiate temporal obligations. To actually enforce these obligations and, consequently, reason on them, we will refine the PEP semantics by appropriately instantiating the predicate $\Downarrow \mathsf{ok}$ introduced in Section 5.4.

We also plan to provide a formally-based analysis technique that system developers can exploit to verify, e.g., history-dependent properties like dynamic separation of duty. To this aim, besides formalising new history-dependent authorisation properties, we want to define and verify properties on conflicts and dependencies among obligations.

Finally, we intend to investigate the use of natural language processing techniques and tools [68] to support elicitation of policies from security requirements expressed in natural, or at least controlled, language. This would permit enriching the FACPL framework with facilities for deriving FACPL policies from high level specifications of access control rights, such as those reported in Table 2.

References

[1] B. W. Lampson, "Protection," *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[2] D. F. Ferraiolo and D. R. Kuhn, "Role-based access control," in *NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.

[3] NIST, "A survey of access control models," 2009, http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.

[4] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-based access control," *IEEE Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[5] X. Jin, R. Krishnan, and R. S. Sandhu, "A unified attribute-based access control model covering DAC, MAC and RBAC," in *DBSec*. Springer, 2012, pp. 41–55.

[6] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, 2012.

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

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

[9] J. Crampton and C. Morisset, "Ptacl: A language for attribute-based access control in open systems," in *POST*, ser. LNCS, P. Degano and J. D. Guttman, Eds., vol. 7215. Springer, 2012, pp. 390–409.

[10] C. D. P. K. Ramli, H. Riis Nielson, and F. Nielson, "Xacml 3.0 in answer set programming," in *LOPSTR*, ser. LNCS, vol. 7844. Springer, 2012, pp. 89–105.

[11] K. Arkoudas, R. Chadha, and C. J. Chiang, "Sophisticated access control via SMT and logical frameworks," *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 4, p. 17, 2014.

[12] M. Masi, R. Pugliese, and F. Tiezzi, "Formalisation and Implementation of the XACML Access Control Mechanism," in *ESSoS*, ser. LNCS 7159. Springer, 2012, pp. 60–74.

[13] A. Margheri, R. Pugliese, and F. Tiezzi, "On Properties of Policy-Based Specifications," in *WWV*, ser. EPTCS, vol. 188, 2015, pp. 33–50.

[14] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE*. ACM, 2005, pp. 196–205.

[15] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[16] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone, "Analysis of XACML policies with SMT," in *POST*, ser. LNCS, vol. 9036. Springer, 2015, pp. 115–134.

[17] V. Kolovski, J. A. Hendler, and B. Parsia, "Analyzing web access control policies," in *WWW*. ACM, 2007, pp. 677–686.

[18] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[19] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A logical language for expressing authorizations," in *Symposium On Security And Privacy*. IEEE, 1997, pp. 31–42.

[20] J. Crampton and C. Williams, "Obligations in ptacl," in *STM*, ser. LNCS, S. Foresti, Ed., vol. 9331. Springer, 2015, pp. 220–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24858-5_14

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

[22] M. C. Tschantz and S. Krishnamurthi, "Towards reasonability properties for access-control policy languages," in *SACMAT*. ACM, 2006, pp. 160–169.

[23] J. Crampton, C. Morisset, and N. Zannone, "On missing attributes in access control: Non-deterministic and probabilistic attribute retrieval," in *SACMAT*. ACM, 2015, pp. 99–109.

[24] S. Krishnamurthi, "The CONTINUE server (or, how I administered PADL 2002 and 2003)," in *PADL*, ser. LNCS, vol. 2562. Springer, 2003, pp. 2–16.

[25] R. Yavatkar, D. Pendarakis, and R. Guerin, "A Framework for Policy-based Admission Control," RFC 2753 (Proposed Standard), Internet Engineering Task Force, 2000. [Online]. Available: https://tools.ietf.org/html/rfc2753

[26] The epSOS project, "An European eHealth Project," http://www.epsos.eu.

[27] M. Kovac, "E-health demystified: An e-government showcase," *IEEE Computer*, vol. 47, no. 10, pp. 34–42, 2014. [Online]. Available: http://dx.doi.org/10.1109/MC.2014.282

[28] European Parliament and Council, "Directive 95/46/EC," 1995, official Journal L 281 , 23/11/1995 P. 0031 - 0050.

http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML.

[29] The Article 29 Data Protection WP, 2013, http://ec.europa.eu/justice/data-protection/article-29/.

[30] Health Level Seven organization, "Hl7 standards," 2017, http://www.hl7.org.

[31] "Electronic health record in austria (elga)," 2017, http://www.elga.gv.at/.

[32] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proc. of CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 171–177.

[33] B. Dutertre, "Yices 2.2," in *Proc. of CAV*, ser. LNCS, vol. 8559. Springer, 2014, pp. 737–744.

[34] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," 2010, http://smtlib.cs.uiowa.edu/.

[35] Microsoft Research, "Z3: a guide," 2017, http://rise4fun.com/z3/tutorial/guide.

[36] FACPL e-Health Case Study, 2017, http://facpl.sf.net/eHealth/.

[37] FACPL Website, 2017, http://facpl.sourceforge.net.

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

[39] WSO2, "Balana: Open source XACML implementation," 2017, https://github.com/wso2/balana.

[40] Xtext, *Language Development Made Easy*, http://www.eclipse.org/Xtext/.

[41] Axiomatics, "Axiomatics Language for Authorization (ALFA)," https://www.axiomatics.com/news/axiomatics-releases-free-plugin-for-the-eclipse-ide-to-author-xacml3-0-policies/.

[42] WSO2, "Balana UI for XACML," 2017, http://xacmlinfo.org/category/xacml-editor/.

[43] FACPL Tools Performance Evaluation, 2017, http://facpl.sf.net/test.

[44] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi, "Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation. A Practical Approach," in *WSFM*, ser. LNCS, vol. 8379. Springer, 2013, pp. 85–105.

[45] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *POLICY*, ser. LNCS 1995. Springer, 2001, pp. 18–38.

[46] K. P. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *ICAS*. IEEE, 2009, pp. 330–335.

[47] C. D. P. K. Ramli, H. Riis Nielson, and F. Nielson, "The logic of XACML," *Sci. Comput. Program.*, vol. 83, pp. 80–105, 2014.

[48] M. Sloman, "Policy driven management for distributed systems," *J. Network Syst. Manage.*, vol. 2, no. 4, pp. 333–360, 1994.

[49] M. Hashimoto, M. Kim, H. Tsuji, and H. Tanaka, "Policy description language for dynamic access control models," in *DASC*. IEEE, 2009, pp. 37–42.

[50] J. DeTreville, "Binder, a logic-based security language," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, ser. SP '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 105–113.

[51] S. Ceri, G. Gottlob, and T. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, 1989.

[52] M. Hilty, D. A. Basin, and A. Pretschner, "On obligations," in *ESORICS*, ser. LNCS, vol. 3679. Springer, 2005, pp. 98–117.

[53] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera, "Provisions and obligations in policy rule management," *J. Netw. Syst. Manage.*, vol. 11, no. 3, pp. 351–372, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1025711105609

[54] A. Pretschner, M. Hilty, and D. Basin, "Distributed usage control," *Commun. ACM*, vol. 49, no. 9, pp. 39–44, Sep. 2006.

[55] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM: probabilistic model checking for performance and reliability analysis," *SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[56] G. J. Ahn, H. Hu, J. Lee, and Y. Meng, "Representing and reasoning about web access control policies," in *COMPSAC*. IEEE Computer Society, 2010, pp. 137–146.

[57] G. Hughes and T. Bultan, "Automated verification of access control policies using a sat solver," *STTT*, vol. 10, no. 6, pp. 503–520, 2008.

[58] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[59] A. K. Bandara, E. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *POLICY*. IEEE, 2003, p. 26.

[60] M. Sloman and E. C. Lupu, "Engineering policy-based ubiquitous systems," *Comput. J.*, vol. 53, no. 7, pp. 1113–1127, 2010.

[61] A. Margheri, R. Pugliese, and F. Tiezzi, "Linguistic abstractions for programming and policing autonomic computing systems," in *UIC/ATC*. IEEE, 2013, pp. 404–409.

[62] A. D. Brucker and H. Petritsch, "Extending access control models with break-glass," in *SACMAT*. ACM, 2009, pp. 197–206.

[63] S. Marinovic, N. Dulay, and M. Sloman, "Rumpole: An introspective break-glass access control language," *ACM TISSEC.*, vol. 17, no. 1, pp. 2:1–2:32, 2014.

[64] "Tiani Spirit - Software Solutions for the best Healthcare," 2017, http://www.tiani-spirit.com/.

[65] FP7 EU project, "ASCENS," 2015, Spring School - http://www.ascens-ist.eu/springschool.html.

[66] A. Lazouski, F. Martinelli, and P. Mori, "Usage control in computer security: A survey," *Computer Science Review*, vol. 4, no. 2, pp. 81–99, 2010.

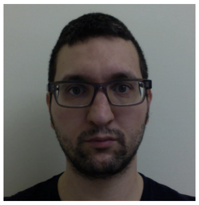
[67] E. Carniani, D. D'Arenzo, A. Lazouski, F. Martinelli, and P. Mori, "Usage control on cloud systems," *Future Generation Comp. Syst.*, vol. 63, pp. 37–55, 2016.

[68] Stanford Natural Language Processing Group, 2017, https://nlp.stanford.edu/.

Andrea Margheri is a Research Fellow in Cyber Security at the University of Southampton. He received his MSc degree in Computer Science from University of Florence and his Ph.D. in Computer Science from University of Pisa. His research covers both theoretical and practical aspects of modern computing systems, with particular emphasis on access control, cloud computing and blockchain-based technologies.

Massimiliano Masi is an IT Security Architect at Tiani "Spirit" GmbH, with more than 13 years of experience. Relevant experiences include specification of IT security measures of the eGP-EGOR and BeS Projects in Austria, and South African eHR.ZA (governmental eHealth initiatives), lead role of epSOS.eu CCD and leader of the Core team of the epSOS Security Experts Group, design security aspects of the LHC computing GRID. Masi has been a member of the OASIS Trust Elevation Committee, and is also the editor of the IHE ITI profile XCF, and participated in the evaluation of the IHE profiles related to security (e.g., XUA, XUA++, SeR, Access Control White Paper, and ATNA). He is actually leading various task forces in the e-SENS project.

Rosario Pugliese is Associate Professor at the Department of Statistic, Computer Science, Applications, University of Florence. He received the Laurea degree from University of Pisa and the Ph.D. degree from University of Rome "La Sapienza". His research interests are focused on today's concurrent, possibly mobile, distributed systems, such as service-oriented and autonomic computing systems. The aim is to devise formal models, languages and tools enabling the study of issues concerning interaction, connectivity, adaptivity and security.



Francesco Tiezzi is Associate Professor at the Computer Science Division of the University of Camerino. He received the Laurea degree from University of Florence and the Ph.D. degree from the same university. His research activities focus on foundational study of distributed systems, and on application of formal methods for developing and analysing them. Particular attention is paid to the definition of formal bases for technologies supporting service-oriented, cloud and autonomic computing.

Appendix A
Definitions for Combining Algorithms

In this section we report all the definitions regarding the combining algorithms. Table 15 shows all the combination matrices defining the binary operators $\otimes$alg for each algorithm alg. Hereafter we report the constraint resulting from the combination of two constraint tuples, say A and B, defined according to the various combining algorithms.

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

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

$$
\begin{array}{l}
\text{d-unless-p}(A,B) = \\
\quad \langle\, \text{permit} : A\downarrow_p \vee B\downarrow_p \\
\quad\;\; \text{deny} : \neg A\downarrow_p \wedge \neg B\downarrow_p \wedge (A\downarrow_d \vee A\downarrow_n \vee A\downarrow_i) \\
\qquad\qquad\qquad \wedge (B\downarrow_d \vee B\downarrow_n \vee B\downarrow_i) \\
\quad\;\; \text{not-app} : \text{false} \\
\quad\;\; \text{indet} : \text{false}\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{p-unless-d}(A,B) = \\
\quad \langle\, \text{permit} : \neg A\downarrow_d \wedge \neg B\downarrow_d \wedge (A\downarrow_p \vee A\downarrow_n \vee A\downarrow_i) \\
\qquad\qquad\qquad \wedge (B\downarrow_p \vee B\downarrow_n \vee B\downarrow_i) \\
\quad\;\; \text{deny} : A\downarrow_d \vee B\downarrow_d \\
\quad\;\; \text{not-app} : \text{false} \\
\quad\;\; \text{indet} : \text{false}\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{first-app}(A,B) = \\
\quad \langle\, \text{permit} : A\downarrow_p \vee (B\downarrow_p \wedge A\downarrow_n) \\
\quad\;\; \text{deny} : A\downarrow_d \vee (B\downarrow_d \wedge A\downarrow_n) \\
\quad\;\; \text{not-app} : A\downarrow_n \wedge B\downarrow_n \\
\quad\;\; \text{indet} : A\downarrow_i \vee (A\downarrow_n \wedge B\downarrow_i)\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{one-app}(A,B) = \\
\quad \langle\, \text{permit} : (A\downarrow_p \wedge B\downarrow_n) \vee\; (A\downarrow_n \wedge B\downarrow_p) \\
\quad\;\; \text{deny} : (A\downarrow_d \wedge B\downarrow_n) \vee\; (A\downarrow_n \wedge B\downarrow_d) \\
\quad\;\; \text{not-app} : A\downarrow_n \wedge B\downarrow_n \\
\quad\;\; \text{indet} : A\downarrow_i \vee B\downarrow_i \vee ((A\downarrow_p \vee A\downarrow_d) \wedge\; (B\downarrow_p \vee B\downarrow_d))\rangle
\end{array}
$$

$$
\begin{array}{l}
\text{weak-con}(A,B) \\
\quad \langle\, \text{permit} : (A\downarrow_p \wedge B\downarrow_p) \vee\; (A\downarrow_p \wedge \neg B\downarrow_d) \vee\; (\neg A\downarrow_d \wedge B\downarrow_p) \\
\quad\;\; \text{deny} : (A\downarrow_d \wedge B\downarrow_d) \vee\; (A\downarrow_d \wedge \neg B\downarrow_p) \vee\; (\neg A\downarrow_p \wedge B\downarrow_d) \\
\quad\;\; \text{not-app} : A\downarrow_n \wedge B\downarrow_n \\
\quad\;\; \text{indet} : (A\downarrow_p \wedge B\downarrow_d) \vee\; (A\downarrow_d \wedge B\downarrow_p) \vee\; A\downarrow_i \vee B\downarrow_i
\end{array}
$$

$$
\begin{array}{l}
\text{strong-con}(A,B) = \\
\quad \langle\, \text{permit} : A\downarrow_p \wedge B\downarrow_p \\
\quad\;\; \text{deny} : A\downarrow_d \wedge B\downarrow_d \\
\quad\;\; \text{not-app} : A\downarrow_n \wedge B\downarrow_n \\
\quad\;\; \text{indet} : A\downarrow_i \vee B\downarrow_i \vee (A\downarrow_n \wedge \neg B\downarrow_n) \vee\; (\neg A\downarrow_n \wedge B\downarrow_n) \\
\qquad\qquad \vee\, (A\downarrow_p \wedge B\downarrow_d) \vee\; (A\downarrow_d \wedge B\downarrow_p)\rangle
\end{array}
$$

Appendix B
Proofs of the Results

Some of the proofs proceed by induction on the *depth* of policies, which is the number of their nesting levels. It is defined by induction on the syntax of policies as follows

$$
\begin{array}{l}
depth((e\;\; \text{target} : expr\;\; \text{obl} : o^*\,)) = 0 \\
depth(\{a\;\; \text{target} : expr\;\; \text{policies} : p^+\; \text{obl-p} : o_p^*\;\; \text{obl-d} : o_d^*\,\}) = \\
\qquad\qquad\qquad\qquad 1 + max(\{depth(p) \mid p \in p^+\})
\end{array}
$$

$\otimes$p-over	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$
$\langle$deny $io_1^*\rangle$	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_1^*\rangle$	indet
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
indet	$\langle$permit $io_2^*\rangle$	indet	indet	indet

$\otimes$d-over	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$permit $io_1^*\rangle$	indet
$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
indet	indet	$\langle$deny $io_2^*\rangle$	indet	indet

$\otimes$d-unless-p	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$
$\langle$deny $io_1^*\rangle$	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$deny $\epsilon\rangle$	$\langle$deny $\epsilon\rangle$
indet	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$deny $\epsilon\rangle$	$\langle$deny $\epsilon\rangle$

$\otimes$p-unless-d	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$
$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$permit $\epsilon\rangle$	$\langle$permit $\epsilon\rangle$
indet	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	$\langle$permit $\epsilon\rangle$	$\langle$permit $\epsilon\rangle$

$\otimes$first-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\rangle$
$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$	$\langle$deny $io_1^*\rangle$
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$one-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	indet	indet	$\langle$permit $io_1^*\rangle$	indet
$\langle$deny $io_1^*\rangle$	indet	indet	$\langle$deny $io_1^*\rangle$	indet
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$weak-con	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	indet	$\langle$permit $io_1^*\rangle$	indet
$\langle$deny $io_1^*\rangle$	indet	$\langle$deny $io_1^*\bullet io_2^*\rangle$	$\langle$deny $io_1^*\rangle$	indet
not-app	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
indet	indet	indet	indet	indet

$\otimes$strong-con	$\langle$permit $io_2^*\rangle$	$\langle$deny $io_2^*\rangle$	not-app	indet
$\langle$permit $io_1^*\rangle$	$\langle$permit $io_1^*\bullet io_2^*\rangle$	indet	indet	indet
$\langle$deny $io_1^*\rangle$	indet	$\langle$deny $io_1^*\bullet io_2^*\rangle$	indet	indet
not-app	indet	indet	not-app	indet
indet	indet	indet	indet	indet

Table 15
Combination matrices for the binary operators $\otimes$alg

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.

B.1 Proofs of Results in Section 5

THEOREM 5.1 (Total and Deterministic Semantics).

1) For all $pas \in PAS$ and $req \in Request$, there exists a $dec \in Decision$, such that $\mathcal{PAS}\llbracket pas, req\rrbracket = dec$.
2) For all $pas \in PAS$, $req \in Request$ and $dec, dec' \in Decision$, it holds that
$$\mathcal{PAS}\llbracket pas, req\rrbracket = dec \ \wedge\ \mathcal{PAS}\llbracket pas, req\rrbracket = dec' \\ \Rightarrow \ \ dec = dec' .$$

Proof. The goal of the proof is to show that $\mathcal{PAS}$ is a total and deterministic function, i.e., it is defined for all possible input pairs and always returns the same decision any time it is applied to a specific pair. If we let pas be $\{\,\mathsf{pep} : ea\ \ \mathsf{pdp} : pdp\,\}$ then, from the clause (S-8), we have that
$$\mathcal{PAS}\llbracket\{\,\mathsf{pep} : ea\ \ \mathsf{pdp} : pdp\,\}, req\rrbracket = \\ \mathcal{EA}\llbracket ea\rrbracket(\mathcal{PDP}\llbracket pdp\rrbracket(\mathcal{R}\llbracket req\rrbracket))$$

Thus, since function composition preserves totality and determinism, we are left to prove that $\mathcal{R}$, $\mathcal{PDP}$ and $\mathcal{E}A$ are total and deterministic functions. Due to their inductive definition (given in Section 5), the proof proceeds by inspecting their defining clauses with the 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 in 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 and deterministic functions/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., is all requests. Moreover, the conditions in 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 and deterministic function.

Case $\mathcal{PDP}$. 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 5, 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 and deterministic function also when extended to sequences of expressions.

Case $\mathcal{O}$. Since $\mathcal{E}$ is a total and deterministic function also on sequences of expressions, from the clauses (S-3a) and (S-3b) it follows that R1 and R2 hold, thus $\mathcal{O}$ is a total and deterministic 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 and deterministic functions. 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 $(p^+)^0$, $\mathcal{P}\llbracket p_j^0\rrbracket r$ is uniquely defined. Thus, since each operator $\otimes\mathsf{alg}$ is total and deterministic by construction, the clauses (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 clauses (S-6b). It satisfies R2 (when all the input policies have depth 0) since its conditions in the right hand side are mutually exclusive by construction (indeed, 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 $\{a\ \mathsf{target}\!:\!expr\ \mathsf{policies}\!:\!(p^+)^n\ \mathsf{obl\text{-}p} : o_p^*\ \mathsf{obl\text{-}d} : o_d^*\}$. 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 $\mathcal{P}\llbracket p_j^k\rrbracket r$ and $\mathcal{A}\llbracket a, (p^+)^n\rrbracket r$ 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 policy).

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

Now, that $\mathcal{PDP}$ is a total and deterministic 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 and deterministic function. □

THEOREM 5.2 (Instantiation Strategy Correspondence). For all $p \in \mathit{Policy}$ and $r \in R$, it holds that

$$\mathcal{P}[\![p]\!]r = \langle \mathit{dec}\ \mathit{io}_1^* \rangle \;\;\Leftrightarrow\;\; \mathcal{P}[\![\texttt{toAll}(p)]\!]r = \langle \mathit{dec}\ \mathit{io}_2^* \rangle\ .$$

Proof. Equivalently, by induction on the depth i of p, we prove that

$$(\mathcal{P}[\![p]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p)]\!]r).\mathit{dec}\,.$$

Base Case ($i = 0$). p^0 has the form $(e\ \mathsf{target} : \mathit{expr}\ \mathsf{obl} : o^*)$, thus, by definition, $p = \texttt{toAll}(p)$ and the thesis for this case obviously follows.

Inductive Case ($i = n+1$). p^{n+1} is of the form $\{a\ \ \mathsf{target} : \mathit{expr}\ \ \mathsf{policies} : (p^+)^n\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\}$ and we have two cases to consider.

a **uses** all **as instantiation strategy.** This means that $a = \mathsf{alg}_{\mathsf{all}}$ for some $\mathsf{alg}_{\mathsf{all}} \in \mathit{Alg}$. Hence, we have that $\texttt{toAll}(p) = \{a\ \ \mathsf{target} : \mathit{expr}\ \mathsf{policies} : \texttt{toAll}((p^+)^n)\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\}$. Now, by the induction hypothesis, for all p_j^k in $(p^+)^n$, with $k \leq n$, it holds that $(\mathcal{P}[\![p_j^k]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p_j^k)]\!]r).\mathit{dec}$. Thus, from the clauses (S-6a) and from the combination matrices for the binary operators shown in Table 15, by easy induction on the length of the sequence $(p^+)^n$, it follows that $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^n]\!]r).\mathit{dec} = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, \texttt{toAll}((p^+)^n)]\!]r).\mathit{dec}$.

a **uses** greedy **as instantiation strategy.** This means that $a = \mathsf{alg}_{\mathsf{greedy}}$ for some $\mathsf{alg}_{\mathsf{greedy}} \in \mathit{Alg}$. Hence, we have that $\texttt{toAll}(p) = \{\mathsf{alg}_{\mathsf{all}}\ \ \mathsf{target} : \mathit{expr}\ \mathsf{policies} : \texttt{toAll}((p^+)^n)\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\}$. Thus, by the induction hypothesis, for all p_j^k in $(p^+)^n$ with $k \leq n$, it holds that $(\mathcal{P}[\![p_j^k]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p_j^k)]\!]r).\mathit{dec}$. Now, by reasoning on induction on the structure of p^+, we prove that

$$\begin{aligned}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, (p^+)^n]\!]r).\mathit{dec}& \\ = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, \texttt{toAll}((p^+)^n)]\!]r).\mathit{dec}&\end{aligned} \tag{1}$$

from which the thesis then follows because of the clause (S-4b) defining the semantics of policies.

Base Case. This means that $(p^+)^n$ is a single policy, say p_1^k, with $k \leq n$. From the clauses (S-6b), we have that $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, p_1^k]\!]r).\mathit{dec} = (\otimes\mathsf{alg}(\mathcal{P}[\![p_1^k]\!]r)).\mathit{dec}$. Now, since by induction on n we can assume that $(\mathcal{P}[\![p_1^k]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p_1^k)]\!]r).\mathit{dec}$, from the clauses (S-6a), we have that $(\otimes\mathsf{alg}(\mathcal{P}[\![\texttt{toAll}(p_1^k)]\!]r)).\mathit{dec} = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, \texttt{toAll}(p_1^k)]\!]r).\mathit{dec}$, which proves (1).

Inductive Case. Let $(p^+)^n$ be $(p_1^+)^{k_1} p_2^{k_2}$, with $k_1, k_2 \leq n$. From the clauses (S-6b), we have two cases to consider.

$\mathit{isFinal}_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r) = \mathsf{true}$**.**
The proof proceeds by case analysis on alg and derives directly from the definitions in Appendix A. We only report the case of the p-over algorithm, as the other ones are similar. In the considered case, due to the definition of predicate $\mathit{isFinal}_{\mathsf{p\text{-}over}}$ in Table 6, the hypothesis $\mathit{isFinal}_{\mathsf{p\text{-}over}}(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r) = \mathsf{true}$ implies that $(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r).\mathit{dec} = \mathsf{permit}$. Therefore, by the structural induction hypothesis, we have that $(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, \texttt{toAll}((p_1^+)^{k_1})]\!]r).\mathit{dec} = \mathsf{permit}$. Now, since by induction on n we can assume that $(\mathcal{P}[\![p_2^{k_2}]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p_2^{k_2})]\!]r).\mathit{dec}$, from the clauses (S-6a) and from the combination matrices for the binary operators shown in Table 15, we get that

$$\begin{aligned}&(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, \texttt{toAll}((p^+)^n)]\!]r).\mathit{dec} = \\ &(\otimes\mathsf{p\text{-}over}(\mathcal{A}[\![\mathsf{p\text{-}over}_{\mathsf{all}}, \texttt{toAll}((p_1^+)^{k_1})]\!]r, \\ &\qquad \mathcal{P}[\![\texttt{toAll}(p_2^{k_2})]\!]r)).\mathit{dec} = \\ &\mathsf{permit}\end{aligned}$$

which proves (1).

$\mathit{isFinal}_{\mathsf{alg}}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r) = \mathsf{false}$**.**
By induction on n, we have that $(\mathcal{P}[\![p_2^{k_2}]\!]r).\mathit{dec} = (\mathcal{P}[\![\texttt{toAll}(p_2^{k_2})]\!]r).\mathit{dec}$. Moreover, by structural induction hypothesis, we have that $(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r).\mathit{dec} = (\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, \texttt{toAll}((p_1^+)^{k_1})]\!]r).\mathit{dec}$. Therefore, from the combination matrices for the binary operators shown in Table 15, we get that

$$\begin{aligned}&(\otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{greedy}}, (p_1^+)^{k_1}]\!]r, \mathcal{P}[\![p_2^{k_2}]\!]r)).\mathit{dec} = \\ &(\otimes\mathsf{alg}(\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, \texttt{toAll}((p_1^+)^{k_1})]\!]r, \\ &\qquad \mathcal{P}[\![\texttt{toAll}(p_2^{k_2})]\!]r)).\mathit{dec}\end{aligned}$$

This, by the clauses (S-6a) and (S-6b), proves (1). □

LEMMA 5.4 (Policy relevant attributes). For all $p \in \mathit{Policy}$ and $r, r' \in R$ such that $r(n) = r'(n)$ for all $n \in \mathit{Names}(p)$ it holds that $\mathcal{P}[\![p]\!]r = \mathcal{P}[\![p]\!]r'$.

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

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

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 $n \in \mathit{Names}(\mathit{expr})$ 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 $\mathit{Names}(\mathit{expr}) \subseteq \mathit{Names}(p)$, from (R), by taking $r_1 = r$ and $r_1' = r'$, it follows that

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

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 $(e\ \mathsf{target} : \mathit{expr}\ \mathsf{obl} : o^*)$, 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 $\{a\ \ \mathsf{target} : \mathit{expr}\ \ \mathsf{policies} : (p^+)^n\ \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\}$, 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). □

B.2 Proofs of results in Section 6

THEOREM 6.1 (Total and Deterministic Constraint Semantics).

1) For all $c \in \mathit{Constr}$ and $r \in R$, there exists an $el \in (\mathit{Value} \cup 2^{\mathit{Value}} \cup \{\mathsf{error}, \bot\})$, such that $\mathcal{C}[\![c]\!]r = el$.
2) For all $c \in \mathit{Constr}$, $r \in R$ and $el, el' \in (\mathit{Value} \cup 2^{\mathit{Value}} \cup \{\mathsf{error}, \bot\})$, it holds that

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

Proof. Similarly to the proof of Theorem 5.1, the proof reduces to showing that $\mathcal{C}$ is a total and deterministic function. We proceed 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 and deterministic function.

Inductive Case. It is not hard to believe that all the defining clauses of $\mathcal{C}$ are such that the conditions in 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 B.1. *For all $expr \in \mathit{Expr}$ 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 10 and that of the operator not in Table 5.

$(expr = \mathsf{op}(expr_1, expr_2))$. 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 10 and that of the constraint operator getOp(op) in Table 5. □

Lemma B.2. *For all $o \in \mathit{Obligation}$ and $r \in R$ it holds that*

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

and

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

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

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

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

$(\mathcal{O}[\![o]\!]r = io)$. 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

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

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 $\mathcal{O}[\![o]\!]r = io$ 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 B.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

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

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

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

Thus, by Lemma B.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 B.3. *For all $\mathsf{alg}_{\mathsf{all}} \in \mathit{Alg}$, $r \in R$ and sequences of policies p^+, such that for all p_i in p^+ it holds $\mathcal{P}[\![p_i]\!]r = \langle dec_i\ io_i^*\rangle \Leftrightarrow \mathcal{C}[\![\mathcal{T}_P\{\!|p_i|\!\}\downarrow_{dec_i}]\!]r = \mathsf{true}$, we have*

$$\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, p^+]\!]r = \langle dec\ io^*\rangle \Leftrightarrow \mathcal{C}[\![\mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, p^+|\!\}\downarrow_{dec}]\!]r = \mathsf{true} .$$

Proof. The proof proceeds by case analysis on alg and derives directly from the definitions in Appendix A. In what follows, we only report the case of the p-over algorithm, as the other ones are similar. We now proceed by reasoning on structural induction on p^+.

Base Case. This means that p^+ is a single policy, say p_1. By definition, we have $\otimes\mathsf{p\text{-}over}(\mathcal{P}\llbracket p_1 \rrbracket r) = \mathcal{P}\llbracket p_1 \rrbracket r$ and $\mathsf{p\text{-}over}(\mathcal{T}_P\{\!|p_1|\!\}) = \mathcal{T}_P\{\!|p_1|\!\}$. The thesis then directly follows from the hypothesis on p_1, that is $\mathcal{P}\llbracket p_1 \rrbracket r = \langle dec_1 \;\, io_1^* \rangle \;\Leftrightarrow\; \mathcal{C}\llbracket \mathcal{T}_P\{\!|p_1|\!\} \downarrow_{dec_1} \rrbracket r = \mathsf{true}$, due to the clauses (S-6a) and (T-4).

Inductive Case. Let p^+ be $(p')^+p''$. By the structural induction hypothesis, we can assume that the thesis holds for the sequence $(p')^+$. That is, if we let $c = \mathcal{T}_A\{\!|\mathsf{p\text{-}over_{all}}, (p')^+|\!\}$, we have

$$\mathcal{A}\llbracket \mathsf{p\text{-}over_{all}}, (p')^+ \rrbracket r = \langle dec' \; (io')^* \rangle \\ \Leftrightarrow \\ \mathcal{C}\llbracket c \downarrow_{dec'} \rrbracket r = \mathsf{true}\,.$$

Moreover, from the hypothesis, we have that

$$\mathcal{P}\llbracket p'' \rrbracket r = \langle dec'' \; (io'')^* \rangle \;\Leftrightarrow\; \mathcal{C}\llbracket \mathcal{T}_P\{\!|p''|\!\} \downarrow_{dec''} \rrbracket r = \mathsf{true}\,.$$

Hence, we are left to prove that

$$\otimes\mathsf{p\text{-}over}(\langle dec' \; (io')^* \rangle, \langle dec'' \; (io'')^* \rangle) = \langle dec \; io^* \rangle \\ \Leftrightarrow \\ \mathcal{C}\llbracket \mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_{dec} \rrbracket r = \mathsf{true}\,.$$

For the sake of simplicity, in the following we omit the sequences of instantiated obligations, as their combination does not affect the decision dec returned by $\otimes\mathsf{p\text{-}over}$. Now, by a case analysis on dec, we determine the form assumed by the constraint $\mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_{dec}$. In all cases, we rely on the combination matrix for the operator $\otimes\mathsf{p\text{-}over}$ defined in Table 15 and on the definition, reported in Appendix A, of the constraint resulting from the combination of two constraint tuples according to the algorithm $\mathsf{p\text{-}over}$.

$(dec = \mathsf{permit})$. It follows that $dec' = \mathsf{permit}$ or $dec'' = \mathsf{permit}$. Moreover, we have $\mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_p = c \downarrow_p \,\vee\, \mathcal{T}_P\{\!|p''|\!\} \downarrow_p$.

$(dec = \mathsf{deny})$. It follows that $dec', dec'' \in \{\mathsf{deny}, \mathsf{not\text{-}app}\}$ but $dec' = dec'' = \mathsf{not\text{-}app}$ does not hold. Moreover, we have $\mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_d = (c \downarrow_d \wedge \mathcal{T}_P\{\!|p''|\!\} \downarrow_d) \vee (c \downarrow_d \wedge \mathcal{T}_P\{\!|p''|\!\} \downarrow_n) \vee (c \downarrow_n \wedge \mathcal{T}_P\{\!|p''|\!\} \downarrow_d)$.

$(dec = \mathsf{not\text{-}app})$. It follows that $dec' = dec'' = \mathsf{not\text{-}app}$. Moreover, we have $\mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_n = c \downarrow_n \wedge \mathcal{T}_P\{\!|p''|\!\} \downarrow_n$.

$(dec = \mathsf{indet})$. It follows that $dec' = \mathsf{indet}$ or $dec'' = \mathsf{indet}$ and $dec', dec'' \neq \mathsf{permit}$. Moreover, we have $\mathsf{p\text{-}over}(c, \mathcal{T}_P\{\!|p''|\!\}) \downarrow_i = (c \downarrow_i \wedge \neg\mathcal{T}_P\{\!|p''|\!\} \downarrow_p) \vee (\neg c \downarrow_p \wedge \mathcal{T}_P\{\!|p''|\!\} \downarrow_i)$.

In any case, the thesis follows from the definition of $\mathcal{C}$ by the structural induction hypothesis and the hypothesis on p''. □

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

$$\mathcal{P}\llbracket p \rrbracket r = \langle dec \; io^* \rangle \;\Leftrightarrow\; \mathcal{C}\llbracket \mathcal{T}_P\{\!|p|\!\} \downarrow_{dec} \rrbracket r = \mathsf{true}$$

Proof. Due to Theorem 5.2 and Lemma 6.4, without loss of generality we can assume that the policy p only encloses algorithms using the all instantiation strategy and prove the statement in this case. Now, 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

$$\mathcal{E}\llbracket expr \rrbracket r = \mathsf{true} \;\wedge\; \mathcal{O}\llbracket o^* \rrbracket r = io^*$$

Thus, by Lemma B.1, it follows that

$$\mathcal{C}\llbracket \mathcal{T}_E\{\!|expr|\!\} \rrbracket r = \mathsf{true}$$

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

$$\mathcal{C}\llbracket \mathcal{T}_{Ob}\{\!|o^*|\!\} \rrbracket r = \mathsf{true}$$

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

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

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

$$\mathcal{C}\llbracket \mathcal{T}_P\{\!|(\mathsf{permit} \;\; \mathsf{target}: expr \;\; \mathsf{obl}: o^* \,)|\!\} \downarrow_p \rrbracket r = \\ \mathcal{C}\llbracket \mathcal{T}_E\{\!|expr|\!\} \rrbracket r \wedge \mathcal{C}\llbracket \mathcal{T}_{Ob}\{\!|o^*|\!\} \rrbracket r = \\ \mathsf{true} \wedge \mathsf{true} {=} \mathsf{true}$$

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-4a), it follows that

$$\mathcal{E}\llbracket expr \rrbracket r = \mathsf{false} \;\vee\; \mathcal{E}\llbracket expr \rrbracket r = \perp$$

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 B.1 and the definition of $\mathcal{C}$.

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

$$\neg(\mathcal{E}\llbracket expr \rrbracket r = \mathsf{true} \;\wedge\; \mathcal{O}\llbracket o^* \rrbracket r = io^*) \\ \wedge \neg(\mathcal{E}\llbracket expr \rrbracket r = \mathsf{false} \;\vee\; \mathcal{E}\llbracket expr \rrbracket r = \perp)$$

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

$$\begin{aligned}
&\neg(\mathcal{E}\llbracket expr \rrbracket r = \mathsf{true} \;\wedge\; \mathcal{O}\llbracket o^* \rrbracket r = io^*) \\
&\quad \wedge \neg(\mathcal{E}\llbracket expr \rrbracket r = \mathsf{false} \;\vee\; \mathcal{E}\llbracket expr \rrbracket r = \perp) \\
&= (\mathcal{E}\llbracket expr \rrbracket r \neq \mathsf{true} \vee \mathcal{O}\llbracket o^* \rrbracket r = \mathsf{error}) \\
&\quad \wedge (\mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{false}, \perp\}) \\
&= \mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{true}, \mathsf{false}, \perp\} \vee (\mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{false}, \perp\} \\
&\quad \wedge \mathcal{O}\llbracket o^* \rrbracket r = \mathsf{error}) \\
&= \mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{true}, \mathsf{false}, \perp\} \vee \\
&\quad (\mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{true}, \mathsf{false}, \perp\} \;\wedge\; \mathcal{O}\llbracket o^* \rrbracket r = \mathsf{error}) \vee \\
&\quad (\mathcal{E}\llbracket expr \rrbracket r = \mathsf{true} \wedge \mathcal{O}\llbracket o^* \rrbracket r = \mathsf{error}) \\
&= \mathcal{E}\llbracket expr \rrbracket r \notin \{\mathsf{true}, \mathsf{false}, \perp\} \\
&\quad \vee(\mathcal{E}\llbracket expr \rrbracket r = \mathsf{true} \wedge \mathcal{O}\llbracket o^* \rrbracket r = \mathsf{error})
\end{aligned}$$

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

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

The thesis then follows by Lemmas B.1 and B.2 and the definition of $\mathcal{C}$.

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

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

$$\begin{aligned}&\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle \mathsf{permit}\ \ io_1^*\rangle\\ &\wedge\ \mathcal{O}\llbracket o_p^*\rrbracket r = io_2^*\end{aligned}$$

Thus, by Lemma B.1, it follows that

$$\mathcal{E}\llbracket \mathit{expr}\rrbracket r =\ \mathcal{C}\llbracket \mathcal{T}_E\{\!|\mathit{expr}|\!\}\rrbracket r = \mathsf{true}$$

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

$$\mathcal{C}\llbracket \mathcal{T}_{Ob}\{\!|o_p^*|\!\}\rrbracket r = \mathsf{true}$$

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

$$\mathcal{P}\llbracket p_i^h\rrbracket r = \langle \mathit{dec}_i\ \ io^*\rangle\ \ \Leftrightarrow\ \ \mathcal{C}\llbracket \mathcal{T}_P\{\!|p_i^h|\!\}\downarrow_{\mathit{dec}_i}\rrbracket r = \mathsf{true}$$

then, by Lemma B.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

$$\begin{aligned}&\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \mathsf{target} : \mathit{expr}\ \mathsf{policies} : (p^+)^k\\ &\qquad\qquad\qquad \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}|\!\}\downarrow_p\\ &= \mathcal{T}_E\{\!|\mathit{expr}|\!\}\wedge\ \mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_p\wedge\,\mathcal{T}_{Ob}\{\!|o_p^*|\!\}\end{aligned}$$

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

$$\begin{aligned}&\mathcal{C}\llbracket \mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \mathsf{target} : \mathit{expr}\ \mathsf{policies} : (p^+)^k\\ &\qquad\qquad\qquad \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}|\!\}\downarrow_p\rrbracket r\\ &= \mathcal{C}\llbracket \mathcal{T}_E\{\!|\mathit{expr}|\!\}\rrbracket r\wedge\mathcal{C}\llbracket \mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_p\rrbracket r\\ &\quad\wedge\ \mathcal{C}\llbracket \mathcal{T}_{Ob}\{\!|o_p^*|\!\}\rrbracket r\\ &= \mathsf{true}\wedge\mathsf{true}\wedge\mathsf{true} = \mathsf{true}\end{aligned}$$

which proves the thesis.

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

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

$$\begin{aligned}&\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{false}\ \vee\ \mathcal{E}\llbracket \mathit{expr}\rrbracket r = \perp\\ &\vee\ (\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \mathsf{not\text{-}app})\end{aligned}$$

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

$$\begin{aligned}&\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \mathsf{target} : \mathit{expr}\ \mathsf{policies} : (p^+)^k\\ &\qquad\qquad\qquad \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}|\!\}\downarrow_n\\ &= \neg\ \mathcal{T}_E\{\!|\mathit{expr}|\!\}\vee(\mathcal{T}_E\{\!|\mathit{expr}|\!\}\wedge\mathcal{T}_A\{\!|\mathsf{alg}_{\mathsf{all}}, (p^+)^k|\!\}\downarrow_n)\end{aligned}$$

The thesis then directly follows by Lemmas B.1 and B.3, due to the induction hypothesis and the definition of $\mathcal{C}$.

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

$$\begin{aligned}&\neg(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\\ &\quad\wedge\ (\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\Longrightarrow\mathcal{O}\llbracket o_e^*\rrbracket r = io_2^*)\\ &\wedge\neg(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{false}\vee\ \mathcal{E}\llbracket \mathit{expr}\rrbracket r = \perp\\ &\quad\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \mathsf{not\text{-}app}))\end{aligned}\tag{C}$$

where, to recall the connection between the effect returned by the combining algorithm and the sequence of obligations that is instantiated, we exploit the tautology

$$\begin{aligned}&\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\wedge\mathcal{O}\llbracket o_e^*\rrbracket r = io_2^*\\ &=\\ &\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\\ &\wedge(\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\Longrightarrow\mathcal{O}\llbracket o_e^*\rrbracket r = io_2^*)\end{aligned}$$

By applying standard boolean laws and reasoning on function codomains, the Condition (C) above can be rewritten as follows

$$\begin{aligned}&\neg(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\\ &\quad\wedge\ (\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\Longrightarrow\mathcal{O}\llbracket o_e^*\rrbracket r = io_2^*)\\ &\wedge\neg(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{false}\vee\ \mathcal{E}\llbracket \mathit{expr}\rrbracket r = \perp\\ &\quad\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r = \mathsf{true}\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \mathsf{not\text{-}app}))\\ &=\\ &(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\neq\mathsf{true}\ \vee\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\in\{\mathsf{not\text{-}app}, \mathsf{indet}\}\\ &\quad\vee\ (\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\wedge\mathcal{O}\llbracket o_e^*\rrbracket r = \mathsf{error}))\\ &\wedge\ (\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\\ &\quad\wedge(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\neq\mathsf{true}\ \vee\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\neq\mathsf{not\text{-}app}))\\ &=\\ &(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\neq\mathsf{true}\ \vee\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\in\{\mathsf{not\text{-}app}, \mathsf{indet}\}\\ &\quad\vee\ (\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\wedge\mathcal{O}\llbracket o_e^*\rrbracket r = \mathsf{error}))\\ &\wedge(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\\ &\quad\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\wedge\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\neq\mathsf{not\text{-}app}))\\ &=\\ &\quad\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\\ &\quad\wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\neq\mathsf{not\text{-}app})\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\\ &\quad\wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\in\{\mathsf{not\text{-}app}, \mathsf{indet}\})\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\\ &\quad\wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\neq\mathsf{not\text{-}app}\\ &\quad\wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\in\{\mathsf{not\text{-}app}, \mathsf{indet}\})\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\wedge\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\\ &\quad\wedge\ \mathcal{O}\llbracket o_e^*\rrbracket r = \mathsf{error})\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\wedge\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r\neq\mathsf{not\text{-}app}\\ &\quad\wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\wedge\mathcal{O}\llbracket o_e^*\rrbracket r = \mathsf{error})\\ &=\\ &\quad\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{true}, \mathsf{false}, \perp\}\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\wedge\mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \mathsf{indet})\\ &\vee(\mathcal{E}\llbracket \mathit{expr}\rrbracket r\notin\{\mathsf{false}, \perp\}\\ &\quad\ \wedge\ \mathcal{A}\llbracket \mathsf{alg}_{\mathsf{all}}, (p^+)^k\rrbracket r = \langle e\ \ io_1^*\rangle\wedge\mathcal{O}\llbracket o_e^*\rrbracket r = \mathsf{error})\\ &=\end{aligned}$$

$$
\begin{array}{l}
\quad \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \\
\lor(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \land \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \land \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \\
\lor(\mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \\
\quad \land\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e \;\; io_1^*\rangle \land \mathcal{O}[\![o_e^*]\!]r = \mathsf{error}) \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \\
\quad \land\mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e \;\; io_1^*\rangle \land \mathcal{O}[\![o_e^*]\!]r = \mathsf{error}) \\
= \\
\quad \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \land \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}) \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \land \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle e \;\; io_1^*\rangle \\
\quad \land\mathcal{O}[\![o_e^*]\!]r = \mathsf{error}) \\
= \\
\quad \mathcal{E}[\![expr]\!]r \notin \{\mathsf{true}, \mathsf{false}, \bot\} \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\land\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \mathsf{indet}\ ) \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\land\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle \mathsf{permit} \;\; io_1^*\rangle \\
\quad \land\mathcal{O}[\![o_p^*]\!]r = \mathsf{error}) \\
\lor(\mathcal{E}[\![expr]\!]r = \mathsf{true} \;\land\; \mathcal{A}[\![\mathsf{alg}_{\mathsf{all}}, (p^+)^k]\!]r = \langle \mathsf{deny} \;\; io_1^*\rangle \\
\quad \land\mathcal{O}[\![o_d^*]\!]r = \mathsf{error})
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{T}_P\{\!|\{\mathsf{alg}_{\mathsf{all}}\ \mathsf{target} : expr\ \mathsf{policies} : (p^+)^k \\
\qquad\qquad\qquad\qquad\quad \mathsf{obl\text{-}p} : o_p^*\ \ \mathsf{obl\text{-}d} : o_d^*\ \}|\!\}\downarrow_i = \\
\quad \neg\,(\mathtt{isBool}(\mathcal{T}_E\{\!|expr|\!\})\ \lor\ \mathtt{isMiss}(\mathcal{T}_E\{\!|expr|\!\})) \\
\quad \lor\,(\mathcal{T}_E\{\!|expr|\!\} \land\ \mathcal{T}_A\{\!|a, (p^+)^k|\!\}\downarrow_i) \\
\quad \lor\,(\mathcal{T}_E\{\!|expr|\!\} \land\ \mathcal{T}_A\{\!|a, (p^+)^k|\!\}\downarrow_p \land \neg\,\mathcal{T}_{Ob}\{\!|o_p^*|\!\}\ ) \\
\quad \lor\,(\mathcal{T}_E\{\!|expr|\!\} \land\ \mathcal{T}_A\{\!|a, (p^+)^k|\!\}\downarrow_d \land \neg\,\mathcal{T}_{Ob}\{\!|o_d^*|\!\}\ )
\end{array}
$$

The thesis then follows by Lemmas B.1, B.2 and B.3, due to the induction hypothesis and the definition of $\mathcal{C}$. □